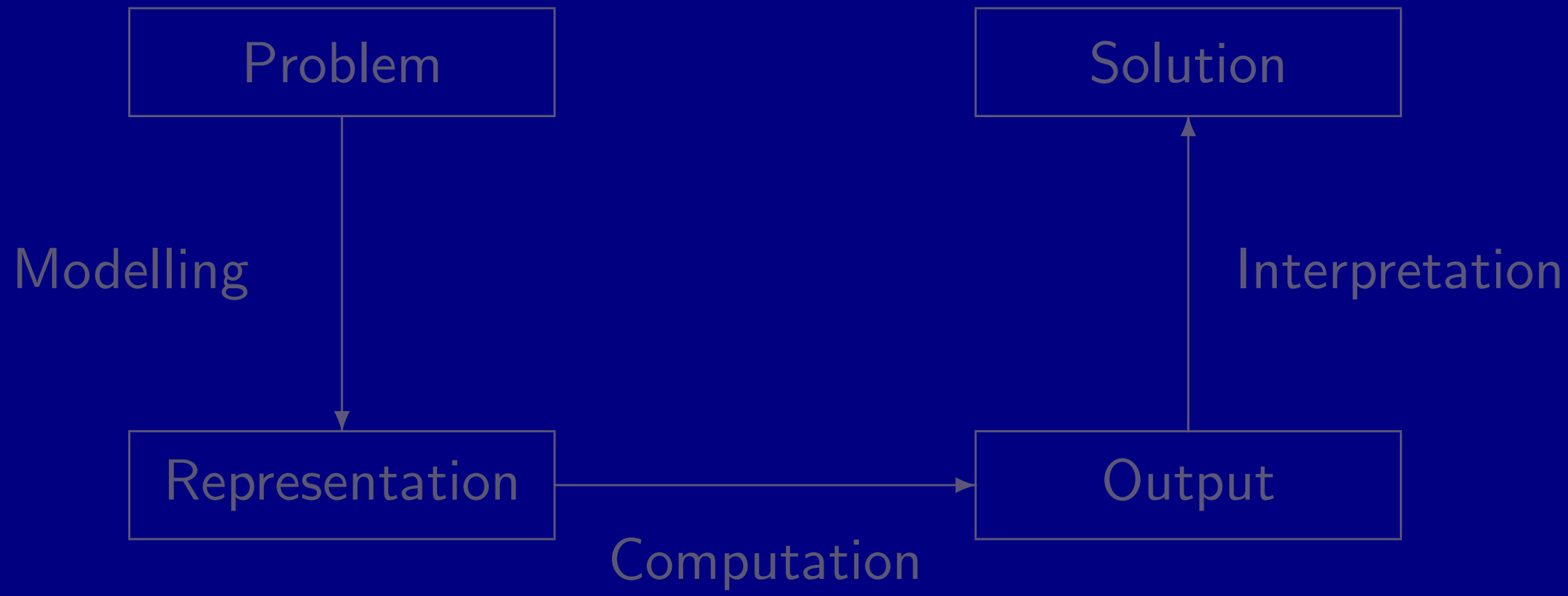
Goal: Declarative problem solving

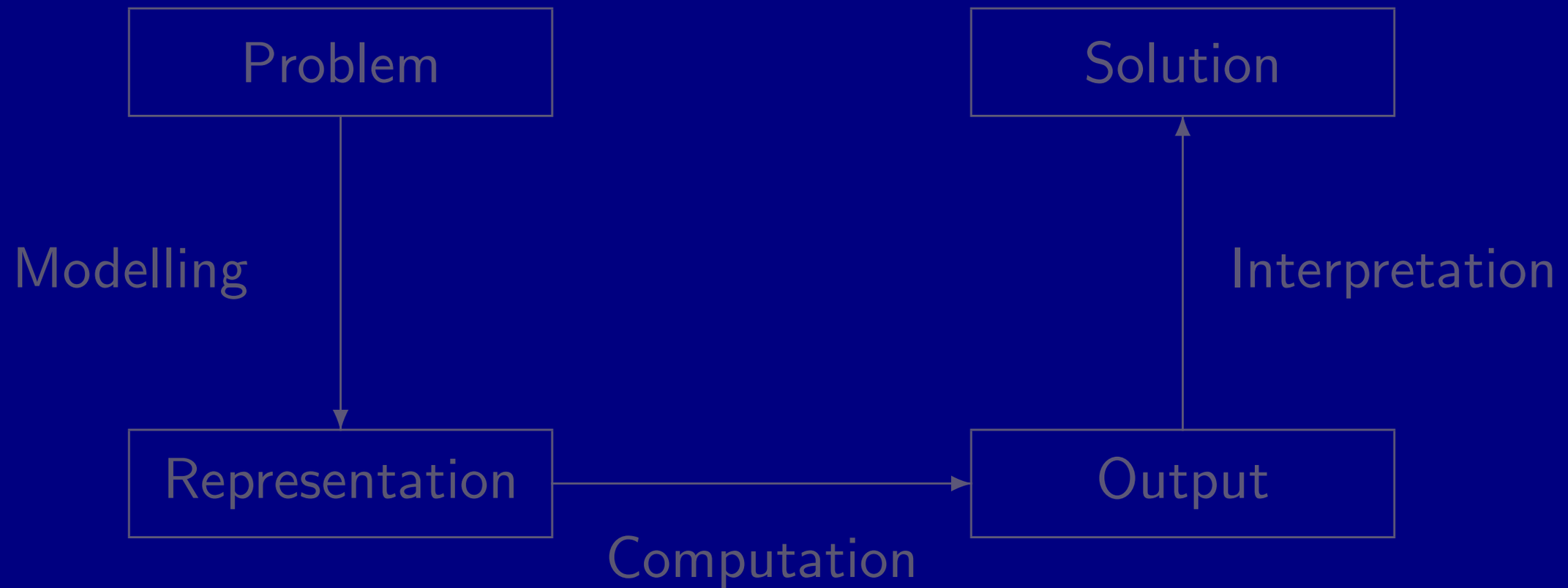
- *"What is the problem?"*
  instead of
- *"How to solve the problem?"*



Goal: Declarative problem solving

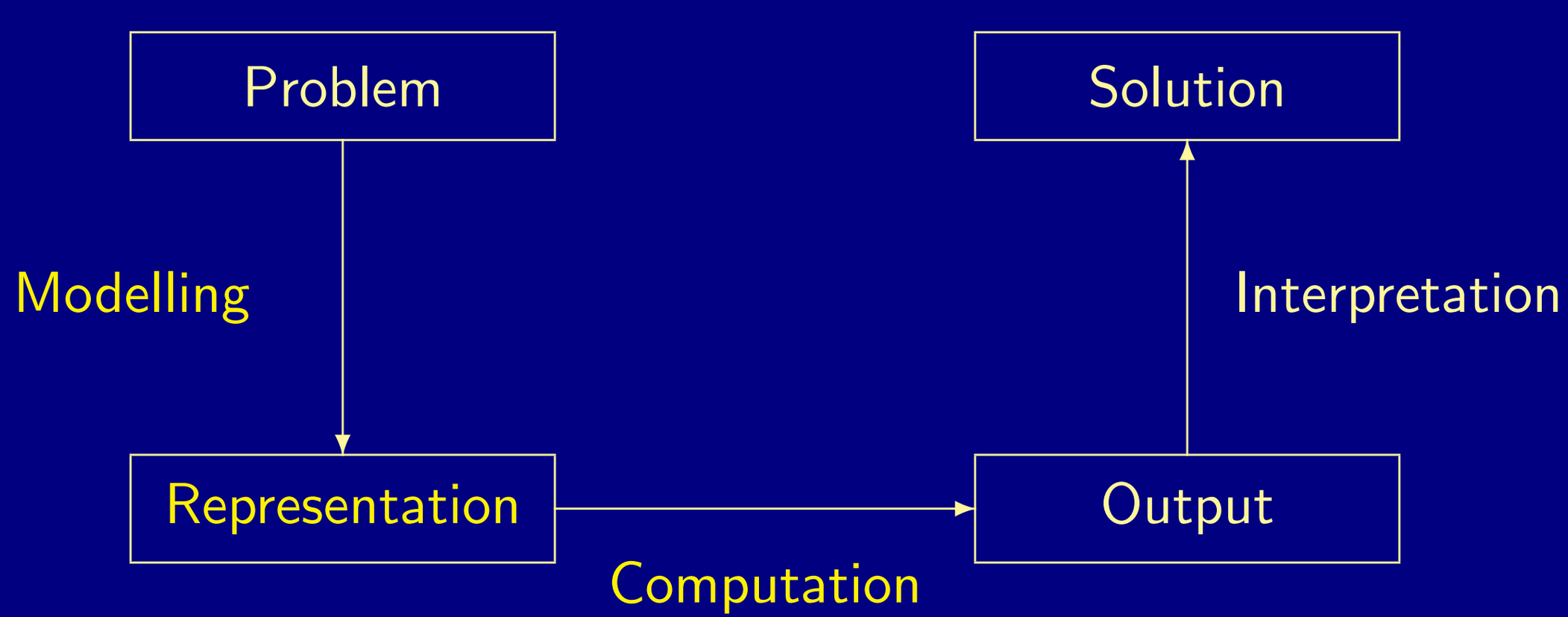
- *"What is the problem?"*

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



Goal: Declarative problem solving

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



Proposal: Answer set programming (ASP)

- has its roots in
  - (logic-based) knowledge representation and reasoning
  - (deductive) databases
  - constraint solving (in particular, SAT solving)
  - logic programming (with negation)
- allows for solving all search problems within NP (and NP^{NP})
  (over finite domains; otherwise Turing-complete)
- allows for using powerful off-the-shelf systems
  (nowadays capable of dealing with millions of variables)

Take Home Lesson

$$\text{ASP} = \text{KR} + \text{DB} + \text{Search}$$

Proposal: Answer set programming (ASP)

- has its roots in
  - (logic-based) knowledge representation and reasoning
  - (deductive) databases
  - constraint solving (in particular, SAT solving)
  - logic programming (with negation)
- allows for solving all search problems within NP (and NP^{NP})
  (over finite domains; otherwise Turing-complete)
- allows for using powerful off-the-shelf systems
  (nowadays capable of dealing with millions of variables)

Take Home Lesson

$$\text{ASP} = \text{KR} + \text{DB} + \text{Search}$$

Proposal: Answer set programming (ASP)

- has its roots in
  - (logic-based) knowledge representation and reasoning
  - (deductive) databases
  - constraint solving (in particular, SAT solving)
  - logic programming (with negation)
- allows for solving all search problems within NP (and NP^{NP})
  (over finite domains; otherwise Turing-complete)
- allows for using powerful off-the-shelf systems
  (nowadays capable of dealing with millions of variables)

Take Home Lesson

$$\text{ASP} = \text{KR} + \text{DB} + \text{Search}$$

Proposal: Answer set programming (ASP)

- has its roots in
  - (logic-based) knowledge representation and reasoning
  - (deductive) databases
  - constraint solving (in particular, SAT solving)
  - logic programming (with negation)
- allows for solving all search problems within NP (and NP^{NP})
  (over finite domains; otherwise Turing-complete)
- allows for using powerful off-the-shelf systems
  (nowadays capable of dealing with millions of variables)

Take Home Lesson

$$\text{ASP} = \text{KR} + \text{DB} + \text{Search}$$

Proposal: Answer set programming (ASP)

- has its roots in
  - (logic-based) knowledge representation and reasoning
  - (deductive) databases
  - constraint solving (in particular, SAT solving)
  - logic programming (with negation)
- allows for solving all search problems within NP (and NP^{NP})
  (over finite domains; otherwise Turing-complete)
- allows for using powerful off-the-shelf systems
  (nowadays capable of dealing with millions of variables)

Take Home Lesson

$$\text{ASP} = \text{KR} + \text{DB} + \text{Search}$$

Root: Logic Programming with negation

- Algorithm = Logic + Control [48]
- Logic as a programming language
  - ➥ Prolog (Colmerauer, Kowalski)
- Features of Prolog
  - Declarative (relational) programming language
  - Based on SLD(NF) Resolution
  - Top-down query evaluation
  - Terms as data structures
  - Parameter passing by unification
  - Solutions are extracted from instantiations of variables occurring in the query

Prolog: Programming in logic

Prolog is great, it's almost declarative!

To see this, consider

```
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z),above(Z,Y).
```

and compare it to

```
above(X,Y) :- above(Z,Y),on(X,Z).
above(X,Y) :- on(X,Y).
```

An interpretation in classical logic amounts to

$$\forall xy(on(x,y) \vee \exists z(on(x,z) \wedge above(z,y)) \rightarrow above(x,y))$$

Prolog: Programming in logic

Prolog is great, it's almost declarative!
To see this, consider

```
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z),above(Z,Y).
```

and compare it to

```
above(X,Y) :- above(Z,Y),on(X,Z).
above(X,Y) :- on(X,Y).
```

An interpretation in classical logic amounts to

$$\forall xy(on(x,y) \vee \exists z(on(x,z) \wedge above(z,y)) \rightarrow above(x,y))$$

Prolog: Programming in logic

Prolog is great, it's almost declarative!
To see this, consider

```
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z),above(Z,Y).
```

and compare it to

```
above(X,Y) :- above(Z,Y),on(X,Z).
above(X,Y) :- on(X,Y).
```

An interpretation in classical logic amounts to

$$\forall xy(on(x,y) \vee \exists z(on(x,z) \wedge above(z,y)) \rightarrow above(x,y))$$

Prolog: Programming in logic

Prolog is great, it's almost declarative!
To see this, consider

```
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z),above(Z,Y).
```

and compare it to

```
above(X,Y) :- above(Z,Y),on(X,Z).
above(X,Y) :- on(X,Y).
```

An interpretation in classical logic amounts to

$$\forall xy(on(x,y) \vee \exists z(on(x,z) \wedge above(z,y)) \rightarrow above(x,y))$$

Prolog (ctd)

Prolog offers negation as failure via operator not.

For instance,

```
info(a).
ask(X) :- not info(X).
```

cannot be captured by

$$info(a) \wedge \forall x(\neg info(x) \rightarrow ask(x))$$

but by appeal to Clark's completion [11] by

$$\forall x(x = a \leftrightarrow info(x)) \wedge \forall x(\neg info(x) \leftrightarrow ask(x))$$
$$\iff \quad info(a) \wedge \forall x(x \neq a \leftrightarrow ask(x))$$

Prolog (ctd)

Prolog offers negation as failure via operator not.

For instance,

```
info(a).
ask(X) :- not info(X).
```

cannot be captured by

$$info(a) \wedge \forall x(\neg info(x) \rightarrow ask(x))$$

but by appeal to Clark's completion [11] by

$$\forall x(x = a \leftrightarrow info(x)) \wedge \forall x(\neg info(x) \leftrightarrow ask(x))$$
$$\iff \quad info(a) \wedge \forall x(x \neq a \leftrightarrow ask(x))$$

Prolog (ctd)

Prolog offers negation as failure via operator not.

For instance,

```
info(a).
ask(X) :- not info(X).
```

cannot be captured by

$$info(a) \wedge \forall x(\neg info(x) \rightarrow ask(x))$$

but by appeal to Clark's completion [11] by

$$\forall x(x = a \leftrightarrow info(x)) \wedge \forall x(\neg info(x) \leftrightarrow ask(x))$$
$$\iff \quad info(a) \wedge \forall x(x \neq a \leftrightarrow ask(x))$$

Prolog (ctd)

Prolog offers negation as failure via operator not.

For instance,

```
info(a).
ask(X) :- not info(X).
```

cannot be captured by

$$info(a) \wedge \forall x(\neg info(x) \rightarrow ask(x))$$

but by appeal to Clark's completion [11] by

$$\forall x(x = a \leftrightarrow info(x)) \wedge \forall x(\neg info(x) \leftrightarrow ask(x))$$
$$\iff \quad info(a) \wedge \forall x(x \neq a \leftrightarrow ask(x))$$

Model-based Problem Solving

Common approach (e.g. Prolog)

1. Provide a specification of the problem.
2. A solution is given by a derivation of an appropriate query.

Model-based approach (e.g. ASP)

1. Provide a specification of the problem.
2. A solution is given by a model of the specification.

Automated planning, Kautz and Selman [46]

Represent planning problems as propositional theories so that models not proofs describe solutions (e.g. `Satplan`)

Model-based Problem Solving

Common approach (e.g. Prolog)

1. Provide a specification of the problem.
2. A solution is given by a derivation of an appropriate query.

Model-based approach (e.g. ASP)

1. Provide a specification of the problem.
2. A solution is given by a model of the specification.

Automated planning, Kautz and Selman [46]

Represent planning problems as propositional theories so that models not proofs describe solutions (e.g. `Satplan`)

Model-based Problem Solving

Common approach (e.g. Prolog)

1. Provide a specification of the problem.
2. A solution is given by a derivation of an appropriate query.

Model-based approach (e.g. ASP)

1. Provide a specification of the problem.
2. A solution is given by a model of the specification.

Automated planning, Kautz and Selman [46]

Represent planning problems as propositional theories so that models not proofs describe solutions (e.g. `Satplan`)

Answer set: Basic idea

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

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

This formula has one stable model, called answer set: $\{p, q\}$

Φ $\quad q \ \wedge \ (q \wedge \neg r \rightarrow p)$

Π_Φ

$$q \ \leftarrow$$
$$p \ \leftarrow \ q, \ not \ r$$

Informally, a set of atoms X is an answer set of a logic program Π

if X is a (classical) model of Π and

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

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

Answer set: Basic idea

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

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

This formula has one stable model, called answer set: $\{p, q\}$

Φ $\quad q \ \wedge \ (q \wedge \neg r \rightarrow p)$

Π_Φ

$$q \ \leftarrow$$
$$p \ \leftarrow \ q, \ not \ r$$

Informally, a set of atoms X is an answer set of a logic program Π

if X is a (classical) model of Π and

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

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

Answer set: Basic idea

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

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

This formula has one stable model, called answer set: $\{p, q\}$

Φ $\quad q \ \wedge \ (q \wedge \neg r \rightarrow p)$

Π_Φ

$$q \ \leftarrow$$
$$p \ \leftarrow \ q, \ not \ r$$

Informally, a set of atoms X is an answer set of a logic program Π

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

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

Answer set: Basic idea

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

$$\Phi \quad q \wedge (q \wedge \neg r \rightarrow p)$$

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

This formula has one stable model, called answer set: $\{p, q\}$

$$\Pi_\Phi \quad \begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q,\ not\ r \end{array}$$

Informally, a set of atoms X is an answer set of a logic program Π

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

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

Answer set: Basic idea

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

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

This formula has one stable model, called answer set: $\{p, q\}$

Φ $\quad q \ \wedge \ (q \wedge \neg r \rightarrow p)$

Π_Φ

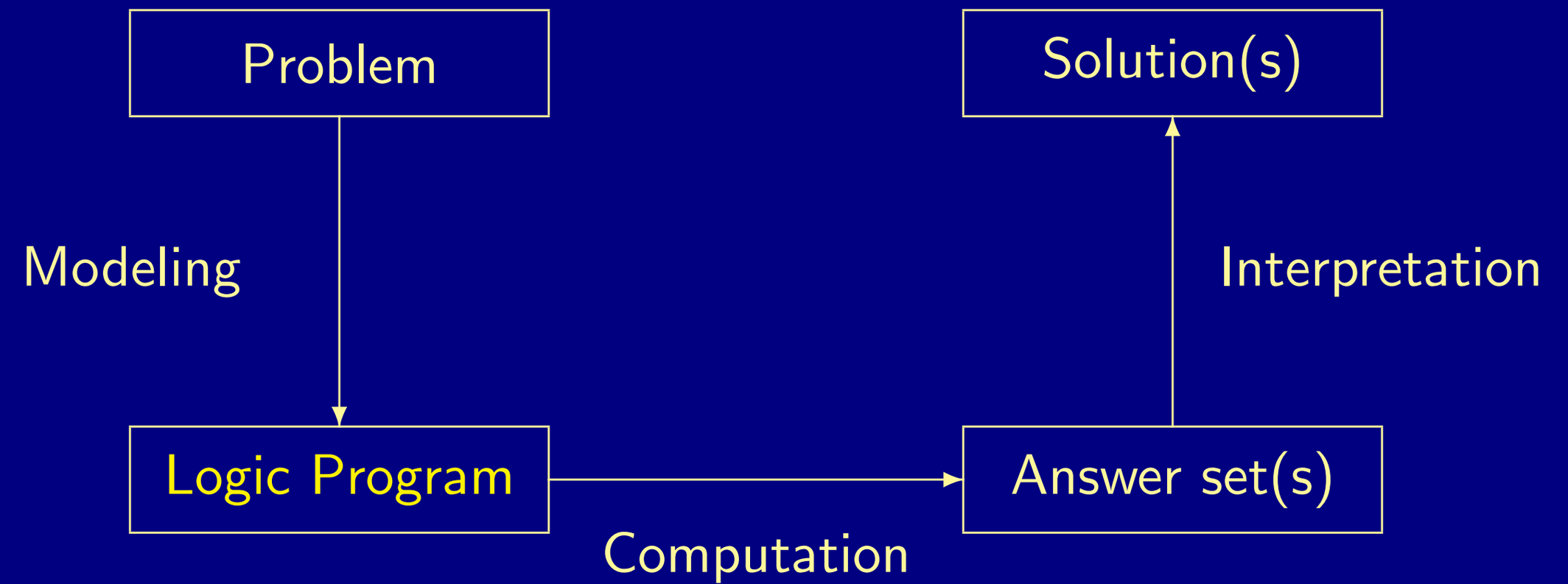
$$q \ \leftarrow$$
$$p \ \leftarrow \ q, \ not \ r$$

Informally, a set of atoms X is an answer set of a logic program Π

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

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

Problem solving in ASP: Syntax



Normal logic programs

- A (normal) rule, r, is an ordered pair of the form

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 0$, and each A_i $(0 \leq i \leq n)$ is an atom.
- A (normal) logic program is a finite set of rules.
- Notation

$$\begin{aligned} \mathit{head}(r) &= A_0 \\ \mathit{body}(r) &= \{A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n\} \\ \mathit{body}^+(r) &= \{A_1, \ldots, A_m\} \\ \mathit{body}^-(r) &= \{A_{m+1}, \ldots, A_n\} \end{aligned}$$

- A program is called positive if $\mathit{body}^-(r) = \emptyset$ for all its rules.

Normal logic programs

- A (normal) rule, r, is an ordered pair of the form

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 0$, and each A_i $(0 \leq i \leq n)$ is an atom.
- A (normal) logic program is a finite set of rules.
- Notation

$$\begin{aligned} \mathit{head}(r) &= A_0 \\ \mathit{body}(r) &= \{A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n\} \\ \mathit{body}^+(r) &= \{A_1, \ldots, A_m\} \\ \mathit{body}^-(r) &= \{A_{m+1}, \ldots, A_n\} \end{aligned}$$

- A program is called positive if $\mathit{body}^-(r) = \emptyset$ for all its rules.

Normal logic programs

- A (normal) rule, r, is an ordered pair of the form

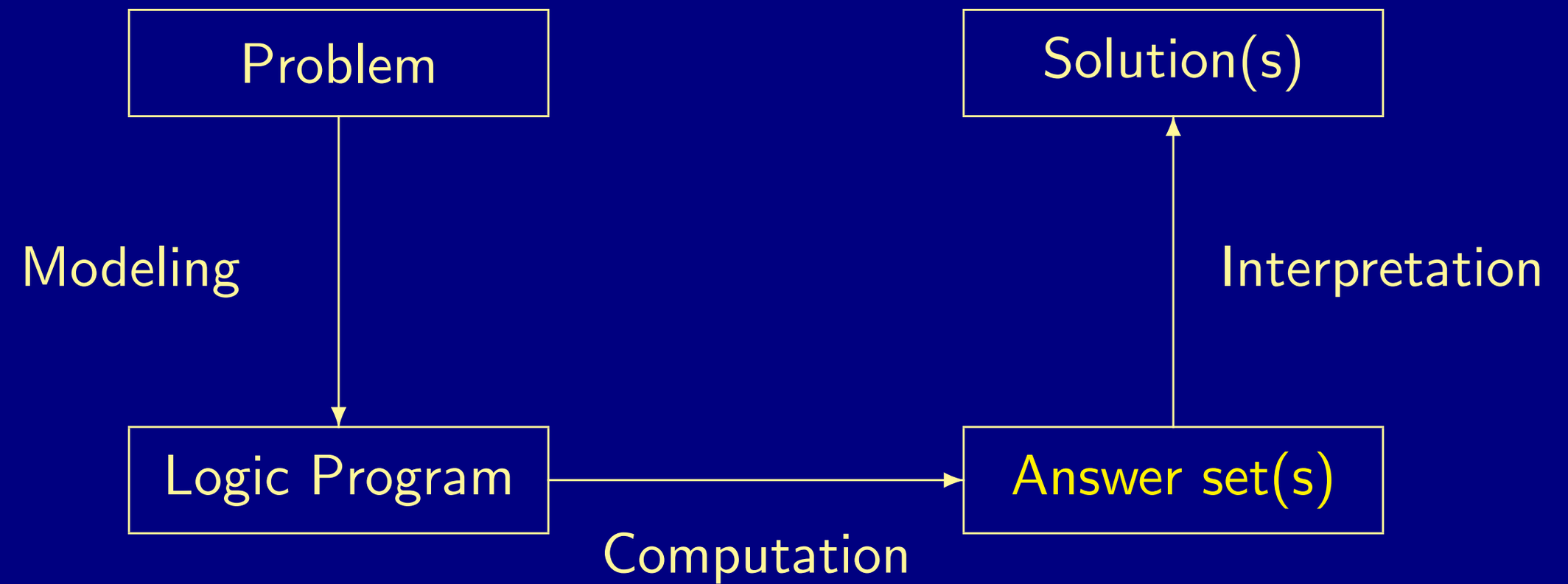
$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 0$, and each A_i $(0 \leq i \leq n)$ is an atom.
- A (normal) logic program is a finite set of rules.
- Notation

$$\begin{aligned} \mathit{head}(r) &= A_0 \\ \mathit{body}(r) &= \{A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n\} \\ \mathit{body}^+(r) &= \{A_1, \ldots, A_m\} \\ \mathit{body}^-(r) &= \{A_{m+1}, \ldots, A_n\} \end{aligned}$$

- A program is called positive if $\mathit{body}^-(r) = \emptyset$ for all its rules.

Problem solving in ASP: Semantics



Answer set: Formal Definition

Positive programs

- A set of atoms X is closed under a positive program Π iff for any $r \in \Pi$, $\mathit{head}(r) \in X$ whenever $\mathit{body}^+(r) \subseteq X$.
  - ➥ X corresponds to a model of Π (seen as a formula).
- The smallest set of atoms which is closed under a positive program Π is denoted by $\mathit{Cn}(\Pi)$.
  - ➥ $\mathit{Cn}(\Pi)$ corresponds to the $\subseteq$-smallest model of Π (ditto).
- The set $\mathit{Cn}(\Pi)$ of atoms is the answer set of a *positive* program Π.

Answer set: Formal Definition

Positive programs

- A set of atoms X is closed under a positive program Π iff for any $r \in \Pi$, $head(r) \in X$ whenever $body^+(r) \subseteq X$.
  - ➥ X corresponds to a model of Π (seen as a formula).
- The smallest set of atoms which is closed under a positive program Π is denoted by $Cn(\Pi)$.
  - ➥ $Cn(\Pi)$ corresponds to the $\subseteq$-smallest model of Π (ditto).
- The set $Cn(\Pi)$ of atoms is the answer set of a *positive* program Π.

Answer set: Formal Definition

Positive programs

- A set of atoms X is closed under a positive program Π iff for any $r \in \Pi$, $head(r) \in X$ whenever $body^+(r) \subseteq X$.
  - ➥ X corresponds to a model of Π (seen as a formula).
- The smallest set of atoms which is closed under a positive program Π is denoted by $Cn(\Pi)$.
  - ➥ $Cn(\Pi)$ corresponds to the $\subseteq$-smallest model of Π (ditto).
- The set $Cn(\Pi)$ of atoms is the answer set of a *positive* program Π.

Answer set: Formal Definition

Positive programs

- A set of atoms X is closed under a positive program Π iff for any $r \in \Pi$, $head(r) \in X$ whenever $body^+(r) \subseteq X$.
  - ➥ X corresponds to a model of Π (seen as a formula).
- The smallest set of atoms which is closed under a positive program Π is denoted by $Cn(\Pi)$.
  - ➥ $Cn(\Pi)$ corresponds to the $\subseteq$-smallest model of Π (ditto).
- The set $Cn(\Pi)$ of atoms is the answer set of a *positive* program Π.

Some "logical" remarks

- Positive rules are also referred to as definite clauses.
  - Definite clauses are disjunctions with exactly one positive atom:

$$A_0 \vee \neg A_1 \vee \cdots \vee \neg A_m$$

  - A set of definite clauses has a (unique) smallest model.
- Horn clauses are clauses with at most one positive atom.
  - Every definite clause is a Horn clause but not vice versa.
  - A set of Horn clauses has a smallest model or none.
- This smallest model is the intended semantics of a set of Horn clauses.
  - ☞ Given a positive program Π, $Cn(\Pi)$ corresponds to the smallest model of the set of definite clauses corresponding to Π.

Some "logical" remarks

- Positive rules are also referred to as definite clauses.
  - Definite clauses are disjunctions with exactly one positive atom:

$$A_0 \vee \neg A_1 \vee \cdots \vee \neg A_m$$

  - A set of definite clauses has a (unique) smallest model.
- Horn clauses are clauses with at most one positive atom.
  - Every definite clause is a Horn clause but not vice versa.
  - A set of Horn clauses has a smallest model or none.
- This smallest model is the intended semantics of a set of Horn clauses.
  - ☞ Given a positive program Π, $Cn(\Pi)$ corresponds to the smallest model of the set of definite clauses corresponding to Π.

Some "logical" remarks

- Positive rules are also referred to as definite clauses.
  - Definite clauses are disjunctions with exactly one positive atom:

$$A_0 \vee \neg A_1 \vee \cdots \vee \neg A_m$$

  - A set of definite clauses has a (unique) smallest model.
- Horn clauses are clauses with at most one positive atom.
  - Every definite clause is a Horn clause but not vice versa.
  - A set of Horn clauses has a smallest model or none.
- This smallest model is the intended semantics of a set of Horn clauses.
  - ☞ Given a positive program Π, $Cn(\Pi)$ corresponds to the smallest model of the set of definite clauses corresponding to Π.

Answer set: Formal Definition

Normal programs [39]

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

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi \text{ and } body^-(r) \cap X = \emptyset\}.$$

- A set X of atoms is an answer set of a program Π if $Cn(\Pi^X) = X$.
  Recall: $Cn(\Pi^X)$ is the $\subseteq$–smallest (classical) model of Π^X.

Intuition: X is stable under *"applying rules from* Π*"*

Note: Every atom in X is justified by an *"applying rule from* Π*"*

Answer set: Formal Definition

Normal programs [39]

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

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi \text{ and } body^-(r) \cap X = \emptyset\}.$$

- A set X of atoms is an answer set of a program Π if $Cn(\Pi^X) = X$.
  Recall: $Cn(\Pi^X)$ is the $\subseteq$–smallest (classical) model of Π^X.

Intuition: X is stable under *"applying rules from* Π*"*

Note: Every atom in X is justified by an *"applying rule from* Π*"*

Answer set: Formal Definition

Normal programs [39]

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

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi \text{ and } body^-(r) \cap X = \emptyset\}.$$

- A set X of atoms is an answer set of a program Π if $Cn(\Pi^X) = X$.
  Recall: $Cn(\Pi^X)$ is the $\subseteq$–smallest (classical) model of Π^X.

Intuition: X is stable under *"applying rules from* Π*"*

Note: Every atom in X is justified by an *"applying rule from* Π*"*

Answer set: Formal Definition

Normal programs [39]

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

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi \text{ and } body^-(r) \cap X = \emptyset\}.$$

- A set X of atoms is an answer set of a program Π if $Cn(\Pi^X) = X$.
  Recall: $Cn(\Pi^X)$ is the $\subseteq$–smallest (classical) model of Π^X.

Intuition: X is stable under *"applying rules from* Π*"*

Note: Every atom in X is justified by an *"applying rule from* Π*"*

Answer set: Formal Definition

Normal programs [39]

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

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi \text{ and } body^-(r) \cap X = \emptyset\}.$$

- A set X of atoms is an answer set of a program Π if $Cn(\Pi^X) = X$.
  Recall: $Cn(\Pi^X)$ is the $\subseteq$–smallest (classical) model of Π^X.

Intuition: X is stable under *"applying rules from* Π*"*

Note: Every atom in X is justified by an *"applying rule from* Π*"*

A first example

$$\Pi = \{p \leftarrow p,\ \ q \leftarrow \mathit{not}\ p\}$$

X	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p\}$	$p \leftarrow p$	$\emptyset$
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p, q\}$	$p \leftarrow p$	$\emptyset$

A first example

$$\Pi = \{p \leftarrow p,\ \ q \leftarrow not\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✗
$\{p\}$	$p \leftarrow p$	$\emptyset$	✗
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✓
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✗

A first example

$$\Pi = \{p \leftarrow p,\ \ q \leftarrow not\ \ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A first example

$$\Pi = \{p \leftarrow p,\ \ q \leftarrow \mathit{not}\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A first example

$$\Pi = \{p \leftarrow p,\ \ q \leftarrow \mathit{not}\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A first example

$$\Pi = \{p \leftarrow p,\ \ q \leftarrow \mathit{not}\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A second example

$$\Pi = \{p \leftarrow not\ q,\ \ q \leftarrow not\ p\}$$

X	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$
$\{p\}$	$p \leftarrow$	$\{p\}$
$\{q\}$	$q \leftarrow$	$\{q\}$
$\{p, q\}$		$\emptyset$

A second example

$$\Pi = \{p \leftarrow \mathit{not}\ q,\ \ q \leftarrow \mathit{not}\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✗
$\{p\}$	$p \leftarrow$	$\{p\}$	✓
$\{q\}$	$q \leftarrow$	$\{q\}$	✓
$\{p, q\}$		$\emptyset$	✗

A second example

$$\Pi = \{p \leftarrow \mathit{not}\ q,\ \ q \leftarrow \mathit{not}\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A second example

$$\Pi = \{p \leftarrow not\ q,\ \ q \leftarrow not\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A second example

$$\Pi = \{p \leftarrow \mathit{not}\ q,\ \ q \leftarrow \mathit{not}\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A second example

$$\Pi = \{p \leftarrow not\ q,\ \ q \leftarrow not\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A third example

$$\Pi = \{p \leftarrow not\ p\}$$

X	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow$	$\{p\}$
$\{p\}$		$\emptyset$

A third example

$$\Pi = \{p \leftarrow not\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$	$\{p\}$	✗
$\{p\}$		$\emptyset$	✗

A third example

$$\Pi = \{p \leftarrow not\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$	$\{p\}$	✘
$\{p\}$		$\emptyset$	✘

A third example

$$\Pi = \{p \leftarrow not\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$	$\{p\}$	✘
$\{p\}$		$\emptyset$	✘

Answer set: Some properties

- A program may have zero, one, or multiple answer sets!
- If X is an answer set of a logic program Π, then X is a model of Π (seen as a formula).
- If X and Y are answer sets of a *normal* program Π, then $X \not\subset Y$.

Answer set: Some properties

- A program may have zero, one, or multiple answer sets!
- If X is an answer set of a logic program Π, then X is a model of Π (seen as a formula).
- If X and Y are answer sets of a *normal* program Π, then $X \not\subset Y$.

Formal Definition

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

$$a \leftarrow b_1, \ldots, b_m, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_n,$$

    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, Π^X, of a program Π relative to a set X of atoms is defined by

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

  A set X of atoms is an answer set of a program Π,
  if $X = Cn(\Pi^X)$,
  where $Cn(\Pi^X)$ is the $\subseteq$–smallest model of Π^X

Formal Definition

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

$$a \leftarrow b_1, \ldots, b_m, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_n,$$

    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, Π^X, of a program Π relative to a set X of atoms is defined by

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

    A set X of atoms is an answer set of a program Π,
    if $X = Cn(\Pi^X)$,
    where $Cn(\Pi^X)$ is the $\subseteq$–smallest model of Π^X

Formal Definition

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

$$a \leftarrow b_1, \ldots, b_m, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_n,$$

    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, Π^X, of a program Π relative to a set X of atoms is defined by

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

  - A set X of atoms is an answer set of a program Π,
    if $X = Cn(\Pi^X)$,
    where $Cn(\Pi^X)$ is the $\subseteq$–smallest model of Π^X

Formal Definition

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

$$a \leftarrow b_1, \ldots, b_m, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_n,$$

    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, Π^X, of a program Π relative to a set X of atoms is defined by

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

  - A set X of atoms is an answer set of a program Π,
    if $X = Cn(\Pi^X)$,
    where $Cn(\Pi^X)$ is the $\subseteq$–smallest model of Π^X

Formal Definition

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

$$a \leftarrow b_1, \ldots, b_m, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_n,$$

    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, Π^X, of a program Π relative to a set X of atoms is defined by

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

  - A set X of atoms is an answer set of a program Π, if $X = Cn(\Pi^X)$, where $Cn(\Pi^X)$ is the $\subseteq$–smallest model of Π^X

Formal Definition

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

$$a \leftarrow b_1, \ldots, b_m, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_n,$$

    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, Π^X, of a program Π relative to a set X of atoms is defined by

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

  - A set X of atoms is an answer set of a program Π,
    if $X = Cn(\Pi^X)$,
    where $Cn(\Pi^X)$ is the $\subseteq$–smallest model of Π^X

Programs with Variables

Let Π be a logic program.

- Herbranduniverse U^Π: Set of constants in Π
- Herbrandbase B^Π: Set of (variable-free) atoms constructible from U^Π

  ☞ We usually denote this as $\mathcal{A}$, and call it alphabet.
- Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in r by elements from U^Π:

$$ground(r) = \{r\theta \mid \theta : var(r) \to U^\Pi\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution.
- Ground Instantiation of Π:

$$ground(\Pi) = \bigcup_{r \in \Pi} ground(r)$$

Programs with Variables

Let Π be a logic program.

- Herbranduniverse U^Π: Set of constants in Π
- Herbrandbase B^Π: Set of (variable-free) atoms constructible from U^Π
  ☞ We usually denote this as $\mathcal{A}$, and call it alphabet.
- Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in r by elements from U^Π:

$$ground(r) = \{r\theta \mid \theta : var(r) \to U^\Pi\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution.
- Ground Instantiation of Π:

$$ground(\Pi) = \bigcup_{r \in \Pi} ground(r)$$

Programs with Variables

Let Π be a logic program.

- Herbranduniverse U^Π: Set of constants in Π
- Herbrandbase B^Π: Set of (variable-free) atoms constructible from U^Π
  ☞ We usually denote this as $\mathcal{A}$, and call it alphabet.
- Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in r by elements from U^Π:

$$ground(r) = \{r\theta \mid \theta : var(r) \to U^\Pi\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution.
- Ground Instantiation of Π:

$$ground(\Pi) = \bigcup_{r \in \Pi} ground(r)$$

Programs with Variables

Let Π be a logic program.

- Herbranduniverse U^Π: Set of constants in Π
- Herbrandbase B^Π: Set of (variable-free) atoms constructible from U^Π
  ☞ We usually denote this as $\mathcal{A}$, and call it alphabet.
- Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in r by elements from U^Π:

  $$ground(r) = \{r\theta \mid \theta : var(r) \to U^\Pi\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution.
- Ground Instantiation of Π:

  $$ground(\Pi) = \bigcup_{r \in \Pi} ground(r)$$

An example

$$\Pi = \{\ r(a,b) \leftarrow,\ \ r(b,c) \leftarrow,\ \ t(X,Y) \leftarrow r(X,Y)\ \}$$

$$U^{\Pi} = \{a, b, c\}$$

$$B^{\Pi} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(\Pi) = \left\{ \begin{array}{llllll} r(a,b) & \leftarrow & , & & & \\ r(b,c) & \leftarrow & , & & & \\ t(a,a) & \leftarrow & r(a,a), & t(b,a) & \leftarrow & r(b,a),\ t(c,a) \leftarrow r(c,a), \\ t(a,b) & \leftarrow & r(a,b), & t(b,b) & \leftarrow & r(b,b),\ t(c,b) \leftarrow r(c,b), \\ t(a,c) & \leftarrow & r(a,c), & t(b,c) & \leftarrow & r(b,c),\ t(c,c) \leftarrow r(c,c) \end{array} \right\}$$

☞ Intelligent Grounding aims at reducing the ground instantiation.

An example

$$\Pi = \{\ r(a,b) \leftarrow,\ \ r(b,c) \leftarrow,\ \ t(X,Y) \leftarrow r(X,Y)\ \}$$

$$U^{\Pi} = \{a, b, c\}$$

$$B^{\Pi} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(\Pi) = \left\{ \begin{array}{llllll} r(a,b) & \leftarrow & , & & & \\ r(b,c) & \leftarrow & , & & & \\ t(a,a) & \leftarrow & r(a,a), & t(b,a) & \leftarrow & r(b,a),\ t(c,a) \leftarrow r(c,a), \\ t(a,b) & \leftarrow & r(a,b), & t(b,b) & \leftarrow & r(b,b),\ t(c,b) \leftarrow r(c,b), \\ t(a,c) & \leftarrow & r(a,c), & t(b,c) & \leftarrow & r(b,c),\ t(c,c) \leftarrow r(c,c) \end{array} \right\}$$

☞ Intelligent Grounding aims at reducing the ground instantiation.

An example

$$\Pi = \{\ r(a,b) \leftarrow,\ \ r(b,c) \leftarrow,\ \ t(X,Y) \leftarrow r(X,Y)\ \}$$

$$U^{\Pi} = \{a, b, c\}$$

$$B^{\Pi} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(\Pi) = \left\{ \begin{array}{l} r(a,b) \leftarrow\ , \\ r(b,c) \leftarrow\ , \\ t(a,a) \leftarrow r(a,a),\ t(b,a) \leftarrow r(b,a),\ t(c,a) \leftarrow r(c,a), \\ t(a,b) \leftarrow r(a,b),\ t(b,b) \leftarrow r(b,b),\ t(c,b) \leftarrow r(c,b), \\ t(a,c) \leftarrow r(a,c),\ t(b,c) \leftarrow r(b,c),\ t(c,c) \leftarrow r(c,c) \end{array} \right\}$$

☞ Intelligent Grounding aims at reducing the ground instantiation.

Answer sets of programs with Variables

Let Π be a normal logic program with variables.

We define a set X of (ground) atoms as an answer set of Π if $Cn(ground(\Pi)^X) = X$.

Programs with Integrity Constraints

- Purpose: Integrity constraints eliminate unwanted solution candidates
- Syntax: An integrity constraints is of the form

$$\leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 1$, and each A_i $(1 \leq i \leq n)$ is a atom.
- Example: $\leftarrow$ monitor(21in), graphics(evil)
- Implementation: For a new symbol x, map

$$\begin{array}{rcl} & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n \\ \mapsto \quad x & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n, \mathit{not}\ x \end{array}$$

- Another example: $\Pi = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p\}$
  versus $\Pi' = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p,\ \leftarrow\ p\}$
  versus $\Pi'' = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p,\ \leftarrow \mathit{not}\ p\}$

Programs with Integrity Constraints

- Purpose: Integrity constraints eliminate unwanted solution candidates
- Syntax: An integrity constraints is of the form

$$\leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 1$, and each A_i $(1 \leq i \leq n)$ is a atom.
- Example: $\leftarrow$ monitor(21in), graphics(evil)
- Implementation: For a new symbol x, map

$$\begin{array}{rcl} & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n \\ \mapsto \quad x & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n, \mathit{not}\ x \end{array}$$

- Another example: $\Pi = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p\}$
  versus $\Pi' = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p,\ \leftarrow\ p\}$
  versus $\Pi'' = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p,\ \leftarrow \mathit{not}\ p\}$

Programs with Integrity Constraints

- Purpose: Integrity constraints eliminate unwanted solution candidates
- Syntax: An integrity constraints is of the form

$$\leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

where $n \geq m \geq 1$, and each A_i $(1 \leq i \leq n)$ is a atom.

- Example: $\leftarrow$ monitor(21in), graphics(evil)
- Implementation: For a new symbol x, map

$$\begin{array}{rrcl} & & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n \\ \mapsto & x & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n, \mathit{not}\ x \end{array}$$

- Another example: $\Pi = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p\}$
  versus $\Pi' = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p,\ \leftarrow\ p\}$
  versus $\Pi'' = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p,\ \leftarrow \mathit{not}\ p\}$

Programs with Integrity Constraints

- Purpose: Integrity constraints eliminate unwanted solution candidates
- Syntax: An integrity constraints is of the form

$$\leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 1$, and each A_i $(1 \leq i \leq n)$ is a atom.
- Example: $\leftarrow$ monitor(21in), graphics(evil)
- Implementation: For a new symbol x, map

$$\begin{array}{rcl} & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n \\ \mapsto \quad x & \leftarrow & A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n, \mathit{not}\ x \end{array}$$

- Another example: $\Pi = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p\}$
  versus $\Pi' = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p,\ \leftarrow\ p\}$
  versus $\Pi'' = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p,\ \leftarrow \mathit{not}\ p\}$

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over $\{$`a, b, c`$\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Quantification
  - `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) : q(X) } 7 :- r(Y)`
- Aggregates
  - `r(Y) :- 2 #count { p(X) : q(X) } 7`
  - also: `#sum, #times, #avg, #min, #max, #even, #odd`

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

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

Graph Coloring: Solving

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

```
clasp version 1.2.1
Reading from stdin
Reading      : Done(0.000s)
Preprocessing: Done(0.000s)
Solving...
Answer: 1
color(1,b) color(2,r) color(3,r) color(4,g) color(5,b) color(6,g) node(1) ... edge(1,2) ... col(r) ...
Answer: 2
color(1,g) color(2,r) color(3,r) color(4,b) color(5,g) color(6,b) node(1) ... edge(1,2) ... col(r) ...
Answer: 3
color(1,b) color(2,g) color(3,g) color(4,r) color(5,b) color(6,r) node(1) ... edge(1,2) ... col(r) ...
Answer: 4
color(1,g) color(2,b) color(3,b) color(4,r) color(5,g) color(6,r) node(1) ... edge(1,2) ... col(r) ...
Answer: 5
color(1,r) color(2,b) color(3,b) color(4,g) color(5,r) color(6,g) node(1) ... edge(1,2) ... col(r) ...
Answer: 6
color(1,r) color(2,g) color(3,g) color(4,b) color(5,r) color(6,b) node(1) ... edge(1,2) ... col(r) ...

Models       : 6
Time         : 0.000  (Solving: 0.000)
```

Graph Coloring: Solving

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

```
clasp version 1.2.1
Reading from stdin
Reading      : Done(0.000s)
Preprocessing: Done(0.000s)
Solving...
Answer: 1
color(1,b) color(2,r) color(3,r) color(4,g) color(5,b) color(6,g) node(1) ... edge(1,2) ... col(r) ...
Answer: 2
color(1,g) color(2,r) color(3,r) color(4,b) color(5,g) color(6,b) node(1) ... edge(1,2) ... col(r) ...
Answer: 3
color(1,b) color(2,g) color(3,g) color(4,r) color(5,b) color(6,r) node(1) ... edge(1,2) ... col(r) ...
Answer: 4
color(1,g) color(2,b) color(3,b) color(4,r) color(5,g) color(6,r) node(1) ... edge(1,2) ... col(r) ...
Answer: 5
color(1,r) color(2,b) color(3,b) color(4,g) color(5,r) color(6,g) node(1) ... edge(1,2) ... col(r) ...
Answer: 6
color(1,r) color(2,g) color(3,g) color(4,b) color(5,r) color(6,b) node(1) ... edge(1,2) ... col(r) ...

Models       : 6
Time         : 0.000  (Solving: 0.000)
```

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 ].
```

What is ASP good for?

- Combinatorial search problems
  (some with substantial amount of data):
  - 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 in large biological networks
  - Home monitoring for risk prevention in assisted living
  - General game playing