Answer Set Programming
Theory and Applications

Tran Cao Son

Department of Computer Science
New Mexico State University
MSC CS, PO Box 30001
Las Cruces, New Mexico 88003

Fall 2018/AI 2
Outline

Logic Programming
  Syntax
  Examples of Propositional Programs
  Programs with FOL Atoms

Answer Set Solver: clingo (How To?)

Reasoning about Dynamic Domains

Logic Programming and Knowledge Representation

Extensions of Logic Programming and Computing Answer Sets

Advanced Problems in ASP
Acknowledgement

This tutorial contains some materials from tutorials on answer set programming and on knowledge representation and logic programming from those provided by

Answer set programming is a new programming paradigm. It was introduced in the late 90’s and manages to attract the intention of different groups of researchers thanks to its:

► **declarativeness**: programs do not specify how answers are computed;
► **modularity**: programs can be developed incrementally;
► **expressiveness**: answer set programming can be used to solve problems in high complexity classes (e.g. $\Sigma^2_P$, $\Pi^2_P$, etc.)

Answer set programming has been applied in several areas: reasoning about actions and changes, planning, configuration, wire routing, phylogenetic inference, semantic web, information integration, etc.
Outline

Logic Programming
  Syntax
  Examples of Propositional Programs
  Programs with FOL Atoms

Answer Set Solver: clingo (How To?)

Reasoning about Dynamic Domains

Logic Programming and Knowledge Representation

Extensions of Logic Programming and Computing Answer Sets

Advanced Problems in ASP
Rules and Constraints

\[ r : b_1 \text{ or } \ldots \text{ or } b_m \leftarrow a_1, \ldots, a_n, \text{not } a_{n+1}, \ldots, \text{not } a_{n+k} \]

- \( a_i, b_j \): atom of a language \( \mathcal{L} \) (\( \mathcal{L} \) can either be propositional or first order)
- \( \text{not } a \): a negation-as-failure atom (naf-atom).
Rules and Constraints

\[ r : \quad b_1 \text{ or } \ldots \text{ or } b_m \leftarrow a_1, \ldots, a_n, \text{not} \ a_{n+1}, \ldots, \text{not} \ a_{n+k} \]

- \( a_i, b_j \): atom of a language \( \mathcal{L} \) (\( \mathcal{L} \) can either be propositional or first order)
- \( \text{not} \ a \): a negation-as-failure atom (naf-atom).

**Reading 1**

If \( a_1, \ldots, a_n \) are true and none of \( a_{n+1}, \ldots, a_{n+k} \) can be proven to be true then at least one of \( b_1, \ldots, b_m \) must be true.
Rules and Constraints

\[ r : b_1 \text{ or } \ldots \text{ or } b_m \leftarrow a_1, \ldots, a_n, \text{not } a_{n+1}, \ldots, \text{not } a_{n+k} \]

- \( a_i, b_j \): atom of a language \( \mathcal{L} \) (\( \mathcal{L} \) can either be propositional or first order)
- \( \text{not } a \): a negation-as-failure atom (naf-atom).

Reading 1

If \( a_1, \ldots, a_n \) are true and none of \( a_{n+1}, \ldots, a_{n+k} \) can be proven to be true then at least one of \( b_1, \ldots, b_m \) must be true.

Reading 2

If \( a_1, \ldots, a_n \) are believed to be true and there is no reason to believe that any of \( a_{n+1}, \ldots, a_{n+k} \) is true then at least one of \( b_1, \ldots, b_m \) must be true.
Notations

\[ r : \underbrace{b_1 \text{ or } \ldots \text{ or } b_m}_{\text{head}(r)} \leftarrow \underbrace{a_1, \ldots, a_n, \text{not } a_{n+1}, \ldots, \text{not } a_{n+k}}_{\text{body}(r)} \]

- head\((r)\) = \(\{b_1, \ldots, b_m\}\)
- pos\((r)\) = \(\{a_1, \ldots, a_n\}\) (also: body\(^+\)(r) = \(\{a_1, \ldots, a_n\}\))
- neg\((r)\) = \(\{a_{n+1}, \ldots, a_{n+k}\}\) (also: body\(^-\)(r) = \(\{a_{n+1}, \ldots, a_{n+k}\}\))

Special cases

- \(n = k = 0\): \(r\) encodes a fact;
- \(k = 0\): \(r\) is a positive rule; and
- \(m = 0\): \(r\) encodes a constraint.
Program

- **Program**: a set of rules.
- **Herbrand universe**: the set of ground terms constructed from function symbols and constants occurring in the program. \((U_\pi)\)
- **Herbrand base**: the set of ground atoms constructed from predicate symbols and ground terms from the Herbrand universe. \((B_\pi)\)
- **Rule with variables**: shorthand for the collection of its ground instances. \((ground(r))\)
- **Program with variables**: collection of ground instances of its rules. \((ground(\pi))\)
$\mathcal{L}$ is a propositional language

$\mathcal{L}$: set of propositions such as $p, q, r, a, b \ldots$

$P_1 = \begin{cases} 
  a \leftarrow \\
  b \leftarrow a, c \\
  c \leftarrow a, p \\
  c \leftarrow 
\end{cases}$

$P_2 = \begin{cases} 
  a \leftarrow \text{not} \ b \\
  b \leftarrow \text{not} \ a, c \\
  p \leftarrow a, \text{not} \ p \\
  c \leftarrow 
\end{cases}$

$P_3 = \begin{cases} 
  a \leftarrow \\
  b \leftarrow c
\end{cases}$
\( \mathcal{L} \) is a first order language

\( \mathcal{L} \) has one function symbol \( f \) (arity: 1) and one predicate symbol \( p \) (arity 1)

\[
Q_1 = \{ \quad p(f(X)) \leftarrow p(X) \}
\]

\[
Q_2 = \{ \quad p(f(f(X)) \leftarrow p(f(X)), \textbf{not} \ p(X) \}
\]

\[
Q_3 = \left\{ \begin{array}{l}
p(f(X)) \leftarrow \\
p(f(f(f(X)))) \leftarrow p(X)
\end{array} \right.
\]
Semantics: Positive Propositional Programs

For a program without \textbf{not} and every rule $m = 1$. So, every rule in $P$ is of the form: $a \leftarrow a_1, \ldots, a_n$

\textbf{Definition}
For a positive program $P$,

\[ T_P(X) = \{ a \mid \exists (a \leftarrow a_1, \ldots, a_n) \in P. [\forall i. (a_i \in X)] \} \]

\textbf{Observations}

\begin{itemize}
  \item every fact in $P$ belongs to $T_P(X)$ for every $X$
  \item If $X \subseteq Y$ then $T_P(X) \subseteq T_P(Y)$
  \item $\emptyset \subseteq T_P(\emptyset) \subseteq T_P(T_P(\emptyset)) \subseteq \ldots \subseteq T_P^n(\emptyset) \subseteq$ and $T_P^n(\emptyset)$ for $n \to \infty$ converges to $\text{lfp}(T_P)$
\end{itemize}
Computing $T_P$: Example 1

$L$: set of propositions such as $p$, $q$, $r$, $a$, $b$ ...

$$P_1 = \begin{cases} 
    a \leftarrow \\
    b \leftarrow a, c \\
    c \leftarrow a, p \\
    c \leftarrow 
\end{cases}$$

$$T_{P_1}(\emptyset) = \{a, c\}$$

$$T_{P_1}^2(\emptyset) = T_{P_1}(T_{P_1}(\emptyset)) = T_{P_1}(\{a, c\}) = \{a, c, b\}$$

$$T_{P_1}^3(\emptyset) = T_{P_1}(T_{P_1}^2(\emptyset)) = T_{P_1}(\{a, c, b\}) = \{a, c, b\} = \text{lfp}(T_{P_1})$$
Computing $T_P$: Example 2

$\mathcal{L}$: set of propositions such as $p, q, r, a, b$ ...

$$P_2 = \{ \begin{array}{l}
a \leftarrow b \\
b \leftarrow a, c \\
p \leftarrow a, p \\
c \leftarrow 
\end{array} \}$$

$$T_{P_2}(\emptyset) = \{c\}$$

$$T_{P_2}^2(\emptyset) = T_{P_2}(T_{P_2}(\emptyset)) = T_{P_2}(\{c\}) = \{c\} = \text{lfp}(T_{P_2})$$
Computing $T_P$: Example 3

$$P_3 = \begin{cases} 
  a \leftarrow \\
  b \leftarrow \\
  c 
\end{cases}$$

$T_{P_3}(\emptyset) = \{a\}$

$T^2_{P_3}(\emptyset) = T_{P_3}(T_{P_3}(\emptyset)) = T_{P_3}(\{a\}) = \{a\} = \text{lfp}(T_{P_3})$
Computing $T_P$: Example 4 and 5

\[
P_4 = \begin{cases} 
  a \leftarrow b \\
  b \leftarrow a 
\end{cases}
\]

\[T_{P_4}(\emptyset) = \emptyset = \text{lfp}(T_{P_4})\]

\[
P_5 = \begin{cases} 
  a \leftarrow \\
  b \leftarrow a, b 
\end{cases}
\]

\[T_{P_5}(\emptyset) = \{a\} \]

\[T^2_{P_5}(\emptyset) = T_{P_5}(T_{P_5}(\emptyset)) = T_{P_5}(\{a\}) = \{a\} = \text{lfp}(T_{P_5})\]
Terminologies – many borrowed from classical logic

- variables: \( X, Y, Z \), etc.
- object constants (or simply constants): \( a, b, c \), etc.
- function symbols: \( f, g, h \), etc.
- predicate symbols: \( p, q \), etc.
- terms: variables, constants, and \( f(t_1, \ldots, t_n) \) such that \( t_i \)'s are terms.
- atoms: \( p(t_1, \ldots, t_n) \) such that \( t_i \)'s are terms.
- literals: atoms or an atom preceded by \( \neg \).
- naf-literals: atoms or an atom preceded by not.
- gen-literals: literals or a literal preceded by not.
- ground terms (atoms, literals): terms (atoms, literals resp.) without variables.
FOL, Herbrand Universe, and Herbrand Base

- $\mathcal{L}$ – a first order language with its usual components (e.g., variables, constants, function symbols, predicate symbols, arity of functions and predicates, etc.)

- $U_{\mathcal{L}}$ – Herbrand Universe of a language $\mathcal{L}$: the set of all ground terms which can be formed with the functions and constants in $\mathcal{L}$.

- $B_{\mathcal{L}}$ – Herbrand Base of a language $\mathcal{L}$: the set of all ground atoms which can be formed with the functions, constants and predicates in $\mathcal{L}$.

Example: Consider a language $\mathcal{L}_1$ with variables $X, Y$; constants $a, b$; function symbol $f$ of arity 1; and predicate symbol $p$ of arity 1.

- $U_{\mathcal{L}_1} = \{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), f(f(f(b))), \ldots\}$.

- $B_{\mathcal{L}_1} = \{p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), p(f(f(b))), p(f(f(f(a)))), p(f(f(f(b)))), \ldots\}$. 
Programs with FOL Atoms

\[ r : \; b_1 \; or \; \ldots \; or \; b_m \leftarrow a_1, \ldots, a_n, \textbf{not} \; a_{n+1}, \ldots, \textbf{not} \; a_{n+k} \]

The language \( \mathcal{L} \) of a program \( \Pi \) is often given \textit{implicitly}.

Rules with Variables

\( \text{ground}(r, \mathcal{L}) \): the set of all rules obtained from \( r \) by all possible substitution of elements of \( U_{\mathcal{L}} \) for the variables in \( r \).

Example

Consider the rule “\( p(f(X)) \leftarrow p(X) \).” and the language \( \mathcal{L}_1 \) (with variables \( X, Y \); constants \( a, b \); function symbol \( f \) of arity 1; and predicate symbol \( p \) of arity 1). Then \( \text{ground}(r, \mathcal{L}_1) \) will consist of the following rules:

\[
\begin{align*}
p(f(a)) & \leftarrow p(a). \\
p(f(b)) & \leftarrow p(b). \\
p(f(f(a))) & \leftarrow p(f(a)). \\
p(f(f(b))) & \leftarrow p(f(b)). \\
& \vdots
\end{align*}
\]
Main Definitions

- \( \text{ground}(r, \mathcal{L}) \): the set of all rules obtained from \( r \) by all possible substitution of elements of \( U_\mathcal{L} \) for the variables in \( r \).
- For a program \( \Pi \):
  - \( \text{ground}(\Pi, \mathcal{L}) = \bigcup_{r \in \Pi} \text{ground}(r, \mathcal{L}) \)
  - \( \mathcal{L}_\Pi \): The language of a program \( \Pi \) is the language consists of the constants, variables, function and predicate symbols (with their corresponding arities) occurring in \( \Pi \). In addition, it contains a constant \( a \) if no constant occurs in \( \Pi \).
  - \( \text{ground}(\Pi) = \bigcup_{r \in \Pi} \text{ground}(r, \mathcal{L}_\Pi) \).
Example 2

\[\Pi:\]
\[p(a). \quad p(b). \quad p(c). \]
\[p(f(X)) \leftarrow p(X).\]

\[\text{ground}(\Pi):\]
\[p(a) \leftarrow .\]
\[p(b) \leftarrow .\]
\[p(c) \leftarrow .\]
\[p(f(a)) \leftarrow p(a).\]
\[p(f(b)) \leftarrow p(b).\]
\[p(f(c)) \leftarrow p(c).\]
\[p(f(f(a))) \leftarrow p(f(a)).\]
\[p(f(f(b))) \leftarrow p(f(b)).\]
\[p(f(f(c))) \leftarrow p(f(c)).\]
\[\vdots\]
\[p(f^{k+1}(x)) \leftarrow p(f^k(x)). \text{ for } x \in \{a, b, c\}\]
Herbrand Interpretation I

Definition
The Herbrand universe (resp. Herbrand base) of Π, denoted by \( U_\Pi \) (resp. \( B_\Pi \)), is the Herbrand universe (resp. Herbrand base) of \( L_\Pi \).

Example For \( \Pi = \{ p(X) \leftarrow q(f(X), g(X)), \ r(Y) \leftarrow \} \) the language of Π consists of two function symbols: \( f \) (arity 1) and \( g \) (arity 2); two predicate symbols: \( p \) (arity 1), \( q \) (arity 2) and \( r \) (arity 1); variables \( X, Y \); and a (added) constant \( a \).
\[
U_\Pi = U_{L_\Pi} = \{ a, f(a), g(a), f(f(a)), g(f(a)), g(f(f(a))), g(g(a)), f(f(f(a))), g(f(f(g(a)))) \ldots \}
\]
\[
B_\Pi = B_{L_\Pi} = \{ p(a), q(a, a), r(a), p(f(a)), q(a, f(a)), r(f(a)), q(f(g(a)), g(f(f(a)))) \ldots \}
\]

Definition (Herbrand Interpretation)
A Herbrand interpretation of a program Π is a set of atoms from its Herbrand base.
Let $\Pi$ be a positive proram and $I$ be a Herbrand interpretation of $\Pi$.

$I$ is called a Herbrand model of $\Pi$ if for every rule “$a_0 \leftarrow a_1, \ldots, a_n$” in $\text{ground}(\Pi)$, $a_1, \ldots, a_n$ are true with respect to $I$ (or $\{a_1, \ldots, a_n\} \subseteq I$) then $a_0$ is also true with respect to $I$ (or $a_0 \in I$).

**Definition**

The least Herbrand model for a program $\Pi$ is called the **minimal model** of $\Pi$ and is denoted by $M_\Pi$.

**Computing $M_\Pi$.** Let $\Pi$ be a program. We define a fixpoint operator $T_\Pi$ that maps a set of atoms (of program $\Pi$) to another set of atoms as follows.

$$T_\Pi(X) = \{a \mid a \in B_\Pi, \text{there exists a rule } a \leftarrow a_1, \ldots, a_n \text{ in } \Pi \text{ s. t. } a_i \in X\}$$ (1)
Note: By $a_0 \leftarrow a_1, \ldots, a_n$ in $\text{ground}(\Pi)$ we mean there exists a rule $b_0 \leftarrow b_1, \ldots, b_n$ in $\Pi$ (that might contain variables) and a ground substitution $\sigma$ such that $a_0 = b_0 \sigma$ and $a_i = b_i \sigma$.

Remark

The operator $T_\Pi$ is often called the van Emden and Kowalski’s iteration operator.
Some Examples

For $\Pi = \{ p(f(X)) \leftarrow p(X), \; q(a) \leftarrow p(X) \}$ we have

$$U_\Pi = \{ a, f(a), f(f(a)), f(f(f(a))), \ldots \} = \{ f^i(a) \mid i = 0, 1, \ldots \}$$

and

$$B_\Pi = \{ q(f^i(a)), p(f^i(a)) \mid i = 0, \ldots \}$$

Computing $T_\Pi(X)$:

- For $X = B_\Pi$, $T_\Pi(X) = \{ q(a) \} \cup \{ p(f(t)) \mid t \in U_\Pi \}$.
- For $X = \emptyset$, $T_\Pi(X) = \emptyset$.
- For $X = \{ p(a) \}$, $T_\Pi(X) = \{ q(a), p(f(a)) \}$.
- We have that $M_\Pi = \emptyset$ (Why?).
Properties of $T_{\Pi}$

- $T_{\Pi}$ is monotonic: $T_{\Pi}(X) \subseteq T_{\Pi}(Y)$ if $X \subseteq Y$.
- $T_{\Pi}$ has a least fixpoint that can be computed as follows.
  1. Let $X_1 = T_{\Pi}(\emptyset)$ and $k = 1$
  2. Compute $X_{k+1} = T_{\Pi}(X_k)$. If $X_{k+1} = X_k$ then stops and return $X_k$.
  3. Otherwise, increase $k$ and repeat the second step.

**Note:** The above algorithm will terminate for positive program $\Pi$ with finite $B_{\Pi}$.
We denote the least fix point of $T_{\Pi}$ with $T_{\Pi}^\infty(\emptyset)$ or $lfp(T_{\Pi})$.

**Theorem**

$M_{\Pi} = lfp(T_{\Pi})$.

**Theorem**

*For every positive program $\Pi$ without constraint, $M_{\Pi}$ is unique.*
More Examples I

For $\Pi_1 = \{ p(X) \leftarrow q(f(X), g(X)). \quad r(Y) \leftarrow \}$

we have that $U_{\Pi_1} =$

$$\{ a, f(a), g(a), f(f(a)), g(f(a)), g(f(a)), g(g(a)), f(f(f(a))), \ldots \}$$

$B_{\Pi_1} = \{ p(a), q(a, a), r(a), p(f(a)), q(a, f(a)), r(f(a)), \ldots \}$

Computing $M_{\Pi_1}$:

$X_0 = T_{\Pi_1}(\emptyset) =$

$$\{ r(a), r(f(a)), r(g(a)), r(f(f(a))), r(g(f(a))), r(f(g(a))), \ldots \}$$

$X_1 = T_{\Pi_1}(X_0) = X_0$

So, $lfp(\Pi_1) = \{ r(a), r(f(a)), r(g(a)), r(f(f(a))), \ldots \}$.

For $\Pi_2 = \{ p(f(X)) \leftarrow p(X). \quad q(a) \leftarrow p(X). \}$

$U_{\Pi_2} = \{ a, f(a), f(f(a)), f(f(f(a))), \ldots \}$

$B_{\Pi_2} = \{ q(a), p(a), p(f(a)), p(f(f(a))), \ldots \}$

Computing $M_{\Pi_2}$:

$X_0 = T_{\Pi_2}(\emptyset) = \emptyset$

$X_1 = T_{\Pi_2}(X_0) = X_0$

So, $lfp(\Pi_2) = \emptyset$. 
More Examples II

- For $\Pi_3 = \{ p(f(X)) \leftarrow p(X). \ q(a) \leftarrow p(X). \ p(b). \}$
  
  $U_{\Pi_3} = \{ a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), \ldots \}$

  $B_{\Pi_3} = \{ q(a), q(b), p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), \ldots \}$

  **Computing $M_{\Pi_3}$:**

  $X_0 = T_{\Pi_3}(\emptyset) = \{ p(b) \}$

  $X_1 = T_{\Pi_3}(X_0) = \{ p(b), q(a), p(f(b)) \}$

  $X_2 = T_{\Pi_3}(X_1) = \{ p(b), q(a), p(f(b)), p(f(f(b))) \}$

  $\ldots$

  So, $\text{lfp}(T_{\Pi_3}) = \{ q(a) \} \cup \{ p(f^i(b)) \mid i = 0, 1, \ldots \}$.

- For $\Pi_4 = \{ p \leftarrow a. \ q \leftarrow b. \ a \leftarrow . \}$, $M_{\Pi_4} = \{ a, p \}$.

- For $\Pi_5 = \{ p \leftarrow p. \}$, $M_{\Pi_5} = \emptyset$

- For $\Pi_6 = \{ p \leftarrow p. \ q \leftarrow . \}$, $M_{\Pi_6} = \{ q \}$.

- For $\Pi_7 = \{ p(b). \ p(c). \ p(f(X)) \leftarrow p(X). \}$,

  $M_{\Pi_7} = \{ p(f^n(b)) \mid n = 0, \ldots, \} \cup \{ p(f^n(c)) \mid n = 0, 1, \ldots, \}$
Entailment

For a program $\Pi$ and an atom $a$, $\Pi$ entails $a$ (with respect to the minimal model semantics), denoted by $\Pi \models a$, iff $a \in M_\Pi$.
We say that $\Pi$ entails $\neg a$ (with respect to the minimal model semantics), denoted by $\Pi \models \neg a$, iff $a \not\in M_\Pi$.

Example

Let

$$\Pi = \left\{ \begin{array}{l} p(f(X)) \leftarrow p(X). \\ q(a) \leftarrow p(X). \\ p(b). \end{array} \right.$$ 

We have that $M_\Pi = \text{lfp}(T_\Pi) = \{q(a)\} \cup \{p(f^i(b)) \mid i = 0, 1, \ldots\}$
where $f^i(b) = f(f(\ldots(f(b))))$ ($f$ repeated $i$ times).
So, we say:
- $\Pi \models q(a)$,
- $\Pi \models \neg q(b)$, and
- $\Pi \models p(f^i(b))$ for $i = 0, 1, \ldots$
Entailment – Another Example I

Consider the parent-child database with facts of the form $p(X, Y)$ ($X$ is a parent of $Y$). We can define the ancestor relationship, $a(X, Y)$ ($X$ is an ancestor of $Y$), using the following rules

$$\Pi_a = \{a(X, Y) \leftarrow p(X, Y), a(X, Y) \leftarrow p(X, Z), a(Z, Y)\}$$

Given the set of facts $I = \{p(a, b), p(b, c), p(c, d)\}$, let $\Pi = \Pi_a \cup I$. We can easily compute

$$M_\Pi = I \cup \{a(X, Y) \mid p(X, Y) \in I\} \cup \{a(a, c), a(a, d), a(b, d)\}$$

So, $\Pi \models a(a, c)$ and $\Pi \models a(a, d)$, i.e., $a$ is an ancestor of $c$ and $d$; on the other hand, $\Pi \models \neg a(d, a)$, i.e., $d$ is not an ancestor of $a$. 

Entailment – Another Example II

Consider a directed graph $G$ described by a set of atoms of the form $\text{edge}(X, Y)$. The following program can be used to determine whether there is a path connecting two nodes of $G$.

$$\Pi_G = \begin{cases} 
\text{reachable}(X, Y) & \leftarrow \text{edge}(X, Y) \\
\text{reachable}(X, Y) & \leftarrow \text{edge}(X, Z), \text{reachable}(Z, Y) \\
\text{edge}(a, b) & \leftarrow \\
\text{edge}(b, c) & \leftarrow \\
\text{edge}(c, a) & \leftarrow \\
\ldots
\end{cases}$$

It can be shown that for every pair of nodes $p$ and $q$ of the graph $G$, $\text{reachable}(p, q)$ belongs to $M_{\Pi_G}$ iff there exists a path from $p$ to $q$ in the graph $G$. 
Remark

*Reasoning using positive programs assumes the closed world assumption (CWA): anything, that cannot be proven to be true, is false.*
Recall that a program is a collection of rules of the form

\[ a \leftarrow a_1, \ldots, a_n, \textbf{not} \ a_{n+1}, \textbf{not} \ a_{n+k}. \]

Let \( \Pi \) be a program and \( X \) be a set of atoms, by \( \Pi^X \) we denote the program obtained from \( \text{ground}(\Pi) \) by

1. Deleting from \( \text{ground}(\Pi) \) any rule
   \[ a \leftarrow a_1, \ldots, a_n, \textbf{not} \ a_{n+1}, \textbf{not} \ a_{n+k} \]
   for that \( \{a_{n+1}, \ldots, a_{n+k}\} \cap X \neq \emptyset \), i.e., the body of the rule contains a naf-atom \( \textbf{not} \ a_l \) and \( a_l \) belongs to \( X \); and

2. Removing all of the naf-atoms from the remaining rules.
Remark
The above transformation is often referred to as the Gelfond-Lifschitz transformation.

Remark
$\Pi^X$ is a positive program.

Definition
A set of atoms $X$ is called an answer set of a program $\Pi$ if $X$ is the minimal model of the program $\Pi^X$.

Theorem
For every positive program $\Pi$, the minimal model of $\Pi$, $M_\Pi$, is also the unique answer set of $\Pi$. 
Consider $\Pi_2 = \{a \leftarrow \text{not } b. \quad b \leftarrow \text{not } a.\}$. We will show that its has two answer sets $\{a\}$ and $\{b\}$

<table>
<thead>
<tr>
<th>$S_1 = \emptyset$</th>
<th>$S_2 = {a}$</th>
<th>$S_3 = {b}$</th>
<th>$S_4 = {a, b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi_2^{S_1}$ :</td>
<td>$\Pi_2^{S_2}$ :</td>
<td>$\Pi_2^{S_3}$ :</td>
<td>$\Pi_2^{S_4}$ :</td>
</tr>
<tr>
<td>$a \leftarrow$</td>
<td>$a \leftarrow$</td>
<td>$b \leftarrow$</td>
<td></td>
</tr>
<tr>
<td>$b \leftarrow$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$M_{\Pi_2^{S_1}} = {a, b}$</td>
<td>$M_{\Pi_2^{S_2}} = {a}$</td>
<td>$M_{\Pi_2^{S_3}} = {b}$</td>
<td>$M_{\Pi_2^{S_4}} = \emptyset$</td>
</tr>
<tr>
<td>$M_{\Pi_2^{S_1}} \neq S_1$</td>
<td>$M_{\Pi_2^{S_2}} = S_2$</td>
<td>$M_{\Pi_2^{S_3}} = S_3$</td>
<td>$M_{\Pi_2^{S_4}} \neq S_4$</td>
</tr>
<tr>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>

Assume that our language contains two object constants $a$ and $b$ and consider $\Pi = \{p(X) \leftarrow \text{not } q(X). \quad q(a) \leftarrow \}$. We show that $S = \{q(a), p(b)\}$ is an answer set of $\Pi$. We have that $\Pi^S = \{p(b) \leftarrow \quad q(a) \leftarrow \}$ whose minimal model is exactly $S$. So, $S$ is an answer set of $\Pi$. 
Detailed Computation II

\[ \Pi_4 = \{ p \leftarrow \text{not } p. \} \] We will show that \( \Pi \) does not have an answer set.

\( S_1 = \emptyset \), then \( \Pi_{S_1}^4 = \{ p \leftarrow \} \) whose minimal model is \( \{ p \} \).

\( \{ p \} \neq \emptyset \) implies that \( S_1 \) is not an answer set of \( \Pi_4 \).

\( S_2 = \{ p \} \), then \( \Pi_{S_2}^4 = \emptyset \) whose minimal model is \( \emptyset \). \( \{ p \} \neq \emptyset \) implies that \( S_2 \) is not an answer set of \( \Pi_4 \).

This shows that \( \Pi \) does not have an answer set.
In computing answer sets, the following theorem is useful:

**Theorem**

*Let* $\Pi$ *be a program.*

1. *Let* $r$ *be a rule in* $\text{ground}(P)$ *whose body contains an atom* $a$ *that does not occur in the head of any rule in* $\text{ground}(P)$. *Then,* $S$ *is an answer set of* $\Pi$ *iff* $S$ *is an answer set of* $\text{ground}(P) \setminus \{r\}$. 

2. *Let* $r$ *be a rule in* $\text{ground}(P)$ *whose body contains a naf-atom* not $a$ *that does not occur in the head of any rule in* $\text{ground}(P)$. *Let* $r'$ *be the rule obtained from* $r$ *by removing* not $a$. *Then,* $S$ *is an answer set of* $P$ *iff* $S$ *is an answer set of* $\text{ground}(P) \setminus \{r\} \cup \{r'\}$. 
More Examples

Remark

A program may have zero, one, or more than one answer sets.

- $\Pi_1 = \{ a \leftarrow \text{not } b. \}$.  
  $\Pi_1$ has a unique answer set $\{ a \}$.

- $\Pi_2 = \{ a \leftarrow \text{not } b. \quad b \leftarrow \text{not } a. \}$.  
  The program has two answer sets: $\{ a \}$ and $\{ b \}$.

- $\Pi_3 = \{ p \leftarrow a. \quad a \leftarrow \text{not } b. \quad b \leftarrow \text{not } a. \}$  
  The program has two answer sets: $\{ a, p \}$ and $\{ b \}$.

- $\Pi_4 = \{ a \leftarrow \text{not } b. \quad b \leftarrow \text{not } c. \quad d \leftarrow . \}$  
  Answer sets: $\{ d, b \}$.

- $\Pi_5 = \{ p \leftarrow \text{not } p. \}$  
  No answer set.

- $\Pi_6 = \{ p \leftarrow \text{not } p, d. \quad r \leftarrow \text{not } d. \quad d \leftarrow \text{not } r. \}$  
  Answer set $\{ r \}$.
Entailment w.r.t. Answer Set Semantics I

- For a program $\Pi$ and an atom $a$, $\Pi$ entails $a$, denoted by $\Pi \models a$, if $a \in S$ for every answer set $S$ of $\Pi$.
- For a program $\Pi$ and an atom $a$, $\Pi$ entails $\neg a$, denoted by $\Pi \models \neg a$, if $a \not\in S$ for every answer set $S$ of $\Pi$.
- If neither $\Pi \models a$ nor $\Pi \models \neg a$, then we say that $a$ is unknown with respect to $\Pi$.

Remark

$\Pi$ does not entail $a$ DOES NOT IMPLY that $\Pi$ entails $\neg a$, i.e., reasoning using answer set semantics does not employ the closed world assumption.

- $\Pi_1 = \{a \leftarrow \text{not } b\}$.
  $\Pi_1$ has a unique answer set $\{a\}$. $\Pi_1 \models a$, $\Pi_1 \models \neg b$.
- $\Pi_2 = \{a \leftarrow \text{not } b. \quad b \leftarrow \text{not } a\}$.
  The program has two answer sets: $\{a\}$ and $\{b\}$. Both $a$ and $b$ are unknown w.r.t. $\Pi_2$. 
Entailment w.r.t. Answer Set Semantics II

- $\Pi_3 = \{p \leftarrow a. \ a \leftarrow \textbf{not} \ b. \ b \leftarrow \textbf{not} \ a.\}$
  The program has two answer sets: $\{a, p\}$ and $\{b\}$. Everything is unknown.

- $\Pi_4 = \{a \leftarrow \textbf{not} \ b. \ b \leftarrow \textbf{not} \ c. \ d \leftarrow \} \}$
  Answer sets: $\{d, b\}$. $\Pi_4 \models b; \Pi_4 \models \neg a$; etc.

- $\Pi_5 = \{p \leftarrow \textbf{not} \ p.\}$
  No answer set. $p$ is unknown.

- $\Pi_6 = \{p \leftarrow \textbf{not} \ p, d. \ r \leftarrow \textbf{not} \ d. \ d \leftarrow \textbf{not} \ r.\}$
  Answer set $\{r\}$. $\Pi_6 \models r; \Pi_6 \models \neg p$; etc.

- $\Pi_7 = \{p \leftarrow \textbf{not} \ a. \ p \leftarrow \textbf{not} \ b. \ a \leftarrow \textbf{not} \ b. \ b \leftarrow \textbf{not} \ a.\}$
  Two answer sets: $\{p, a\}$ and $\{p, b\}$. So, $\Pi_7 \models p$ but $\Pi_7 \not\models a$ and $\Pi_7 \not\models \neg a$? (likewise $b$).

- $\Pi_8 = \{q \leftarrow \textbf{not} \ r. \ r \leftarrow \textbf{not} \ q. \ p \leftarrow \textbf{not} \ p. \ p \leftarrow \textbf{not} \ r.\}$
  One stable model: $\{p, q\}$. So, $\Pi_7 \models p$ and $\Pi_7 \models q$. 
Brave Reasoning vs. Skeptical Reasoning The entailment defined earlier required that $\Pi \models a$ iff $a$ belongs to every answer set of $\Pi$. This is termed as skeptical reasoning w.r.t. answer set semantics.

Brave reasoning is an alternative that can be useful in different situations. Brave reasoning relaxes the notion of entailment as follows: for a program $\Pi$ and an atom $a$, $\Pi$ bravely entails $a$ (resp. $\neg a$), denoted by $\Pi \models^b a$ (resp. $\Pi \models^b \neg a$), if $a \in S$ (resp. $a \notin S$) for some answer set $S$ of $\Pi$.

Brave reasoning is the semantics underlying answer set programming.
Consider $\Pi_3 = \{ p \leftarrow a. \ a \leftarrow \text{not } b. \ b \leftarrow \text{not } a. \}$
The Herbrand base of this program is $\{p, a, b\}$. So, this program has eight possible answer sets. Each answer set is a subset of $\{p, a, b\}$. Let us consider each of them.

1. $X_0 = \emptyset$. Computing $\Pi_3^{X_0}$, we have that

\[
\Pi_3^{X_0} = \left\{ \begin{array}{l}
p \leftarrow a \\
a \leftarrow \\
b \leftarrow 
\end{array} \right. 
\]

The minimal model of this program is: $M_{\Pi_3^{X_0}} = \{p, a, b\}$. $M_{\Pi_3^{X_0}} = \{p, a, b\} \neq X_0$. This implies that $X_0$ is not an answer set.
Computation of Answer Sets II

2. $X_1 = \{p\}$. Computing $\Pi_{X_1}^3$, we have that

$$\Pi_{X_1}^3 = \begin{cases} 
  p \leftarrow a \\
  a \leftarrow \\
  b \leftarrow 
\end{cases}$$

The minimal model of this program is: $M_{\Pi_{X_1}^3} = \{p, a, b\}$.
$M_{\Pi_{X_1}^3} = \{p, a, b\} \neq X_1$. This implies that $X_1$ is not an answer set.

3. $X_2 = \{a\}$. Computing $\Pi_{X_2}^3$, we have that

$$\Pi_{X_2}^3 = \begin{cases} 
  p \leftarrow a \\
  a \leftarrow
\end{cases}$$

The minimal model of this program is: $M_{\Pi_{X_2}^3} = \{p, a\}$.
$M_{\Pi_{X_2}^3} = \{p, a\} \neq X_2$. This implies that $X_2$ is not an answer set.
4. $X_3 = \{b\}$. Computing $\Pi_3^{X_3}$, we have that

\[
\Pi_3^{X_3} = \begin{cases} 
    p \leftarrow a \\
    b \leftarrow 
\end{cases}
\]

The minimal model of this program is: $M_{\Pi_3^{X_3}} = \{b\}$. $M_{\Pi_3^{X_3}} = \{b\} = X_3$. This implies that $X_3$ is an answer set.

5. $X_4 = \{p, a\}$. Computing $\Pi_3^{X_4}$, we have that

\[
\Pi_3^{X_4} = \begin{cases} 
    p \leftarrow a \\
    a \leftarrow 
\end{cases}
\]

The minimal model of this program is: $M_{\Pi_3^{X_4}} = \{p, a\}$. $M_{\Pi_3^{X_4}} = \{p, a\} = X_4$. This implies that $X_4$ is an answer set.
6. $X_5 = \{p, b\}$. Computing $\Pi^X_3$, we have that

$$\Pi^X_3 = \begin{cases} p \leftarrow a \\ b \leftarrow \end{cases}$$

The minimal model of this program is: $M_{\Pi^X_3} = \{b\}$. $M_{\Pi^X_3} = \{b\} \neq X_5$. This implies that $X_5$ is not an answer set.

7. $X_6 = \{a, b\}$. Computing $\Pi^X_3$, we have that

$$\Pi^X_3 = \{ p \leftarrow a \}$$

The minimal model of this program is: $M_{\Pi^X_3} = \emptyset$. $M_{\Pi^X_3} = \emptyset \neq X_6$. This implies that $X_6$ is not an answer set.
8. \( X_7 = \{p, a, b\} \). Computing \( \Pi_{X_7}^3 \), we have that

\[
\Pi_{X_7}^3 = \{ p \leftarrow a \}
\]

The minimal model of this program is: \( M_{\Pi_{X_7}^3} = \emptyset \).
\( M_{\Pi_{X_7}^3} = \emptyset \neq X_7 \). This implies that \( X_7 \) is not an answer set.

So, the program has two answer sets: \( \{b\} \) and \( \{p, a\} \).

Notice that in the computation, I did not detail how the minimal model of a positive program is computed. One should notice the difference between minimal model of the program \( \Pi_{X_5}^3 \) and \( \Pi_{X_5}^3 \) and know the reason why it is so.
Further intuitions behind the semantics I

- A set of atoms $S$ is **closed under** a program $\Pi$ if for all rules of the form
  
  $a_0 \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n$. 

  in $\Pi$, $\{a_1, \ldots, a_m\} \subseteq S$ and $\{a_{m+1}, \ldots, a_n\} \cap S = \emptyset$ implies that $a_0 \in S$.

- A set of atoms $S$ is said to be **supported by** $\Pi$ if for all $p \in S$ there is a rule of the form
  
  $p \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n$. 

  in $\Pi$, such that $\{a_1, \ldots, a_m\} \subseteq S$ and $\{a_{m+1}, \ldots, a_n\} \cap S = \emptyset$.

- A set of atoms $S$ is an answer set of a program $\Pi$ **iff** (i) $S$ is closed under $\Pi$ and (ii) there exists a level mapping function $\lambda$ (that maps atoms in $S$ to a number) such that for each $p \in S$ there is a rule in $\Pi$ of the form
  
  $p \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n$. such that

  $\{a_1, \ldots, a_m\} \subseteq S$, $\{a_{m+1}, \ldots, a_n\} \cap S = \emptyset$ and $\lambda(p) > \lambda(a_i)$, for $1 \leq i \leq m$. 
Further intuitions behind the semantics II

Note that (ii) above implies that $S$ is supported by $\Pi$.

It is known that if $S$ is an answer set of $\Pi$ then

1. $S$ must be closed under $\Pi$ and
2. $S$ must be supported by $\Pi$.

The above notions are useful for the computation of answer sets. They allow us to eliminate possible answer sets quickly. Let us look at our computation of the answer sets of $\Pi_3$.

1. Let us consider $X_0 = \emptyset$. Consider the rule $a \leftarrow \text{not } b$ in $\Pi_3$. We have that the set of positive atoms in the body of this rule is empty (i.e., the set $\{a_1, \ldots, a_m\}$ in the definition of closedness) and the set of negative atoms in the body of this rule is $\{b\}$ and $\{b\} \cap X_0 = \emptyset$. This means that $X_0$ violates the closedness condition, i.e., $X_0$ is not closed under the rule $a \leftarrow \text{not } b$ of $\Pi_3$ and hence it is not closed under $\Pi_3$. As such, $X_0$ cannot be an answer set of $\Pi_3$. 
2. Let us consider $X_7 = \{p, a, b\}$. Consider the atom $a \in X_7$. The only rule in $\Pi_3$ whose head is $a$ is the rule $a \leftarrow \text{not } b$. We have that the set of positive atoms in the body of this rule is empty (i.e., the set $\{a_1, \ldots, a_m\}$ in the definition of supportedness) and the set of negative atoms in the body of this rule is $\{b\}$ and $\{b\} \cap X_7 \neq \emptyset$. This means that $a$ has no rule to support it in $\Pi_3$, i.e., $X_7$ is not supported by $\Pi_3$. As such, $X_7$ cannot be an answer set of $\Pi_3$. 
Answer Sets of Programs with Constraints I

For a set of ground atoms \( S \) and a constraint \( c \) of the form

\[
\leftarrow a_0, \ldots, a_n, \text{not } a_{n+1}, \ldots, \text{not } a_{n+k}
\]

we say that \( c \) is satisfied by \( S \) if \( \{a_0, \ldots, a_n\} \setminus S \neq \emptyset \) or \( \{a_{n+1}, \ldots, a_{n+k}\} \cap S \neq \emptyset \).

Let \( \Pi \) be a program with constraints. Let

\[
\Pi_O = \{ r \mid r \in \Pi, \ r \text{ has non-empty head} \}
\]

(\( \Pi_O \) is the set of normal logic program rules in \( \Pi \)) and

\[
\Pi_C = \Pi \setminus \Pi_O
\]

(\( \Pi_C \) is the set of constraints in \( \Pi \)).

**Definition**

A set of atoms \( S \) is an answer sets of a program \( \Pi \) if it is an answer set of \( \Pi_O \) and satisfies all the constraints in \( \text{ground}(\Pi_C) \).
Answer Sets of Programs with Constraints II

\[ \Pi_2 = \{ a \leftarrow \text{not } b. \; b \leftarrow \text{not } a. \}: \text{two answer sets } \{ a \} \text{ and } \{ b \}. \]

- \( R_1 = \{ a \leftarrow \text{not } b. \; b \leftarrow \text{not } a. \; \leftarrow \text{not } a. \} \) has only one answer set \( \{ a \} \).

- \( R_2 = \{ a \leftarrow \text{not } b. \; b \leftarrow \text{not } a. \; \leftarrow \text{not } a, \text{not } b. \} \) has again two answer sets \( \{ a \} \) and \( \{ b \} \).

- \( R_3 = \{ a \leftarrow \text{not } b. \; b \leftarrow \text{not } a. \; p \leftarrow a. \; \leftarrow \text{not } p. \} \) has only one answer set \( \{ b \} \).
Disjunctive Logic Programs I

We define the general logic program rule as

\[ b_1 \text{ or } \ldots \text{ or } b_m \leftarrow a_1, \ldots, a_n, \text{not } a_{n+1}, \ldots, \text{not } a_{n+k} \]

This rule is often called a disjunctive logic program rule (or simply disjunction rule). A rule of this form says that whenever \( a_1, \ldots, a_n \) are true and \( a_{n+1}, \ldots, a_{n+k} \) are false then at least one of \( b_1, \ldots, b_m \) must be true. There are several situations where disjunctive logic programs are necessary since disjunctive logic program (DLP) is strictly more expressive than normal logic program.

An example is the map coloring problem:

\[ \text{colored}(N, \text{red}) \text{ or colored}(N, \text{blue}) \text{ or colored}(N, \text{yellow}) \leftarrow \text{node}(N). \]
Disjunctive Logic Programs II

Semantics of DLP is also defined by answer sets. The definition also makes use of the Gelfond-Lifschitz reduct: given a DLP program $\pi$ and a set of atoms $X$, $\pi^X$ is the result of the Gelfond-Lifschitz reduct of $\pi$ over $X$.

The program $\pi^X$ does not contain negation-as-failure atoms.

A set of atoms $X$ is an answer set of $\pi$ if and only if $X$ is a minimal set of atoms satisfying $\pi^X$.

Examples

1. $\pi_1 = \{ a \lor b \leftarrow c. \ a \leftarrow c. \ b \leftarrow c. \ c. \}$
   $\pi_1$ is a positive program. It has only one answer set: $\{c, a, b\}$.

2. $\pi_2 = \{ a \lor b \leftarrow c. \ c. \}$
   $\pi_2$ is also a positive program. It has two answer sets: $\{c, a\}$ and $\{c, b\}$.
It is useful for representing incomplete knowledge. For example, consider the story

**John’s Injury**

John ran and broke one of his arms. We do not know whether his left or right arm was broken but we are sure that only one was broken. This story can be represented by the disjunctive program consisting of the fact:

\[ \text{broken(} \text{left} \text{)} \text{ or } \text{broken(} \text{right} \text{)} \leftarrow \]
Outline

Logic Programming
  Syntax
  Examples of Propositional Programs
  Programs with FOL Atoms

Answer Set Solver: clingo (How To?)

Reasoning about Dynamic Domains

Logic Programming and Knowledge Representation

Extensions of Logic Programming and Computing Answer Sets

Advanced Problems in ASP
Running clingo

- Download from https://potassco.org, set up, etc.
- Run clingo <params>
- Set of examples
- Let solve some problems
- Some syntactical notes:
  - “←” is replaced by “:-”
  - “or” is replaced by “|”
Graph Coloring I

Problem
Given a undirected graph $G$. Color each node of the graph by red, yellow, or blue so that no two adjacent nodes have the same color.

Approach
We will solve the problem by writing a logic program $\Pi_G$ such that each answer set of $\Pi_G$ gives us a solution to the problem. Furthermore, each solution of the problem corresponds to an answer set.

The program $\Pi_G$ needs to contain information about the graph and then the definition of the problem. So,

- Graph representation:
  - The nodes: $node(1), \ldots, node(n)$.
  - The edges: $edge(i, j)$.

- Solution representation: use the predicate $colored(X, Y)$ - node $X$ is assigned the color $Y$. 
Graph Coloring II

- Generating the solutions: Each node is assigned one color. The three rules

\[
\begin{align*}
\text{colored}(X, \text{red}) & \leftarrow \text{not colored}(X, \text{blue}), \text{not colored}(X, \text{yellow}) \quad (2) \\
\text{colored}(X, \text{blue}) & \leftarrow \text{not colored}(X, \text{red}), \text{not colored}(X, \text{yellow}) \quad (3) \\
\text{colored}(X, \text{yellow}) & \leftarrow \text{not colored}(X, \text{blue}), \text{not colored}(X, \text{red}) \quad (4)
\end{align*}
\]

- Checking for a solution: needs to make sure that no edge connects two nodes of the same color. This can be represented by a constraint:

\[
\leftarrow \text{edge}(X, Y), \text{colored}(X, C), \text{colored}(Y, C). \quad (5)
\]
Graph Coloring III

%%% graphcolor.lp
%%% the graph
node(1). node(2). node(3).
node(4). node(5). edge(1,2).
edge(1,3). edge(2,4).
edge(2,5). edge(3,4). edge(3,5).

%%% each node is assigned a color
colored(X,red):- node(X), not colored(X,blue), not colored(X, yellow).
colored(X,blue):- node(X), not colored(X,red), not colored(X, yellow).
colored(X,yellow):- node(X), not colored(X,blue), not colored(X, red).

%%% constraint checking
:- edge(X,Y), colored(X,C), colored(Y,C).

Try with
    clingo graphcolor.lp  or
    clingo 0 graphcolor.lp
and see the result. The atom node(X) in the rules with the head colored(X, _) is needed because the solver’s “unsafe variable” error.
graphcolor.lp can be divided into three group of rules

- Rules for *describing the graph*;
- Rules for *generating the hypothetical solutions*; and
- Rules for *checking the correctness of the solutions*.

It is a good practice to separate the rules into two files: (a) the first file contains rules for describing the graph (let us called this file graphcolor1.lp, for our example, this program contains the first three lines of graphcolor.lp); and (b) the second file contains other rules. (let us called this file graphcolor2.lp which contains the other rules of graphcolor2.lp).

Command line: clingo graphcolor1.lp graphcolor2.lp
Correctness of graphcolor.lp

Let us denote the program graphcolor.lp developed for a graph $G$ by $\Pi_G$.

- **What is to prove?** (one-to-one mapping between solutions of $G$ and answer sets of $\Pi_G$.) Intuitively, this means
  - If $\Pi_G$ has an answer set then the 3-coloring problem for $G$ has a solution and vice versa.
  - If $\Pi_G$ does not have an answer set then the coloring problem of $G$ has no solution.

- **How can we prove this?**
  - Take an answer set $S$ of $\Pi_G$, construct a solution for the coloring problem of $G$ from $S$.
  - Take a color mapping $M$, which is a solution for the problem, construct an answer set for $\Pi_G$. 
Correctness of graphcolor.lp II

We can prove the following theorems.

**Theorem**

*Let $S$ be an answer set of $\Pi_G$. Then, the 3-coloring problem of $G$ has a solution corresponds to $S$.*

**Proof.** We prove the following:

1. For every node $k$ of the graph $G$, $S$ contains one and only one atom from the set
   
   $C = \{\text{colored}(k, \text{red}), \text{colored}(k, \text{blue}), \text{colored}(k, \text{yellow})\}$, i.e., $S \cap C$ has only one element. Assume that it is not the case. Then, there are only three cases: $S$ contains zero, two, or three elements of the set $C$. Assume that

   **Case 1:** $S$ does not contain any element from $C$. Then, $\Pi^S_G$ contains
   
   $\text{colored}(k, \text{red}) \leftarrow \text{node}(k)$. (because of rule (2))
   
   $\text{colored}(k, \text{blue}) \leftarrow \text{node}(k)$. (because of rule (3))
   
   $\text{colored}(k, \text{yellow}) \leftarrow \text{node}(k)$. (because of rule (4))
Correctness of graphcolor.lp III

Because $k$ is a node of $G$, we can easily see that $\text{colored}(k, \text{red})$, $\text{colored}(k, \text{yellow})$, and $\text{colored}(k, \text{blue})$ belong to the minimal model of $\Pi^S_G$. Thus, $S$ cannot be an answer set of $\Pi_G$ because it cannot be equal the minimal model of $\Pi^S_G$. This contradicts the assumption that $S$ is an answer set of $\Pi_G$. Hence, this case cannot happen.

**Case 2:** $S$ contains two elements from $C$. Since the three colors are equivalent and so, without loss of generality, we assume that $S$ contains $\text{colored}(k, \text{red})$ and $\text{colored}(k, \text{blue})$; and it does not contain $\text{colored}(k, \text{yellow})$. Then, $\Pi^S_G$ will not contain any rule whose head is an atom belonging to $C$. (all the rules of the form (2)-(4) for $X = k$ are removed). This implies that the minimal model of $\Pi^S_G$ cannot contain any element of $C$. This implies that $S$ cannot be an answer set of $\Pi_G$ because it cannot be equal the minimal model of $\Pi^S_G$. Again, this contradicts the assumption that $S$ is an answer set of $\Pi_G$. Hence, this case cannot happen.
Correctness of graphcolor.lp IV

**Case 3:** $S$ contains all elements of $C$. This is similar to the second case, i.e., $\Pi^S_G$ does not contain any rule whose head is a member of $C$, and hence, $S$ would not be an answer set of $\Pi_G$.

The three cases show that for each node $k$, $S$ contains 1-and-only-1 member of the set
\[
\{\text{colored}(k, \text{red}), \text{colored}(k, \text{blue}), \text{colored}(k, \text{yellow})\}.
\]

2. Now we need to show that the color mapping specified by the answer set $S$ is a solution to the coloring problem.

Let $1, \ldots, n$ be the nodes of the graph and $c_1, \ldots, c_n$ be the color such that $\text{colored}(i, c_i) \in S$ for $i = 1, \ldots, n$. We need to show that if $\text{edge}(i, j)$ belongs to $G$ then $c_i \neq c_j$. Again, we prove by contradiction. Let assume that there is an edge $(p, q)$ in $G$ and $c_p = c_q$. This means that the body of the rule (5) for $\text{edge}(p, q)$, $\text{colored}(p, c_p)$, and $\text{colored}(q, c_q)$ is satisfied. This means that $S$ is not an answer set of $\Pi_G$, i.e., our assumption contradicts the fact that $S$ is an answer set of $\Pi_G$. Thus, our assumption is incorrect, i.e., we have proved that for every edge $(i, j)$ of $G$, $c_i \neq c_j$.
Correctness of graphcolor.lp

shows that $S$ corresponds to a solution of the 3-coloring problem for $G$.

**Theorem**

*If the 3-coloring problem for $G$ has a solution $M$ then $\Pi_G$ has an answer set corresponds to $M$.*

**Proof.** Let $1,\ldots, n$ be the nodes of the graph. Consider a solution for the 3-coloring problem for $G$. Let $c_1,\ldots, c_n$ be the color of the node $i = 1,\ldots, n$, respectively. We will show that the set of atoms

$$S = \{\text{node}(i) \mid i = 1,\ldots, n\} \cup \{\text{edge}(i,j) \mid (i,j) \text{ is an edge of } G\} \cup \{\text{colored}(i, c_i) \mid i = 1,\ldots, n\}$$

is an answer set of $\Pi_G$.

Let us compute $\Pi_G^S$. We can see that $\Pi_G^S$ consists of the following rules:
Correctness of graphcolor.lp VI

- the rules defining the graph, i.e., the rules \( \text{node}(1) \leftarrow \ldots \)
  \( \text{node}(n) \leftarrow \) (nodes of the graph) and the rules \( \text{edge}(i, j) \leftarrow \) if \((i, j)\) is an edge of \(G\).

- for each node \(i\), one rule of the form \( \text{colored}(i, c_i) \leftarrow \text{node}(i) \)
  which comes from one of the rules (2)-(4).

- for each edge \((i, j)\), three constraints:
  \( \leftarrow \text{edge}(i, j), \text{colored}(i, \text{red}), \text{colored}(j, \text{red}) \)
  \( \leftarrow \text{edge}(i, j), \text{colored}(i, \text{yellow}), \text{colored}(j, \text{yellow}) \)
  \( \leftarrow \text{edge}(i, j), \text{colored}(i, \text{blue}), \text{colored}(j, \text{blue}) \)

We need to show that \(S\) is the minimal model of \(\Pi_G^S\) that does not violate any of the constraints.

Let
\[
X_0 = \{ \text{node}(i) \mid i = 1, \ldots, n \} \cup \{ \text{edge}(i, j) \mid (i, j) \text{ is an edge of } G \}.
\]

Obviously, \( T_{\Pi_G^S}(\emptyset) = X_0 \) and
\[
T_{\Pi_G^S}(X_0) = S, \text{ and } T_{\Pi_G^S}(S) = S. \tag{\ast}
\]
Furthermore, because for every edge \((i, j)\), \(c_i \neq c_j\), we can conclude that there exists no edge \((i, j)\) in \(G\) and a color \(C \in \{\text{red, blue, yellow}\}\) such that \(S\) contains \(\text{colored}(i, C)\) and \(\text{colored}(j, C)\). This is equivalent to say that the constraints in \(\Pi_G^S\) are satisfied by \(S\). Together with (*)\), we conclude that \(S\) is an answer set of \(\Pi_G\).
Problem
Ann, John, Sally, and Mike often have lunch in the break room of the department. The room has two tables.

- Mike wants to be alone or with Ann.
- John does not care.
- Ann likes to be either with Mike and Sally or with Sally and John.
- Sally does not want to be alone.

Find a seat arrangement that can make everybody happy.
Seating Arrangements II

- Problem representation: \textit{table}(1) and \textit{table}(2); \textit{person}(	extit{mike}), \textit{person}(	extit{sally}), \textit{person}(	extit{ann}), and \textit{person}(	extit{john}).

- Result representation: \textit{at}(	extit{mike}, 1), \textit{at}(	extit{mike}, 2), etc.

- The rules:
  \[
  \text{at}(X, 1) \leftarrow \text{not } \text{at}(X, 2).
  \]
  \[
  \text{at}(X, 2) \leftarrow \text{not } \text{at}(X, 1).
  \]

- \texttt{clingo} and see "X is unsafe"

- add "what is X" to the rule: \textit{person}(X) or \textit{table}(X)"

- run the command \texttt{clingo 0 seats.lp} and see the possible arrangements.

- add \texttt{constraints} to remove unwanted answer sets.
Choice Atoms
colored(X,red):- node(X), not colored(X,blue), not colored(X, yellow).
colored(X,blue):- node(X), not colored(X,red), not colored(X, yellow).
colored(X,yellow):- node(X), not colored(X,blue), not colored(X, red).
replaced by
\[ 1 \{\text{colored}(X, C) : \text{is\_color}(C)\} 1 :- \text{node}(X). \]
and a set of atoms
\[ \text{is\_color(yellow)}. \quad \text{is\_color(red)}. \quad \text{is\_color(blue)}. \]
Choice atoms allow for a succinct representation. General form of choice atoms is
\[ l \{ p_1, p_2, \ldots, p_k \} u \]
where \( 0 \leq l \leq u \) are integers and \( p_i \)'s are atoms. Expression of the from \( \{p(X) : q(Y)\} \) where all variables in \( Y \) appear in \( X \). A choice atom is true with respect to a set of atoms \( S \) if \( l \leq |\{p_i \mid p_i \in S\}| \leq u \).
Weighted Atoms

\[ l \{ l_0 = w_0, \ldots, l_k = w_k, \text{not} \ l_{k+1} = w_{k+1}, \ldots, \text{not} \ l_{k+n} = w_{k+n} \} u \]

where \( l_i \)’s are atoms, \( w_i \) are integers, and \( l \leq u \) are integers. This atom is true with respect to a set of literals \( S \) if

\[
l \leq \sum_{l_j \in S} w_j + \sum_{l_j \not\in S} w_j \leq u
\]

Special case: \textit{choice atom} – \( w_i = 1 \) for every \( i \).

Aggregates

\textit{Sum}(\( \Omega \)), \textit{Count}(\( \Omega \)), \textit{Average}(\( \Omega \)), \textit{Min}(\( \Omega \)), \textit{Max}(\( \Omega \)) where \( \Omega \) denotes a multiset (e.g., \( \{ p(a, X) \mid X \in \{1, 2, 3\} \} \))
Problem: Place $n$ queens on a $n \times n$ chess board so that no queen is attacked (by another one).
n-Queens

- **Representation:** the chess board can be represented by a set of cells $cell(i, j)$ and the size $n$.

- **Solution:** Each cell is assigned a number 1 or 0. $cell(i, j) = 1$ means that a queen is placed at the position $(i, j)$ and $cell(i, j) = 0$ if no queen is placed at the position $(i, j)$.

- **Generating a possible solution:**
  - $cell(i, j)$ is either true or false
  - select $n$ cells, each on a column, assign 1 to these cells.

- **Checking for the solution:** ensures that no queen is attacked
n-Queens – writing a program

Use a constant \( n \) to represent the size of the board

\[
\text{col}(1..n). \quad // \ n \ columns
\]
\[
\text{row}(1..n). \quad // \ n \ rows
\]

Since two queens cannot be on the same column, we know that each column has to have one and only one queen. Thus, using the choice atom in the rule

\[
1 \{ \text{cell}(I, J) : \text{row}(J) \} 1 \leftarrow \text{col}(I).
\]

we can make sure that only one queen is placed on one column.

To complete the program, we need to make sure that the queens do not attack each other.

- No two queens on the same row
  \[
  \leftarrow \text{cell}(I, J1), \text{cell}(I, J2), J1 \neq J2.
  \]
- No two queens on the same column (not really needed)
  \[
  \leftarrow \text{cell}(I1, J), \text{cell}(I2, J), I1 \neq I2.
  \]
- No two queens on the same diagonal
  \[
  \leftarrow \text{cell}(I1, J1), \text{cell}(I2, J2), |I1 - I2| = |J1 - J2|
  \]
% representing the board, using n as a constant
col(1..n). % n column
row(1..n). % n row
% generating solutions
1 {cell(I,J) : row(J) } 1:- col(I).
% two queens cannot be on the same row/column
:- col(I), row(J1), row(J2), J1!=J2, cell(I,J1), cell(I,J2).
:- row(J), col(I1), col(I2), I1!=I2, cell(I1,J), cell(I2,J).
% two queens cannot be on a diagonal
:- row(J1), row(J2), J1 > J2, col(I1), col(I2), I1 > I2, cell(I1,J1),
cell(I2,J2), I1 - I2 == J1 - J2.
:- row(J1), row(J2), J1 > J2, col(I1), col(I2), I1 < I2, cell(I1,J1),
cell(I2,J2), I2 - I1 == J1 - J2.
Command line: clingo -c n=input queens.lp
Sudoku

Facts:  \( c(1..9). \)  \( r(1..9). \)  \( num(1..9). \)
\( an(1, 1, 2). \)  \( an(1, 3, 9). \)  \( an(1, 4, 6) \ldots \)
\( ar(1..3, 1..3, 1). \)  \( ar(1..3, 4..6, 2). \)  \( ar(1..3, 7..9, 3). \)
\( ar(4..6, 1..3, 4). \)  \( ar(4..6, 4..6, 5). \)  \( ar(4..6, 7..9, 6). \)
\( ar(7..9, 1..3, 7). \)  \( ar(7..9, 4..6, 8). \)  \( ar(7..9, 7..9, 9). \)

Generation:
1{\( an(X, Y, N) : num(N) \)}1 : \( -c(X), r(Y) \).

Check:
: \( -c(Y), c(X), X \neq Y, r(R), an(R, X, N), an(R, Y, N). \)
: \( -r(Y), r(X), X \neq Y, c(C), an(Y, C, N), an(X, C, N). \)
: \( -ar(X, Y, Z), ar(P, Q, Z), X \neq P, an(X, Y, N), an(P, Q, N). \)
: \( -ar(X, Y, Z), ar(P, Q, Z), Y \neq Q, an(X, Y, N), an(P, Q, N). \)

(c - column, r - row, num - number, an - answer, ar - area)
Knapsack Problem

**Problem**
There are several items. Each has some weight and value. There is a backpack that can only carry a maximal weight of $M$. Identify the set of items that should be put in the backpack so that the total value is maximal.

**Example**
Five items: $i_1$ (12kg, $4), i_2$ (2kg, $2), i_3$ (1kg, $1), i_4$ (4kg, $10), i_5$ (1kg, $2$).
Backpack: maximal 15 kg.
Knapsack Problem

- Problem representation: items, values, weights
- Solution generation:
- Checking for solution:
Knapsack Problem

- Problem representation: items, values, weights
  item(1..n). value(1,4). weight(1,12). ...

- Solution generation:
  \{\text{in}(I)\} :- \text{item}(I).

- Checking for solution:
  \textbf{Compute the total weight}
  \text{total\_weight}(W) :- W = \#\text{sum} \{N : \text{in}(I), \text{weight}(I,N)\}.
  :- \text{total\_weight}(W), W > 15.

  \textbf{Compute total value}
  \text{total\_value}(V) :- V = \#\text{sum} \{VA : \text{in}(I), \text{value}(I, VA)\}.
  \#\text{maximize} \{V : \text{total\_value}(V)\}.
Knapsack Problem: Code

item(1..5).
value(1,4). value(2,2).
value(3,1). value(4,10).
value(5,2).
weight(1,12). weight(2,2).
weight(3,1). weight(4,4).
weight(5,1).
{in(I)} :- item(I).
total_weight(W) :- W = #sum {N : in(I), weight(I,N)}.
:- total_weight(W), W > 15.
total_value(V) :- V = #sum {VA : in(I), value(I, VA)}.
#maximize {V : total_value(V)}. 
Some Notes on Syntax used by clingo

item(1..5).
value(1,4). value(2,2). value(3,1). value(4,10). value(5,2).
weight(1,12). weight(2,2). weight(3,1). weight(4,4). weight(5,1).
{in(I)} :- item(I).

Old

total_weight(W) :- W = #sum \{N : in(I), weight(I,N)\}.
:- total_weight(W), W > 15.
total_value(V) :- V = #sum \{VA : in(I), value(I, VA)\}.
#maximize \{V : total_value(V)\}.

New

total_weight(W) :- W = #sum \{N,I : in(I), weight(I,N)\}.
:- total_weight(W), W > 15.
total_value(V) :- V = #sum \{VA,I : in(I), value(I, VA)\}.
#maximize \{V : total_value(V)\}.
General Syntax: guide.pdf (2.0)

\[ s_1 \prec_1 \alpha \{ t_1 : L_1; \ t_2 : L_2; \ldots; \ t_n : L_n \} \prec_2 s_2 \]

where

\[ \begin{align*} &\text{\textbullet } t_i \text{ and } L_i \text{ are non-empty tuples of terms and atoms, respectively.} \\
&\text{\textbullet } \alpha \text{ can either be } \#\text{count}, \#\text{sum}, \#\text{min}, \#\text{max}, \#\text{sum+} \text{ (sum only positive weights)} \\
&\text{\textbullet } \prec_1 \text{ and } \prec_2 : \text{comparison operator } (=, \leq, \geq, <, >, \text{ default is } \leq) \end{align*} \]

Example

\[ \#\text{sum} \ \{ \ 3 : \text{bananas}; \ 25 : \text{cigars}; \ 10 : \text{broom} \ \} \ \leq 30. \]
\[ \text{\textbullet if bananas, cigars, broom are true then it results in } 38 \leq 30 \text{ (false)} \]
\[ \text{\textbullet if bananas, cigars are true (broom is false) it results in } 28 \leq 30 \text{ (true)} \]
\[ \#\text{count} \ \{ \ 42 : \text{a}; \ 42 : \text{a}; \ t : \text{not b}; \ t : \text{not b} \ \} = 2. \]
\[ \text{\textbullet if a and b are true then it results in } \{42\} = 2 \text{ (false)} \]
\[ \text{\textbullet if a is true and b is false then it results in } \{42;t\} = 2 \text{ (true)} \]
Set atom with variable

\[
\text{total_weight}(W) : \ W = \#\text{sum} \ \{N, I : \text{in}(I), \ \text{weight}(I, N)\}.
\]

The variables: \( I \) and \( N \)

This stands for

\[
\text{total_weight}(W) : \ W = \#\text{sum} \ \{12, 1 : \text{in}(1); 2, 2 : \text{in}(2); 1, 3 : \text{in}(3); 4, 4 : \text{in}(4); 1, 5 : \text{in}(5)\}.
\]
K-clique

- **Problem:** Given a number $k$ and a graph $G$. $G$ has a clique of size $k$ if there is a set of $k$ different vertices (nodes) in $G$ such that each pair of vertices from this set is connected through an edge.

- **Representation:**
  - Graph ($node()$ and $edge()$)
  - Clique ($clique(N)$) to say that node $N$ belongs to the clique if $clique(N)$ is true; otherwise it does not belong to the clique.

- **Generating a solution:** Selecting $k$ nodes – this is equivalent to assigning $k$ atoms of the set \{clique(1), \ldots, clique(n)\} the truth value true. This can be achieved by the rule

- **Checking for a solution:** if every pair of the selected nodes is connected then this is a solution; otherwise it is not a solution. This means that there exists no pair $(I, J)$ such that $clique(I)$ and $clique(J)$ are true but $edge(I, J)$ is not true.
K-clique

- **Problem**: Given a number $k$ and a graph $G$. $G$ has a clique of size $k$ if there is a set of $k$ different vertices (nodes) in $G$ such that each pair of vertices from this set is connected through an edge.

- **Representation**:
  - Graph ($node()$ and $edge()$)
  - Clique ($clique(N)$) to say that node $N$ belongs to the clique if $clique(N)$ is true; otherwise it does not belong to the clique.

- **Generating a solution**: Selecting $k$ nodes – this is equivalent to assigning $k$ atoms of the set $\{clique(1), \ldots, clique(n)\}$ the truth value true. This can be achieved by the rule $k\{clique(N) : node(N)\}k$.

- **Checking for a solution**: if every pair of the selected nodes is connected then this is a solution; otherwise it is not a solution. This means that there exists no pair $(I, J)$ such that $clique(I)$ and $clique(J)$ are true but $edge(I, J)$ is not true. $\leftarrow clique(I), clique(J), I \neq J, \textbf{not} edge(I, J), \textbf{not} edge(J, I)$.
% Graph
node(1..5).
edge(1,2). edge(1,3). edge(2,3).
edge(1,4). edge(1,5).
edge(N1,N2):- edge(N2,N1).
% generating solution
k {clique(N):node(N)} k.
% checking solution
:- clique(N1), clique(N2), N1 !≡ N2,
    not edge(N1,N2).
Tile covering problem

**Problem**: Given a slightly damaged chess board \((n \times n)\). Find a covering of the board using \(m 1 \times 2\)-tiles so that all the good squares on the board are covered. If no such covering exists, report there is no solution.
Tile covering problem

**Problem:** Given a slightly damaged chess board \((n \times n)\). Find a covering of the board using \(m\) 1 \times 2-tiles so that all the good squares on the board are covered. If no such covering exists, report there is no solution.
Tile covering problem I

- **Problem:** Given a slightly damaged chess board \((n \times n)\). Find a covering of the board using \(m\) \(1 \times 2\)-tiles so that all the good squares on the board are covered. If no such covering exists, report there is no solution.

- **Representation:**
  - the board
    - dimension
    - damaged/good cells
  - the tiles with their coverage

As with N-queens problem, we can use the two predicates \(\text{row}()\) and \(\text{col}()\) to represent the possible board cells. We can use \(\text{bad}(i, j)\) to indicate that the cell \((i, j)\) is damaged. We can use the predicate \(\text{cell}(i, j, t)\) to represent the information that the cell \((i, j)\) is covered by the tile \(t\).
Generating a possible solution: We assign each tile a number. Then we can assign a tile to a location. Since each tile covers two and only two cells, we can use the weighted-rule:

\[ 2\{\text{cell}(I, J, T) : \text{col}(I), \text{row}(J)\} \leftarrow \text{tile}(T). \]

to generate a possible solution.

Checking for a solution: We need to check for the following constraints:

- bad cell needs not be covered
- no two tiles on the same cell
- The cells covered by a tile must be neighbor
- The cells covered by a tile cannot lie on a diagonal
% Board representation
col(1..n). row(1..n). tile(1..ntiles).
bad(1,1). bad(3,2).
% generating solution
2 {cell(I,J,T) : col(I),row(J) } 2 :- tile(T).
% checking solution
% bad cell needs not ..
:- bad(I,J), cell(I,J,T).
% one cell one tile ....
:- cell(I,J,T1),cell(I,J,T2),T1!=T2.
% neightbor only ....
:- cell(I1,J1,T), cell(I2,J2,T), I1 - I2 > 1.
:- cell(I1,J1,T), cell(I2,J2,T), J1 - J2 > 1.
% no diagonal ...
:- cell(P,Q,T), cell(R,S,T), P!=R, Q!=S.
Outline

Logic Programming
  Syntax
  Examples of Propositional Programs
  Programs with FOL Atoms

Answer Set Solver: clingo (How To?)

Reasoning about Dynamic Domains

Logic Programming and Knowledge Representation

Extensions of Logic Programming and Computing Answer Sets

Advanced Problems in ASP
Block World Domain

Diagram: 
- b (blue) is on top of c (red) which is on the ground.
- a (green) is to the right of b and c.
Block World Domain

```
move(a, b)
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
</tbody>
</table>
```
Block World Domain

```
move(a, b)
```

Diagram:

```
| b | c | a |
-|-|--
```

```
| a | b |
-|-|--
```

Block World Domain

move(a, b)

before

after
Dynamic Domains

The state of the world continuously changes through the execution of actions (by some agent). State of the world can be described by a set of fluents (properties of the world whose truth values change over time). We are interested in

- predicting the state of the world after the execution of an action sequence;
- identifying ways to change the world to a pre-defined state;
- explaining the discrepancies that we observe.
Modeling the Block World Domain: Fluents and Actions

Fluents

- $\text{on}(X, Y)$: block $X$ is on block $Y$
- $\text{onTable}(X)$: block $X$ is on the table
- $\text{clear}(X)$: block $X$ is clear
- $\text{holding}(X)$: the agent holds the block $X$
- $\text{handEmpty}$: the agent does not hold anything

Actions

- $\text{un}/\text{stack}(X, Y)$: un/stack block $X$ is on block $Y$
- $\text{pickup}/\text{putdown}(X)$: pickup or put down block $X$ from/on to the table
Modeling the Block World Domain: Action Description I

Action Preconditions and Effects:

- **stack**(X, Y)
  - Precondition:
    - block Y is clear
    - the agent holds the block X
  - Effects:
    - X is clear
    - X is on Y
    - Y is no longer clear
    - the agent does not hold anything

- **putdown**(X)
  - Precondition:
    - the agent holds the block X
  - Effects:
    - X is clear
    - X is on the table
    - the agent does not hold anything
Modeling the Block World Domain: Action Description II

Action Preconditions and Effects:

- **unstack**($X$, $Y$)
  - **Precondition:**
    - $X$ is clear
    - $X$ is on $Y$
    - the agent does not hold anything
  - **Effects:**
    - the agent holds the block $X$
    - $Y$ becomes clear
    - $X$ is not clear

- **pickup**($X$)
  - **Precondition:**
    - $X$ is clear
    - $X$ is on the table
    - the agent does not hold anything
  - **Effects:**
    - the agent holds the block $X$
    - $X$ is no longer on the table and not clear
% Defining the time constants and objects
time(0..length).
block(a). block(b). block(c).

% Defining fluents
fluent(on(X,Y)):- block(X), block(Y), X!=Y.
fluent(onTable(X)):- block(X).
fluent(clear(X)):- block(X).
fluent(holding(X)):- block(X).
fluent(handEmpty).

% Defining actions
action(stack(X,Y)):- block(X), block(Y), X!=Y.
action(unstack(X,Y)):- block(X), block(Y), X!=Y.
action(putdown(X)):- block(X).
action(pickup(X)):- block(X).
Block World Domain: Encoding II

% Action effects & precondition: pickup
executable(pickup(X), T):- holds(clear(X), T),
    holds(handEmpty, T),
    holds(onTable(X), T).
:- occ(pickup(X), T), not executable(pickup(X), T).

holds(neg(clear(X)), T+1):- occ(pickup(X), T).
holds(neg(onTable(X)), T+1):- occ(pickup(X), T).
holds(neg(handEmpty), T+1):- occ(pickup(X), T).
holds(holding(X), T+1):- occ(pickup(X), T).

% unstack
executable(unstack(X,Y), T) :- holds(clear(X), T), holds(on(X,Y), T),
    holds(handEmpty, T).
holds(neg(clear(X)), T+1):- occ(unstack(X,Y), T).
holds(neg(on(X,Y)), T+1):- occ(unstack(X,Y), T).
holds(holding(X), T+1):- occ(unstack(X,Y), T).
holds(neg(handEmpty), T+1):- occ(pickup(X), T).
holds(clear(Y), T+1):- occ(unstack(X,Y), T).
:- occ(unstack(X,Y), T), not executable(unstack(X,Y), T).
% Action effects & precondition: putdown
executable(putdown(X), T):- holds(holding(X), T).
holds(clear(X), T+1):- occ(putdown(X), T).
holds(onTable(X), T+1):- occ(putdown(X), T).
holds(neg(holding(X)), T+1):- occ(putdown(X), T).
holds(handEmpty, T+1):- occ(putdown(X), T).
:- occ(putdown(X), T), not executable(putdown(X), T).

% stack
executable(stack(X,Y), T):- holds(clear(Y), T), holds(holding(X), T).
holds(clear(X), T+1):- occ(stack(X,Y), T).
holds(on(X,Y), T+1):- occ(stack(X,Y), T).
holds(neg(holding(X)), T+1):- occ(stack(X,Y), T).
holds(neg(clear(Y)), T+1):- occ(stack(X,Y), T).
holds(handempty, T+1) :- occ(stack(X,Y), T).
:- occ(stack(X,Y), T), not executable(stack(X,Y), T).
Block World Domain: Initial State, Inertial Axiom

holds(on(b,c), 0).  holds(onTable(b), 0).
holds(onTable(a), 0).  holds(clear(a), 0).
holds(clear(b), 0).  holds(handEmpty, 0).
holds(neg(F), 0) :- fluent(F), not holds(F, 0).

% inertial axioms
holds(F, T+1) :- fluent(F), holds(F, T), not holds(neg(F), T+1).
holds(neg(F), T+1) :- fluent(F), holds(neg(F), T), not holds(F, T+1).

Planning

% action occurrences generation
1 {occ(A,T) : action(A) } 1 :- time(T), T ¡ length.
% Goal checking
goal(T) :- holds(on(b,a), T), holds(on(a,c), T), holds(onTable(c), T).
:- not goal(length).
Let $\Pi_{\text{block}}(n)$ be the program for the block world domain with length sets to $n$ without the set of rules for planning (the rules in the block below, denoted by $Plan(n)$).

**Plan(n)**

```
% action occurrences generation
1 {occ(A,T) : action(A) } 1 :- time(T), T < length.
% Goal checking
goal(T) :- holds(on(b,a), T), holds(on(a,c), T), holds(onTable(c), T).
:- not goal(length).
```

This program can be used for two tasks: planning and hypothetical reasoning.
Planning in the Block World Domain

Planning is the problem of computing an action sequence that transforms the state of the world from the initial state to the goal state.

To use the program for planning, we need to add the planning module (\(Plan(n)\)) to the program. This module generates the action occurrences, defines the goal, and checks for the satisfiability of the goal.

To run the code, we need to provide an estimated length or use a script to run with length=1, 2, ...:

\[
\text{clingo -c length=<n> block.lp}
\]

Answer sets of \(\Pi_{block}(n) \cup Plan(n)\) contain atoms of the form

- \(\text{occ}(a, t)\) where \(a\) is an action and \(t\) is an integer.
- \(\text{holds}(l, t)\) where \(l\) is a fluent literal (\(f/\neg f\)) and \(t\) is an integer.
- others such as \(\text{executable}(a, t)\), etc.

We can show that if \(A\) is an answer set of the program \(\Pi_{block}(n) \cup Plan(n)\) then \(A\) will contain the atoms \(\text{occ}(a_0, 0), \ldots, \text{occ}(a_{n-1}, n-1)\) and \([a_0, \ldots, a_{n-1}]\) is a solution (soundness); and if the problem has a plan \([b_0, \ldots, b_{n-1}]\), then \(\Pi_{block}(n) \cup Plan(n)\) has an answer set \(B\) containing \(\text{occ}(b_0, 0), \ldots, \text{occ}(b_{n-1}, n-1)\) (completeness).
Example 1

How do we encode the problem?

► the initial state
► the goal state

► Initial state:
% Expressing what are true!
holds(on(b,c),0). holds(onTable(c),0). holds(onTable(a),0).
holds(clear(a),0). holds(clear(b),0). holds(handEmpty,0).
% Expressing what we do not state as true are false!
holds(neg(F),0) :- fluent(F), not holds(F,0).

► The goal state:
% Defining what should be the goal!
goal(T) :- holds(on(b,a),T), holds(on(a,c),T), holds(onTable(c),T).
Example 2

- What should be changed?
- How to encode the changes?

There is one additional block (Block d).
The initial state and the goal state change.

- The object: add block(d).

% Initial state: Expressing what are true!
holds(on(d,c),0). holds(onTable(c),0). holds(clear(d),0).
holds(on(b,a),0). holds(onTable(a),0). holds(clear(b),0).
holds(handEmpty,0).

% Expressing what we do not state as true are false!
holds(neg(F),0) :- fluent(F), not holds(F,0).

% The goal state: Defining what should be the goal!
goal(T) :- holds(on(d,b),T), holds(on(b,a),T),
holds(on(a,c),T), holds(onTable(c),T).
Organizing Code

The ASP code for the block world domain is often organized into different files:

- The action description: this contains the description of the fluents, the actions, the action effects, the inertial axioms.
- The instance: this includes the rules defining the objects, the initial state, and the goal state.

Alternative Goal Encoding

For a set of literal \( \{l_1, \ldots, l_n\} \), the goal \( l_1 \land l_2 \ldots \land l_n \) can be encoded using the set of rules:

\[
\begin{align*}
goal(l_1). & \quad \ldots \quad goal(l_n). \\
:- \ goal(l), \ not \ holds(l, \ length). \end{align*}
\]
Hypothetical reasoning in the Block World Domain

Hypothetical reasoning: predicting what state of the world will be after the execution of an action sequence.

Assume that we would like to execute the sequence of actions $\alpha = [a_0, \ldots, a_{k-1}]$. To use $\Pi_{\text{block}}(n)$ for hypothetical reasoning we need to:

- set the value of $\text{length}$ to be $k$ the length of the action sequence
- add to the program the collection of facts $OCC(\alpha) = \{occ(a_0, 0), \ldots, occ(a_{k-1}, k - 1)\}$.

To find out whether a fluent $f$ is true (false) after the execution of the action sequence $\alpha$, we need to

- compute all answer sets of $\Pi_{\text{block}}(k) \cup OCC(\alpha)$
- check whether $\text{holds}(f, k)$ is in each answer set; if it is, then answer YES.
Examples

▶ unstack(b); putdown(b)
▶ unstack(b); putdown(b); stack(b, a)
▶ unstack(b); putdown(b); pickup(a); stack(a, c); pickup(b); stack(b, a)
▶ unstack(b); putdown(b); pickup(a); stack(a, c); pickup(b); stack(b, a)
▶ pickup(a); putdown(a); unstack(b); putdown(b); pickup(a); stack(a, c); pickup(b); stack(b, a)
A Variation of the Yale Shooting Problem

Consider the story: *Matt* – a turkey – is walking along the road. *Jimmy* – a hunter – is coming from the opposite direction. He takes out a loaded gun and shoots at *Matt*. 
A Variation of the Yale Shooting Problem

Consider the story: *Matt* – a turkey – is walking along the road. *Jimmy* – a hunter – is coming from the opposite direction. He takes out a loaded gun and shoots at *Matt*. There are several questions that arise given the above story:

▶ Is *Matt* still alive?
▶ Is the gun still loaded?
▶ Is *Jimmy* walking?
▶ Does *Jimmy* have the same number of guns?
▶ etc.

How do we answer the above questions?
Frame and Ramification Problem

Consider the story: *Matt* – a turkey – is walking along the road. *Jimmy* – a hunter – is coming from the opposite direction. He takes out one of his loaded guns and shoots at *Matt*.

**Frame Problem**

From the story, we know that the action “shoot” occurs. Commonsense tells us that the turkey will be dead and the gun becomes unloaded if it can hold at most one bullet. However, several other properties of the environment stay unchanged after the action has completed. For instance,

- the number of bullets in other guns of Jimmy does not change;
- the number of guns belonging to Jimmy does not change;
- the amount of water in the lake nearby does not change; etc.

**Ramification Problem**

We also know that if Matt is hit by the gun, he will be dead and if he is dead he cannot not continue walking.
We consider the situation where Jimmy has only one gun for now.

- **Fluents:** walking, loaded, dead
  
  We know that *alive* and *dead* are “two sides of a coin.” For now, let us consider only one of them.

- **Actions:** shoot, load
  
  - **shoot:** if the is loaded then the turkey to be dead; otherwise, nothing happens
    
    - Precondition: *True*
    
    - Effects: two situations (loaded then dead; ¬loaded then nothing)
  
  - **load:**
    
    - Precondition: the gun is unloaded
    
    - Effects: the gun is loaded
ASP Encoding

% Defining the steps
time(0..length).
% Defining fluents
fluent(loaded). fluent(dead). fluent(walking).
% Defining actions
action(load). action(shoot).
% Representing action’s effects
executable(shoot, T):- time(T).
holds(dead, T+1) :- occ(shoot, T), holds(loaded, T).
holds(-loaded, T+1) :- time(T), occ(shoot, T).
executable(load, T):- time(T), holds(-loaded, T).
holds(loaded, T+1) :- time(T), occ(load, T).
% The inertial rule
holds(F,T+1):- time(T), fluent(F), holds(F,T), not holds(-F,T+1).
holds(-F,T+1):- time(T), fluent(F),holds(-F,T),not holds(F,T+1).
% The initial state
holds(loaded, 0). holds(-dead, 0). holds(walking, 0).
Let us denote this program with $\Pi_{Turkey}(length)$.
Answering Questions

- Is *Matt* still alive? (after the shoot action!)
- Is the gun still loaded?
- Is *Jimmy* walking?
- Does *Jimmy* have the same number of guns?
Answering Questions

- Is Matt still alive? (after the shoot action!)
  - \( length = 1 \) (only one action occurrence)
  - Add the \( occ(shoot, 0) \) to \( \Pi_{Turkey}(1) \)
  - Computer answer sets of the resulting program and check for answer

- Is the gun still loaded?
- Is Jimmy walking?
- Does Jimmy have the same number of guns?
Answering Questions

- Is Matt still alive? (after the shoot action!)
  - \(\text{length} = 1\) (only one action occurrence)
  - Add the \(\text{occ}(\text{shoot}, 0)\) to \(\Pi_{\text{Turkey}}(1)\)
  - Computer answer sets of the resulting program and check for answer

- Is the gun still loaded? Same as above
- Is Jimmy walking?
- Does Jimmy have the same number of guns?
Answering Questions

- Is \textit{Matt} still alive? (after the shoot action!)
  
  \begin{itemize}
  \item \textit{length} = 1 (only one action occurrence)
  \item Add the \textit{occ}(\textit{shoot}, 0) to $\Pi_{\text{Turkey}}(1)$
  \item Computer answer sets of the resulting program and check for answer
  \end{itemize}

- Is the gun still loaded? Same as above

- Is \textit{Jimmy} walking?
  We cannot answer the question as our modeling does not consider this! How to change this?

- Does \textit{Jimmy} have the same number of guns?
  Again, our modeling does not consider this! How to change this?
Answering Questions

- Is *Matt* still alive? (after the shoot action!)
  - *length* = 1 (only one action occurrence)
  - Add the *occ*(shoot, 0) to $\Pi_{Turkey}(1)$
  - Computer answer sets of the resulting program and check for answer

- Is the gun still loaded?  Same as above

- Is *Jimmy* walking?
  Change the fluent *walking* to *walking(.)* and model it as follows.
  - individual(matt). individual(jimmy).
  - fluent(walking(l)):- individual(l).

- Does *Jimmy* have the same number of guns?
Answering Questions

- Is *Matt* still alive? (after the shoot action!)
  - \( \text{length} = 1 \) (only one action occurrence)
  - Add the \( \text{occ}(\text{shoot}, 0) \) to \( \Pi_{\text{Turkey}}(1) \)
  - Computer answer sets of the resulting program and check for answer

- Is the gun still loaded? Same as above

- Is *Jimmy* walking?
  Change the fluent \( \text{walking} \) to \( \text{walking(}.\) and model it as follows.
  \[
  \text{individual(matt). individual(jimmy).}
  \text{fluent(walking(I)):- individual(I).}
  \]

- Does *Jimmy* have the same number of guns?
  Add a fluent \( \text{number_of_guns(}.\) and represent it as follows.
  \[
  \text{number(0..10).}
  \text{fluent(number_of_guns(I)):- number(I).}
  \]
After adding a fluent

▶ specify the truth values of these fluents in the initial state
▶ make sure that they are modeled correctly in action effects

Jimmy walking?

\[
\begin{align*}
\text{individual}(\text{matt}). & \quad \text{individual}(\text{jimmy}). \\
\text{fluent}(\text{walking}(I)) & \quad :- \quad \text{individual}(I). \\
\text{holds}(\text{walking}(\text{jimmy}), 0). & \quad \text{holds}(\text{walking}(\text{matt}, 0).
\end{align*}
\]

There exists no action that affects walking ...

number_of_guns

\[
\begin{align*}
\text{number}(0..10). & \\
\text{fluent}(\text{number_of_guns}(I)) & \quad :- \quad \text{number}(I). \\
\text{holds}(\text{number_of_guns}(1), 0).
\end{align*}
\]

There exists no action that affects number_of_guns ...
It does not matter how we change the program, the current formalization will not allow us to conclude that Matt is not walking, i.e., for every answer set of the program $\Pi_{\text{turkey}}(1) \cup \{\text{occ}(\text{shoot}, 0)\}$ will contain $h(\text{walking}, 1)$. What is the reason? How to correct this?
Is Matt Walking?

It does not matter how we change the program, the current formalization will not allow us to conclude that Matt is not walking, i.e., for every answer set of the program

\[ \Pi_{\text{turkey}}(1) \cup \{\text{occ}(\text{shoot}, 0)\} \] will contain \( h(walking, 1) \).

What is the reason? there is no rule with the head -walking

How to correct this?

- Because dead implies -walking
- We add the rule
  \[ h(-\text{walking}, T) :- \text{time}(T), h(\text{dead}, T). \]
We know that if Matt is hit by the gun, he will be dead and if he is dead he cannot not continue walking. Therefore, we add the rule:

$$h(-\text{walking}, T) :- \text{time}(T), h(\text{dead}, T).$$

This rule represents a static causal law (or state constraint). It states a relationship between fluents and can cause a fluent to change its value even though the action does not directly change it. In general form, a static causal law is given by the statement of the form

$$\text{If } l_1, \ldots, l_n \text{ are true then so is } l.$$  

This static causal law is written in answer set programing by the rule:

$$h(l, T) : -\text{time}(T), h(l_1, T), \ldots, h(l_n, T).$$

Representing and reasoning with static causal laws are often called the ramification problem.
River-Crossing Problem
Three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries). The boat cannot cross the river by itself with no people on board. And, in some variations, one of the cannibals has only one arm and cannot row.

Jealous Husbands Problem
Three married couples, with the constraint that no woman can be in the presence of another man unless her husband is also present. Under this constraint, there cannot be both women and men present on a bank with women outnumbering men, since if there were, some woman would be husbandless. Therefore, upon changing men to missionaries and women to cannibals, any solution to the jealous husbands problem will also become a solution to the missionaries and cannibals problem.
% Defining the constants
\[
\begin{align*}
time(0..\text{length}). & \quad \text{number}(0..3). \\
\text{location}(l1). & \quad \text{location}(l2). \quad \% \text{two banks}
\end{align*}
\]

% X missionaries and Y cannibals are at location L
\[
\text{fluent}(\text{at}(X,Y,L)):- \text{missionary}(X),\text{cannibal}(Y),\text{location}(L).
\]

% the boat is at location L
\[
\text{fluent}(\text{boat\_at}(L)):- \text{location}(L).
\]

% X missionaries and Y cannibals move from one bank to another
\[
\text{action}(\text{cross}(I,J,L)):- \\
\quad \text{number}(I), \text{number}(J), \text{location}(L), \quad I+J \leq 2, \quad I+J > 0.
\]
% Effects of \((M,N)\) crossing from one bank to another

\[
\text{holds}(\text{at}(M+P,N+Q,L1),T+1) :\text{- time}(T), \text{number}(P), \text{number}(Q),
\]
\[
\text{action}(\text{cross}(M,N,L)), \text{location}(L1), L\neq L1,
\]
\[
\text{occ}(\text{cross}(M,N,L), T), \text{holds}(\text{at}(P,Q,L1), T).
\]

\[
\text{holds}(\text{at}(P-M,Q-N,L),T+1) :\text{- time}(T), \text{number}(P), \text{number}(Q),
\]
\[
\text{action}(\text{cross}(M,N,L)), \text{occ}(\text{cross}(M,N,L), T), \text{holds}(\text{at}(P,Q,L), T).
\]

\[
\text{holds}(\text{boat}_\text{at}(L1),T+1) :\text{- time}(T), \text{location}(L1), L\neq L1,
\]
\[
\text{action}(\text{cross}(M,N,L)), \text{occ}(\text{cross}(M,N,L), T).
\]

% executability condition

\[
\text{executable}(\text{cross}(I,J,L), T) :\text{- time}(T), \text{number}(P), \text{number}(Q),
\]
\[
\text{action}(\text{cross}(I,J,L)), \text{holds}(\text{boat}_\text{at}(L), T),
\]
\[
\text{holds}(\text{at}(P, Q, L), T), P \geq I, Q \geq J.
\]

% constraint

\[
:\text{- time}(T), \text{action}(\text{cross}(I,J,L)),
\]
\[
\text{occ}(\text{cross}(I,J,L), T), \text{not executable}(\text{cross}(I,J,L), T).
\]

\[
:\text{- number}(I), \text{number}(J), \text{location}(L),
\]
\[
\text{time}(T), \text{holds}(\text{at}(I,J,L), T), I < J, I > 0.
\]
% initial condition
holds(at(3,3,l1),0).
holds(at(0,0,l2),0).
holds(boat_at(l1),0).

% goal
:- not goal(length).
goal(T):- holds(at(3,3,l2), T).
1 { occ(A, T): action(A) } 1 :- time(T), not goal(T).

Differences from previous encoding

- We ignore the negative information (neg(.)). Why? (the negative information is implicit!)

- We add an extra constraint to represent the requirement “…for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals…” (we could have included it in the executability condition but we make use of the flexibility of logic programming)
The Second Encoding I

% Defining the constants
time(0..length).
% Three missionaries/three cannibals
number(0..3).
% Two banks of the river
location(l1). location(l2).
% X missionaries and Y cannibals are at location L
fluent(at(X,Y,L)):- number(X),number(Y),location(L).
% X missionaries and Y cannibals are in the boat
fluent(in_boat(X,Y)):- number(X),number(Y).
% the boat is at location L
fluent(boat_at(L)):- location(L).
% The boat moves from one bank to another
action(cross(L)):- location(L).
% X missionaries and Y cannibals depart the boat
action(depart(I,J)):- number(I), number(J), I+J>0.
% X missionaries and Y cannibals get on the boat
action(board(I,J)):- number(I), number(J), I+J>0.
The Second Encoding II

% Effects of departing from the boat
% increasing the number of individuals on the bank
holds(at(M+P,N+Q, L),T+1):- time(T), occ(depart(M,N), T),
    holds(boat_at(L), T), holds(at(P,Q,L), T),
    holds(in_boat(X,Y), T), M ≤ X, N ≤ Y.
% reducing the number in the boat
holds(in_boat(P,Q),T+1):- time(T), occ(depart(M,N), T),
    holds(boat_at(L), T), fluent(in_boat(P,Q)),
    holds(in_boat(M+P,N+Q), T).
% does not change the number of individuals on the other bank
holds(at(P,Q,L1),T+1):- time(T), occ(depart(M,N), T),
    holds(boat_at(L), T), holds(at(P,Q,L1), T),
    location(L1), L1!=L.
% does not change the location of the boat
holds(boat_at(L), T+1):- time(T), location(L),
    holds(boat_at(L), T), action(depart(M,N)),
    occ(depart(M,N), T).
The Second Encoding III

% boarding to the boat

% reducing the number of individuals on the bank
holds(at(M-P,N-Q,L),T+1):- time(T), occ(board(P,Q), T),
    holds(boat_at(L), T), holds(at(M,N,L), T),
    M >= P, N >= Q.

% increasing the number of individuals in the boat
holds(in_boat(M+P,N+Q),T+1):- time(T), occ(board(P,Q), T),
    holds(boat_at(L), T), holds(in_boat(M,N), T).

% does not change the number of individuals on the other bank
holds(at(M,N,L1),T+1):- time(T), occ(board(P,Q), T),
    holds(boat_at(L), T), holds(at(M,N,L1), T),
    location(L1), L1!=L.

% does not change the location of the boat
holds(boat_at(L), T+1):- time(T), location(L),
    action(board(P,Q)), holds(boat_at(L), T),
    occ(board(P,Q), T).
The Second Encoding IV

% boat crossing
% change the location of the boat
holds(boat_at(L1), T+1):- time(T), location(L1), L!=[L1],
    occ(cross(L), T).
% does not change the number of individuals in the boat
holds(in_boat(M,N), T+1):- time(T), holds(in_boat(M,N), T),
    occ(cross(L), T).
% does not change the number of individuals on the banks
holds(at(M,N,L1), T+1):- time(T), occ(cross(L), T),
    holds(at(M,N,L1), T), location(L1).
The Second Encoding V

% executability condition

executable(cross(L), T):-
    time(T),
    action(cross(L)),
    holds(boat_at(L), T).

executable(board(M,N), T):-
    time(T),
    action(board(M,N)),
    holds(boat_at(L), T),
    holds(in_boat(P,Q), T),
    holds(at(X,Y,L), T),
    X >= M, Y >= N, M+N > 0, M+N+P+Q <= 2.

executable(depart(M,N), T):-
    time(T),
    action(depart(M,N)),
    holds(in_boat(P,Q), T),
    M <= P, N <= Q, M+N > 0.
The Second Encoding VI

% initial condition
holds(at(3,3,l1),0).
holds(at(0,0,l2),0).
holds(boat_at(l1),0).
holds(in_boat(0,0),0).

% constraint
:- time(T), action(A), occ(A, T), not executable(A, T).
:- number(I), number(J), location(L),
   time(T), holds(at(I,J,L), T), I<J, I>0.

% goal
:- not goal(length).
goal(T):- holds(at(3,3,l2), T).
1 { occ(A, T): action(A) } 1 :- time(T), T < length.

This program has the shortest plan with 28 actions. clingo takes more than 1 hour and cannot even verify it!
The right level of abstraction: one type of actions \((cross(I, J, L))\) vs. three types \((cross(L), board(X, Y), depart(X, Y))\). The second provides more details. However, the program would not even run!

Trade off: details vs. computability!

Use the right constructs and vocabularies: When we have one type of actions \((cross(X, Y, L))\), we need to encode all information (who is crossing and from which bank), we could have used \(cross(X, Y, L1, L2)\) to denote the direction of movement of the boat. However, the second representation is not necessary in this problem. It will be necessary to introduce the second parameter if the setting changes. For example, the river is replaced by a triangle pond!
A is a high-level action language for representing and reasoning about actions and change. It has a simple and independent semantics based on transition system. It is introduced in


Several extensions of $A$ have been proposed. We will use $AL$, which is introduced in

In $\mathcal{AL}$, an action theory is defined over two disjoint sets, a set of fluents (a fluent is a property whose value changes over time) and a set of actions, and is a set of propositional propositions of the form

\[
\begin{align*}
\text{a causes } f & \text{ if } p_1, \ldots, p_n \\
\text{f if } p_1, \ldots, p_n & \\
\text{initially } f & \\
\text{a executable if } p_1 \ldots, p_n
\end{align*}
\]

(6), referred as dynamic law, represents the (conditional) effect of action $a$. (7) is a static law which represents the relationship between fluents. Propositions of the form (8), also called $\nu$-propositions, are used to describe the initial situation. An action theory is given by a pair $(D, I)$ where $D$ consists of propositions of the form (6)-(8) and $I$ consists of propositions of the form (8). $D$ and $I$ will be called the domain description and initial state, respectively. We assume that for each fluent $f$, initially $f$ or initially $\neg f$ belongs to $I$ but not both.
To say that initially, the turkey is walking and not dead, we write
\[
\text{initially } \neg \text{dead and} \\
\text{initially } \text{walking}
\]

Initially, the gun is loaded:
\[
\text{initially } \text{loaded}
\]

Shooting causes the turkey to be dead if the gun is loaded can be expressed by
\[
\text{shoot causes dead if } \text{loaded and} \\
\text{shoot causes } \neg \text{loaded if } \text{loaded}
\]

Un/Loading the gun causes the gun to be un/loaded
\[
\text{load causes leaded and} \\
\text{unload causes } \neg \text{loaded}
\]

Dead turkeys cannot walk
\[
\neg \text{walking if dead}
\]

A gun can be loaded only when it is not loaded
\[
\text{load executable if } \neg \text{loaded}
\]
So, the action theory is

\[ I_y = \{ \text{initially } \neg \text{dead}, \text{initially } \text{walking}, \text{initially } \text{loaded} \} \]

and

\[ D_y = \{ \text{shoot causes dead if } \text{loaded} \\
\text{shoot causes } \neg \text{loaded if } \text{loaded} \\
\text{load causes leaded} \\
\neg \text{walking if } \text{dead} \\
\text{shoot executable if } \text{true} \\
\text{load executable if } \neg \text{loaded} \} \]
Why High Level Action Languages?

Each action theory can be easily translated into an ASP program. In fact,

- each law \textbf{a causes} \textbf{f} \textbf{if} \ p_1, \ldots, p_k is translated into the rule:
  \[ \text{holds}(f, T + 1) \leftarrow \text{occ}(a, T), \text{holds}(p_1, T), \ldots, \text{holds}(p_k, T). \]

- each law \textbf{f} \textbf{if} \ p_1, \ldots, p_k is translated into the rule:
  \[ \text{holds}(f, T) \leftarrow \text{holds}(p_1, T), \ldots, \text{holds}(p_k, T). \]

- each law \textbf{a executable if} \ p_1, \ldots, p_k is translated into the rules:
  \[ \text{executable}(a, T) \leftarrow \text{holds}(p_1, T), \ldots, \text{holds}(p_k, T). \]

- each law \textbf{initially} \ f is translated into the rule:
  \[ \text{holds}(f, 0). \]

The other rules that we have seen are the same for all theories:

\[ \leftarrow \text{occ}(a, T), \textbf{not executable}(a, T). \]

\[ 1\{\text{occ}(a, T) : \text{action}(A)\}1. \]
Consider the following narrative

- **9am**: John arrived at work and turned the light on. As usual, the light went on and John started his daily work.
- **12 pm**: John turned off the light and went to lunch.
- **1pm**: John turned on the light when he got back from lunch. The light did not go on.
- **1:15pm**: The company’s electrician arrived. He replaced the bulb and turned on the light, the light did not go on. He then checked the fuses and replaced one which was blown. The light is back.
The narrative can be described by a triple \((SD, COMPS, OBS)\) where

- **SD** is an action theory describing actions (e.g. \(\text{turn\_on}, \text{turn\_off}, \text{replace\_bulb}, \text{etc.}\)) and their expected outcomes and relationships between fluents. It includes also actions that are beyond the control of the agents such as \(\text{break}(\text{bulb})\) causes the bulb to be broke.

- **COMP** is a set of objects that can be broken (\(\text{bulb}, \text{fuse}, \text{...}\)); for each object \(o\), an action of the form \(\text{break}(o)\) with the outcome \(ab(o)\), indicating that \(o\) is broken, is included in \(SD\).

- **OBS** is a set of observations that describes the history of the world (might be incomplete) in term of which actions were executed, when they were executed relative to each other, and what are the outcomes of the actions;
The Narrative as a System

We illustrate the concepts using the example. Let $Sys = (SD, \{\text{bulb}\}, OBS)$ be a system with

$$SD = \begin{cases} 
(r1) & \text{turn}_\text{on} \text{ causes light}_\text{on if } \neg ab(\text{bulb}) \\
(r2) & \text{turn}_\text{off} \text{ causes } \neg \text{light}_\text{on} \\
(r3) & \neg \text{light}_\text{on if } \text{ab}(\text{bulb}) \\
(r4) & \text{break}(\text{bulb}) \text{ causes } \text{ab}(\text{bulb}) 
\end{cases}$$

and

$$OBS = \begin{cases} 
(o1) & \text{turn}_\text{on occurs at} \ s_0 \\
(o2) & \text{turn}_\text{off occurs at} \ s_1 \\
(o3) & \text{turn}_\text{on between} \ s_2, s_3 \\
(o4) & s_0 \text{ precedes } s_1 \\
(o5) & s_1 \text{ precedes } s_2 \\
(o6) & s_2 \text{ precedes } s_3 \\
(o7) & \neg \text{light}_\text{on at} \ s_0 \\
(o8) & \text{light}_\text{on at} \ s_1 \\
(o9) & \neg \text{light}_\text{on at} \ s_2 \\
(o10) & \neg \text{light}_\text{on at} \ s_3 
\end{cases}$$
Diagnostic Reasoning Process

Address the questions

▶ when does a system need a diagnosis?

Answer: When there are inconsistency between observations and expected outcomes; or when the system does not have a model if we remove all actions $break(o)$ from $SD$.

▶ what are the diagnoses?

Answer: Additional action occurrences that help explain the discrepancies between observations and expected outcomes.

▶ how to fix a system that needs a diagnosis?

Answer: Collecting enough information and executing test actions if necessary so that a diagnosis is singed out. Thereafter, executing the repair actions necessary to fix the system.
Diagnostic Reasoning Process – A Summary

Answer the question: when a system needs a diagnosis and what are the diagnoses.

▶ generating candidate diagnoses based on an incomplete history of events that have occurred and observations that have been made.

▶ in the event of multiple candidate diagnoses, performing actions to enable observations that will discriminate candidate diagnoses. The selection of a particular action is often biased towards confirming the most likely diagnosis, or the one that is easiest to test.

▶ generating (possibly with conditional) plans, comprising both world-altering actions and sensing actions, to discriminate candidate diagnoses.

▶ updating the space of diagnoses in the face of changes in the state of the world, and in the face of new observations.
The Narrative as a Logic Program – General Idea

Given a system description \((SD, COMPS, OBS)\) with

- \(SD\): a set of propositions of the form \(a \text{ causes } f \text{ if } p_1, \ldots, p_n\) or \(f \text{ if } p_1, \ldots, p_n\); this set of propositions describes the normal system behavior;
- \(COMPS\): a set of components that can be broken
- \(OBS\): a set of observations representing a narrative of the system.

We will write a logic program \(\Pi(SD, COMPS, OBS)\) (or \(\Pi\) for short) to compute diagnoses. \(\Pi\) will need to contain the following parts:

- rules for generating a sequence of actions
- rules for checking if the generated sequence of actions is a possible diagnosis; this includes
  - rules that assign situations to time moments – this assignment must respect the ordering between situations in the observations
  - rules that make sure that observations are satisfied.
The narrative happens over a course of time (0, 1, 2, ....).

► Our first task is to identify the time that the events happened. We can do so by assigning a time step for each situation. The assignment must satisfy the ordering of the situations. Some assignments are good and some are not good.
The narrative happens over a course of time (0, 1, 2, ....).

- Our first task is to identify the time that the events happened. We can do so by assigning a time step for each situation. The assignment must satisfy the ordering of the situations. Some assignments are good and some are not good.
The narrative happens over a course of time (0, 1, 2, ....).

Our first task is to identify the time that the events happened. We can do so by assigning a time step for each situation. The assignment must satisfy the ordering of the situations. Some assignments are good and some are not good.
The narrative happens over a course of time (0, 1, 2, ....).

- Our first task is to identify the time that the events happened. We can do so by assigning a time step for each situation. The assignment must satisfy the ordering of the situations. Some assignments are good and some are not good.

- Assume that the assignment is good. We then need to generate action occurrences and ensure that observations are true.
Elements of Π I

% Constants: time, situations, actions, fluents
time(0..length).
% situations
sit(s0). sit(s1). sit(s2). sit(s3).
% actions
action(turn_on). action(turn_off). action(break(bulb)).
% fluents
fluent(light_on). fluent(ab(bulb)).
% Effects of actions (similar to what has been done in the part on reasoning about actions/planning)
holds(light_on, T+1):- time(T), occ(turn_on, T), holds(neg(ab(bulb)), T).
holds(neg(light_on), T+1):- time(T), occ(turn_off, T).
holds(ab(bulb), T+1):- time(T), occ(break(bulb), T).
holds(neg(light_on), T):- time(T), holds(ab(bulb), T).
% inertial axiom
holds(F, T+1):- time(T), fluent(F), holds(F, T), not holds(neg(F), T+1).
holds(neg(F), T+1):- time(T), fluent(F), holds(neg(F), T), not holds(F, T+1)
Elements of $\Pi$ II

% specifying the set of observations situation order
prec(s0,s1). prec(s1,s2). prec(s2,s3).

% fluent observations
at(neg(light_on), s0). at(neg(ab(bulb)), s0). at(light_on, s1).
at(neg(light_on), s2). at(neg(light_on), s3).

% action observations
between(s0,s1,turn_on). between(s2,s3,turn_on).
exec_at(s1,turn_off).

% satisfying all the observations
% generating situation order each situation happens at a time moment
1 {happens(S, T): time(T) } 1 :- sit(S).
% s0 is always at the time moment 0
:- time(T), happens(s0,T), T > 0.
:- time(T), happens(s3,T), T < length.
:- time(T1), time(T2), sit(S1), sit(S2), happens(S1,T1),
   happens(S2,T2), prec(S1,S2), T1 > T2.
:- time(T1), time(T2), sit(S1), sit(S2), action(A),
   happens(S1,T1), happens(S2,T2), between(S1,S2,A), T2 != T1 + 1.
% generating action occurrences
1{occ(A, T): action(A) } 1 :- time(T), T < length.
% satisfying all the observations
% fluent observations
holds(F, T):- time(T), fluent(F), sit(S), at(F, S), happens(S, T).
holds(neg(F), T):- time(T), fluent(F), sit(S), at(neg(F), S), happens(S, T).
% constraints, making sure that action occurrences happen as they are observed
:- time(T1), time(T2), action(A), action(A1), between(S1, S2, A),
happens(S1, T1), occ(A1, T1), A1 != A.
occ(A, T1):- time(T1), action(A), between(S1, S2, A), happens(S1, T1).
% constraints that eliminate inconsistency model
:- time(T), fluent(F), holds(F, T), holds(neg(F), T).
Running the program

clingo -c length=5 bulb.lp
that gives an answer set with the following:
occ(turn_on,0)  occ(turn_off,1)  occ(turn_on,2)
oc(break(bulb),3)  occ(turn_on,4)
happens(s3,5)   happens(s2,4)   happens(s1,1)   happens(s0,0)
Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!

Reason

It is the inertial axiom!

What is it? The inertial law says that an object at rest stays at rest and an object in motion stays in motion with the same speed and in the same direction unless acted upon by an unbalanced force.
Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!
- I add the action, I specify the effects, my program does not work still!

Reason

It is the inertial axiom!

What is it? The inertial law says that object at rest stays at rest. Translated to reasoning about effects of actions and changes: what is true/false stays true/false if it has not been changed by the action execution.
Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!
- I add the action, I specify the effects, my program does not work still!
- I did ...., my program does not work!
Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!
- I add the action, I specify the effects, my program does not work still!
- I did ...., my program does not work!

Reason

It is the inertial axiom!
Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!
- I add the action, I specify the effects, my program does not work still!
- I did ...., my program does not work!

Reason

It is the inertial axiom!
Missionaries and Cannibals, Blocks, etc.

Why?

- My program does not work!
- I add the action, I specify the effects, my program does not work still!
- I did ..., my program does not work!

Reason

It is the inertial axiom!
Missionaries and Cannibals, Blocks, etc.

Why?

▶ My program does not work!
▶ I add the action, I specify the effects, my program does not work still!
▶ I did ..., my program does not work!

Reason

It is the inertial axiom!

It is the inertial axiom!

It is the inertial axiom!

What is it? The inertial law says that object at rest stays at rest. Translated to reasoning about effects of actions and changes: what is true/false stays true/false if it has not been changed by the action execution.
Different ways to do it

- See [Formalization in the block world domain](#)
  
  holds(F, T+1) :- fluent(F), holds(F, T), not holds(neg(F), T+1).
  holds(neg(F),T+1):-fluent(F),holds(neg(F),T),not holds(F,T+1).

- See [Formalization in the Yale shooting problem](#)
  
  Same as in the block world domain.

- The formalization in the missionaries and cannibals is different. See [MC problem](#)
  
  It seems that we forgot inertial rules! NO, we just did it in a different way.
Frame Axioms in MC Problem 1

We have fluents: at(M,N,L), boat_at(L). 0 ≤ M, N ≤ 3. They can be divided into three types: at(M,N,l1), at(M,N,l2), boat_at(L) (color coded: red, blue, black)

If you display the fluents from the program you will see the following fluents:

fluent(at(0,0,l1)) fluent(at(1,0,l1)) fluent(at(2,0,l1)) fluent(at(3,0,l1))
fluent(at(0,1,l1)) fluent(at(1,1,l1)) fluent(at(2,1,l1)) fluent(at(3,1,l1))
fluent(at(0,2,l1)) fluent(at(1,2,l1)) fluent(at(2,2,l1)) fluent(at(3,2,l1))
fluent(at(0,3,l1)) fluent(at(1,3,l1)) fluent(at(2,3,l1)) fluent(at(3,3,l1))
fluent(at(0,0,l2)) fluent(at(1,0,l2)) fluent(at(2,0,l2)) fluent(at(3,0,l2))
fluent(at(0,1,l2)) fluent(at(1,1,l2)) fluent(at(2,1,l2)) fluent(at(3,1,l2))
fluent(at(0,2,l2)) fluent(at(1,2,l2)) fluent(at(2,2,l2)) fluent(at(3,2,l2))
fluent(at(0,3,l2)) fluent(at(1,3,l2)) fluent(at(2,3,l2)) fluent(at(3,3,l2))
fluent(boat_at(l1)) fluent(boat_at(l2))
Frame Axioms in MC Problem II

For each action $\text{cross}(I, J, L)$ we have one rule that changes the value of

$\text{holds}($at$(I, J, L), T+1) :\text{... occ(cross(P,Q,L),T), ...}$

$\text{holds}($at$(I, J, L1), T+1) :\text{... occ(cross(P,Q,L),T), } L1!\equiv L,$...

$\text{holds(boat at(L1), T+1) :... occ(cross(P,Q,L),T), } L1!\equiv L,$...

The three rules guarantee that there will be one fluent of each type that is true in the next time step ($T+1$) if the action is executable. Since we represent negative information implicitly (there is no $\text{neg(.)}$ in the program), we account for all the fluents in the problem ($\text{holds}($at$(.,.,l), t)$) records what is true; the missing fluents are false).

For example, when we add $\text{occ(cross(1,1,l1),0)}$ to the program and display atoms of the form $\text{holds(.,.)}$ we can see the following:

$\text{holds(at(3,3,l1),0)}$  $\text{holds(at(0,0,l2),0)}$  $\text{holds(boat at(l1),0)}$

$\text{holds(at(2,2,l1),1)}$  $\text{holds(at(1,1,l2),1)}$  $\text{holds(boat at(l2),1)}$
Frame Axioms in MC Problem III

Why a new way?

We could represent the negative information to the MC program by the rule (Closed World Assumption):

\[ \text{holds}(\neg(F), 0) :- \text{fluent}(F), \text{not holds}(F, 0). \]

and the inertial rules

\[ \text{holds}(F, T+1) :- \text{fluent}(F), \text{holds}(F, T), \text{not holds}(\neg(F), T+1). \]

\[ \text{holds}(\neg(F), T+1) :- \text{fluent}(F), \text{holds}(\neg(F), T), \text{not holds}(F, T+1). \]

then we get the following results after the action \( \text{cross}(1, 1, l1) \) occurs, i.e., \( \text{occ}(\text{cross}(1, 1, l1), 0) \) is added to the program (red for positive information and black for negative information, which are not good!):

\[ \text{holds}(\text{at}(3, 3, l1), 0) \]
\[ \text{holds}(\text{at}(0, 0, l2), 0) \]
\[ \text{holds}(\text{boat} \text{at}(l1), 0) \]
\[ \text{holds}(\text{boat} \text{at}(l2), 1) \]
\[ \text{holds}(\text{at}(2, 2, l1), 1) \]
\[ \text{holds}(\text{at}(1, 1, l2), 1) \]
\[ \text{holds}(\text{at}(3, 3, l1), 1) \]
\[ \text{holds}(\text{at}(0, 0, l2), 1) \]
\[ \text{holds}(\text{boat} \text{at}(l2), 2) \]
\[ \text{holds}(\text{boat} \text{at}(l1), 2) \]
\[ \text{holds}(\text{at}(2, 2, l1), 2) \]
\[ \text{holds}(\text{at}(1, 1, l2), 2) \]
\[ \text{holds}(\text{at}(3, 3, l1), 2) \]
\[ \text{holds}(\text{at}(0, 0, l2), 2) \]
\[ \text{holds}(\neg(\text{at}(0, 0, l1)), 0) \]
\[ \text{holds}(\neg(\text{at}(1, 0, l1)), 0) \]
\[ \text{holds}(\neg(\text{at}(2, 0, l1)), 0) \]
Frame Axioms in MC Problem IV

\[
\begin{align*}
\text{holds}(\neg(\text{at}(3,0,l1)),0) & \quad \text{holds}(\neg(\text{at}(0,1,l1)),0) & \quad \text{holds}(\neg(\text{at}(1,1,l1)),0) \\
\text{holds}(\neg(\text{at}(2,1,l1)),0) & \quad \text{holds}(\neg(\text{at}(3,1,l1)),0) & \quad \text{holds}(\neg(\text{at}(0,2,l1)),0) \\
\text{holds}(\neg(\text{at}(1,2,l1)),0) & \quad \text{holds}(\neg(\text{at}(2,2,l1)),0) & \quad \text{holds}(\neg(\text{at}(3,2,l1)),0) \\
\text{holds}(\neg(\text{at}(0,3,l1)),0) & \quad \text{holds}(\neg(\text{at}(1,3,l1)),0) & \quad \text{holds}(\neg(\text{at}(2,3,l1)),0) \\
\text{holds}(\neg(\text{at}(1,0,l2)),0) & \quad \text{holds}(\neg(\text{at}(2,0,l2)),0) & \quad \text{holds}(\neg(\text{at}(3,0,l2)),0) \\
\text{holds}(\neg(\text{at}(0,1,l2)),0) & \quad \text{holds}(\neg(\text{at}(1,1,l2)),0) & \quad \text{holds}(\neg(\text{at}(2,1,l2)),0) \\
\text{holds}(\neg(\text{at}(3,1,l2)),0) & \quad \text{holds}(\neg(\text{at}(0,2,l2)),0) & \quad \text{holds}(\neg(\text{at}(1,2,l2)),0) \\
\text{holds}(\neg(\text{at}(2,2,l2)),0) & \quad \text{holds}(\neg(\text{at}(3,2,l2)),0) & \quad \text{holds}(\neg(\text{at}(0,3,l2)),0) \\
\text{holds}(\neg(\text{at}(1,3,l2)),0) & \quad \text{holds}(\neg(\text{at}(2,3,l2)),0) & \quad \text{holds}(\neg(\text{at}(3,3,l2)),0) \\
\text{holds}(\neg(\text{boat}_\text{at}(l2)),0) & \quad \text{holds}(\neg(\text{at}(0,0,l1)),1) & \\
\text{holds}(\neg(\text{at}(1,0,l1)),1) & \quad \text{holds}(\neg(\text{at}(2,0,l1)),1) & \quad \text{holds}(\neg(\text{at}(3,0,l1)),1) \\
\text{holds}(\neg(\text{at}(0,1,l1)),1) & \quad \text{holds}(\neg(\text{at}(1,1,l1)),1) & \quad \text{holds}(\neg(\text{at}(2,1,l1)),1) \\
\text{holds}(\neg(\text{at}(3,1,l1)),1) & \quad \text{holds}(\neg(\text{at}(0,2,l1)),1) & \quad \text{holds}(\neg(\text{at}(1,2,l1)),1) \\
\text{holds}(\neg(\text{at}(2,2,l1)),1) & \quad \text{holds}(\neg(\text{at}(3,2,l1)),1) & \quad \text{holds}(\neg(\text{at}(0,3,l1)),1) \\
\text{holds}(\neg(\text{at}(1,3,l1)),1) & \quad \text{holds}(\neg(\text{at}(2,3,l1)),1) & \quad \text{holds}(\neg(\text{at}(3,3,l1)),1) \\
\text{holds}(\neg(\text{at}(1,0,l2)),1) & \quad \text{holds}(\neg(\text{at}(2,0,l2)),1) & \quad \text{holds}(\neg(\text{at}(3,0,l2)),1) \\
\text{holds}(\neg(\text{at}(0,1,l2)),1) & \quad \text{holds}(\neg(\text{at}(1,1,l2)),1) & \quad \text{holds}(\neg(\text{at}(2,1,l2)),1) \\
\text{holds}(\neg(\text{at}(3,1,l2)),1) & \quad \text{holds}(\neg(\text{at}(0,2,l2)),1) & \quad \text{holds}(\neg(\text{at}(1,2,l2)),1)
\end{align*}
\]
Frame Axioms in MC Problem V

holds(neg(at(2,2,l2)),1) holds(neg(at(3,2,l2)),1) holds(neg(at(0,3,l2)),1)
holds(neg(at(1,3,l2)),1) holds(neg(at(2,3,l2)),1) holds(neg(at(3,3,l2)),1)
holds(neg(at(0,0,l1)),2) holds(neg(at(1,0,l1)),2) holds(neg(at(2,0,l1)),2)
holds(neg(at(3,0,l1)),2) holds(neg(at(0,1,l1)),2) holds(neg(at(1,1,l1)),2)
holds(neg(at(2,1,l1)),2) holds(neg(at(3,1,l1)),2) holds(neg(at(0,2,l1)),2)
holds(neg(at(1,2,l1)),2) holds(neg(at(3,2,l1)),2) holds(neg(at(0,3,l1)),2)
holds(neg(at(1,3,l1)),2) holds(neg(at(2,3,l1)),2) holds(neg(at(3,3,l1)),2)
holds(neg(at(2,1,l2)),2) holds(neg(at(3,1,l2)),2) holds(neg(at(0,2,l2)),2)
holds(neg(at(1,2,l2)),2) holds(neg(at(3,2,l2)),2) holds(neg(at(0,3,l2)),2)
holds(neg(at(1,3,l2)),2) holds(neg(at(2,3,l2)),2) holds(neg(at(3,3,l2)),2)
Frame Axioms in MC Problem VI

The reason for the wrong results lies in that we do not have rules for negative effects. For example, if we add
holds(neg(boat_at(L)), T+1):- time(T),
    action(cross(M,N,L)), occ(cross(M,N,L), T).
Then we solve the problem with holds(boat_at(l2), 1) holds(boat_at(l1), 1).

How do we solve the problem with holds(at(3,3,l1), 0) holds(at(3,3,l1), 1)?
holds(neg(at(P,Q,L)), T+1):- time(T), action(cross(M,N,L)),
    occ(cross(M,N,L), T), holds(at(P,Q,L), T).
How do we solve the problem with holds(at(1,1,l2), 1) holds(at(0,0,l2), 1)?
holds(neg(at(P,Q,L1)), T+1):- time(T), action(cross(M,N,L)),
    occ(cross(M,N,L), T), L!=L1, holds(at(P,Q,L1), T).

We get the following results:
holds(at(3,3,l1), 0) holds(at(0,0,l2), 0) holds(boat_at(l1), 0)
holds(boat_at(l2), 1) holds(at(2,2,l1), 1) holds(at(1,1,l2), 1)
holds(at(2,2,l1), 2) holds(at(1,1,l2), 2) holds(neg(boat_at(l1)), 1)
holds(neg(at(0,0,l1)), 0) holds(neg(at(1,0,l1)), 0) holds(neg(at(2,0,l1)), 0)
holds(neg(at(3,0,l1)), 0) holds(neg(at(0,1,l1)), 0) holds(neg(at(1,1,l1)), 0)
holds(neg(at(2,1,l1)), 0) holds(neg(at(3,1,l1)), 0) holds(neg(at(0,2,l1)), 0)
holds(neg(at(2,2,l1)), 0) holds(neg(at(2,2,l1)), 0) holds(neg(at(3,2,l1)), 0)
Frame Axioms in MC Problem VII

holds(neg(at(0,3,l1)),0) holds(neg(at(1,3,l1)),0) holds(neg(at(2,3,l1)),0)
holds(neg(at(1,0,l2)),0) holds(neg(at(2,0,l2)),0) holds(neg(at(3,0,l2)),0)
holds(neg(at(0,1,l2)),0) holds(neg(at(1,1,l2)),0) holds(neg(at(2,1,l2)),0)
holds(neg(at(3,1,l2)),0) holds(neg(at(0,2,l2)),0) holds(neg(at(1,2,l2)),0)
holds(neg(at(2,2,l2)),0) holds(neg(at(3,2,l2)),0) holds(neg(at(0,3,l2)),0)
holds(neg(at(1,3,l2)),0) holds(neg(at(2,3,l2)),0) holds(neg(at(3,3,l2)),0)
holds(neg(boat_at(l2)),0) holds(neg(boat_at(l1)),2) holds(boat_at(l2),2)
holds(neg(at(0,0,l1)),1) holds(neg(at(1,0,l1)),1) holds(neg(at(2,0,l1)),1)
holds(neg(at(3,0,l1)),1) holds(neg(at(0,1,l1)),1) holds(neg(at(1,1,l1)),1)
holds(neg(at(2,1,l1)),1) holds(neg(at(3,1,l1)),1) holds(neg(at(0,2,l1)),1)
holds(neg(at(1,2,l1)),1) holds(neg(at(2,2,l1)),1) holds(neg(at(3,2,l1)),1)
holds(neg(at(0,3,l1)),1) holds(neg(at(1,3,l1)),1) holds(neg(at(2,3,l1)),1)
holds(neg(at(3,3,l1)),1) holds(neg(at(0,0,l2)),1) holds(neg(at(1,0,l2)),1)
holds(neg(at(2,0,l2)),1) holds(neg(at(3,0,l2)),1) holds(neg(at(0,1,l2)),1)
holds(neg(at(2,1,l2)),1) holds(neg(at(3,1,l2)),1) holds(neg(at(0,2,l2)),1)
holds(neg(at(1,2,l2)),1) holds(neg(at(2,2,l2)),1) holds(neg(at(3,2,l2)),1)
holds(neg(at(0,3,l2)),1) holds(neg(at(1,3,l2)),1) holds(neg(at(2,3,l2)),1)
holds(neg(at(3,3,l2)),1) holds(neg(at(0,0,l1)),2) holds(neg(at(1,0,l1)),2)
holds(neg(at(2,0,l1)),2) holds(neg(at(3,0,l1)),2) holds(neg(at(0,1,l1)),2)
holds(neg(at(1,1,l1)),2) holds(neg(at(3,1,l1)),2) holds(neg(at(0,2,l1)),2)
Frame Axioms in MC Problem VIII

holds(neg(at(2,1,l1)),2) holds(neg(at(3,1,l1)),2) holds(neg(at(0,2,l1)),2)
holds(neg(at(1,2,l1)),2) holds(neg(at(3,2,l1)),2) holds(neg(at(0,3,l1)),2)
holds(neg(at(1,3,l1)),2) holds(neg(at(2,3,l1)),2) holds(neg(at(1,0,l2)),2)
holds(neg(at(2,0,l2)),2) holds(neg(at(3,0,l2)),2) holds(neg(at(0,1,l2)),2)
holds(neg(at(2,1,l2)),2) holds(neg(at(3,1,l2)),2) holds(neg(at(0,2,l2)),2)
holds(neg(at(1,2,l2)),2) holds(neg(at(2,2,l2)),2) holds(neg(at(3,2,l2)),2)
holds(neg(at(0,3,l2)),2) holds(neg(at(1,3,l2)),2) holds(neg(at(2,3,l2)),2)
holds(neg(at(3,3,l2)),2) holds(neg(at(3,3,l1)),2) holds(neg(at(0,0,l2)),2)

holds(neg(at(2,1,l1)),2) holds(neg(at(3,1,l1)),2) holds(neg(at(0,2,l1)),2)
holds(neg(at(1,2,l1)),2) holds(neg(at(3,2,l1)),2) holds(neg(at(0,3,l1)),2)
holds(neg(at(1,3,l1)),2) holds(neg(at(2,3,l1)),2) holds(neg(at(1,0,l2)),2)
holds(neg(at(2,0,l2)),2) holds(neg(at(3,0,l2)),2) holds(neg(at(0,1,l2)),2)
holds(neg(at(2,1,l2)),2) holds(neg(at(3,1,l2)),2) holds(neg(at(0,2,l2)),2)
holds(neg(at(1,2,l2)),2) holds(neg(at(2,2,l2)),2) holds(neg(at(3,2,l2)),2)
holds(neg(at(0,3,l2)),2) holds(neg(at(1,3,l2)),2) holds(neg(at(2,3,l2)),2)
holds(neg(at(3,3,l2)),2) holds(neg(at(3,3,l1)),2) holds(neg(at(0,0,l2)),2)
Frame Axioms in MC Problem IX

Two views of the same coin

holds(at(3,3,l1),0)
holds(at(0,0,l2),0)
holds(boat_at(l1),0)

only what is true is inside!

holds(neg(boat_at(l2)),0)
holds(at(0,0,l2),0)
holds(neg(at(1,1,l2)),0)
holds(boat_at(l1),0)

... 

everything is inside
Adding Actions and Programs Stop Working I

Problem
I added an action that allows for the boat to move to an island. The program stops working.

The changes you might have made:
number(0..4). % increase number of MC
location(island). % add the island
action(cross(I,J,L,L1)):- number(I), number(J), I+J ≤ 2, I+J > 0,
   location(L), L!=L1, location(L1). % change actions
... replace cross(I,J,L) in the old program to cross(I,J,L,L1) ...
holds(at(0,0,island),0). % specify number of MC on island
goal(T):- holds(at(4,4,l2), T). % make the goal
Adding Actions and Programs Stop Working II

Why so?

Remove the goal and adding \texttt{occ(cross(1,1,l1,l2),0)} to the program, run with \textit{length} = 1, you will see
\begin{align*}
\text{holds(at(4,4,l1),0)} & \quad \text{holds(at(0,0,l2),0)} \quad \text{holds(boat\_at(l1),0)} \\
\text{holds(at(0,0,island),0)} & \quad \text{holds(at(3,3,l1),1)} \quad \text{holds(at(1,1,l2),1)} \quad \text{holds(boat\_at(l2),1)}
\end{align*}

Oh, I expect that \texttt{holds(at(0,0,island),1)} be there!

Where does it go?

In fact, if you add \texttt{occ(cross(1,1,l1,island),0)} to the program, then \texttt{holds(at(0,0,l2),1)} is expected but not present. And, that was the source of the problem: \texttt{something we expect to be true but disappear!} The reason was inertial axioms are missing. By introducing the island, we create the fourth type of fluents (analogous to our previous analysis, \texttt{at(P,Q,island)}). Our program change the value of only three types of fluents after one action is executed. The fourth type must stay the same.
Scheduling

Problem
We have several tasks $t_1, \ldots, t_n$. For every $i$, we have

- a unique atom $\text{duration}(t_i, d_i)$ that encodes the duration of the task $t_i$ (we assume that $d_i$ is a positive integer);
- a collection of atoms of the form $\text{prec}(t_i, t_j)$ which says that $t_i$ has to be completed before $t_j$ can start.
- a collection of atoms of the form $\text{non_overlap}(t_i, t_j)$ which says that $t_i$ and $t_j$ cannot be overlapped.

Goal: find a schedule to complete the $t_1, \ldots, t_n$ with minimal span (total time).

duration(t_1, 3)  
duration(t_2, 3)  
duration(t_3, 2)

$t_1, t_2$: OVERLAPPING  
t_1 \text{ BEFORE } t_3
A Schedule for a Set of Tasks: Definition

A schedule for the set of tasks \( T = \{t_1, \ldots, t_n\} \) is a mapping of \( T \) to the set of non-negative integers \( \mathbb{N} \), denoted by \( \text{start} : T \rightarrow \mathbb{N} \), such that

- if \( \text{prec}(t_i, t_j) \) is true then \( \text{start}(t_i) + d_i \leq \text{start}(t_j) \) (\( t_i \) completed before \( t_j \))

- if \( \text{non\_overlap}(t_i, t_j) \) is true then \( \text{start}(t_i) + d_i \leq \text{start}(t_j) \) or \( \text{start}(t_j) + d_j \leq \text{start}(t_i) \)

Given three tasks \( t_1, t_2, t_3 \) with \( \text{duration}(t_1, 3), \text{duration}(t_2, 3), \text{duration}(t_3, 2) \), and the constraints \( \text{prec}(t_1, t_3), \text{non\_overlap}(t_1, t_2) \) then the assignment represents in the top half of the figure is not a schedule for the set of tasks \( \{t_1, t_2, t_3\} \);
A Schedule for a Set of Tasks: Definition

A schedule for the set of tasks $T = \{t_1, \ldots, t_n\}$ is a mapping of $T$ to the set of non-negative integers $\mathbb{N}$, denoted by $\text{start} : T \rightarrow \mathbb{N}$, such that

- if $\text{prec}(t_i, t_j)$ is true then $\text{start}(t_i) + d_i \leq \text{start}(t_j)$ ($t_i$ completed before $t_j$)

- if $\text{non\_overlap}(t_i, t_j)$ is true then $\text{start}(t_i) + d_i \leq \text{start}(t_j)$ or $\text{start}(t_j) + d_j \leq \text{start}(t_i)$

Given three tasks $t_1, t_2, t_3$ with $\text{duration}(t_1, 3)$, $\text{duration}(t_2, 3)$, $\text{duration}(t_3, 2)$, and the constraints $\text{prec}(t_1, t_3)$, $\text{non\_overlap}(t_1, t_2)$ then the assignment represents in the top half of the figure is not a schedule for the set of tasks $\{t_1, t_2, t_3\}$; the assignment represents in the bottom half is.
Span of a Schedule: Definition

A schedule for the set of tasks $T = \{t_1, \ldots, t_n\}$ is a mapping of $T$ to the set of non-negative integers $\mathbb{N}$, denoted by $\text{start} : T \rightarrow \mathbb{N}$, such that

- if $\text{prec}(t_i, t_j)$ is true then $\text{start}(t_i) + d_i \leq \text{start}(t_j)$ ($t_i$ completed before $t_j$)

- if $\text{non_overlap}(t_i, t_j)$ is true then $\text{start}(t_i) + d_i \leq \text{start}(t_j)$ or $\text{start}(t_j) + d_j \leq \text{start}(t_i)$

The span of a schedule is defined by the formula $\text{span} = \max_{\text{end}} - \min_{\text{start}}$ where

- $\max_{\text{end}} = \max\{\text{start}(t_i) + d_i \mid i = 1, \ldots, n\}$ and
- $\min_{\text{start}} = \min\{\text{start}(t_i) \mid i = 1, \ldots, n\}$.

![Diagram showing two schedules with spans 6 and 7](image-url)
ASP Encoding for Scheduling

Input: assume that the problem is given ...

\( \text{task}(t_1), \ldots, \text{task}(t_n), \text{duration}(t_1, d_1), \ldots, \text{duration}(t_n, d_n) \)
\( \text{prec}(t_i, t_j), \ldots, \text{non_overlap}(t_i, t_j), \ldots, \)

Code
ASP Encoding for Scheduling

Input: assume that the problem is given ...

\( task(t_1), \ldots, task(t_n), \ duration(t_1, d_1), \ldots, duration(t_n, d_n) \)
\( prec(t_i, t_j), \ldots, non\_overlap(t_i, t_j), \ldots, \)

Code

time(0..length).
% generating start time
1 \{ start(T, S) : time(S) \} 1 :- task(T).
ASP Encoding for Scheduling

Input: assume that the problem is given ...
\( task(t_1), \ldots, task(t_n), \ duration(t_1, d_1), \ldots, duration(t_n, d_n) \)
\( prec(t_i, t_j), \ldots, non\_overlap(t_i, t_j), \ldots, \)

Code

time(0..length).
% generating start time
1 \{ start(T, S) : time(S) \} 1 :- task(T).
% checking prec
:- prec(T1,T2),start(T1,S1),start(T2,S2),duration(T1,D1), S2\lt S1+D1.
ASP Encoding for Scheduling

Input: assume that the problem is given ...

\[ task(t_1), \ldots, task(t_n), \ duration(t_1, d_1), \ldots, duration(t_n, d_n) \]
\[ prec(t_i, t_j), \ldots, non\_overlap(t_i, t_j), \ldots, \]

Code

time(0..length).
% generating start time
1 \{ start(T, S) : time(S) \} 1 :- task(T).
% checking prec
:- prec(T1,T2),start(T1,S1),start(T2,S2),duration(T1,D1), S2 < S1 + D1.
% non-overlap
:- non_overlap(T1,T2), start(T1, S1), start(T2,S2), duration(T1, D1),
   S2 < S1 + D1, S2 \geq S1.
:- non_overlap(T1,T2), start(T1, S1), start(T2,S2), duration(T2, D2),
   S1 < S2 + D1, S1 \geq S2.
ASP Encoding for Scheduling

Input: assume that the problem is given ...

\[
\text{task}(t_1), \ldots, \text{task}(t_n), \text{duration}(t_1, d_1), \ldots, \text{duration}(t_n, d_n) \\
\text{prec}(t_i, t_j), \ldots, \text{non_overlap}(t_i, t_j), \ldots,
\]

Code

\[
\text{time}(0..\text{length}). \\
% \text{ generating start time} \\
1 \{ \text{start}(T, S) : \text{time}(S) \} 1 :- \text{task}(T). \\
% \text{ checking prec} \\
:- \text{prec}(T1, T2), \text{start}(T1, S1), \text{start}(T2, S2), \text{duration}(T1, D1), S2 < S1 + D1. \\
% \text{ non-overlap} \\
:- \text{non_overlap}(T1, T2), \text{start}(T1, S1), \text{start}(T2, S2), \text{duration}(T1, D1), S2 < S1 + D1, S2 \geq S1. \\
:- \text{non_overlap}(T1, T2), \text{start}(T1, S1), \text{start}(T2, S2), \text{duration}(T2, D2), S1 < S2 + D1, S1 \geq S2. \\
% \text{ minimizing span} \\
\text{max_end}(M) :- M = \#\text{max} \{ D + S : \text{task}(T), \text{duration}(T, D), \text{start}(T, S) \}. \\
\text{min_start}(MS) :- MS = \#\text{min} \{ S : \text{task}(T), \text{start}(T, S) \}. \\
\text{span}(MA - MS) :- \text{max_end}(MA), \text{min_start}(MS). \\
\text{\#minimize} \{ S : \text{span}(S) \}. 
\]
Planning with Durative Actions and Numerical Fluents

Assumptions in Planning

- Actions are **duration-less**: every action takes one time step to complete (sometime, it is also referred to as **instantaneous**) (not realistic, e.g., driving takes time)
- Fluents are **boolean**: simplified the modeling but could increase the grounding (e.g., the amount of gasoline in the tank is a real number)

Example 1

John is at the train station and wants to go to class. It is 7am and John has a class at 8am. The train starts at 7:05 am and will take 40 minutes to get to the school. Walking from the train station to the class will require 15 minutes. John would like to have a cup of coffee before the class (otherwise, he will just sleep in the class :-). Standing in line to get the coffee at the coffee stand will take 10 minutes. He could also buy coffee on the train. What should John do?
Planning with Durative Actions and Numerical Fluents

Assumptions in Planning

- Actions are **duration-less**: every action takes one time step to complete (sometime, it is also referred to as **instantaneous**) (not realistic, e.g., driving takes time)

- Fluents are **boolean**: simplified the modeling but could increase the grounding (e.g., the amount of gasoline in the tank is a real number)

Example 2

John needs to catch his flight out of El Paso airport at 9 am. Driving from Las Cruces to El Paso airport takes 45 minutes during non-rush hours. It could take 1 hour and 15 minutes during rush hours. Rush hours is between 7:30 am and 9:30 am. John needs to be at the airport 15 minutes earlier to check in and pass through security. When should John start driving?
Planning with Durative Actions and Numerical Fluents

Problems and Ideas

- Representation of durative actions: this can be done similar to what we use in scheduling; however, there is something more than that

  - **fixed duration**: the train trip between two stations (John’s home and school) takes 40 minutes. In this case, we can use a fact of the form $\text{duration}(a, d)$ to denote that the execution of $a$ takes $d$ time units.

  - **variable duration**: the time needed for driving from Las Cruces to El Paso depends on how fast John drives. In this case, we need to introduce a rule that defines $\text{duration}(a, d)$, for example,
    
    $\text{duration(\text{drive}_{lc_{elp}}, 45/\text{Speed}) :- \text{speed(\text{drive}_{lc_{elp}}, \text{Speed})}}.$
    $\text{duration(\text{drive}(A,B),D/S):-\text{speed(A,B,S)},\text{distance(A,B,D})}.$

- Representation of numerical fluents: use equality, for example,
  
  $\text{holds(\text{amount}_{\text{gasoline}} = 10, T+1) :- \text{...}}$
The encoding of dynamic domains using ASP is developed by several authors. It has its root in the investigation of reasoning about actions and changes and the development of action languages. Some representative references are:

- Tran Cao Son, Chitta Baral, Tran Hoai Nam, and Sheila McIlraith, Domain-Dependent Knowledge in Answer Set Planning, ACM TOCL, 2006.
Outline

Logic Programming
  Syntax
  Examples of Propositional Programs
  Programs with FOL Atoms

Answer Set Solver: clingo (How To?)

Reasoning about Dynamic Domains

Logic Programming and Knowledge Representation

Extensions of Logic Programming and Computing Answer Sets

Advanced Problems in ASP
Commonsense Reasoning

Our knowledge

► is often incomplete (it does not contain complete information about the world), and

► contains defaults (rules which have exceptions, also called normative sentences).

► contains preferences between defaults (prefer a conclusion/default).

For this reasons, we often jump to conclusions (ignore what we do not know), and deal with exceptions and preferences whenever they arise.
We know

- Normally, birds fly.
- Normally, computer science students can program.
- Normally, students work hard.
- Normally, things do not change.
- Normally, students do not watch TV.
- Normally, the speed limit on highways is 70 mph.
- Normally, it is cold in December.

From this, we make conclusions such as *Tweety* flies if we know that *Tweey is a bird; Monica* can program if *she is a computer science student*; etc.
Representing Defaults: Example 1

Normally, a’s are b’s. \[ b(X) \leftarrow a(X), \textbf{not} \ ab(r, X) \]

Normally, birds fly. \[ fly(X) \leftarrow bird(X), \textbf{not} \ ab(\text{bird}_\text{fly}, X) \]

Normally, animals have four legs. \[ \text{numberoflegs}(X, 4) \leftarrow \text{animal}(X), \textbf{not} \ ab(\text{animal}, X) \]

Normally, fishs swim. \[ \text{swim}(X) \leftarrow \text{fish}(X), \textbf{not} \ ab(\text{fish}, X) \]

Normally, computer science students can program. \[ \text{can\_program}(X) \leftarrow \text{student}(X), \text{is\_in}(X, \text{cs}), \textbf{not} \ ab_p(X) \]

Normally, students work hard. \[ \text{hard\_working}(X) \leftarrow \text{student}(X), \textbf{not} \ ab(X) \]

Typically, classes start at 8am. \[ \text{start\_time}(X, 8am) \leftarrow \text{class}(X), \textbf{not} \ ab(X) \]

\(^{1}r\) is the 'name' of the statement; \(ab(X)\) or \(ab_r(X)\) can also be used.
Suppose that we know

- $r_1$: Normally, birds fly.
- $r_3$: Penguins are birds.
- $r_3$: Penguins do not fly.

- $r_4$: Tweety is a penguin.
- $r_5$: Tim is a bird.

\[
\Pi_b = \begin{cases} 
  r_1 : & fly(X) \leftarrow bird(X), \textbf{not} \ ab(r_1, X) \\
  r_2 : & bird(X) \leftarrow penguin(X) \\
  r_3 : & ab(r_1, X) \leftarrow penguin(X) \\
  r_4 : & penguin(tweety) \leftarrow \\
  r_5 : & bird(tim) \leftarrow 
\end{cases}
\]

Answer set of $\Pi_b$: ?
Representing Defaults: Example 2 (Birds and Penguins) II

\[ \Pi_b = \left\{ \begin{array}{l}
  r_1 : \quad fly(X) \quad \leftarrow \quad bird(X), \textbf{not} \ ab(r_1, X) \\
  r_2 : \quad bird(X) \quad \leftarrow \quad penguin(X) \\
  r_3 : \quad ab(r_1, X) \quad \leftarrow \quad penguin(X) \\
  r_4 : \quad penguin(tweety) \quad \leftarrow \\
  r_5 : \quad bird(tim) \quad \leftarrow \\
\end{array} \right. \]

Answer set of \( \Pi_b \):
\{bird(tim), fly(tim), penguin(tweety), ab(r_1, tweety), bird(tweety)\}
\( \Pi_b \models fly(tim) \) and \( \Pi_b \models \neg fly(tweety) \)
Consider the following information:
- Normally lions and tigers are cats.
- Sam and John are lions.
- Sam is not a cat.
- Sam is a sea lion.

This information can be represented by the following program

\[
\Pi_a = \begin{cases} 
    r_1 : & \text{cat}(X) \leftarrow \text{lion}(X), \neg \text{ab}(r_1, X) \\
    r_2 : & \text{cat}(X) \leftarrow \text{tiger}(X), \neg \text{ab}(r_2, X) \\
    r_3 : & \text{lion}(X) \leftarrow \text{sea\_lion}(X) \\
    r_4 : & \text{ab}(r_1, X) \leftarrow \text{sea\_lion}(X) \\
    r_5 : & \text{sea\_lion}(sam) \leftarrow \\
    r_6 : & \text{lion}(john) \leftarrow 
\end{cases}
\]

We can check that \( \Pi_a \) entails that \textit{sam} is a lion but not a cat while \textit{john} is a cat which is a lion.
We know that computers are normally fast machines, the commodore is a slow machine because it is an old one. This can be represented by the following program:

\[
\Pi_c = \left\{ \begin{array}{l}
r_1: \text{fast\_machine}(X) \leftarrow \text{computer}(X), \textbf{not} \ ab(X) \\
r_2: \text{old}(X) \leftarrow \text{comodore}(X) \\
r_3: \text{computer}(X) \leftarrow \text{comodore}(X) \\
r_4: \ ab(X) \leftarrow \text{old}(X) \\
\end{array} \right. 
\]
The Boeing 747, concord, and FA21314 are airplanes. Airplanes normally fly unless they are out of order. FA21314 is out of order. \( \Pi_{airplanes} \) is given below:

\[
\begin{align*}
\textit{r}_1 & : \text{fly}(X) & \leftarrow \text{airplane}(X), \textbf{not} \ ab(X) \\
\textit{r}_2 & : \text{ab}(X) & \leftarrow \text{out_of_order}(X) \\
\textit{r}_3 & : \text{airplane} (\text{boeing\_747}) & \leftarrow \\
\textit{r}_4 & : \text{airplane} (\text{concord}) & \leftarrow \\
\textit{r}_5 & : \text{airplane} (\text{fa21324}) & \leftarrow \\
\textit{r}_6 & : \text{out\_of\_order} (\text{fa21324}) & \leftarrow 
\end{align*}
\]
Inheritance Reasoning I

In practical applications, there might be more than one defaults and they interact with each others.

Example

Consider the knowledge base with the following statements:

- Normally, students are young adults.
- Normally, students are single.
- Young adults are adults.
- Normally, adults are married.

How to answer the following questions?

- Sam is a young adult. Is Sam married?
- Marry is an adult and she is a student.
- John is an adult. Is he a student?
Inheritance Reasoning II

The normal encoding of $\Pi_{\text{adult\_student}}$ is given below:

\[
\begin{align*}
    r_1 : & \quad \text{young\_adult}(X) \leftarrow \text{student}(X), \text{not}\ ab(r_1, X) \\
    r_2 : & \quad \neg\text{married}(X) \leftarrow \text{student}(X), \text{not}\ ab(r_2, X) \\
    r_3 : & \quad \text{adult}(X) \leftarrow \text{young\_adult}(X) \\
    r_4 : & \quad \text{married}(X) \leftarrow \text{adult}(X), \text{not}\ ab(r_4, X) \\
    r_5 : & \quad \text{young\_adult}(\text{sam}) \leftarrow \\
    r_6 : & \quad \text{adult}(\text{marry}) \leftarrow \\
    r_7 : & \quad \text{student}(\text{marry}) \leftarrow \\
    r_8 : & \quad \text{adult}(\text{john}) \leftarrow \\
\end{align*}
\]

This program is inconsistent because of the conflict information about Marry! The information that Marry is a student is more specific than that of her being an adult. Therefore, information about her being single should override that she is married.
Inheritance Reasoning III

Adding rules that allow for information about Marry being married to be overridden:

\[ r_1 : \quad \text{young}_\text{adult}(X) \quad \leftarrow \quad \text{student}(X), \text{not} \ ab(r_1, X) \]
\[ r_2 : \quad \neg \text{married}(X) \quad \leftarrow \quad \text{student}(X), \text{not} \ ab(r_2, X) \]
\[ r_3 : \quad \text{adult}(X) \quad \leftarrow \quad \text{young}_\text{adult}(X) \]
\[ r_4 : \quad \text{married}(X) \quad \leftarrow \quad \text{adult}(X), \text{not} \ ab(r_4, X) \]
\[ r_4' : \quad ab(r_4, X) \quad \leftarrow \quad \text{student}(X), \text{not} \ ab(r_4', X) \]
\[ r_5 : \quad \text{young}_\text{adult}(\text{sam}) \quad \leftarrow \quad \]
\[ r_6 : \quad \text{adult}(\text{marry}) \quad \leftarrow \quad \]
\[ r_7 : \quad \text{student}(\text{marry}) \quad \leftarrow \quad \]
\[ r_8 : \quad \text{adult}(\text{john}) \quad \leftarrow \quad \]

This rule says that normally, students are exceptions to the default \( r_4 \)
(normally, adults are married!). It is derived from the principle:

**Specificity principle**

normally, more specific information overrides less specific one!
Outline

Logic Programming
  Syntax
  Examples of Propositional Programs
  Programs with FOL Atoms

Answer Set Solver: clingo (How To?)

Reasoning about Dynamic Domains

Logic Programming and Knowledge Representation

Extensions of Logic Programming and Computing Answer Sets

Advanced Problems in ASP
Additional Features for KRR

To add expressiveness and to make logic programming more suitable for knowledge representation and reasoning, additional features and constructors are introduced:

- **classical negation**: instead of atoms, literals are used in the rule

  \[ l_0 \leftarrow l_1, \ldots, l_n, \textbf{not} \ l_{n+1}, \ldots, \textbf{not} \ l_{n+k} \]

  where \( l_i \) is a literal (an atom \( a \) or its negation \( \neg a \)). This will allow us to represent and reason with negative information.

- **nested expression**: allowing \( \textbf{not} \ \textbf{not} \ l \).
Using Classical Negation I

\[ \Pi_b = \begin{cases} 
    r_1 : & \text{fly}(X) \leftarrow \text{bird}(X), \textbf{not} \ ab(r_1, X) \\
    r_2 : & \text{bird}(X) \leftarrow \text{penguin}(X) \\
    r_3 : & \text{ab}(r_1, X) \leftarrow \text{penguin}(X) \\
    r_4 : & \text{penguin}(\text{tweety}) \leftarrow \\
    r_5 : & \text{bird}(\text{tim}) \leftarrow \\
\end{cases} \]

In the bird example, “Penguins do not fly” is more intuitive than “Normally, penguins do not fly.” So, \( r_3 \) of \( \Pi_b \) should be changed to

\[ r'_3 : \neg \text{fly}(X) \leftarrow \text{penguin}(X). \]

Doing so will make the program \( \Pi_b \) inconsistent! Obviously, \( r_1 \) does not account for the class of birds who do not fly. We should change the rule \( r_1 \) to

\[ r'_1 : \text{fly}(X) \leftarrow \text{bird}(X), \textbf{not} \ ab(r_1, X), \textbf{not} \ \neg \text{fly}(X) \]

The new program, \( \Pi'_b = \Pi_b \setminus \{ r_1, r_3 \} \cup \{ r'_1, r'_3 \} \) will correctly answer the same questions such as “does Tim fly” and “does Tweety fly?”
Using Classical Negation II

Suppose that we have a list of professors and their courses:

<table>
<thead>
<tr>
<th>Professor</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>mike</td>
<td>ai</td>
</tr>
<tr>
<td>sam</td>
<td>db</td>
</tr>
<tr>
<td>staff</td>
<td>C</td>
</tr>
</tbody>
</table>

where staff stands for “someone” – an unknown professor – who might be different than mike and sam. This list can be expressed by a set of atoms of the form teach(P, C) (P teaches C): teach(mike, ai), teach(sam, db), and teach(staff, c).

By default, we know that if a professor P teaches the course C, then (P, C) will be listed (and hence the atom teach(P, C) will be present.) Thus, by default, professor P does not teach the course C if teach(P, C) is not present. The exception to this rule are the courses taught by “staff”. This leads to the following two rules:

\[
\neg teach(P, C) \leftarrow \text{not teach}(P, C), \text{not ab}(P, C).
\]

\[
ab(P, C) \leftarrow \text{teach}(staff, C)
\]

This will allow us to conclude that mike teaches ai but we do not know whether he teaches c or not.
The Closed World Assumption is invented by Ray Reiter in his study on deductive database. It has since been used in several areas of knowledge representation. The key idea of the CWA is the presumption that whatever is true is also known to be true. In other words, whatever is not true is false.

We have used the CWA in representing the initial state of planning problems. For example, to describe the initial state of the block world domain, we list all fluents that are true and add a rule that completes the initial state:

- holds(on(a,b), 0).
- holds(ontable(b), 0).

....

- holds(clear(a), 0).

```
 holds(neg(F), 0) :- fluent(F), not holds(F, 0).
```

The last rule is often called CWA rule.
Outline

Logic Programming
  Syntax
  Examples of Propositional Programs
  Programs with FOL Atoms

Answer Set Solver: clingo (How To?)

Reasoning about Dynamic Domains

Logic Programming and Knowledge Representation

Extensions of Logic Programming and Computing Answer Sets

Advanced Problems in ASP
Advanced Problems in ASP

- Planning with durative actions and resources
- Integration of ASP with external solvers
- Multi-agent systems with ASP
Planning with durative actions and resources

Problem
Given a planning problem with durative actions and resources. How to solve this type of problems using ASP?
The general idea is simple but scalable and efficient solution is yet to be developed.

Example 1
In the block world domain, we can imagine that each block has some weights. The cost of moving a block to the table or atop of another block is given in some way (e.g., the robot arm needs gasoline to move around and the amount of gasoline is proportional to the weight of the block).
Planning with durative actions and resources

Problem
Given a planning problem with durative actions and resources. How to solve this type of problems using ASP?
The general idea is simple but scalable and efficient solution is yet to be developed.

Example 2
Suppose that we are at the headquarter of FedEx. We need to create a delivery plan for packets that we receive in the day. Assume that the packets have been sorted, i.e., origins and destinations and type (e.g., express, normal, one-day, etc.) of packets are known. FedEx has airplanes and trucks for transportation. The delivery plan would need to take into consideration information such as

- the duration of each action ($fly(from, to)$ or $drive(from, to)$)
- the type of packet which implies the dateline for delivery
- the availability of the transportation means
Integration of ASP with external solvers

- Early ASP-solver is developed to run in batch mode. There is no interaction with users during the answer set computation process. This might not be an optimal way to deal with problems that are, for example, interactive in nature. For instance, if we were to use ASP to analyze video stream of an airport to identify terrorists, we cannot wait until we get the complete video. It will never be complete or if it is complete then the terrorist might have been escaped already.

- The method of computing answer sets by ASP-solvers is dominantly two steps: first the system grounds the program and then it computes the answer sets. As such, scalability is a challenge for ASP-based applications. For example, we cannot consider variables with continuous domains such as the amount of gasoline, probability, etc.
Integration of ASP with external solvers: Current Directions

- Reactive ASP: for stream/reactive reasoning (e.g., elevator controller)
- Multi-shot ASP: connection with Python (e.g., user interface, game)
- Integration with external solver
  - database (or ontology) solver for Semantic Web applications
  - constraint solver for dealing with real number
- Integration with natural language processor to acquire knowledge, represent it as an ASP program, and reason about it. See presentation on ASP and HRI.
Multi-Agent Systems

- See agent based modeling slides
- Platform for agent based modeling