Answer Set Solving in Practice

Torsten Schaub

University of Potsdam

Outline

Outline

Goal: Declarative problem solving

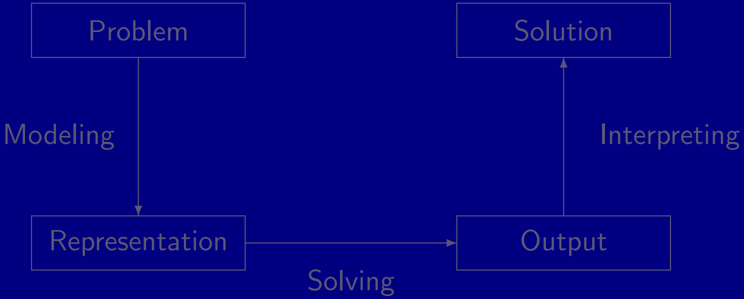
- *"What is the problem?"*

  instead of
- *"How to solve the problem?"*

$$\begin{tikzcd}
\text{Problem} \arrow[d, "\text{Modeling}"'] & \text{Solution} \\
\text{Representation} \arrow[r, "\text{Solving}"'] & \text{Output} \arrow[u, "\text{Interpreting}"']
\end{tikzcd}$$

Goal: Declarative problem solving

- *"What is the problem?"*
  instead of
- *"How to solve the problem?"*



Goal: Declarative problem solving

- *"What is the problem?"*

  instead of
- *"How to solve the problem?"*

$$\begin{tikzcd}
\boxed{\text{Problem}} \arrow[d, "\text{Modeling}"'] & & \boxed{\text{Solution}} \\
\boxed{\text{Representation}} \arrow[rr, "\text{Solving}"'] & & \boxed{\text{Output}} \arrow[u, "\text{Interpreting}"']
\end{tikzcd}$$

Answer Set Programming (ASP) *in a Nutshell*

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities

  tailored to Knowledge Representation and Reasoning
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way (being more compact than SAT)
- The versatility of ASP is reflected by the ASP solver `clasp`, winning first places at ASP, CASC, MISC, PB, and SAT
  - `http://potassco.sourceforge.net`
- ASP embraces many emerging application areas, eg.
  - RoboCup@Home at USTC, Beijing
  - Configuration at SIEMENS, Vienna

Answer Set Programming (ASP) *in a Nutshell*

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities

  tailored to Knowledge Representation and Reasoning
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way (being more compact than SAT)
- The versatility of ASP is reflected by the ASP solver `clasp`, winning first places at ASP, CASC, MISC, PB, and SAT
  - `http://potassco.sourceforge.net`
- ASP embraces many emerging application areas, eg.
  - RoboCup@Home at USTC, Beijing
  - Configuration at SIEMENS, Vienna

Answer Set Programming (ASP) *in a Nutshell*

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities

  tailored to Knowledge Representation and Reasoning
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way (being more compact than SAT)
- The versatility of ASP is reflected by the ASP solver `clasp`, winning first places at ASP, CASC, MISC, PB, and SAT
  - `http://potassco.sourceforge.net`
- ASP embraces many emerging application areas, eg.
  - RoboCup@Home at USTC, Beijing
  - Configuration at SIEMENS, Vienna

Answer Set Programming (ASP) *in a Nutshell*

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities

  tailored to Knowledge Representation and Reasoning
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way (being more compact than SAT)
- The versatility of ASP is reflected by the ASP solver `clasp`, winning first places at ASP, CASC, MISC, PB, and SAT
  - `http://potassco.sourceforge.net`
- ASP embraces many emerging application areas, eg.
  - RoboCup@Home at USTC, Beijing
  - Configuration at SIEMENS, Vienna

Answer Set Programming (ASP) *in a Nutshell*

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities

  tailored to Knowledge Representation and Reasoning
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way (being more compact than SAT)
- The versatility of ASP is reflected by the ASP solver `clasp`, winning first places at ASP, CASC, MISC, PB, and SAT
  - `http://potassco.sourceforge.net`
- ASP embraces many emerging application areas, eg.
  - RoboCup@Home at USTC, Beijing
  - Configuration at SIEMENS, Vienna

Outline

Declarative problem solving

- *"What is the problem?"*

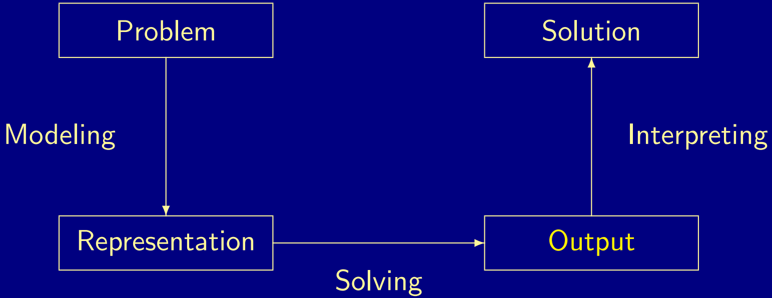
  instead of
- *"How to solve the problem?"*

$$\begin{tikzcd}
\boxed{\text{Problem}} \arrow[d, swap, "\text{Modeling}"] & & \boxed{\text{Solution}} \\
\boxed{\text{Representation}} \arrow[rr, swap, "\text{Solving}"] & & \boxed{\text{Output}} \arrow[u, swap, "\text{Interpreting}"]
\end{tikzcd}$$

Declarative problem solving

- *"What is the problem?"*

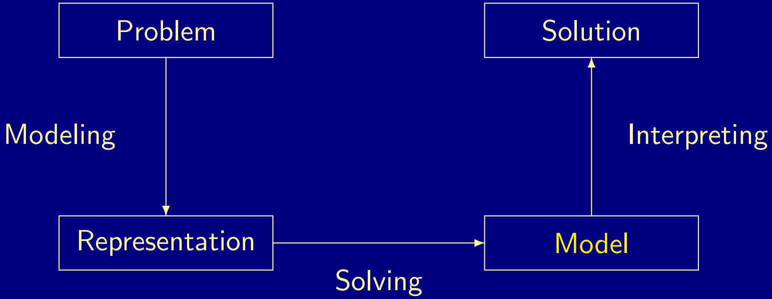
  instead of
- *"How to solve the problem?"*



Declarative model-based problem solving

- *"What is the problem?"*

  instead of
- *"How to solve the problem?"*



Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

$$\Phi \quad \boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$$

Formula Φ has one stable model, often called answer set:

$$\{p, q\}$$

$$P_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}}$$

Informally, a set X of atoms is an stable model of a logic program P

if X is a (classical) model of P and

if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

$$\Phi \quad \boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$$

Formula Φ has one stable model, often called answer set:

$$\{p, q\}$$

$$P_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q,\ \sim r \end{array}}$$

Informally, a set X of atoms is an stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$\Phi \quad \boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$

$\{p, q\}, \{q, r\},$ and $\{p, q, r\}$

Formula Φ has one stable model, often called answer set:

$$\boxed{\begin{array}{ccc} p & \mapsto & 1 \\ q & \mapsto & 1 \\ r & \mapsto & 0 \end{array}}$$

$\{p, q\}$

$P_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}}$

Informally, a set X of atoms is an stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\{p,q\}, \{q,r\}, \text{ and } \{p,q,r\}$$

Φ $\boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$

Formula Φ has one stable model, often called answer set:

$$\{p,q\}$$

P_Φ $\boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}}$

Informally, a set X of atoms is an stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Φ $\boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$

Consider the logical formula Φ and its three (classical) models:

$\{p, q\}, \{q, r\}$, and $\{p, q, r\}$

Formula Φ has one stable model, often called answer set:

$\{p, q\}$

P_Φ $\boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}}$

Informally, a set X of atoms is an stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

$$\Phi \quad \boxed{q \ \land \ (q \land \neg r \rightarrow p)}$$

Formula Φ has one stable model, often called answer set:

$$\{p, q\}$$

$$P_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \ \sim r \end{array}}$$

Informally, a set X of atoms is an stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

$$\Phi \quad \boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$$

Formula Φ has one stable model, often called answer set:

$$\{p, q\}$$

$$P_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}}$$

Informally, a set X of atoms is an stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\{p,q\}, \{q,r\}, \text{ and } \{p,q,r\}$$

$$\Phi \quad \boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$$

Formula Φ has one stable model, often called answer set:

$$\{p,q\}$$

$$P_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q,\ \sim r \end{array}}$$

Informally, a set X of atoms is an stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

ASP versus SAT

ASP	SAT
Model generation	
Bottom-up	
Constructive Logic	Classical Logic
Closed (and open) world reasoning	Open world reasoning
Modeling language	—
Complex reasoning modes	Satisfiability testing
Satisfiability	Satisfiability
Enumeration/Projection	—
Optimization	—
Intersection/Union	—
(Turing +) $NP(^{NP})$	NP

ASP versus SAT

ASP	SAT
Model generation	
Bottom-up	
Constructive Logic	Classical Logic
Closed (and open) world reasoning	Open world reasoning
Modeling language	—
Complex reasoning modes	Satisfiability testing
Satisfiability	Satisfiability
Enumeration/Projection	—
Optimization	—
Intersection/Union	—
(Turing +) $NP(^{NP})$	NP

Formal Definition

- Syntax
  - A rule, r, is an expression of the form

    $$a \leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_n,$$

    where $0 \leq m, n$ and each a, b_i, c_j is an atom
  - A logic program is a finite set of rules

- Semantics

  The reduct, P^X, of a program P relative to a set X of atoms is defined by

  $$P^X = \{\, a \leftarrow b_1, \dots, b_m \mid r \in P \text{ and } \{c_1, \dots, c_n\} \cap X = \emptyset \}$$

  The $\subseteq$–smallest model of P^X is denoted by $Cn(P^X)$
  A set X of atoms is an stable model of a program P, if

  $$X = Cn(P^X)$$

Formal Definition

- Syntax
  - A rule, r, is an expression of the form

$$a \leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_n,$$

    where $0 \leq m, n$ and each a, b_i, c_j is an atom
  - A logic program is a finite set of rules

- Semantics
  - The reduct, P^X, of a program P relative to a set X of atoms is defined by

$$P^X = \{\, a \leftarrow b_1, \dots, b_m \mid r \in P \text{ and } \{c_1, \dots, c_n\} \cap X = \emptyset \}$$

  - The $\subseteq$–smallest model of P^X is denoted by $Cn(P^X)$
  - A set X of atoms is an stable model of a program P, if

$$X = Cn(P^X)$$

Formal Definition

- Syntax
  - A rule, r, is an expression of the form

    $$a \leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_n,$$

    where $0 \leq m, n$ and each a, b_i, c_j is an atom
  - A logic program is a finite set of rules

- Semantics
  - The reduct, P^X, of a program P relative to a set X of atoms is defined by

    $$P^X = \{\, a \leftarrow b_1, \dots, b_m \mid r \in P \text{ and } \{c_1, \dots, c_n\} \cap X = \emptyset \}$$

  - The $\subseteq$–smallest model of P^X is denoted by $Cn(P^X)$
  - A set X of atoms is an stable model of a program P, if

    $$X = Cn(P^X)$$

Formal Definition

- Syntax
  - A rule, r, is an expression of the form

    $$a \leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_n,$$

    where $0 \leq m, n$ and each a, b_i, c_j is an atom
  - A logic program is a finite set of rules
- Semantics
  - The reduct, P^X, of a program P relative to a set X of atoms is defined by

    $$P^X = \{\, a \leftarrow b_1, \dots, b_m \mid r \in P \text{ and } \{c_1, \dots, c_n\} \cap X = \emptyset \}$$
  - The $\subseteq$–smallest model of P^X is denoted by $Cn(P^X)$
  - A set X of atoms is an stable model of a program P, if

    $$X = Cn(P^X)$$

Formal Definition

- Syntax
  - A rule, r, is an expression of the form

    $$a \leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_n,$$

    where $0 \leq m, n$ and each a, b_i, c_j is an atom
  - A logic program is a finite set of rules
- Semantics
  - The reduct, P^X, of a program P relative to a set X of atoms is defined by

    $$P^X = \{\, a \leftarrow b_1, \dots, b_m \mid r \in P \text{ and } \{c_1, \dots, c_n\} \cap X = \emptyset\}$$
  - The $\subseteq$–smallest model of P^X is denoted by $Cn(P^X)$
  - A set X of atoms is an stable model of a program P, if

    $$X = Cn(P^X)$$

Formal Definition

- Syntax
  - A rule, r, is an expression of the form

    $$a \leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_n,$$

    where $0 \leq m, n$ and each a, b_i, c_j is an atom
  - A logic program is a finite set of rules
- Semantics
  - The reduct, P^X, of a program P relative to a set X of atoms is defined by

    $$P^X = \{\, a \leftarrow b_1, \dots, b_m \mid r \in P \text{ and } \{c_1, \dots, c_n\} \cap X = \emptyset \}$$
  - The $\subseteq$–smallest model of P^X is denoted by $Cn(P^X)$
  - A set X of atoms is an stable model of a program P, if

    $$X = Cn(P^X)$$

Formal Definition

- Syntax
  - A rule, r, is an expression of the form

    $$a \leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_n,$$

    where $0 \leq m, n$ and each a, b_i, c_j is an atom
  - A logic program is a finite set of rules
- Semantics
  - The reduct, P^X, of a program P relative to a set X of atoms is defined by

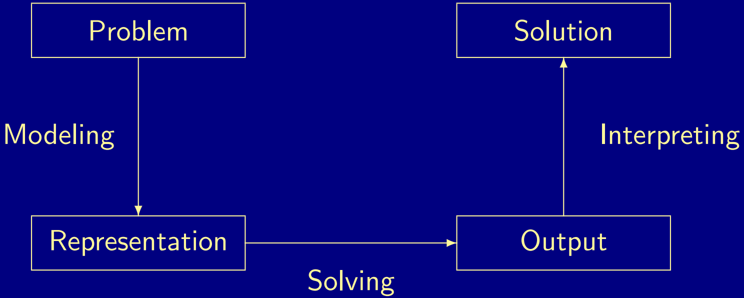
    $$P^X = \{\ a \leftarrow b_1, \dots, b_m \mid r \in P \text{ and } \{c_1, \dots, c_n\} \cap X = \emptyset\}$$
  - The $\subseteq$–smallest model of P^X is denoted by $Cn(P^X)$
  - A set X of atoms is an stable model of a program P, if

    $$X = Cn(P^X)$$

Declarative problem solving

- *"What is the problem?"*

  instead of
- *"How to solve the problem?"*



Declarative problem solving

- *"What is the problem?"*

  instead of
- *"How to solve the problem?"*

$$\begin{tikzcd}
\boxed{\text{Problem}} \arrow[d, "\text{Modeling}"'] & & \boxed{\text{Solution}} \\
\boxed{\text{Representation}} \arrow[rr, "\text{Solving}"'] & & \boxed{\text{Output}} \arrow[u, "\text{Interpreting}"']
\end{tikzcd}$$

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a,b,c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) ; q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #times, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) ; q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #times, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) ; q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #times, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) ; q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#times`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) ; q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#times`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) ; q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #times, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) ; q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#times`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) ; q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#times`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Declarative problem solving

- *"What is the problem?"*

  instead of

- *"How to solve the problem?"*

$$\begin{tikzcd}\boxed{\text{Problem}} \arrow[d, "\text{Modeling}"'] & \boxed{\text{Solution}} \\ \boxed{\text{Representation}} \arrow[r, "\text{Solving}"'] & \boxed{\text{Output}} \arrow[u, "\text{Interpreting}"']\end{tikzcd}$$

Declarative problem solving

- *"What is the problem?"*
  instead of
- *"How to solve the problem?"*

$$\begin{tikzcd}
\boxed{\text{Problem}} \arrow[d, "\text{Modeling}"'] & \boxed{\text{Solution}} \\
\boxed{\text{Representation}} \arrow[r, "\text{Solving}"'] & \boxed{\text{Output}} \arrow[u, "\text{Interpreting}"']
\end{tikzcd}$$

Reasoning Modes

- Satisfiability
- Enumeration†
- Projection†
- Intersection‡
- Union‡
- Optimization

† without solution recording

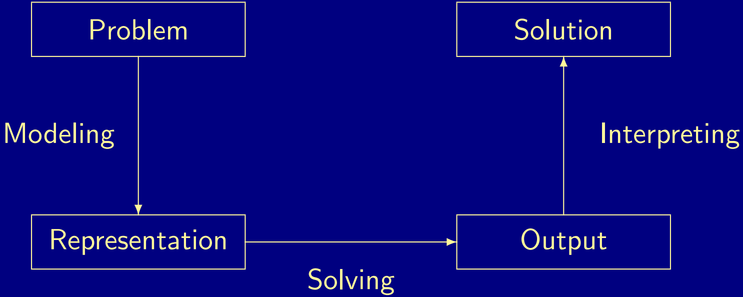
‡ without solution enumeration

Outline

Declarative problem solving

- *"What is the problem?"*

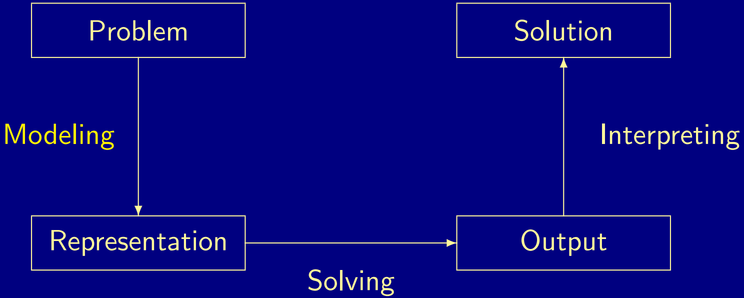
  instead of
- *"How to solve the problem?"*


$$\begin{tikzcd} \boxed{\text{Problem}} \arrow[d, "\text{Modeling}"'] & \boxed{\text{Solution}} \\ \boxed{\text{Representation}} \arrow[r, "\text{Solving}"'] & \boxed{\text{Output}} \arrow[u, "\text{Interpreting}"'] \end{tikzcd}$$

Declarative problem solving

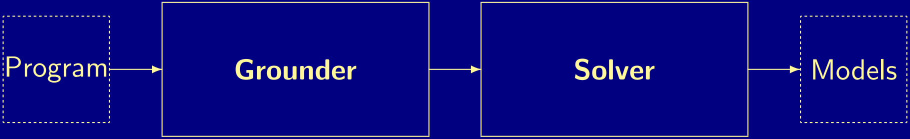
- *"What is the problem?"*

  instead of
- *"How to solve the problem?"*

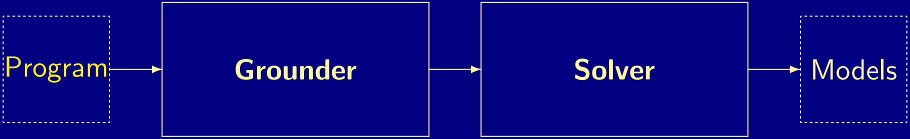


$$\begin{tikzcd}
\text{Problem} \arrow[d, "\text{Modeling}"'] & & \text{Solution} \\
\text{Representation} \arrow[rr, "\text{Solving}"'] & & \text{Output} \arrow[u, "\text{Interpreting}"']
\end{tikzcd}$$

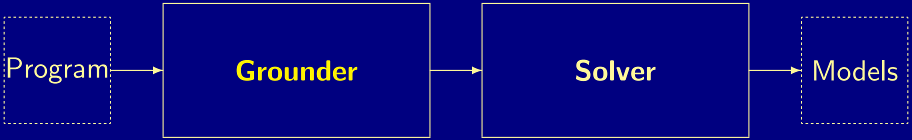
ASP Solving Process



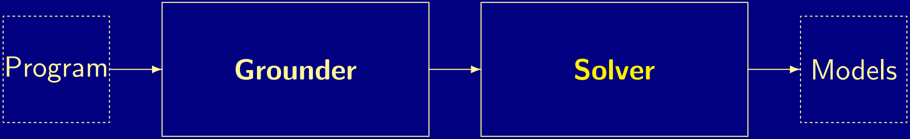
ASP Solving Process



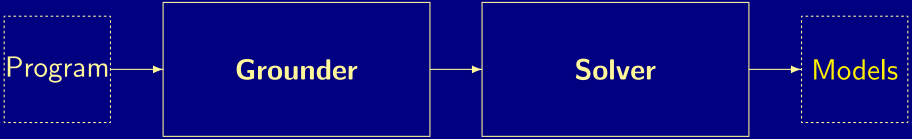
ASP Solving Process



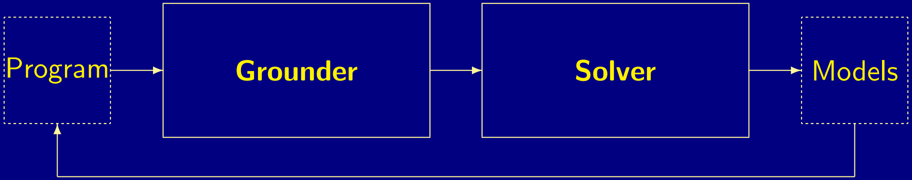
ASP Solving Process



ASP Solving Process

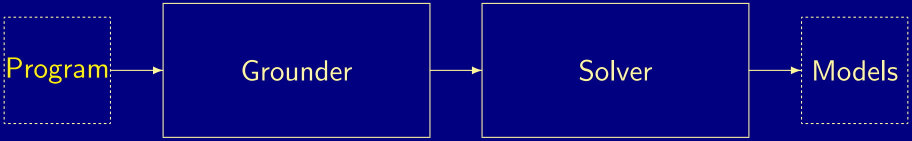


ASP Solving Process



Outline

ASP Solving Process



Graph Coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).


1 { color(X,C) : col(C) } 1 :- node(X).


:- edge(X,Y), color(X,C), color(Y,C).
```

Graph Coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).


1 { color(X,C) : col(C) } 1 :- node(X).


:- edge(X,Y), color(X,C), color(Y,C).
```

Graph Coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).


1 { color(X,C) : col(C) } 1 :- node(X).


:- edge(X,Y), color(X,C), color(Y,C).
```

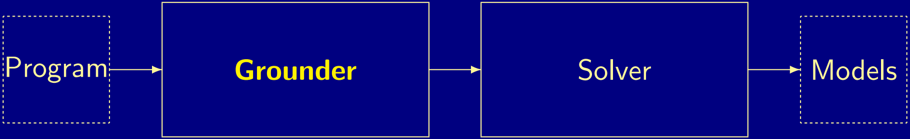
Graph Coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).


1 { color(X,C) : col(C) } 1 :- node(X).


:- edge(X,Y), color(X,C), color(Y,C).
```

Graph Coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).


1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

ASP Solving Process



Graph Coloring: Grounding

```
$ gringo -t color.lp
```

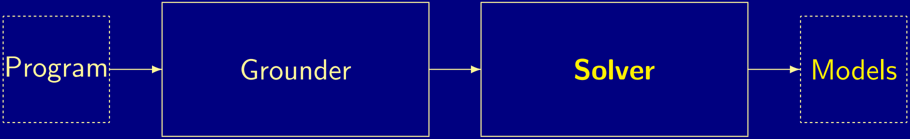
```
node(1).  node(2).  node(3).  node(4).  node(5).  node(6).

edge(1,2).  edge(1,3).  edge(1,4).  edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).  edge(4,1).  edge(4,2).  edge(5,3).
edge(5,4).  edge(5,6).  edge(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).

1 {color(1,r), color(1,b), color(1,g)} 1.
1 {color(2,r), color(2,b), color(2,g)} 1.
1 {color(3,r), color(3,b), color(3,g)} 1.
1 {color(4,r), color(4,b), color(4,g)} 1.
1 {color(5,r), color(5,b), color(5,g)} 1.
1 {color(6,r), color(6,b), color(6,g)} 1.

 :- color(1,r), color(2,r).  :- color(2,g), color(5,g).  ...  :- color(6,r), color(2,r).
 :- color(1,b), color(2,b).  :- color(2,r), color(6,r).       :- color(6,b), color(2,b).
 :- color(1,g), color(2,g).  :- color(2,b), color(6,b).       :- color(6,g), color(2,g).
 :- color(1,r), color(3,r).  :- color(2,g), color(6,g).       :- color(6,r), color(3,r).
 :- color(1,b), color(3,b).  :- color(3,r), color(1,r).       :- color(6,b), color(3,b).
 :- color(1,g), color(3,g).  :- color(3,b), color(1,b).       :- color(6,g), color(3,g).
 :- color(1,r), color(4,r).  :- color(3,g), color(1,g).       :- color(6,r), color(5,r).
 :- color(1,b), color(4,b).  :- color(3,r), color(4,r).       :- color(6,b), color(5,b).
 :- color(1,g), color(4,g).  :- color(3,b), color(4,b).       :- color(6,g), color(5,g).
 :- color(2,r), color(4,r).  :- color(3,g), color(4,g).
 :- color(2,b), color(4,b).  :- color(3,r), color(5,r).
 :- color(2,g), color(4,g).  :- color(3,b), color(5,b).
```

Graph Coloring: Grounding

```
$ gringo -t color.lp
```

```
node(1).  node(2).  node(3).  node(4).  node(5).  node(6).

edge(1,2).  edge(1,3).  edge(1,4).  edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).  edge(4,1).  edge(4,2).  edge(5,3).
edge(5,4).  edge(5,6).  edge(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).

1 {color(1,r), color(1,b), color(1,g)} 1.
1 {color(2,r), color(2,b), color(2,g)} 1.
1 {color(3,r), color(3,b), color(3,g)} 1.
1 {color(4,r), color(4,b), color(4,g)} 1.
1 {color(5,r), color(5,b), color(5,g)} 1.
1 {color(6,r), color(6,b), color(6,g)} 1.

 :- color(1,r), color(2,r).  :- color(2,g), color(5,g).  ...  :- color(6,r), color(2,r).
 :- color(1,b), color(2,b).  :- color(2,r), color(6,r).       :- color(6,b), color(2,b).
 :- color(1,g), color(2,g).  :- color(2,b), color(6,b).       :- color(6,g), color(2,g).
 :- color(1,r), color(3,r).  :- color(2,g), color(6,g).       :- color(6,r), color(3,r).
 :- color(1,b), color(3,b).  :- color(3,r), color(1,r).       :- color(6,b), color(3,b).
 :- color(1,g), color(3,g).  :- color(3,b), color(1,b).       :- color(6,g), color(3,g).
 :- color(1,r), color(4,r).  :- color(3,g), color(1,g).       :- color(6,r), color(5,r).
 :- color(1,b), color(4,b).  :- color(3,r), color(4,r).       :- color(6,b), color(5,b).
 :- color(1,g), color(4,g).  :- color(3,b), color(4,b).       :- color(6,g), color(5,g).
 :- color(2,r), color(4,r).  :- color(3,g), color(4,g).
 :- color(2,b), color(4,b).  :- color(3,r), color(5,r).
 :- color(2,g), color(4,g).  :- color(3,b), color(5,b).
```

ASP Solving Process



Graph Coloring: Solving

```
$ gringo color.lp | clasp 0
```

```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
edge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,g) color(4,b) color(3,r) color(2,r) color(1,g)
Answer: 2
edge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,g) color(4,r) color(3,b) color(2,b) color(1,g)
Answer: 3
edge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,b) color(4,g) color(3,r) color(2,r) color(1,b)
Answer: 4
edge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,b) color(4,r) color(3,g) color(2,g) color(1,b)
Answer: 5
edge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,r) color(4,g) color(3,b) color(2,b) color(1,r)
Answer: 6
edge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,r) color(4,b) color(3,g) color(2,g) color(1,r)
SATISFIABLE

Models      : 6
Time        : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
```

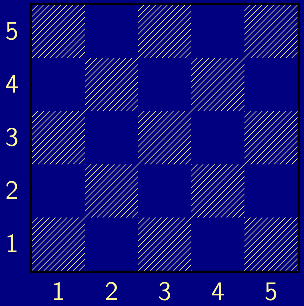
Graph Coloring: Solving

```
$ gringo color.lp | clasp 0
```

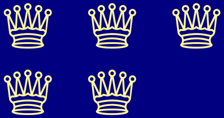
```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
edge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,g) color(4,b) color(3,r) color(2,r) color(1,g)
Answer: 2
edge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,g) color(4,r) color(3,b) color(2,b) color(1,g)
Answer: 3
edge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,b) color(4,g) color(3,r) color(2,r) color(1,b)
Answer: 4
edge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,b) color(4,r) color(3,g) color(2,g) color(1,b)
Answer: 5
edge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,r) color(4,g) color(3,b) color(2,b) color(1,r)
Answer: 6
edge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,r) color(4,b) color(3,g) color(2,g) color(1,r)
SATISFIABLE

Models      : 6
Time        : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
```

Outline

The n-Queens Problem



- Place n queens on an $n \times n$ chess board
- Queens must not attack one another



Defining the Field

queens.lp

```
row(1..n).
col(1..n).
```

- Create file queens.lp
- Define the field
  - *n* rows
  - *n* columns

Defining the Field

Running . . .

```
$ clingo queens.lp -c n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5)
SATISFIABLE

Models      : 1
Time        : 0.000
  Prepare   : 0.000
  Prepro.   : 0.000
  Solving   : 0.000
```

Placing some Queens

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
```

- Guess a solution candidate
  by placing some queens on the board

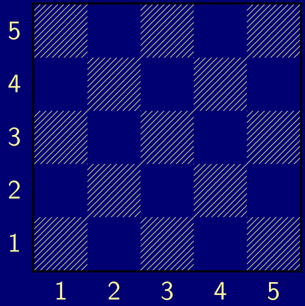
Placing some Queens

Running . . .

```
$ clingo queens.lp -c n=5 3
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) queen(1,1)
Answer: 3
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) queen(2,1)
SATISFIABLE

Models       : 3+
```

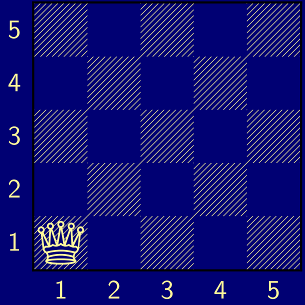
Placing some Queens: Answer 1

Answer 1



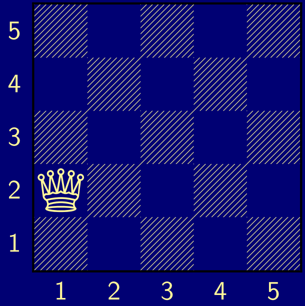
Placing some Queens: Answer 2

Answer 2



Placing some Queens: Answer 3

Answer 3



Placing *n* Queens

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
```

- Place exactly *n* queens on the board

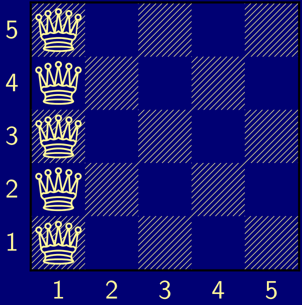
Placing *n* Queens

Running . . .

```
$ clingo queens.lp -c n=5 2
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,1) queen(4,1) queen(3,1) \
queen(2,1) queen(1,1)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(1,2) queen(4,1) queen(3,1) \
queen(2,1) queen(1,1)
...
```

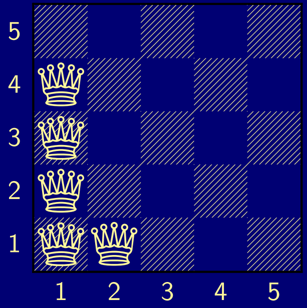
Placing *n* Queens: Answer 1

Answer 1



Placing *n* Queens: Answer 2

Answer 2



Horizontal and vertical Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
:- queen(I,J), queen(I,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
```

- Forbid horizontal attacks
- Forbid vertical attacks

Horizontal and vertical Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
:- queen(I,J), queen(I,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
```

- Forbid horizontal attacks
- Forbid vertical attacks

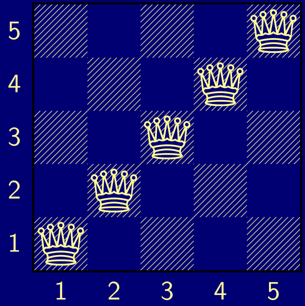
Horizontal and vertical Attack

Running . . .

```
$ clingo queens.lp -c n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,5) queen(4,4) queen(3,3) \
queen(2,2) queen(1,1)
...
```

Horizontal and vertical Attack: Answer 1

Answer 1



Diagonal Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
:- queen(I,J), queen(I,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ), I-J == II-JJ.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ), I+J == II+JJ.
```

- Forbid diagonal attacks

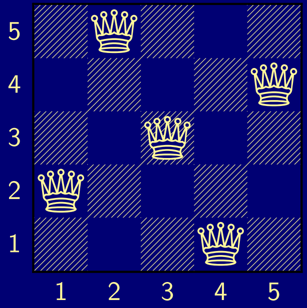
Diagonal Attack

Running . . .

```
$ clingo queens.lp -c n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(4,5) queen(1,4) queen(3,3) \
queen(5,2) queen(2,1)
SATISFIABLE

Models      : 1+
Time        : 0.000
  Prepare   : 0.000
  Prepro.   : 0.000
  Solving   : 0.000
```

Diagonal Attack: Answer 1

Answer 1



Optimizing

queens-opt.lp

```
1 { queen(I,1..n) } 1 :- I = 1..n.
1 { queen(1..n,J) } 1 :- J = 1..n.
 :- 2 { queen(D-J,J) }, D =   2..2*n.
 :- 2 { queen(D+J,J) }, D = 1-n..n-1.
```

- Encoding can be optimized
- Much faster to solve

Outline

Traveling Salesperson

```
node(1..6).

edge(1,2;3;4).  edge(2,4;5;6).  edge(3,1;4;5).
edge(4,1;2).    edge(5,3;4;6).  edge(6,2;3;5).

cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
cost(4,1,1).  cost(4,2,2).
cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
cost(6,2,4).  cost(6,3,3).  cost(6,5,1).
```

Traveling Salesperson

```
node(1..6).

edge(1,2;3;4).  edge(2,4;5;6).  edge(3,1;4;5).
edge(4,1;2).    edge(5,3;4;6).  edge(6,2;3;5).

cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
cost(4,1,1).  cost(4,2,2).
cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
cost(6,2,4).  cost(6,3,3).  cost(6,5,1).
```

Traveling Salesperson

```
node(1..6).

edge(1,2;3;4).  edge(2,4;5;6).  edge(3,1;4;5).
edge(4,1;2).    edge(5,3;4;6).  edge(6,2;3;5).

cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
cost(4,1,1).  cost(4,2,2).
cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
cost(6,2,4).  cost(6,3,3).  cost(6,5,1).
```

Traveling Salesperson

```
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize { cycle(X,Y) : cost(X,Y,C) = C }.
```

Traveling Salesperson

```
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize { cycle(X,Y) : cost(X,Y,C) = C }.
```

Traveling Salesperson

```
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize { cycle(X,Y) : cost(X,Y,C) = C }.
```

Traveling Salesperson

```
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize { cycle(X,Y) : cost(X,Y,C) = C }.
```

What is ASP good for?

- Combinatorial search problems in the realm of P, NP, and NP^{NP} (some with substantial amount of data), like
  - For instance, auctions, bio-informatics, computer-aided verification, configuration, constraint satisfaction, diagnosis, information integration, planning and scheduling, security analysis, semantic web, wire-routing, zoology and linguistics, and many more
- My favorite: Using ASP as a basis for a decision support system for NASA's space shuttle (Gelfond et al., Texas Tech)
- Our own applications:
  - Automatic synthesis of multiprocessor systems
  - Inconsistency detection, diagnosis, repair, and prediction in large biological networks
  - Home monitoring for risk prevention in ambient assisted living
  - General game playing

What is ASP good for?

- Combinatorial search problems in the realm of P, NP, and NP^{NP} (some with substantial amount of data), like
  - For instance, auctions, bio-informatics, computer-aided verification, configuration, constraint satisfaction, diagnosis, information integration, planning and scheduling, security analysis, semantic web, wire-routing, zoology and linguistics, and many more
- My favorite: Using ASP as a basis for a decision support system for NASA's space shuttle (Gelfond et al., Texas Tech)
- Our own applications:
  - Automatic synthesis of multiprocessor systems
  - Inconsistency detection, diagnosis, repair, and prediction in large biological networks
  - Home monitoring for risk prevention in ambient assisted living
  - General game playing

What is ASP good for?

- Combinatorial search problems in the realm of P, NP, and NP^{NP} (some with substantial amount of data), like
  - For instance, auctions, bio-informatics, computer-aided verification, configuration, constraint satisfaction, diagnosis, information integration, planning and scheduling, security analysis, semantic web, wire-routing, zoology and linguistics, and many more
- My favorite: Using ASP as a basis for a decision support system for NASA's space shuttle (Gelfond et al., Texas Tech)
- Our own applications:
  - Automatic synthesis of multiprocessor systems
  - Inconsistency detection, diagnosis, repair, and prediction in large biological networks
  - Home monitoring for risk prevention in ambient assisted living
  - General game playing

What does ASP offer?

- Integration of KR, DB, and SAT techniques
- Succinct, elaboration-tolerant problem representations
    - Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
    - including: data, frame axioms, exceptions, defaults, closures, etc.

What does ASP offer?

- Integration of KR, DB, and SAT techniques
- Succinct, elaboration-tolerant problem representations
  - Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
  - including: data, frame axioms, exceptions, defaults, closures, etc.

$$\textbf{ASP} = \textbf{DB}+\textbf{LP}+\textbf{KR}+\textbf{SAT}$$

Outline

Declarative problem solving

- *"What is the problem?"*

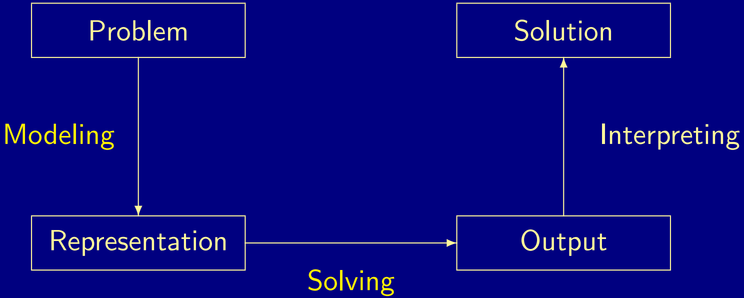
  instead of
- *"How to solve the problem?"*

$$\begin{tikzcd}
\boxed{\text{Problem}} \arrow[d, "\text{Modeling}"'] & & \boxed{\text{Solution}} \\
\boxed{\text{Representation}} \arrow[rr, "\text{Solving}"'] & & \boxed{\text{Output}} \arrow[u, "\text{Interpreting}"']
\end{tikzcd}$$

Declarative problem solving

- *"What is the problem?"*

  instead of
- *"How to solve the problem?"*


$$\begin{tikzcd}
\boxed{\text{Problem}} \arrow[d, "\text{Modeling}"'] & \boxed{\text{Solution}} \\
\boxed{\text{Representation}} \arrow[r, "\text{Solving}"'] & \boxed{\text{Output}} \arrow[u, "\text{Interpreting}"']
\end{tikzcd}$$

Grounding by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

gringo --text easy.lp

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

Grounding by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

gringo --text easy.lp

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

Solving by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

gringo easy.lp | clasp 0

```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
p(1) p(2) p(3) q(2)
SATISFIABLE

Models      : 1
Time        : 0.000s
CPU Time    : 0.000s
```

Solving by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

gringo easy.lp | clasp 0

```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
p(1) p(2) p(3) q(2)
SATISFIABLE

Models       : 1
Time         : 0.000s
CPU Time     : 0.000s
```

Reifying by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

gringo --reify easy.lp

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

gringo --reify easy.lp

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

gringo --reify easy.lp

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

gringo --text easy.lp

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

gringo --reify easy.lp

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
    #count{ p(1), p(2), p(3) }.
    :- #count{ p(3), p(2), p(1) } 2.
    q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
    wlist(0,0,pos(atom(p(1))),1).
    wlist(0,1,pos(atom(p(2))),1).
    wlist(0,2,pos(atom(p(3))),1).
    rule(pos(sum(0,0,3)),pos(conjunction(0))).
    set(1,pos(sum(0,0,2))).
    rule(pos(false),pos(conjunction(1))).
    set(2,pos(atom(p(2)))).
    set(2,pos(atom(p(3)))).
    rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
    #count{ p(1), p(2), p(3) }.
    :- #count{ p(3), p(2), p(1) } 2.
    q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
    wlist(0,0,pos(atom(p(1))),1).
    wlist(0,1,pos(atom(p(2))),1).
    wlist(0,2,pos(atom(p(3))),1).
    rule(pos(sum(0,0,3)),pos(conjunction(0))).
    set(1,pos(sum(0,0,2))).
    rule(pos(false),pos(conjunction(1))).
    set(2,pos(atom(p(2)))).
    set(2,pos(atom(p(3)))).
    rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

gringo --text easy.lp

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

gringo --reify easy.lp

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

gringo --text easy.lp

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

gringo --reify easy.lp

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
    #count{ p(1), p(2), p(3) }.
    :- #count{ p(3), p(2), p(1) } 2.
    q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
    wlist(0,0,pos(atom(p(1))),1).
    wlist(0,1,pos(atom(p(2))),1).
    wlist(0,2,pos(atom(p(3))),1).
    rule(pos(sum(0,0,3)),pos(conjunction(0))).
    set(1,pos(sum(0,0,2))).
    rule(pos(false),pos(conjunction(1))).
    set(2,pos(atom(p(2)))).
    set(2,pos(atom(p(3)))).
    rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
    #count{ p(1), p(2), p(3) }.
    :- #count{ p(3), p(2), p(1) } 2.
    q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
    wlist(0,0,pos(atom(p(1))),1).
    wlist(0,1,pos(atom(p(2))),1).
    wlist(0,2,pos(atom(p(3))),1).
    rule(pos(sum(0,0,3)),pos(conjunction(0))).
    set(1,pos(sum(0,0,2))).
    rule(pos(false),pos(conjunction(1))).
    set(2,pos(atom(p(2)))).
    set(2,pos(atom(p(3)))).
    rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

gringo --text easy.lp

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

gringo --reify easy.lp

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
    #count{ p(1), p(2), p(3) }.
    :- #count{ p(3), p(2), p(1) } 2.
    q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
    wlist(0,0,pos(atom(p(1))),1).
    wlist(0,1,pos(atom(p(2))),1).
    wlist(0,2,pos(atom(p(3))),1).
    rule(pos(sum(0,0,3)),pos(conjunction(0))).
    set(1,pos(sum(0,0,2))).
    rule(pos(false),pos(conjunction(1))).
    set(2,pos(atom(p(2)))).
    set(2,pos(atom(p(3)))).
    rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
    #count{ p(1), p(2), p(3) }.
    :- #count{ p(3), p(2), p(1) } 2.
    q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
    wlist(0,0,pos(atom(p(1))),1).
    wlist(0,1,pos(atom(p(2))),1).
    wlist(0,2,pos(atom(p(3))),1).
    rule(pos(sum(0,0,3)),pos(conjunction(0))).
    set(1,pos(sum(0,0,2))).
    rule(pos(false),pos(conjunction(1))).
    set(2,pos(atom(p(2)))).
    set(2,pos(atom(p(3)))).
    rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

gringo --text easy.lp

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

gringo --reify easy.lp

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

gringo --text easy.lp

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

gringo --reify easy.lp

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

gringo --text easy.lp

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

gringo --reify easy.lp

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
    #count{ p(1), p(2), p(3) }.
    :- #count{ p(3), p(2), p(1) } 2.
    q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
    wlist(0,0,pos(atom(p(1))),1).
    wlist(0,1,pos(atom(p(2))),1).
    wlist(0,2,pos(atom(p(3))),1).
    rule(pos(sum(0,0,3)),pos(conjunction(0))).
    set(1,pos(sum(0,0,2))).
    rule(pos(false),pos(conjunction(1))).
    set(2,pos(atom(p(2)))).
    set(2,pos(atom(p(3)))).
    rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
    #count{ p(1), p(2), p(3) }.
    :- #count{ p(3), p(2), p(1) } 2.
    q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
    wlist(0,0,pos(atom(p(1))),1).
    wlist(0,1,pos(atom(p(2))),1).
    wlist(0,2,pos(atom(p(3))),1).
    rule(pos(sum(0,0,3)),pos(conjunction(0))).
    set(1,pos(sum(0,0,2))).
    rule(pos(false),pos(conjunction(1))).
    set(2,pos(atom(p(2)))).
    set(2,pos(atom(p(3)))).
    rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

gringo --text easy.lp

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

gringo --reify easy.lp

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
    #count{ p(1), p(2), p(3) }.
    :- #count{ p(3), p(2), p(1) } 2.
    q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
    wlist(0,0,pos(atom(p(1))),1).
    wlist(0,1,pos(atom(p(2))),1).
    wlist(0,2,pos(atom(p(3))),1).
    rule(pos(sum(0,0,3)),pos(conjunction(0))).
    set(1,pos(sum(0,0,2))).
    rule(pos(false),pos(conjunction(1))).
    set(2,pos(atom(p(2)))).
    set(2,pos(atom(p(3)))).
    rule(pos(atom(q(2))),pos(conjunction(2))).
```

Reifying by example

```
gringo --text easy.lp
```

```
#count{ p(1), p(2), p(3) }.
:- #count{ p(3), p(2), p(1) } 2.
q(2) :- p(2), p(3).
```

```
gringo --reify easy.lp
```

```
wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).
```

Basic meta-encoding
`meta.lp`

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding `meta.lp`

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
`meta.lp`

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Basic meta-encoding
meta.lp

```
litb(B) :- rule(_,B).
litb(E) :- litb(pos(conjunction(S))), set(S,E).
litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

eleb(P) :- litb(pos(P)).
eleb(N) :- litb(neg(N)).

elem(E) :- eleb(E).
elem(E) :- rule(pos(E),_).
elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

hold(conjunction(S)) :- eleb(conjunction(S)),
                            hold(P)     : set(S,pos(P)),
                        not hold(N)     : set(S,neg(N)).

hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
                        L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
                                 not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).

 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
                     :- rule(pos(sum(L,S,U)),pos(B)), hold(B).

                     :- rule(pos(false),     pos(B)), hold(B).

#hide. #show hold(atom(A)).
```

Reified Grounding by example

```
gringo --reify easy.lp | gringo - meta.lp
```

```
eleb(atom(p(1))).       litb(pos(atom(p(1)))).       elem(atom(p(1))).       elem(false).
eleb(atom(p(2))).       litb(pos(atom(p(2)))).       elem(atom(p(2))).       elem(sum(0,0,2)).
eleb(atom(p(3))).       litb(pos(atom(p(3)))).       elem(atom(p(3))).       elem(sum(0,0,3)).
eleb(conjunction(0)).   litb(pos(conjunction(0))).   elem(atom(q(2))).
eleb(conjunction(1)).   litb(pos(conjunction(1))).   elem(conjunction(0)).
eleb(conjunction(2)).   litb(pos(conjunction(2))).   elem(conjunction(1)).
eleb(sum(0,0,2)).       litb(pos(sum(0,0,2))).       elem(conjunction(2)).

wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).

hold(conjunction(2)) :- hold(atom(p(3))),hold(atom(p(2))).
hold(conjunction(1)) :- hold(sum(0,0,2)).
hold(conjunction(0)).
hold(sum(0,0,2)) :- 0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 2.
hold(atom(q(2))) :- hold(conjunction(2)).
0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 3.
 :- hold(conjunction(1)).

#hide.
```

Reified Grounding by example

```
gringo --reify easy.lp | gringo - meta.lp
```

```
eleb(atom(p(1))).       litb(pos(atom(p(1)))).       elem(atom(p(1))).       elem(false).
eleb(atom(p(2))).       litb(pos(atom(p(2)))).       elem(atom(p(2))).       elem(sum(0,0,2)).
eleb(atom(p(3))).       litb(pos(atom(p(3)))).       elem(atom(p(3))).       elem(sum(0,0,3)).
eleb(conjunction(0)).   litb(pos(conjunction(0))).   elem(atom(q(2))).
eleb(conjunction(1)).   litb(pos(conjunction(1))).   elem(conjunction(0)).
eleb(conjunction(2)).   litb(pos(conjunction(2))).   elem(conjunction(1)).
eleb(sum(0,0,2)).       litb(pos(sum(0,0,2))).       elem(conjunction(2)).

wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).

hold(conjunction(2)) :- hold(atom(p(3))),hold(atom(p(2))).
hold(conjunction(1)) :- hold(sum(0,0,2)).
hold(conjunction(0)).
hold(sum(0,0,2)) :- 0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 2.
hold(atom(q(2))) :- hold(conjunction(2)).
0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 3.
 :- hold(conjunction(1)).

#hide.
```

Reified Grounding by example

```
gringo --reify easy.lp | gringo - meta.lp
```

```
eleb(atom(p(1))).       litb(pos(atom(p(1)))).       elem(atom(p(1))).       elem(false).
eleb(atom(p(2))).       litb(pos(atom(p(2)))).       elem(atom(p(2))).       elem(sum(0,0,2)).
eleb(atom(p(3))).       litb(pos(atom(p(3)))).       elem(atom(p(3))).       elem(sum(0,0,3)).
eleb(conjunction(0)).   litb(pos(conjunction(0))).   elem(atom(q(2))).
eleb(conjunction(1)).   litb(pos(conjunction(1))).   elem(conjunction(0)).
eleb(conjunction(2)).   litb(pos(conjunction(2))).   elem(conjunction(1)).
eleb(sum(0,0,2)).       litb(pos(sum(0,0,2))).       elem(conjunction(2)).

wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).

hold(conjunction(2)) :- hold(atom(p(3))),hold(atom(p(2))).
hold(conjunction(1)) :- hold(sum(0,0,2)).
hold(conjunction(0)).
hold(sum(0,0,2)) :- 0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 2.
hold(atom(q(2))) :- hold(conjunction(2)).
0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 3.
 :- hold(conjunction(1)).

#hide.
```

Reified Grounding by example

```
gringo --reify easy.lp | gringo - meta.lp
```

```
eleb(atom(p(1))).       litb(pos(atom(p(1)))).       elem(atom(p(1))).       elem(false).
eleb(atom(p(2))).       litb(pos(atom(p(2)))).       elem(atom(p(2))).       elem(sum(0,0,2)).
eleb(atom(p(3))).       litb(pos(atom(p(3)))).       elem(atom(p(3))).       elem(sum(0,0,3)).
eleb(conjunction(0)).   litb(pos(conjunction(0))).   elem(atom(q(2))).
eleb(conjunction(1)).   litb(pos(conjunction(1))).   elem(conjunction(0)).
eleb(conjunction(2)).   litb(pos(conjunction(2))).   elem(conjunction(1)).
eleb(sum(0,0,2)).       litb(pos(sum(0,0,2))).       elem(conjunction(2)).

wlist(0,0,pos(atom(p(1))),1).
wlist(0,1,pos(atom(p(2))),1).
wlist(0,2,pos(atom(p(3))),1).
rule(pos(sum(0,0,3)),pos(conjunction(0))).
set(1,pos(sum(0,0,2))).
rule(pos(false),pos(conjunction(1))).
set(2,pos(atom(p(2)))).
set(2,pos(atom(p(3)))).
rule(pos(atom(q(2))),pos(conjunction(2))).

hold(conjunction(2)) :- hold(atom(p(3))),hold(atom(p(2))).
hold(conjunction(1)) :- hold(sum(0,0,2)).
hold(conjunction(0)).
hold(sum(0,0,2)) :- 0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 2.
hold(atom(q(2))) :- hold(conjunction(2)).
0 #sum [ hold(atom(p(3)))=1, hold(atom(p(2)))=1, hold(atom(p(1)))=1 ] 3.
 :- hold(conjunction(1)).

#hide.
```

Solving by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

```
gringo --reify easy.lp | gringo - meta.lp | clasp 0
```

```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
hold(atom(p(3))) hold(atom(p(2))) hold(atom(p(1))) hold(atom(q(2)))
SATISFIABLE

Models      : 1
Time        : 0.000s
CPU Time    : 0.000s
```

Solving by example

easy.lp

```
{ p(1..3) }.
:- { p(X) } 2.
q(X) :- p(X), p(X+1), X>1.
```

gringo --reify easy.lp | gringo - meta.lp | clasp 0

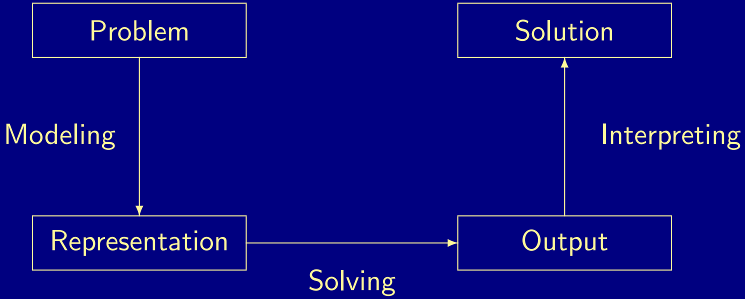
```
clasp version 2.0.0
Reading from stdin
Solving...
Answer: 1
hold(atom(p(3))) hold(atom(p(2))) hold(atom(p(1))) hold(atom(q(2)))
SATISFIABLE

Models      : 1
Time        : 0.000s
CPU Time    : 0.000s
```

Outline

Declarative problem solving

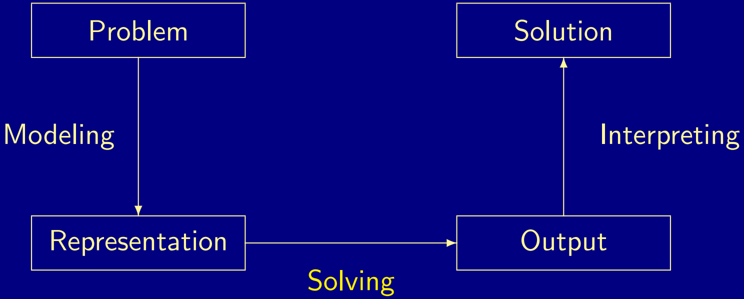
- *"What is the problem?"*

  instead of
- *"How to solve the problem?"*



Declarative problem solving

- *“What is the problem?”*
  instead of
- *“How to solve the problem?”*



Towards conflict-driven search

Boolean constraint solving algorithms pioneered for SAT led to:

- Traditional DPLL-style approach
  (DPLL stands for 'Davis-Putnam-Logemann-Loveland')
  - (Unit) propagation
  - (Chronological) backtracking
  - in ASP, eg *smodels*
- Modern CDCL-style approach
  (CDCL stands for 'Conflict-Driven Constraint Learning')
  - (Unit) propagation
  - Conflict analysis (via resolution)
  - Learning + Backjumping + Assertion
  - in ASP, eg *clasp*

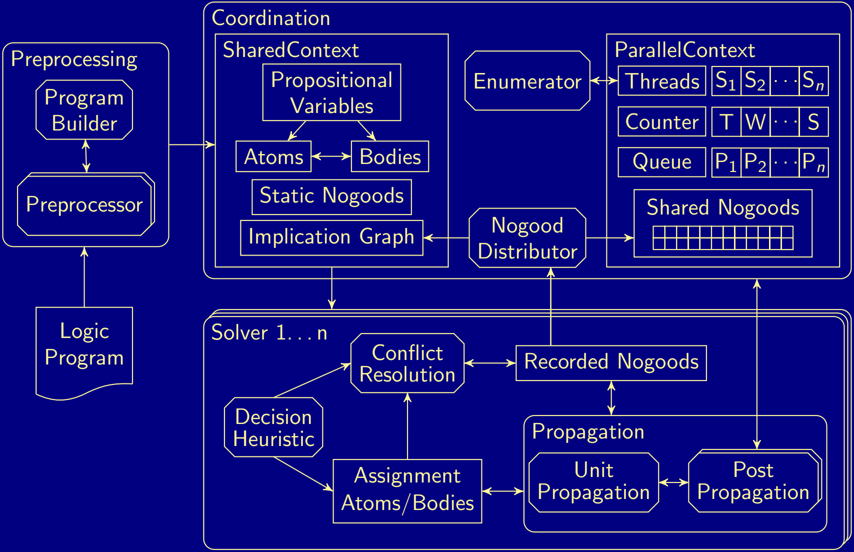
DPLL-style solving

```
loop
    propagate                                  // deterministically assign literals
    if no conflict then
        if all variables assigned then return solution
        else decide                            // non-deterministically assign some literal
    else
        if top-level conflict then return unsatisfiable
        else
            backtrack                          // unassign literals made after last decision
            flip                               // assign complement of last decision literal
```

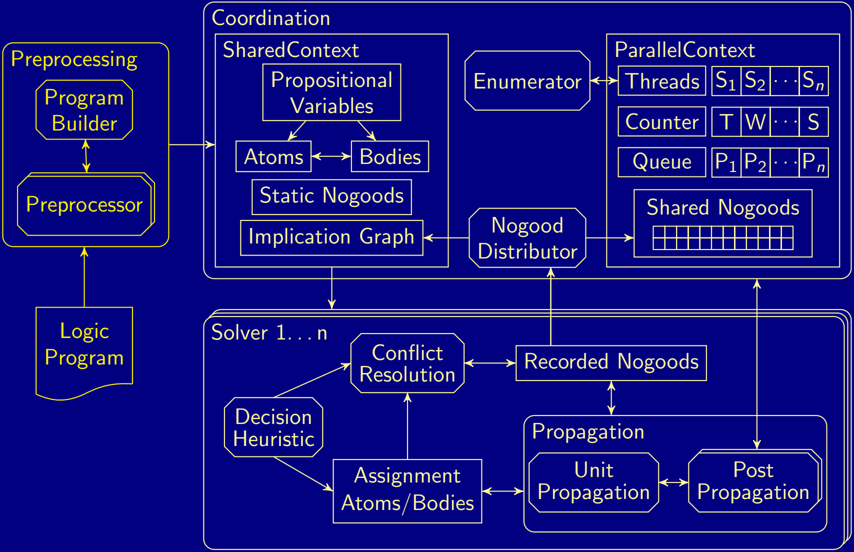
CDCL-style solving

```
loop
    propagate                                  // deterministically assign literals
    if no conflict then
        if all variables assigned then return solution
        else decide                            // non-deterministically assign some literal
    else
        if top-level conflict then return unsatisfiable
        else
            analyze                            // analyze conflict and add conflict constraint
            backjump                           // unassign literals until conflict constraint is unit
```

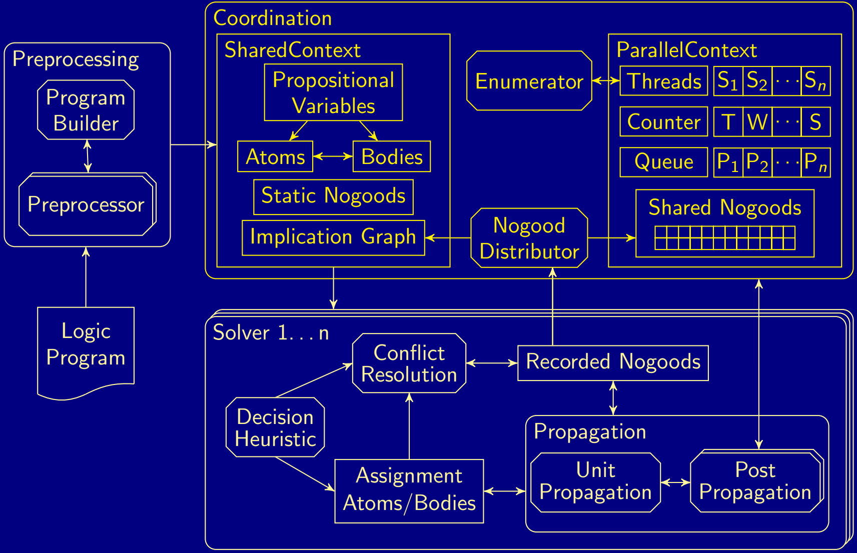
Multi-threaded architecture of *clasp*



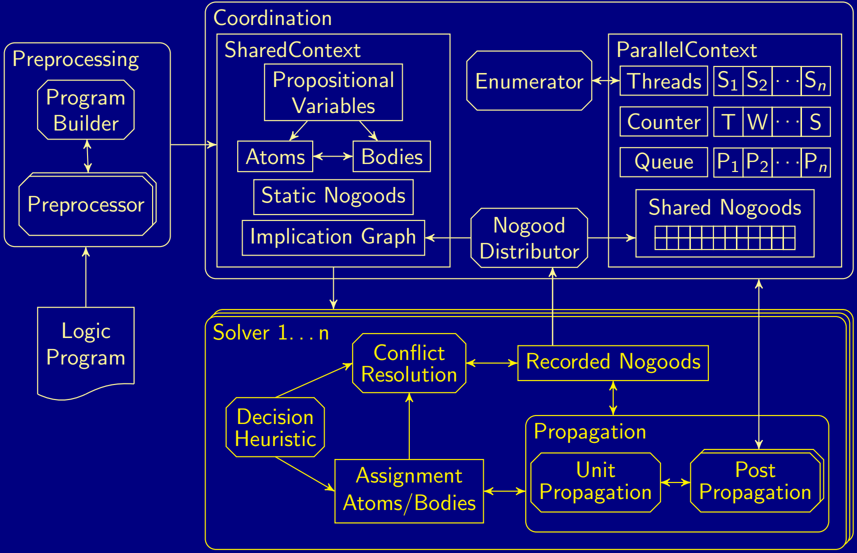
Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Outline

`http://potassco.sourceforge.net`

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: gringo, pyngo
- *Solver*: clasp, {a,u,h}clasp, claspD, claspfolio, claspar, aspeed
- *Grounder+Solver*: Clingo, iClingo, oClingo, Clingcon
- *Further Tools*: as{un}cud, coala, fimo, metasp, plasp, etc.

Benchmarking: `http://asparagus.cs.uni-potsdam.de`

http://potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: gringo, pyngo
- *Solver*: clasp, {a,u,h}clasp, claspD, claspfolio, claspar, aspeed
- *Grounder+Solver*: Clingo, iClingo, oClingo, Clingcon
- *Further Tools*: as{un}cud, coala, fimo, metasp, plasp, etc.
- *Benchmarking*: `http://asparagus.cs.uni-potsdam.de`

`http://potassco.sourceforge.net`

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: gringo, pyngo
- *Solver*: clasp, {a,u,h}clasp, claspD, claspfolio, claspar, aspeed
- *Grounder+Solver*: Clingo, iClingo, oClingo, Clingcon
- *Further Tools*: as{un}cud, coala, fimo, metasp, plasp, etc.
- *Benchmarking*: `http://asparagus.cs.uni-potsdam.de`

Outline

Summary

- ASP is emerging as a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
  - `http://potassco.sourceforge.net`
  - ASP, CASC, MISC, PB, and SAT competitions
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

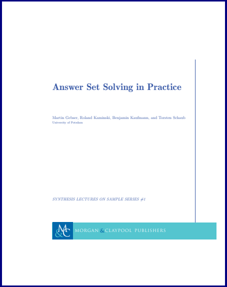
Summary

- ASP is emerging as a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
  - `http://potassco.sourceforge.net`
  - ASP, CASC, MISC, PB, and SAT competitions
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

ASP = DB+LP+KR+SAT

The (forthcoming) Potassco Book

1. Motivation
2. Introduction
3. Basic modeling
4. Grounding
5. Characterizations
6. Solving
7. Systems
8. Advanced modeling
9. Conclusions



Answer Set Solving in Practice

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub
University of Potsdam

SYNTHESIS LECTURES ON SAMPLE SERIES #1

MORGAN & CLAYPOOL PUBLISHERS

`http://potassco.sourceforge.net/teaching.html`