Introduction to Answer Set Programming

Sheila McIlraith
CSC2542
Department of Computer Science
University of Toronto
April 19, 2006

Acknowledgements

The material in this presentation was drawn from papers and slides by the following people: Chitta Baral, Michael Gelfond, Vladimir Lifschitz, Ilka Niemela, Aarati Pamar, Torsten Schwaub, and Son Cao Tran.

Answer Set Programming

Answer Set Programming (ASP) is a form of **declarative programming** oriented towards combinatorial problems (i.e., search).

Syntactically: Looks like logic programming
Computationally: Similar to SAT solving

Example Applications:

- plan generation
- product configuration
- diagnosis
- default reasoning
- graph theory problems in VLSI

…basically anything where the solutions to the problem can be characterized as (preferred) models.

What's so great about ASP?

Effective computational machinery for problem domains that require combinatorial search together with

- Nonmonotonic reasoning (including closed world assumption, frame problem, default reasoning, etc.)
- Reasoning with incomplete information

Approach

E.g., plan generation, product configuration diagnosis, verification

Recall that in SAT

Problem → propositional theory → SAT Solver

The solutions to the problem are the **models** of the propositional theory.

Similarly in ASP,

Problem → Logic Program → ASP Solver

The solutions to the problem are the **answer sets** of the logic program

Answer Set Programming

…So what *are* answer sets

and how do we compute them?

Outline

- Origins of ASP
- Quick Review of Logic Programming Semantics
- Answer Set Semantics
  - Introduction
  - Examples
  - Equivalence to Default Logic
- Computing Answer Sets
  - smodels
  - dlv
  - assat, cmodels, noMoRe
- Smodels Example

Origins of ASP

Proposed in the late 1990s as a new logic programming paradigm.

(e.g., Lifschitz, 1999; Marek and Truszczynski, 1999; Niemela 1999)

Emerged from the interaction of 2 lines of research:

1. Semantics of negation in logic programming
   *(I.e., **stable model semantics** for logic programs*
   (Gelfond and Lifschitz, 1988))

2. Application of **SAT solvers** to search problems
   (e.g., Kautz and Selman, 1992)

"**Stable Model Semantics"** are the same as "**Answer Set Semantics"**

Review of Logic Pgming Semantics

$\mathcal{L}$ – a first order language with its usual components (e.g., variables, constants, function symbols, predicate symbols, etc.

$U_{\mathcal{L}}$ – **Herbrand Universe** of the 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 atom which can be formed with the functions, constants and predicates in $\mathcal{L}$.

E.g., 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))),…}

$B_{\mathcal{L}_1}$ = {p(a),p(b),p(f(a)),p(f(b)),p(f(f(a)),p(f(f(b))),p(f(f(f(a))), …}

A **Herbrand Interpretation** of a logic program P is a set of atoms from its Herbrand Base.

The **Least Herbrand Model** of a program P is called the **minimal model** of P.

Logic Programming Semantics *(cont)*

Definite logic programs, having rules of the form (no not's in the body):

$R_i: \quad L_1 \leftarrow L_{1+1}, \ldots, L_m.$

have a **unique** intended Herbrand model – the least Herbrand model.

Logic Programming Semantics *(cont)*

Unfortunately, when you add **negation**, things get complicated.
General (aka) **Normal logic programs** have rules of the form

$\mathbf{R_i}: \quad L_1 \leftarrow L_{l+1}, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n$

where the **not** is **negation as failure** *(Clark, 1978);(Reiter, 1978))*

- Usually there is no unique least Herbrand model.
- Choosing a single intended model is difficult.
- Logic programming schism:
    1. **Single intended model** approach, e.g.,
        - Perfect semantics of stratified programs
        - 3-valued well-founded semantics for (arbitrary) programs.
    2. **Multiple preferred model** approach, e.g.,
        - Stable model semantics (aka answer set semantics)

Answer Set Semantics

Intuitively,

An answer set is the **minimal set of atoms** (using set inclusion) that satisfies a set of prolog-style rules.

$$\mathbf{R_i}: \quad L_1; \ldots; L_k; \text{not } L_{k+1}; \ldots; \text{not } L_l \leftarrow L_{l+1}, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n$$

Answer Set Semantics

Intuitively,

An answer set is the **minimal set of atoms** (using set inclusion) that satisfies a set of prolog-style rules.

$$\mathtt{R_i:}\quad \mathtt{L_1;\ldots;L_k;not\ L_{k+1};\ldots;not\ L_l} \leftarrow \mathtt{L_{l+1},\ldots,L_m,not\ L_{m+1},\ldots,not\ L_n}$$

Satisfaction of the rules is defined in terms of the concept of **reducts** of the rules – **transformations** of the rules $\{\mathbf{R_i}\}$ that are used to check if an answer set $\mathbf{M}$ satisfies $\{\mathbf{R_i}\}$. *(Gelfond and Lifschitz, 1988).*

Intuitively, a literal is only in the answer set if it is justified by a rule in the program P.

Answer Set Semantics – Definite Programs

If there is **no negation** (no `nots`), the definition of an answer set is easy. Given, rules of the form:

$$\mathtt{R_i:}\quad \mathtt{L_1;\ldots;L_k} \leftarrow \mathtt{L_{l+1},\ldots,L_m}$$

$\mathbf{M}$ is an answer set when for each $\mathbf{R_i}$,

if all of $\mathtt{L_{l+1},\ldots,L_m}$ are in $\mathbf{M}$, then at least one of $\mathtt{L_1,\ldots,L_k}$ is in $\mathbf{M}$, and $\mathbf{M}$ is the **minimal** such set (under set inclusion).

Another way to think about this is that every literal in the answer set has to have a reason to be in $\mathbf{M}$.

The **answer set of a definite program** P is the smallest subset $\mathbf{M}$ of the Herbrand Base such that for any rule $\mathtt{L_k} \leftarrow \mathtt{L_{l+1},\ldots,L_m}$ from P, if $\mathtt{L_{l+1},\ldots,L_m}$ is in $\mathbf{M}$ then $\mathtt{L_k}$ is in $\mathbf{M}$.

Answer Set Semantics - Normal Logic Programs

Now let's add negation. For **normal logic programs,** given

$\mathbf{P} = \{\mathbf{R_i}\}$ – a propositional normal logic program

$$\mathbf{R_i}: \quad L_1 \leftarrow L_2, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n$$

$\mathbf{M}$ – set of atoms (which are the potential answer set)

Reduct $\mathbf{P}^{\mathbf{M}}$ (aka $\{\mathbf{R_i}\}^{\mathbf{M}}$) is a definite program constructed as follows:

- For each atom $L \in M$, **remove rules** with **not L** in the body
- Remove literals **not L** from all other rules

M is an **answer set** of **P** if **M** is a **least model of the reduct**.

Again, intuitively, a literal is only in the answer set if it is justified by a rule in the program P.

Answer Set Semantics – The General Case

In the most **general case**,if there **is** negation in the head and body of the rules, and disjunction in the head, i.e., if rules are of the form:

$$\mathbf{R_i}: \quad L_1; \ldots; L_k; \text{not } L_{k+1}; \ldots; \text{not } L_l \leftarrow L_{l+1}, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n$$

Take the reduct $\{\mathbf{R_i}\}^{\mathbf{M}}$. I.e.,

- take the set of rules where all of $L_{k+1}, \ldots, L_l$ are in **M** and none of $L_{m+1}, \ldots, L_n$ are.
- remove $\text{not } L_{m+1}, \ldots, \text{not } L_n$ from all remaining rules.

Resulting rules in the **reduct** will be of the form:

$$\mathbf{R_i}': \quad L_1; \ldots; L_k \leftarrow L_{l+1}, \ldots, L_m$$

M is an **answer set** for $\{\mathbf{R_i}\}$ iff it is for $\{\mathbf{R_i}\}^{\mathbf{M}}$

Reduct Intuition *(Gelfond)*

Take a set of literals $\mathbf{M}$. For each rule $\mathbf{R_i}$ of the form:

$$\mathbf{R_i}: \quad L_1;\ldots;L_k;\texttt{not}\ L_{k+1};\ldots;\texttt{not}\ L_l \leftarrow$$
$$L_{l+1},\ldots,L_m,\texttt{not}\ L_{m+1},\ldots,\texttt{not}\ L_n$$

Either at least one of the $L_{m+1},\ldots,L_n$ is in $\mathbf{M}$, or none are.

- If at least one of the $L_{m+1},\ldots,L_n$ is in $\mathbf{M}$ then the rule can't possibly fire, **so we throw it out of the reduct**.
- If none of $L_{m+1},\ldots,L_n$ is in $\mathbf{M}$ then the rule is equivalent to

$$\mathbf{R_i}': \quad L_1;\ldots;L_k;\texttt{not}\ L_{k+1};\ldots;\texttt{not}\ L_l \leftarrow L_{l+1},\ldots,L_m$$

Reduct Intuition (continued)

Continuing with a subset of the rules of the form:

$$\mathbf{R_i}': \quad L_1;\ldots;L_k;\texttt{not}\ L_{k+1};\ldots;\texttt{not}\ L_l \leftarrow L_{l+1},\ldots,L_m$$

- If at least one of the $L_{k+1},\ldots,L_l$ is *missing* from $\mathbf{M}$ then the rule is automatically satisfied, **so we throw it out of the reduct**.
- If all of the $L_{k+1},\ldots,L_l$ are in $\mathbf{M}$ then the rule is still not satisfied, and we can write it as:

$$\mathbf{R_i}'': \quad L_1;\ldots;L_k \leftarrow L_{l+1},\ldots,L_m$$

The intuition is that if $\mathbf{M}$ still satisfies $\{\mathbf{R_i}\}^{\mathbf{M}}$ after these transformations, then it should satisfy $\{\mathbf{R_i}\}$.

Answer Set Semantics

Let $Cn(P^M)$ denote the smallest set of atoms that is closed under the reduct P^M. A set **M** of atoms is an answer set (aka stable model) of a program P iff $Cn(P^M)$= **M**.

I.e., an answer set is closed under the rules of P, and it is grounded in P, that is, each of its atoms has a derivation using applicable rules from P. Answer set was originally called **stable model** because it was indeed "stable".

Obviously we can have many answer sets. As with SAT problem encodings, these models correspond to different solutions to a problem (e.g., different plans, different diagnoses, etc.), and again as with SAT, P entails a formula *f* if *f* is true in all answer set of P.

So for query answering, the answer to the ground query *q* is

- yes if *q* is true in all answer sets of P
- no if –*q* is true in all answer sets of P and
- unknown otherwise.

Example 1

An easy first example with no negation in the body.

```
    p,q.   (1)
-r ← p.    (2)
```

Either **p** or **q** has to be in **M**, yielding the following candidate answer sets:

{ **p**},{ **q** },{ **p**, **q** }

Then rule (2) fires **p** so we must include **-r**, yielding

{ **p** , **-r** },{ **q** },{ **p**, **q** , **-r** }

Applying the minimality criterion yields two answer sets:

{ **p** , **-r** } and { **q** }

Example 2

A second example:

```
p_next ← p, not p_next'.      (1)
p_next' ← p', not p_next.     (2)
        ← p,p'.               (3)
        ← p_next,p_next'.     (4)
p,p'.                         (5)
```

No matter what **M** we choose, **(3)-(5)** will be in the reduct.
(5) tells us we need an answer set for both **p** and **p'** but not together since **(3)** tells us they are mutually exclusive, as are **p_next,p_next'**.
So the candidate answer sets are:

{**p** },{**p'** },{ **p**, **p_next**}, { **p'**, **p_next**},
{ **p**, **p_next'**}, { **p'**, **p_next'**}.

Example 2 (continued)

Now let's compute the reducts for each candidate **M**.

```
p_next ← p, not p_next'.      (1)
p_next' ← p', not p_next.     (2)
        ← p,p'.               (3)
        ← p_next,p_next'.     (4)
p,p'.                         (5)
```

M = {**p** }
M = {**p'** }
M = { **p**, **p_next**}
M = { **p'**, **p_next**}
M = { **p**, **p_next'**}
M = { **p'**, **p_next'**}.

Example 2 (continued)

Now let’s compute the reducts for each candidate **M**.

p_next ← p, not p_next’. (1)
p_next’ ← p’, not p_next. (2)
← p,p’. (3)
← p_next,p_next’. (4)
p,p’. (5)

M = {p} $\{Ri\}^M$= {p_next ← p.,p_next’ ← p’.,(3),(4),(5).}
M = {p’} $\{Ri\}^M$= {p_next ← p.,p_next’ ← p’.,(3),(4),(5).}
M = { p, p_next} $\{Ri\}^M$= {p_next ← p.,(3),(4),(5).}
M = { p’, p_next} $\{Ri\}^M$= {p_next ← p.,(3),(4),(5).}
M = { p, p_next’} $\{Ri\}^M$= {p_next’ ← p’.,(3),(4),(5).}
M = { p’, p_next’}. $\{Ri\}^M$= {p_next’ ← p’.,(3),(4),(5).}

Example 2 (continued)

Now let’s compute the reducts for each candidate **M**.

p_next ← p, not p_next’. (1)
p_next’ ← p’, not p_next. (2)
← p,p’. (3)
← p_next,p_next’. (4)
p,p’. (5)

~~M = {p} {Ri}^M= {p_next ← p.,p_next’ ← p’.,(3),(4),(5).}~~
~~M = {p’} {Ri}^M= {p_next ← p.,p_next’ ← p’.,(3),(4),(5).}~~
M = { p, p_next} $\{Ri\}^M$= {p_next ← p.,(3),(4),(5).}
~~M = { p’, p_next} {Ri}^M= {p_next ← p.,(3),(4),(5).}~~
~~M = { p, p_next’} {Ri}^M= {p_next’ ← p’.,(3),(4),(5).}~~
M = { p’, p_next’}. $\{Ri\}^M$= {p_next’ ← p’.,(3),(4),(5).}

Equivalence to Default Logic

Recall default logic statements of the form

$$\frac{L_1, \dots, L_m \;:\; -L_{m+1}, \dots, -L_n}{L_0} \qquad \textbf{(1)}$$

State that if $L_1, \dots, L_m$ are true
and we can consistently assume $-L_{m+1}, \dots, -L_n$ then infer L_0

Equivalence to Default Logic

Recall default logic statements of the form

$$\frac{L_1, \dots, L_m \;:\; -L_{m+1}, \dots, -L_n}{L_0} \qquad \textbf{(1)}$$

State that if $L_1, \dots, L_m$ are true
and we can consistently assume $-L_{m+1}, \dots, -L_n$ then infer L_0

(1) is equivalent to the answer set of

$$\textbf{R:}\; L_0 \leftarrow L_1, \dots, L_m, \textbf{not}\; L_{m+1}, \dots, \textbf{not}\; L_n$$

Equivalence to Default Logic

Recall default logic statements of the form

$$\frac{L_1,\dots,L_m \; : \; -\,L_{m+1},\dots,-\,L_n}{L_0} \qquad \textbf{(1)}$$

State that if $L_1,\dots,L_m$ are true
and we can consistently assume $-\,L_{m+1},\dots,-\,L_n$ then infer L_0

(1) is equivalent to the answer set of

$$\mathbf{R:}\; L_0 \leftarrow L_1,\dots,L_m,\mathbf{not}\; L_{m+1},\dots,\mathbf{not}\; L_n$$

This makes intuitive sense: **M** is an answer set of **R** if **M** lacks $L_{m+1},\dots,L_n$ and includes $L_1,\dots,L_m$ and L_0.

Equivalence to Default Logic

Recall default logic statements of the form

$$\frac{L_1,\dots,L_m \; : \; -\,L_{m+1},\dots,-\,L_n}{L_0} \qquad \textbf{(1)}$$

State that if $L_1,\dots,L_m$ are true
and we can consistently assume $-\,L_{m+1},\dots,-\,L_n$ then infer L_0

(1) is equivalent to the answer set of

$$\mathbf{R:}\; L_0 \leftarrow L_1,\dots,L_m,\mathbf{not}\; L_{m+1},\dots,\mathbf{not}\; L_n$$

This makes intuitive sense: **M** is an answer set of **R** if **M** lacks $L_{m+1},\dots,L_n$ and includes $L_1,\dots,L_m$ and L_0.

More formally

> **M** is an answer set of $\{\mathbf{R_i}\}$ iff the deductive closure of **M** is a consistent extension of the equivalent default theory.

Equivalence to Default Logic (cont.)

Example 2 can be interpreted as a default theory by replacing `pred'` by `-pred`.

Answer Set Pgm	Default Theory
`p_next ← p, not p_next'.`	$\dfrac{p : p\_next}{p\_next}$
`p_next' ← p', not p_next.`	
`← p,p'.`	$\dfrac{-p : -p\_next}{-p\_next}$
`← p_next,p_next'.`	
`p,p'.`	$p \vee - p$

Notice that this is just the commonsense law of inertia for p.
Extensions: {`p` ,`p_next` } and {`-p`,`-p_next` }

Outline

- ✓ Origins of ASP
- ✓ Quick Review of Logic Programming Semantics
- ✓ Answer Set Semantics
  - – Introduction
  - – Examples
  - – Equivalence to Default Logic
- Computing Answer Sets
  - – smodels
  - – dlv
  - – assat, cmodels, noMoRe
- Smodels Example

Encoding a Problem

Answer Set generation follows a generate-and-test strategy.

To encode a problem as an ASP

- Write a group of rules whose answer sets would correspond to candidate solutions **(generators)**
- Add a second group of rules, mainly consisting of integrity constraints (rules of the form $\leftarrow L_{l+1}, \ldots, L_m, \mathbf{not}\ L_{m+1}, \ldots, \mathbf{not}\ L_n$) that eliminate candidates representing invalid solutions **(testers)**

Encoding a Problem

Answer Set generation follows a generate-and-test strategy.

To encode a problem as an ASP

- Write a group of rules whose answer sets would correspond to candidate solutions **(generators)**
- Add a second group of rules, mainly consisting of integrity constraints (rules of the form $\leftarrow L_{l+1}, \ldots, L_m, \mathbf{not}\ L_{m+1}, \ldots, \mathbf{not}\ L_n$) that eliminate candidates representing invalid solutions **(testers)**

In addition to logic programming, there are a number of ASP language **extensions** allowing: classical negation, disjunctive logic programs, nested logic programs, cardinality constraints, preferences, rule preferences, ordered disjunction, aggregate functions, etc.

CSC2542 ASP Lecture – April, 2006

ASP Systems

There are a number of "ASP systems":

- **smodels** *(Simons, Niemela, Soininen et al.)*
- **dlv** *(Leone, Faver, Eiter, Gottlob, Koch, et al.)*
- **noMoRe** (Linke)
- **assat** *(Lin, Zhou et al.)*
- **cmodels** *(Lifschitz et al.)*

In order to deal with programs containing variables, the systems rely on a two phase implementation consisting of:

1. Grounding to eliminate variables (and to deal with extensions)
2. Computation of answer sets for propositional programs

The "ASP systems" predominantly refer to the 2nd phase of computation. Most ASPs use a program called **lparse** to do the grounding.

ASP Systems

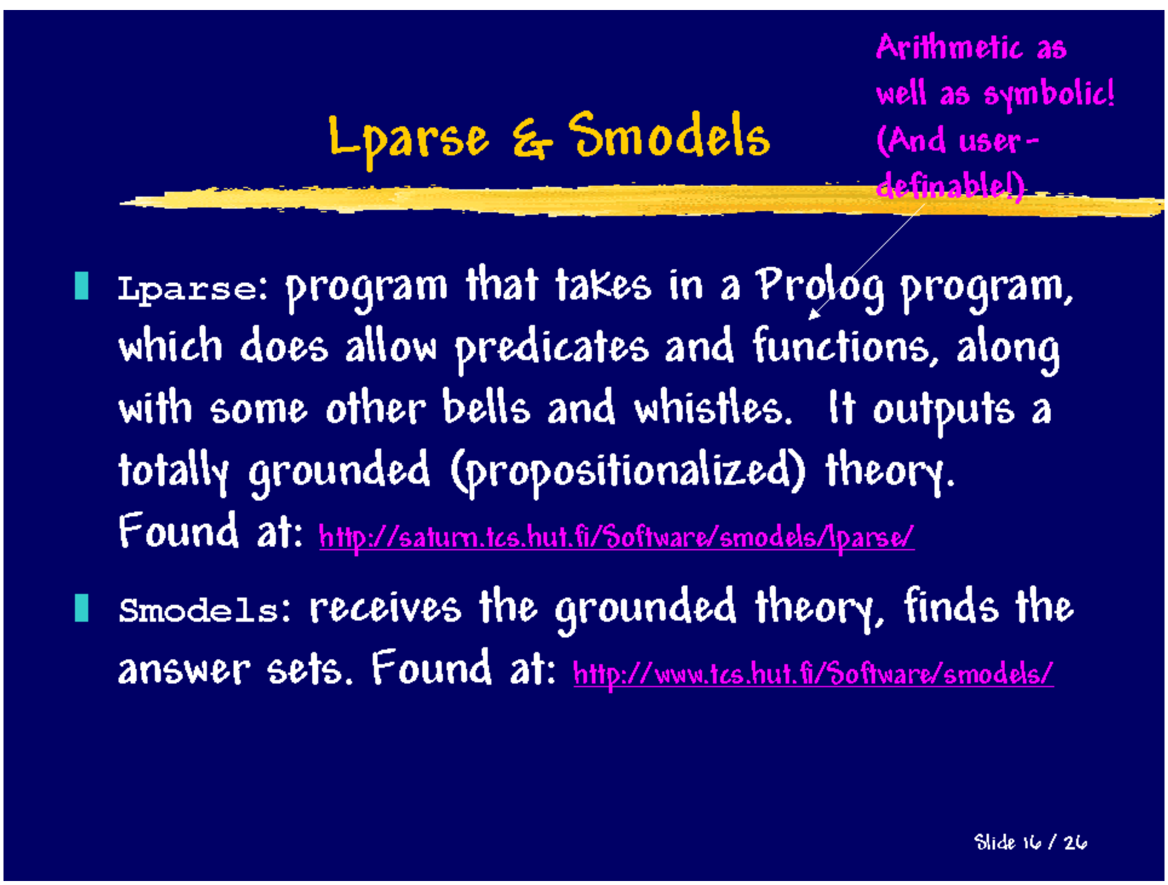
To **compute answer sets**, most of these systems use a procedure **similar to DPLL**, where a partial model is constructed and refined. Some editorial comments:

- **smodels** *(first and most popular)*
- **dlv** *(also very popular. core similar to smodels. extended for disjunctive logic programs, negation and aggregates)*
- **noMoRe** *(graph based approach, originating in default logic. I know nothing else about it)*
- **assat** *(uses SAT solver. Fast but unfortunately mapping creates a blow up in the representation. Uses Clark's completion to deal w/ the closure in the translation)*
- **cmodels** *(also uses SAT solver in a similar way)*

The relationship between SAT and ASP is a topic of growing interest.

Acknowledgements

Thanks to Aarati Pamar for the slides that follow…



Smodels

- Takes in grounded output of lparse, returns as many answers set as you like.
- Stable model semantics
- Handles:
  - constraint rules,
  - choice rules,
  - weight rules, and
  - optimize statements.

DLV (Datalog w Disjunction)

- Unlike `lparse+smodels`, does handle true negation
- Also disjunction (in head):

```
color(X,red) v color(X,green) v color(X,blue) :- node(X).
```

- Arithmetic relations, limited arithmetic functions
- Constants
- Support for brave vs. cautious reasoning, planning, diagnosis, SQL3, prioritized logic programs.

Example: Blocksworld (Lifschitz)

```
const grippers=2.
const lasttime=3.

block(1..6).

% Initial state:    Goal:
%
%                   3 6
% 1 3 5             2 5
% 2 4 6             1 4
% ------------- ---------

% DEFINE


on(1,2,0).
on(2,table,0).
on(3,4,0).
on(4,table,0).
on(5,6,0).
on(6,table,0).
```

```
% TEST

:- not on(3,2,lasttime).
:- not on(2,1,lasttime).
:- not on(1,table,lasttime).
:- not on(6,5,lasttime).
:- not on(5,4,lasttime).
:- not on(4,table,lasttime).
```


Example: Blocksworld (Lifschitz)

```
time(0..lasttime).

location(B) :- block(B).
location(table).

% GENERATE
{move(B,L,T) : block(B) : location(L)} grippers :- time(T),
            T<lasttime.

% DEFINE
% effect of moving a block
on(B,L,T+1) :- move(B,L,T), block(B), location(L), time(T),
                    T<lasttime.

% inertia
on(B,L,T+1) :- on(B,L,T), not on'(B,L,T+1), location(L),
                    block(B), time(T), T<lasttime.
```

Example: Blocksworld (Lifschitz)

```
% uniqueness of location
on'(B,L1,T) :- on(B,L,T), L!=L1, block(B), location(L),
               location(L1), time(T).

% TEST
% on' is the negation of on
:- on(B,L,T), on'(B,L,T), block(B), location(L), time(T).

% two blocks cannot be on top of the same block
:- 2 {on(B1,B,T) : block(B1)}, block(B), time(T).

% a block can't be moved unless it is clear
:- move(B,L,T), on(B1,B,T), block(B), block(B1), location(L),
               time(T), T<lasttime.

% a block can't be moved onto a block that is being moved also
:- move(B,B1,T), move(B1,L,T), block(B), block(B1),
               location(L), time(T), T<lasttime.
```


Example: Blocksworld (Lifschitz)

```
smodels version 2.25. Reading...done
Answer: 1
Stable Model: move(1,table,0) move(3,table,0) move(2,1,1)
move(5,4,1) move(3,2,2) move(6,5,2)
False
Duration: 0.190
Number of choice points: 0
Number of wrong choices: 0
Number of atoms: 507
Number of rules: 3026
Number of picked atoms: 24
Number of forced atoms: 13
Number of truth assignments: 944
Size of searchspace (removed): 0 (0)
```

```
% Initial state:    Goal:
%
%                   3 6
% 1 3 5             2 5
% 2 4 6             1 4
% -------------     ---------
```

Example: Some Points

- A different way to program action theories -- allow all actions, but then restrict models via constraints. (Qualification, ramification problems)
- CWA, UNA automatically taken care of via grounding.
- Negation as failure semantics allows solution to frame problem.
- Difference between `p.` (Assertion) and `:- not p.` (Filter)
- Semantics well understood and SIMPLE!

Recap

- Origins of ASP
- Quick Review of Logic Programming Semantics
- Answer Set Semantics
  - Introduction
  - Examples
  - Equivalence to Default Logic
- Computing Answer Sets
  - smodels
  - dlv
  - assat, cmodels, noMoRe
- Smodels Example

Want to learn more?

There are some excellent resources and references, including text books, good theoretical papers and well-documented systems to experiment with. I'll update the postings on our Web page to reflect these.