Hannes Strass (based on slides by Martin Gebser & Torsten Schaub (CC-BY 3.0))

Faculty of Computer Science, Institute of Artificial Intelligence, Computational Logic Group

ASP: Language Extensions and Modelling

Lecture 11, 8th Jan 2024 // Foundations of Logic Programming, WS 2023/24

Previously . . .

- PROLOG-based logic programming focuses on **theorem proving**.
- LP based on stable model semantics focuses on **model generation**.
- The **stable model** of a positive program is its least (Herbrand) model.
- The **stable models** of a normal logic program P are those sets X for which X is the stable model of the positive program P^X (the reduct).
- The **well-supported** model semantics equals **stable** model semantics.

Example

Logic program $\{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ has stable models $\{p\}$ and $\{q\}$.

Remember

A stable model is a supported model in which every true atom has well-founded support.

Overview

Language Extensions

Basic Language Extensions

Fact

The expressiveness and/or usability of a language can be enhanced by adding new language constructs.

Questions

- What is the syntax of the new language construct?
- What is the semantics of the new language construct?
- How to implement the new language construct?

Answers

- A way of providing semantics is to furnish a translation removing the new constructs. ($\rightsquigarrow$ New constructs are merely "syntactic sugar".)
- This translation might also be used for implementing the extension.

Integrity Constraint

Purpose: Eliminate unwanted solution candidates

Definition

An **integrity constraint** is of the form

$$\leftarrow a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n$$

where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$.

Example: `:- edge(3,7), colour(3,red), colour(7,red).`

Example Programs

$\{ a \leftarrow {\sim}b,\ b \leftarrow {\sim}a \}$	$\{a\}$	$\{b\}$
$\{ a \leftarrow {\sim}b,\ b \leftarrow {\sim}a \} \cup \{ \leftarrow a \}$		$\{b\}$
$\{ a \leftarrow {\sim}b,\ b \leftarrow {\sim}a \} \cup \{ \leftarrow {\sim}a \}$	$\{a\}$	

Embedding in Normal Rules

Translation

An integrity constraint of the form

$$\leftarrow a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n$$

can be translated into the normal rule

$$x \leftarrow a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n, {\sim}x$$

where x is a new symbol.

Example Programs

$\{\ a \leftarrow {\sim}b,\ b \leftarrow {\sim}a\ \}$	$\{a\}$	$\{b\}$
$\{\ a \leftarrow {\sim}b,\ b \leftarrow {\sim}a\ \} \cup \{\ x \leftarrow a, {\sim}x\ \}$		$\{b\}$
$\{\ a \leftarrow {\sim}b,\ b \leftarrow {\sim}a\ \} \cup \{\ x \leftarrow {\sim}a, {\sim}x\ \}$	$\{a\}$	

Choice Rule

Purpose: Provide choices over subsets of atoms

Definition

A **choice rule** is of the form

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o$$

where $0 \leq m \leq n \leq o$ and each a_i is an atom for $1 \leq i \leq o$

Informal meaning: If the body is satisfied by the stable model, any subset of $\{a_1, \ldots, a_m\}$ can be included in the stable model.

Example: `{ buy(pizza); buy(wine); buy(corn) } :- at(grocery).`

Example Program

$$\{\ \{a\} \leftarrow b, \quad b \leftarrow\ \} \qquad\qquad \{b\} \quad \{a, b\}$$

Embedding in Normal Rules

Translation

A choice rule of the form

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o$$

can be translated into $2m + 1$ normal rules

$$\begin{array}{rcl} x & \leftarrow & a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o \\ a_1 & \leftarrow & x, \sim x_1 \quad \ldots \quad a_m \leftarrow x, \sim x_m \\ x_1 & \leftarrow & \sim a_1 \qquad \ldots \quad x_m \leftarrow \sim a_m \end{array}$$

by introducing new atoms $x, x_1, \ldots, x_m$.

Example Program

$$\{\ \{a\} \leftarrow b, \quad b \leftarrow\ \} \qquad \{b\} \quad \{a, b\}$$

$$\left\{ \begin{array}{rcl} x & \leftarrow & b \\ a & \leftarrow & x, \sim x_1 \\ x_1 & \leftarrow & \sim a \end{array} \right\} \cup \{\ b \leftarrow\ \} \qquad \{b, x, x_1\} \quad \{a, b, x\}$$

Cardinality Rule

Purpose: Control (lower) cardinality of subsets of literals

Definition

A **cardinality rule** is the form

$$a_0 \leftarrow l \; \{ \; a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n \; \}$$

where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$; and l is a non-negative integer called **lower bound**.

Informal meaning: The head belongs to the stable model, if at least l positive/negative body literals are in/excluded in the stable model.

Example: `pass(c42) :- 2 { pass(a1); pass(a2); pass(a3) }.`

Example Program

$$\{ \; a \leftarrow 1 \; \{b, c\}, \; b \leftarrow \; \} \qquad\qquad \{a, b\}$$

Embedding in Normal Rules

Translation

A cardinality rule of the form

$$a_0 \leftarrow l\ \{\ a_1, \dots, a_m, {\sim}a_{m+1}, \dots, {\sim}a_n\ \}$$

is translated into the normal rule $a_0 \leftarrow x(1, l)$ and for $0 \leq k \leq l$ the rules

$$\begin{array}{rcll}
x(i, k{+}1) & \leftarrow & x(i+1, k), a_i & \\
x(i, k) & \leftarrow & x(i+1, k) & \text{for } 1 \leq i \leq m \\
x(j, k{+}1) & \leftarrow & x(j+1, k), {\sim}a_j & \\
x(j, k) & \leftarrow & x(j+1, k) & \text{for } m+1 \leq j \leq n \\
x(n+1, 0) & \leftarrow & &
\end{array}$$

Idea: The atom $x(i, j)$ represents that at least j of the literals having an equal or greater index than i are in a stable model.

An Example

- Program $\{\ a \leftarrow 1\ \{b, c\},\quad b \leftarrow\ \}$ has the stable model $\{a, b\}$.
- Translating the cardinality rule yields the rules

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
a & \leftarrow & x(1,1) & b & \leftarrow & \\
x(1,2) & \leftarrow & x(2,1), b & & & \\
x(1,1) & \leftarrow & x(2,1) & & & \\
x(2,2) & \leftarrow & x(3,1), c & & & \\
x(2,1) & \leftarrow & x(3,1) & & & \\
x(1,1) & \leftarrow & x(2,0), b & & & \\
x(1,0) & \leftarrow & x(2,0) & & & \\
x(2,1) & \leftarrow & x(3,0), c & & & \\
x(2,0) & \leftarrow & x(3,0) & & & \\
x(3,0) & \leftarrow & & & &
\end{array}
$$

having stable model $\{a, b, x(3,0), x(2,0), x(1,0), x(1,1)\}$.

Cardinality Rules with Upper Bounds

Translation

A rule of the form

$$a_0 \leftarrow l \; \{ \; a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \; \} \; u$$

where $0 \leq m \leq n$, each a_i is an atom for $1 \leq i \leq n$,
and l and u are non-negative integers

is translated into

$$\begin{array}{rcl} a_0 & \leftarrow & x, \sim y \\ x & \leftarrow & l \; \{ \; a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \; \} \\ y & \leftarrow & u{+}1 \; \{ \; a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \; \} \end{array}$$

where x and y are new symbols.

The expression in the body of the cardinality rule is referred to as a cardinality constraint with lower and upper bound l and u.

Cardinality Constraints as Heads

Translation

A rule of the form

$$l\ \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}\ u \leftarrow a_{n+1}, \ldots, a_o, \sim a_{o+1}, \ldots, \sim a_p$$

where $0 \leq m \leq n \leq o \leq p$, each a_i is an atom for $1 \leq i \leq p$, and l and u are non-negative integers

is translated into

$$\begin{aligned} x &\leftarrow a_{n+1}, \ldots, a_o, \sim a_{o+1}, \ldots, \sim a_p \\ \{a_1, \ldots, a_m\} &\leftarrow x \\ y &\leftarrow l\ \{a_1, \ldots, a_m, , \sim a_{m+1}, \ldots, \sim a_n\}\ u \\ &\leftarrow x, \sim y \end{aligned}$$

where x and y are new symbols.

Example: `1 {colour(2,red); colour(2,green); colour(2,blue)} 1.`

Conditional Literals

Definition

A **conditional literal** is of the form

$$K : L_1, \dots, L_n$$

where K and L_i are literals for $0 \leq i \leq n$.

Informal meaning: A (non-ground) conditional literal can be regarded as the collection of elements in the set $\{K \mid L_1, \dots, L_n\}$.

Note: The expansion of this collection is context dependent.

Example

Assume '`p(1..3). q(2).`', then '`r(X) : p(X), not q(X)`' yields `r(1)` and `r(3)`.

The constraint

```
:- r(X) : p(X), not q(X); 1 { r(X) : p(X), not q(X) }.
```

is instantiated to

```
:- r(1), r(3), 1 { r(1); r(3) }.
```

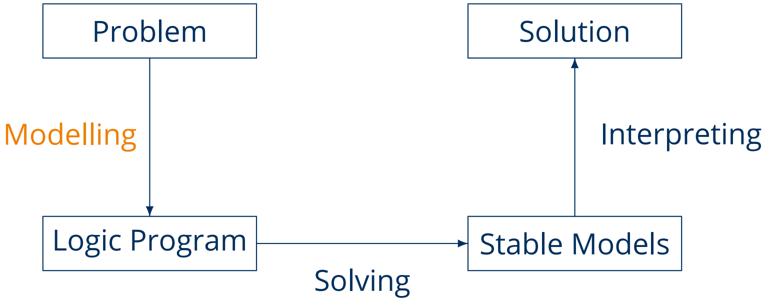
Quiz: Programs with New Constructs

Quiz

Consider the following answer set program P: ...

Modelling

Modelling


Problem
Solution
Modelling
Interpreting
Logic Program
Stable Models
Solving

Guiding principle

Elaboration Tolerance (McCarthy, 1998)

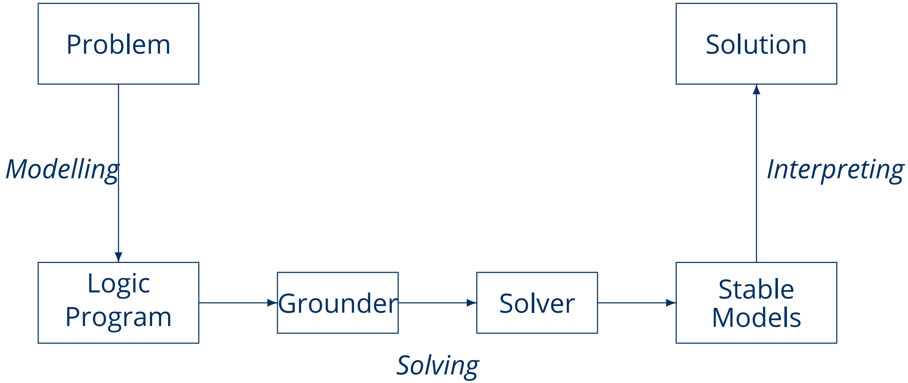
"A formalism is elaboration tolerant [if] *it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances."*

Uniform Problem Representation

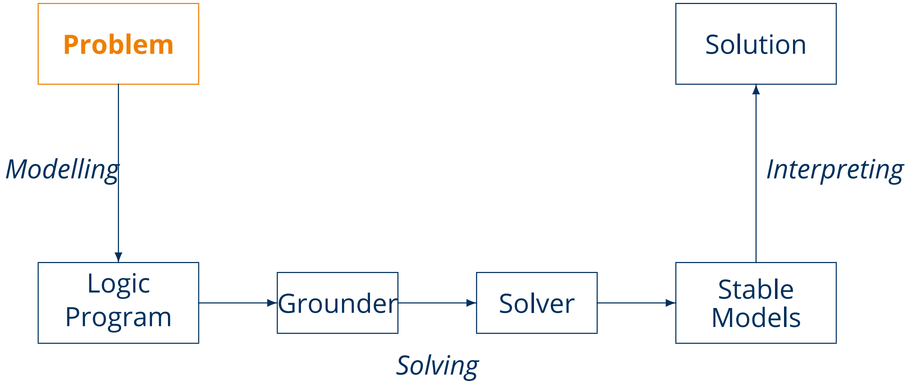
For solving a problem instance **I** of a problem class **C**,

- **I** is represented as a set of facts $P_{\mathbf{I}}$,
- **C** is represented as a set of rules $P_{\mathbf{C}}$, and
- $P_{\mathbf{C}}$ can be used to solve all problem instances in **C**

ASP workflow



ASP workflow: Problem



A Case Study: Graph Colouring

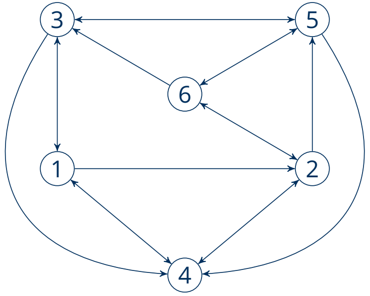
Problem instance:

A graph consisting of nodes and edges:

- facts using predicates `node/1` and `edge/2`
- facts using predicate `colour/1`

Problem class:

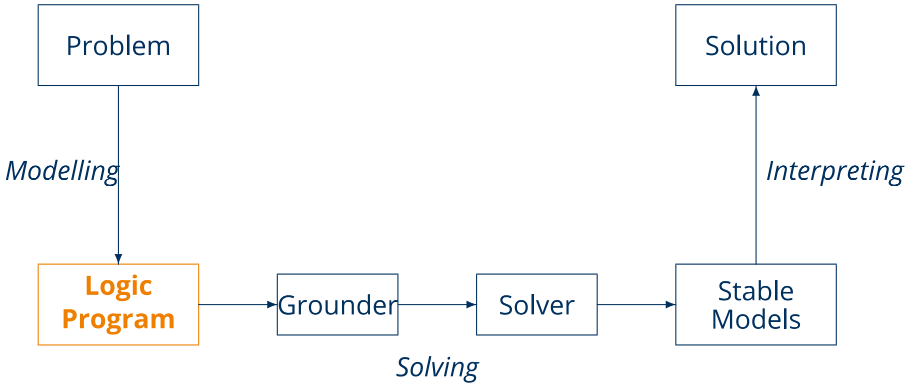
Assign each node one colour such that no two nodes connected by an edge have the same colour.



In other words:

1. Each node has one colour
2. Two connected nodes must not have the same colour

ASP Workflow: Problem Representation



Graph Colouring

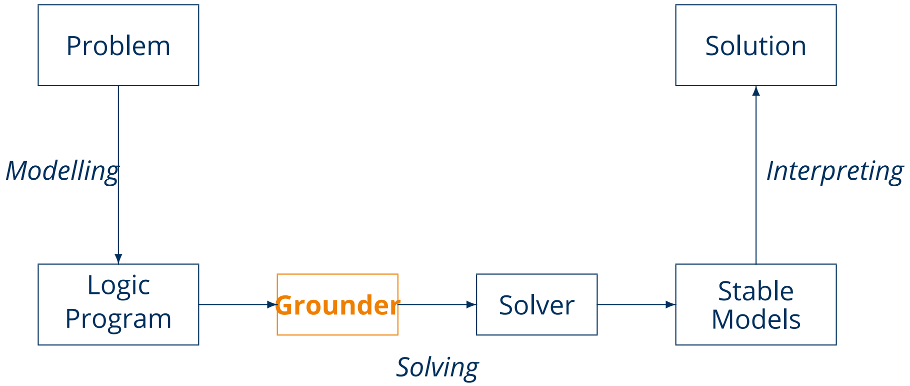
```
node(1..6).

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

colour(r). colour(b). colour(g).
```

Problem instance `graph.lp`

```
1 { assign(N,C) : colour(C) } 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem encoding `colour.lp`

ASP Workflow: Grounding

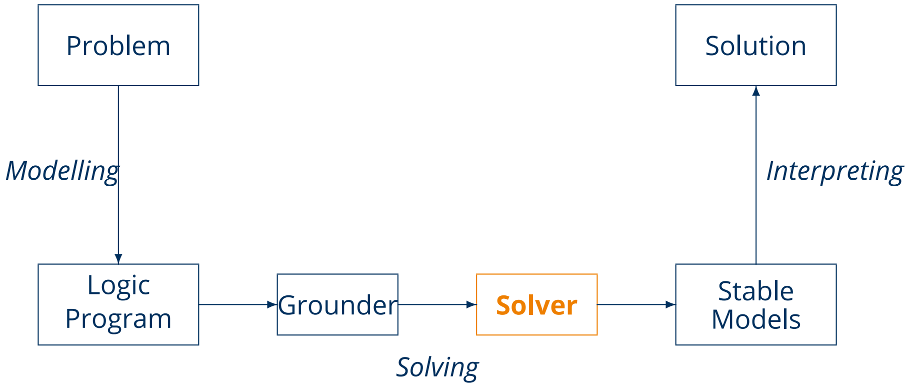


Graph Colouring: Grounding

```
$ clingo −text graph.lp colour.lp
```

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

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

colour(r). colour(b). colour(g).

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

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

ASP Workflow: Solving

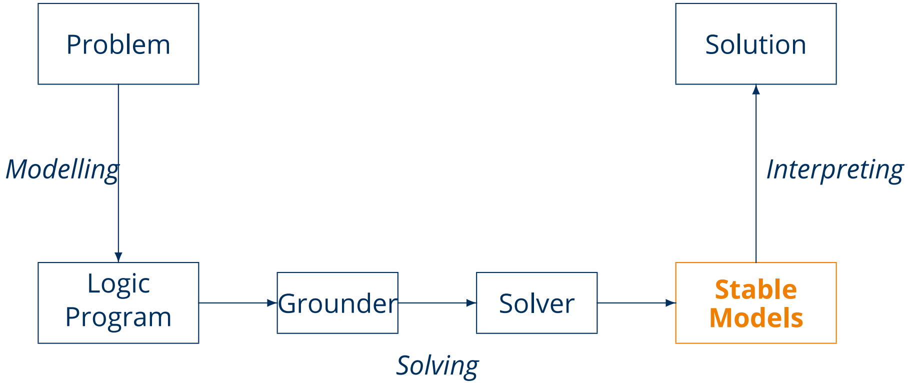


Graph Colouring: Solving

```
$ clingo graph.lp colour.lp 0
```

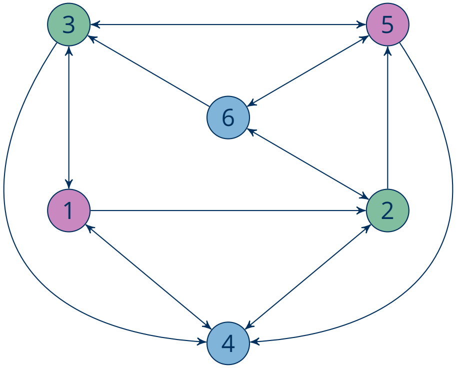
```
   clasp version 2.1.0
   Reading from stdin
   Solving...
   Answer: 1
node(1) [...] assign(6,b) assign(5,g) assign(4,b) assign(3,r) assign(2,r) assign(1,g)
   Answer: 2
node(1) [...] assign(6,r) assign(5,g) assign(4,r) assign(3,b) assign(2,b) assign(1,g)
   Answer: 3
node(1) [...] assign(6,g) assign(5,b)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

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

ASP Workflow: Stable models

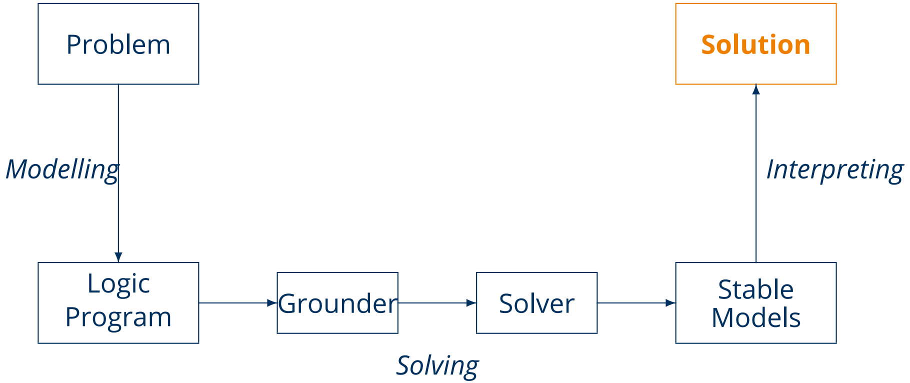


A Colouring

```
Answer: 6
node(1)   [...]    \
assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(1,r)
```



ASP Workflow: Solutions



Basic Methodology

Methodology

Generate and Test (or: Guess and Check)

Generator Generate potential stable model candidates
(typically through non-deterministic constructs)

Tester Eliminate invalid candidates
(typically through integrity constraints)

Nutshell

Logic program = Data + Generator + Tester (+ Optimizer)

Graph Colouring

```
node(1..6).

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

colour(r). colour(b). colour(g).
```

Data

```
1 {assign(N,C) : colour(C) } 1 :- node(N).
```

Generator

```
:- edge(N,M), assign(N,C), assign(M,C).
```

Tester

History (1)

- Early 1970s: definite LPs with unique least models (SLD resolution)
- Default negation: operational semantics only (SLDNF resolution)
- 1978 (Clark): Program completion for normal LPs
- 1980 (Reiter): Default Logic for non-monotonic reasoning
  (one default theory can have zero or more *extensions*)
- 1987 (Bidoit & Froidevaux): Semantics for normal LPs via translation to default logic (effectively first definition of stable model semantics)
- 1988 (Gelfond & Lifschitz): Stable Model semantics
- 1995 (Marek & Truszczyński): DeReS (Default Reasoning System)
  (modelling search problems via default logic; with solver implementation)
- 1996 (Niemelä & Simons): first ASP grounder (lparse) and solver (smodels)
- 1999 (Marek & Truszczyński; Niemelä): ASP paradigm

History (2)

Michael Gelfond (b. 1945)

- Russian-American mathematician/computer scientist
- PhD in Mathematics from Steklov Institute (1974)
- emigrated to the US in 1978
- stable model semantics, KR languages
- AAAI Fellow

(C) Michael Gelfond

Vladimir Lifschitz (b. 1947)

- Russian-American mathematician/computer scientist
- PhD in Mathematics from Steklov Institute (1971)
- emigrated to the US in 1976
- stable model semantics, KR languages

(C) Vladimir Lifschitz

History (3)

Victor Witold (Witek) Marek (b. 1943)

- Polish-American mathematician/computer scientist
- PhD from Warsaw University (1968)
- Non-monotonic reasoning, ASP paradigm

(C) Victor Marek

Mirosław (Mirek) Truszczyński (b. 1950s?)

- Polish-American mathematician/computer scientist
- PhD from Warsaw University of Technology (1980)
- Non-monotonic reasoning, ASP paradigm
- AAAI Fellow (2013), Dov Gabbay Prize (2023)

(C) Mirosław Truszczyński

History (4)

Ilkka Niemelä (b. 1961)

- Finnish computer scientist
- PhD from Helsinki University of Technology (1993)
- Non-monotonic reasoning, ASP paradigm
- co-developed (with Patrik Simons) the first ASP grounder (lparse) and solver (smodels)

(C) Ilkka Niemelä

- Marek & Truszczyński (1999):
  Stable models and an alternative logic programming paradigm
- Niemelä (1999):
  Logic programming with stable model semantics as a constraint programming paradigm

Conclusion

Summary

- The language of normal logic programs can be extended by constructs:
  - **Integrity constraints** for eliminating unwanted solution candidates
  - **Choice rules** for choosing subsets of atoms
  - **Cardinality rules** for counting certain present/absent atoms
  - **Conditional literals** for improving conciseness
- All of them can be translated back into normal logic program rules.
- The modelling methodology of ASP is **generate and test**:

  Generate solution candidates & eliminate infeasible ones

Suggested action points:

- Model solving Sudoku puzzles using a ternary predicate `num`(i, j, k) expressing that the field in row i and column j of the Sudoku grid contains the number k ($i, j, k \in \{1, \ldots, 9\}$). Initial hints are given by `num/3` facts.