MARIO ALVIANO

DeMaCS@UNICAL

UNIVERSITÀ
DELLA
CALABRIA

il Campus per eccellenza

Deep Reasoning in AI with ASP

Joint effort with Francesco Ricca

FAIR Future Artificial Intelligence Research

PRODE Probabilistic declarative process mining

ESSAI & ACAI 2024
ATHENS - GREECE

TECH4YOU The more you change, the less climate changes.

CYBER CHALLENGE.IT

SERICS SECURITY AND RIGHTS IN THE CYBERSPACE

LAIA intelligenza artificiale

SISTEMA INTEGRATO
DI LABORATORI
PER L'AMBIENTE
SILA

https://asp-chef.alviano.net/s/dimacs-to-truth-table

https://asp-chef.alviano.net/s/tutorials/essai2024/1

https://asp-chef.alviano.net/s/tutorials/essai2024/2

Suggested Book

Potassco Guide

https://github.com/potassco/guide/releases/download/v2.2.0/guide.pdf

L2

https://asp-chef.alviano.net/s/asp-playground

https://asp-chef.alviano.net/s/grounding-playground

https://asp-chef.alviano.net/s/tutorials/basic-usage

https://asp-chef.alviano.net/s/tutorials/aquarium https://asp-chef.alviano.net/s/tutorials/billy-the-kid

https://asp-chef.alviano.net/s/tutorials/fortress

https://asp-chef.alviano.net/s/tutorials/essai2024/3

Structured Declarative Language (SDL)

Explainability Issues

ASP and LLMs interaction

L5

Structured Declarative Language

https://asp-chef.alviano.net/s/SDL

GitHub Repository
https://github.com/dodaro/SDL

Declarative Programming

Expresses the **logic** of a computation without describing its **control flow**

Declarative Programming

Expresses the **logic** of a computation without describing its **control flow**

Answer Set Programming is (at its core) declarative

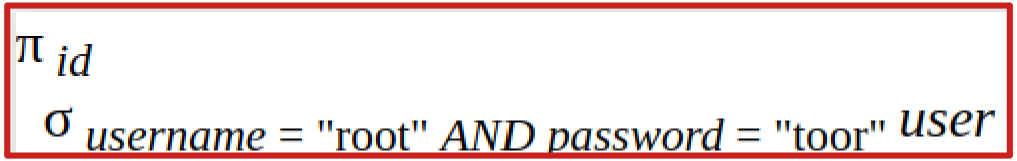
Declarative Programming

Expresses the **logic** of a computation without describing its **control flow**

Answer Set Programming is (at its core) declarative

But is ASP a
good speaker?

Relational Algebra vs Structured Query Language (SQL)



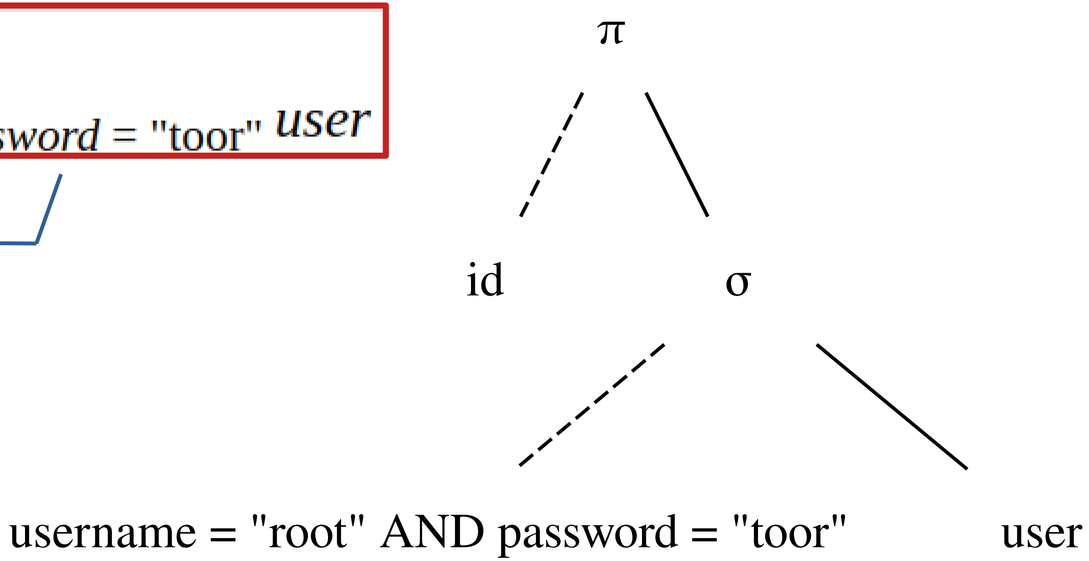
$$\pi_{id}\ \sigma_{username = \text{"root"}\ AND\ password = \text{"toor"}}\ user$$

Relational Algebra vs Structured Query Language (SQL)

$$\pi_{id}\ \sigma_{username = \text{"root"}\ AND\ password = \text{"toor"}}\ user$$

OK in papers (for compactness)

OK for the engine (to optimize)


π

id

σ

username = "root" AND password = "toor"

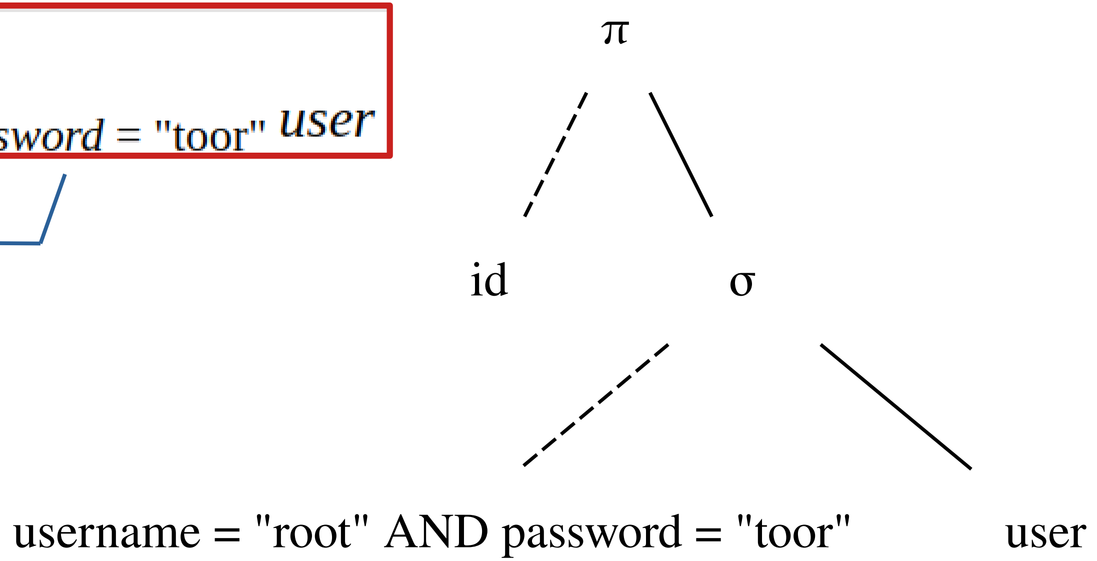
user

Relational Algebra vs Structured Query Language (SQL)

$$\pi_{\,id}\ \sigma_{\,username\ =\ \text{"root"}\ AND\ password\ =\ \text{"toor"}}\ user$$

OK in papers (for compactness)

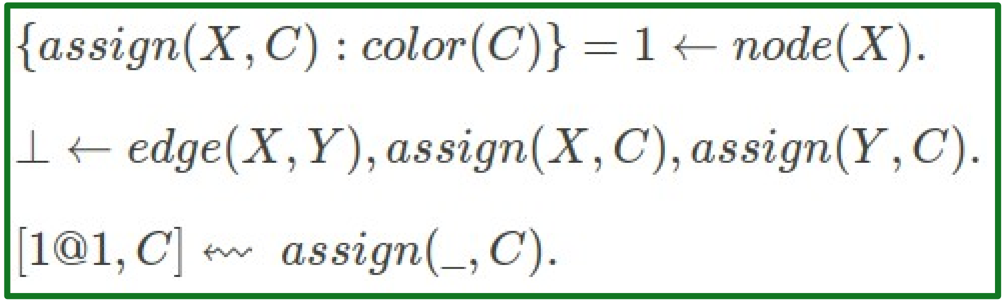
OK for the engine (to optimize)



But programmers prefer to express themselves this way

```
SELECT id
FROM user
WHERE username = 'root' AND password = 'toor'
```

ASP for Papers vs ASP for Programmers

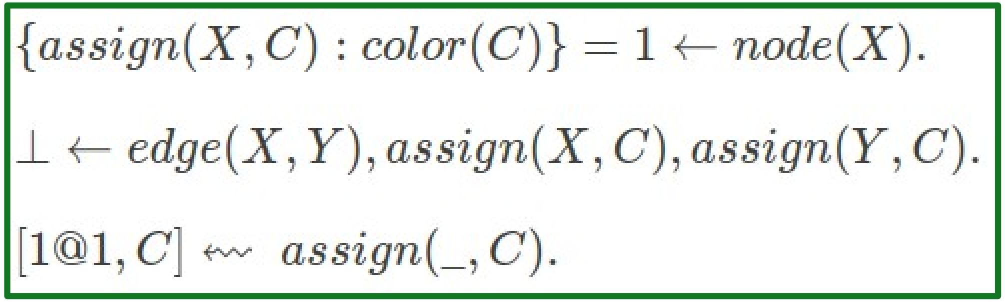


$$\{assign(X,C) : color(C)\} = 1 \leftarrow node(X).$$

$$\bot \leftarrow edge(X,Y), assign(X,C), assign(Y,C).$$

$$[1@1, C] \leftsquigarrow\ assign(\_, C).$$

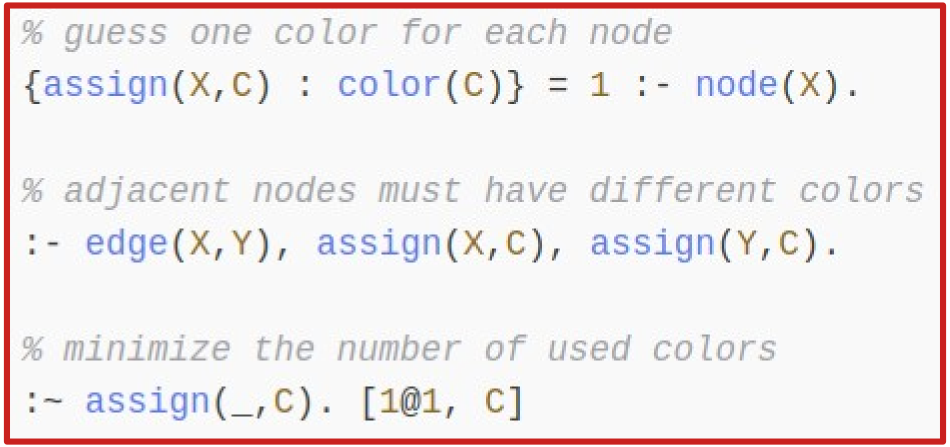
ASP for Papers vs ASP for Programmers



$$\{assign(X,C) : color(C)\} = 1 \leftarrow node(X).$$

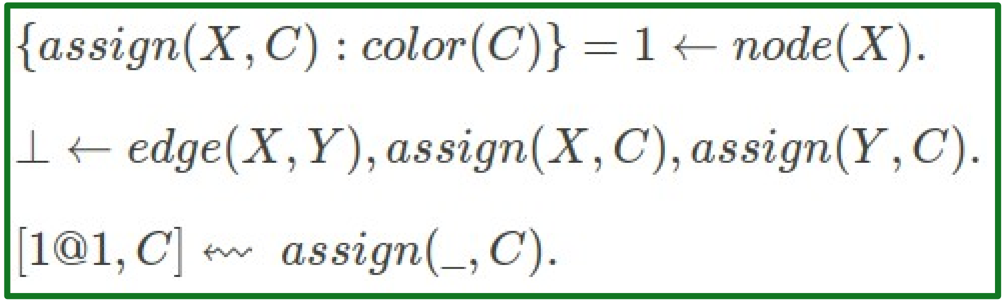
$$\bot \leftarrow edge(X,Y), assign(X,C), assign(Y,C).$$

$$[1@1, C] \leftsquigarrow assign(\_, C).$$



```
% guess one color for each node
{assign(X,C) : color(C)} = 1 :- node(X).

% adjacent nodes must have different colors
:- edge(X,Y), assign(X,C), assign(Y,C).

% minimize the number of used colors
:~ assign(_,C). [1@1, C]
```

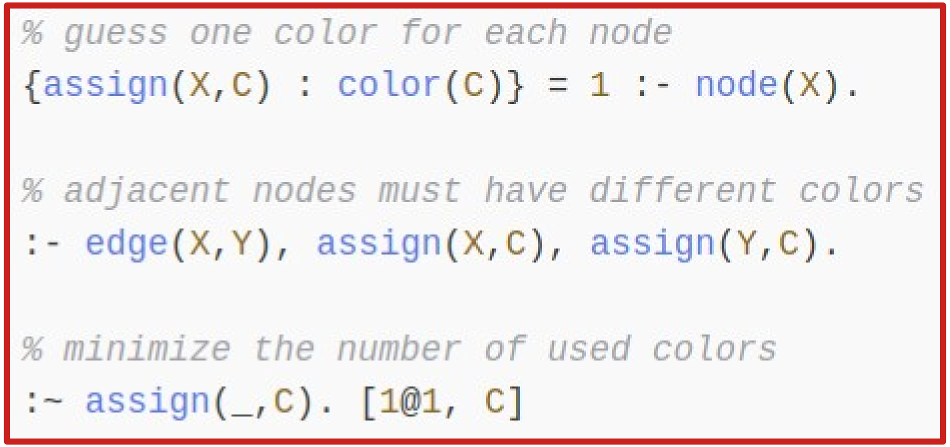
ASP for Papers vs ASP for Programmers



$$\{assign(X,C) : color(C)\} = 1 \leftarrow node(X).$$

$$\bot \leftarrow edge(X,Y), assign(X,C), assign(Y,C).$$

$$[1@1, C] \leftsquigarrow assign(\_, C).$$



```
% guess one color for each node
{assign(X,C) : color(C)} = 1 :- node(X).

% adjacent nodes must have different colors
:- edge(X,Y), assign(X,C), assign(Y,C).

% minimize the number of used colors
:~ assign(_,C). [1@1, C]
```

But... they are essentially

the same!

The SAME… but for Different Purposes!

In a paper, you want to be concise

The SAME… but for Different Purposes!

In a paper, you want to be concise

But a long-standing codebase needs maintenance, readability, and robustness

The SAME… but for Different Purposes!

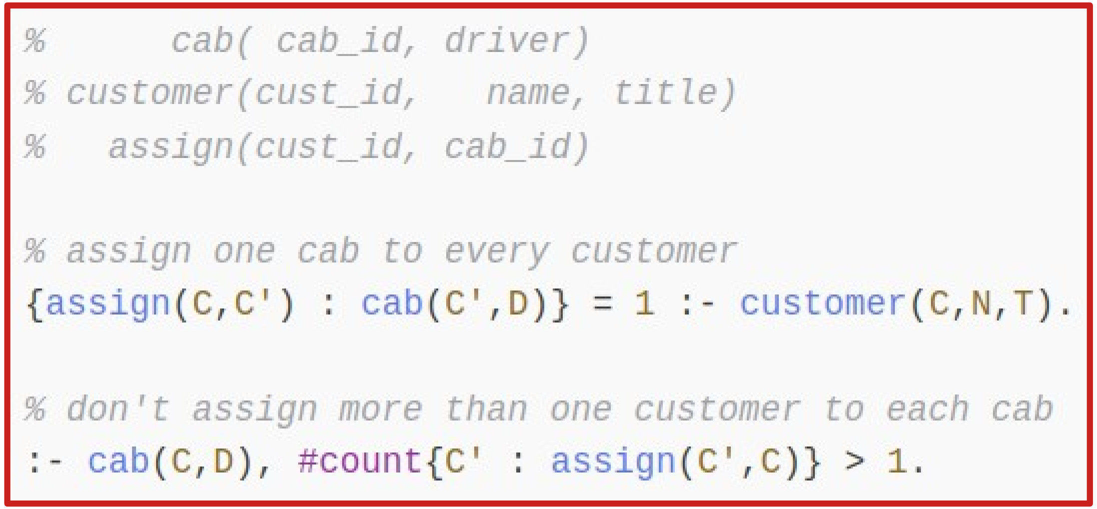
In a paper, you want to be concise

But a long-standing codebase needs maintenance, readability, and robustness

Three downsides of long-standing ASP codebases

1. **Propagating Changes**
2. **Use of Object Variables**
3. **Lack of Semantic Annotations**

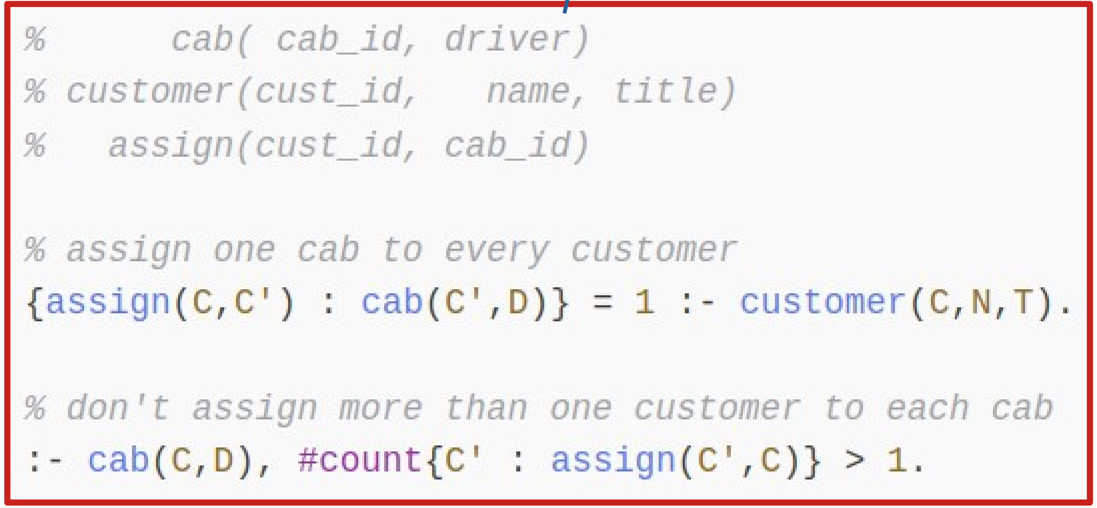
ASP Example



```
%      cab( cab_id, driver)
% customer(cust_id,   name, title)
%   assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

ASP Example

If driver is moved in a different predicate, both rules HAVE TO be changed!

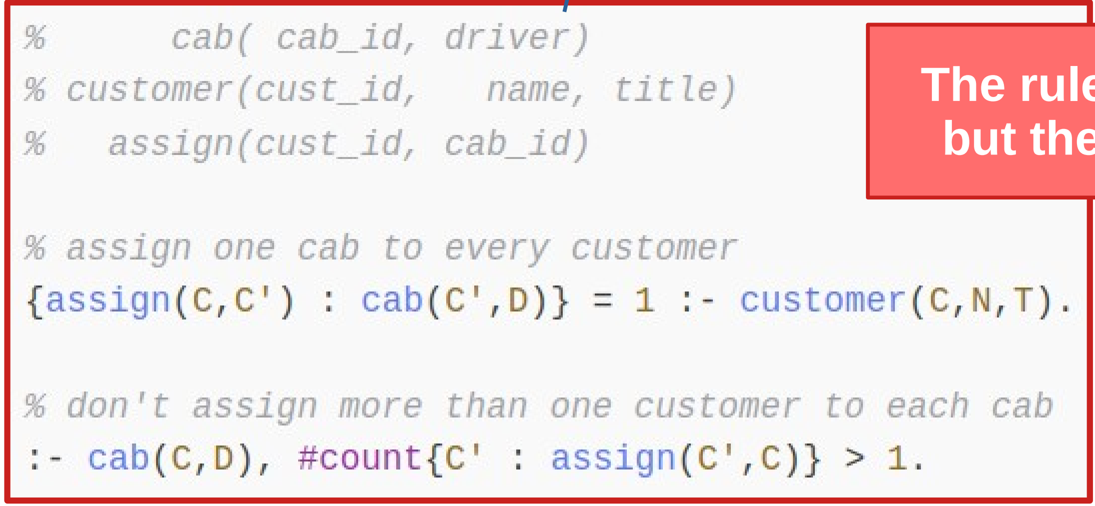


```
%      cab( cab_id, driver)
% customer(cust_id,   name, title)
%   assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

ASP Example

If driver is moved in a different predicate, both rules HAVE TO be changed!

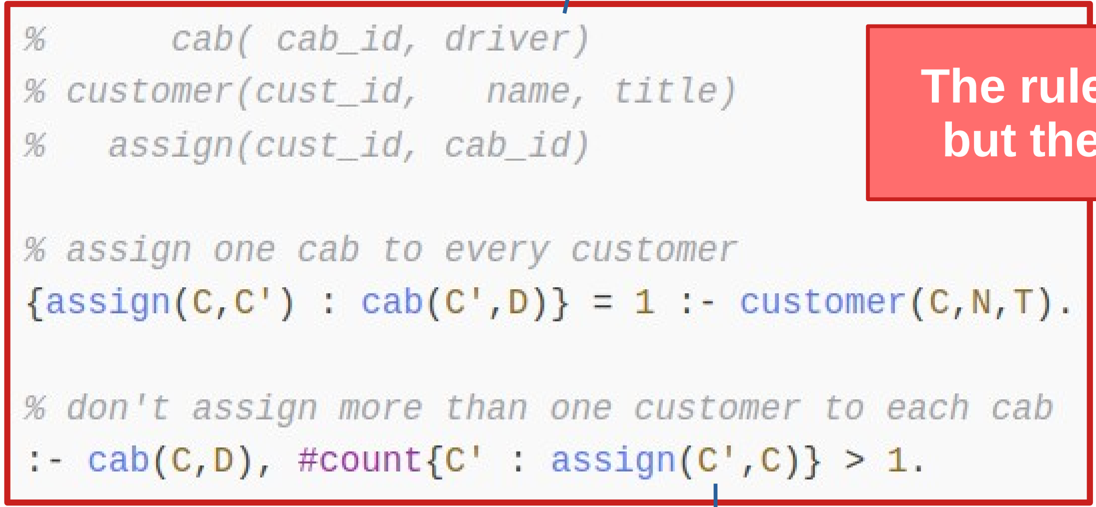
The rules don’t use the driver… but the driver affect the rules!



```
%      cab( cab_id, driver)
% customer(cust_id,   name, title)
%   assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

ASP Example

If driver is moved in a different predicate, both rules HAVE TO be changed!



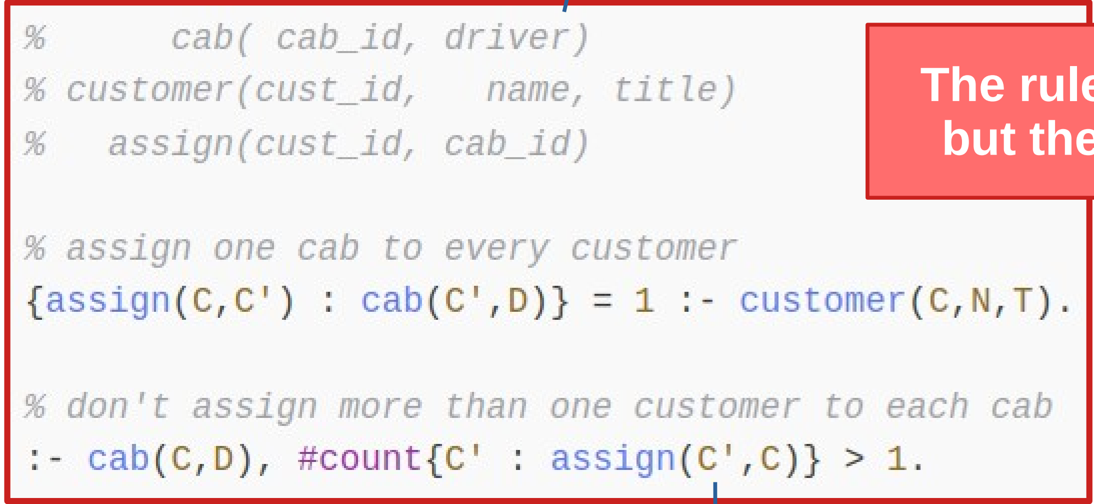
```
%      cab( cab_id, driver)
% customer(cust_id,   name, title)
%   assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

The rules don’t use the driver… but the driver affect the rules!

Object variables are often shorts and used inconsistently (e.g., C and C’)

ASP Example

If driver is moved in a different predicate, both rules HAVE TO be changed!



```
%       cab( cab_id, driver)
% customer(cust_id,    name, title)
%   assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

The rules don't use the driver… but the driver affect the rules!

Object variables are often shorts and used inconsistently (e.g., C and C')

BTW… is assign(cust, cab) or assign(cab, cust)? There is no semantic annotation within the syntax!

Structured Declarative Language (SDL)

Like SQL… but mapping to ASP!

Structured Declarative Language (SDL)

Like SQL… but mapping to ASP!

Abstraction and Simplification

Problems are expressed in a language closer to English

Structured Declarative Language (SDL)

Like SQL… but mapping to ASP!

Abstraction and Simplification

Problems are expressed in a language closer to English

Qualifying Names for Attribute Access

Dot notation, as in OOP!
No order of attributes,
no object variables

Structured Declarative Language (SDL)

Like SQL… but mapping to ASP!

Abstraction and Simplification

Problems are expressed in a language closer to English

Qualifying Names for Attribute Access

Dot notation, as in OOP!
No order of attributes,
no object variables

Automatic Attribute Tracking

References to records are tracked to ease debugging, and to avoid comparing apples and oranges

Structured Declarative Language (SDL)

Like SQL… but mapping to ASP!

Abstraction and Simplification

Problems are expressed in a language closer to English

Qualifying Names for Attribute Access

Dot notation, as in OOP!
No order of attributes,
no object variables

Automatic Attribute Tracking

References to records are tracked to ease debugging, and to avoid comparing apples and oranges

SDL maps to semantically annotated atoms

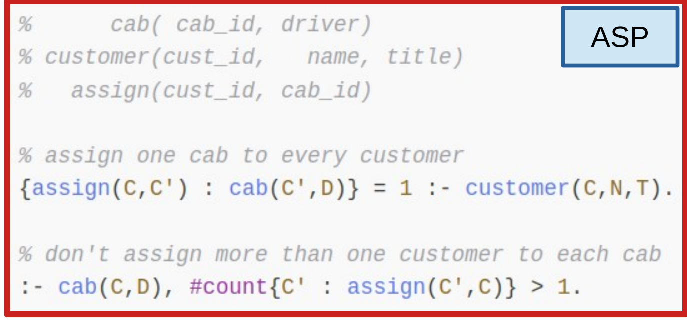
```
assign(
  customer(CUST_ID, NAME, TITLE),
  cab(CAB_ID, DRIVER)
)
```

vs

```
assign(C,C')
```

SDL Example

ASP

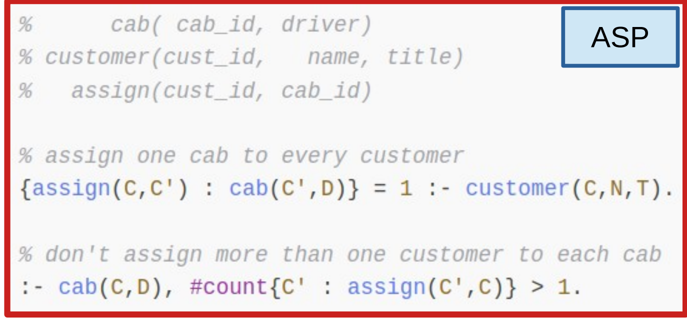


```
%       cab( cab_id, driver)
% customer(cust_id,   name, title)
%   assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

```
record Cab:        id: int, driver: str;
record Customer:   id: int, name: str, title: str;
record Assign:     customer: Customer, cab: Cab;

guess from Customer exactly 1
    Assign from Cab
        where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
    count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

SDL Example

ASP

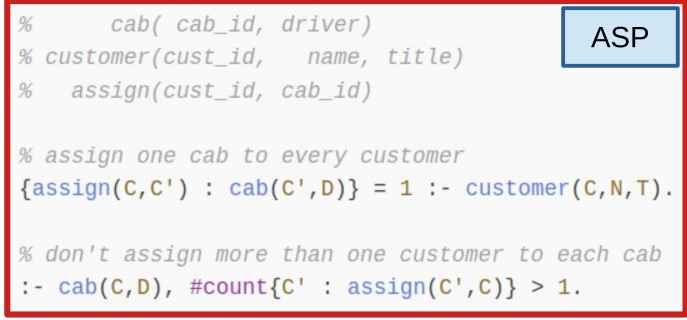


```
%       cab( cab_id, driver)
% customer(cust_id,   name, title)
%   assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

A Cab has attributes id (an integer) and driver (a string)

```
record Cab:        id: int, driver: str;
record Customer:   id: int, name: str, title: str;
record Assign:     customer: Customer, cab: Cab;

guess from Customer exactly 1
    Assign from Cab
        where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
    count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

SDL Example


ASP

```
%      cab( cab_id, driver)
% customer(cust_id,   name, title)
%   assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

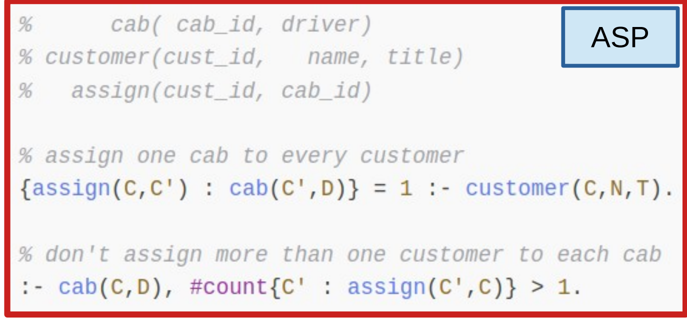
A Cab has attributes
id (an integer) and
driver (a string)

Assign has attributes
customer (being a Customer) and
cab (being a Cab)

```
record Cab:        id: int, driver: str;
record Customer:   id: int, name: str, title: str;
record Assign:     customer: Customer, cab: Cab;

guess from Customer exactly 1
    Assign from Cab
        where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
    count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

SDL Example

ASP



```
%       cab( cab_id, driver)
% customer(cust_id,   name, title)
%   assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

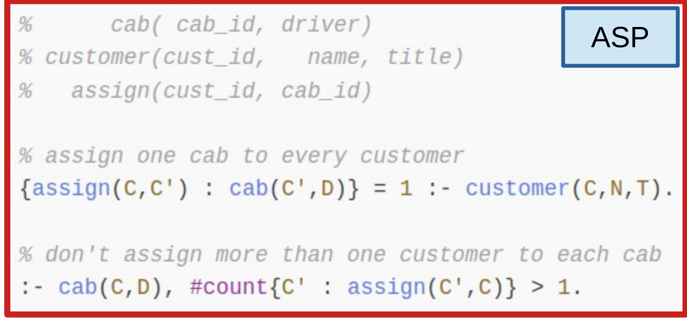
A Cab has attributes
id (an integer) and
driver (a string)

Assign has attributes
customer (being a Customer) and
cab (being a Cab)

```
record Cab:         id: int, driver: str;
record Customer:    id: int, name: str, title: str;
record Assign:      customer: Customer, cab: Cab;

guess from Customer exactly 1
    Assign from Cab
        where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
    count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

Assign one cab to every customer

SDL Example



ASP

```
%      cab( cab_id, driver)
% customer(cust_id,   name, title)
%   assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

A Cab has attributes id (an integer) and driver (a string)

Assign has attributes customer (being a Customer) and cab (being a Cab)

```
record Cab:       id: int, driver: str;
record Customer:  id: int, name: str, title: str;
record Assign:    customer: Customer, cab: Cab;

guess from Customer exactly 1
    Assign from Cab
        where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
    count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

Assign one cab to every customer

Don’t assign more than one customer to each cab

Just two little observations...

```
record Cab:       id: int, driver: str;
record Customer:  id: int, name: str, title: str;
record Assign:    customer: Customer, cab: Cab;

guess from Customer exactly 1
    Assign from Cab
        where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
    count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

Just two little observations...

```
record Cab:        id: int, driver: str;
record Customer:   id: int, name: str, title: str;
record Assign:     customer: Customer, cab: Cab;

guess from Customer exactly 1
    Assign from Cab
        where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
    count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

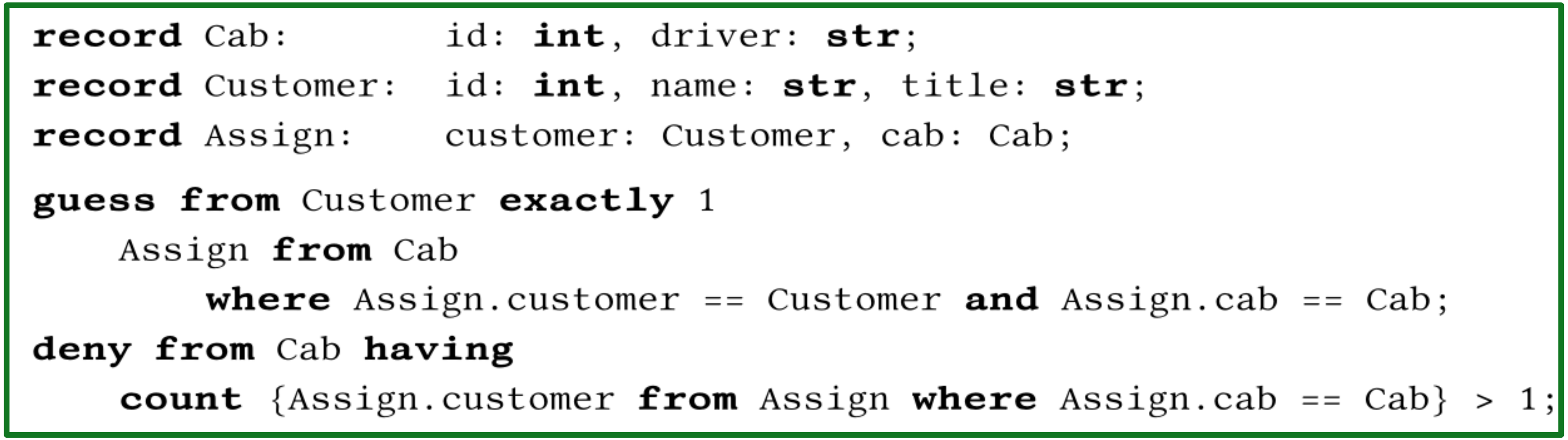
Doesn't use the driver… and
has no idea about the driver!

Just two little observations...

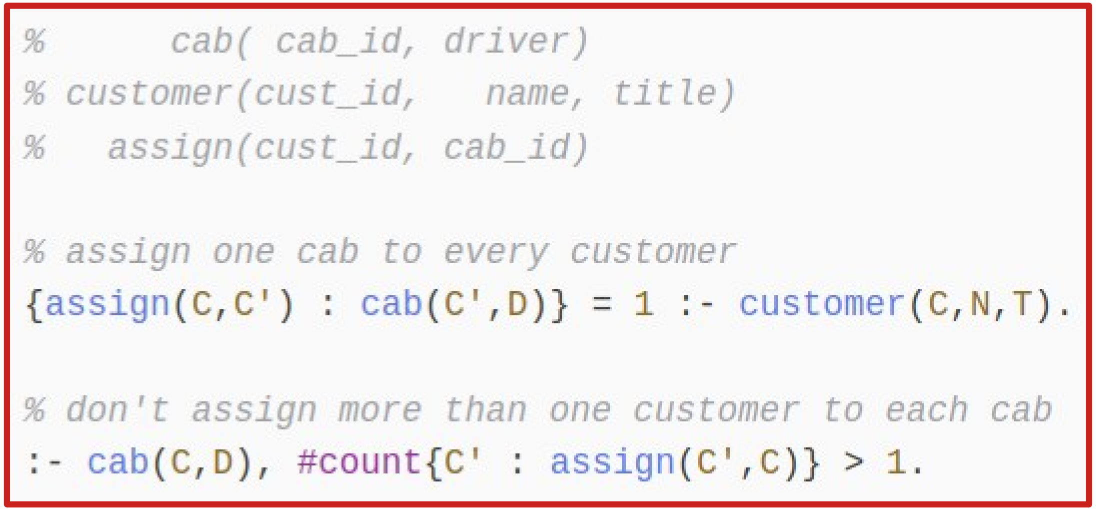
```
record Cab:        id: int, driver: str;
record Customer:   id: int, name: str, title: str;
record Assign:     customer: Customer, cab: Cab;

guess from Customer exactly 1
    Assign from Cab
        where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
    count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

Doesn’t use the driver… and has no idea about the driver!

Attributes accessed by name, not by position!



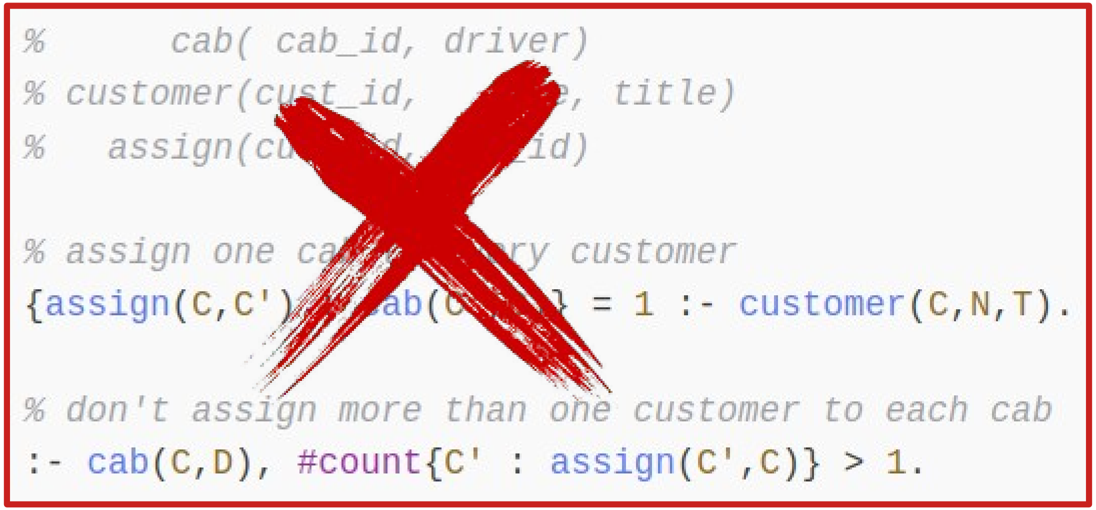
```
record Cab:        id: int, driver: str;
record Customer:   id: int, name: str, title: str;
record Assign:     customer: Customer, cab: Cab;

guess from Customer exactly 1
    Assign from Cab
        where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
    count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```



```
%      cab( cab_id, driver)
% customer(cust_id,   name, title)
%   assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

```
record Cab:        id: int, driver: str;
record Customer:   id: int, name: str, title: str;
record Assign:     customer: Customer, cab: Cab;

guess from Customer exactly 1
    Assign from Cab
        where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
    count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

```
{assign(
    customer(ID,NAME,TITLE),
    cab(ID,DRIVER)
 ) : cab(ID,DRIVER)} = 1 :-
  customer(ID,NAME,TITLE).

:- cab(ID,DRIVER), #count{
    customer(ID,NAME,TITLE) :
    assign(
      customer(ID,NAME,TITLE),
      cab(ID,DRIVER)
    )
  } > 1.
```



```
%       cab( cab_id, driver)
% customer(cust_id, [illegible]e, title)
%   assign(cu[illegible]d, [illegible]id)

% assign one ca[illegible]ry customer
{assign(C,C') [illegible]ab(C[illegible]} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

Structure Instruction

$$\textbf{\texttt{record}}\ \mathit{RecordName}\colon \mathit{Attributes};$$

List of **name: type** pairs

int, str or a record name

Structure Instruction

$$\textbf{\texttt{record}}\ \mathit{RecordName}\colon \mathit{Attributes};$$

List of **name: type** pairs

int, str or a record name

Everything MUST be declared
(SDL targets long-standing codebases)

Structure Instruction

$$\texttt{\textbf{record}}\ \mathit{RecordName}\colon \mathit{Attributes};$$

List of **name: type** pairs

int, str or a record name

Everything MUST be declared
(SDL targets long-standing codebases)

Acyclicity of structure instructions is required and checked

Query Instruction

$$\texttt{\textbf{show}}\ RecordNames;$$

List of record names

Query Instruction

$$\mathtt{\textbf{show}}\ \mathit{RecordNames};$$

List of record names

We may include more expressive query instructions in the future, but for now… KISS!

Query Instruction

$$\textbf{show}\ \mathit{RecordNames};$$

List of record names

We may include more expressive query instructions in the future, but for now… KISS!

By default, we only show SAT/UNSAT...

Query Instruction

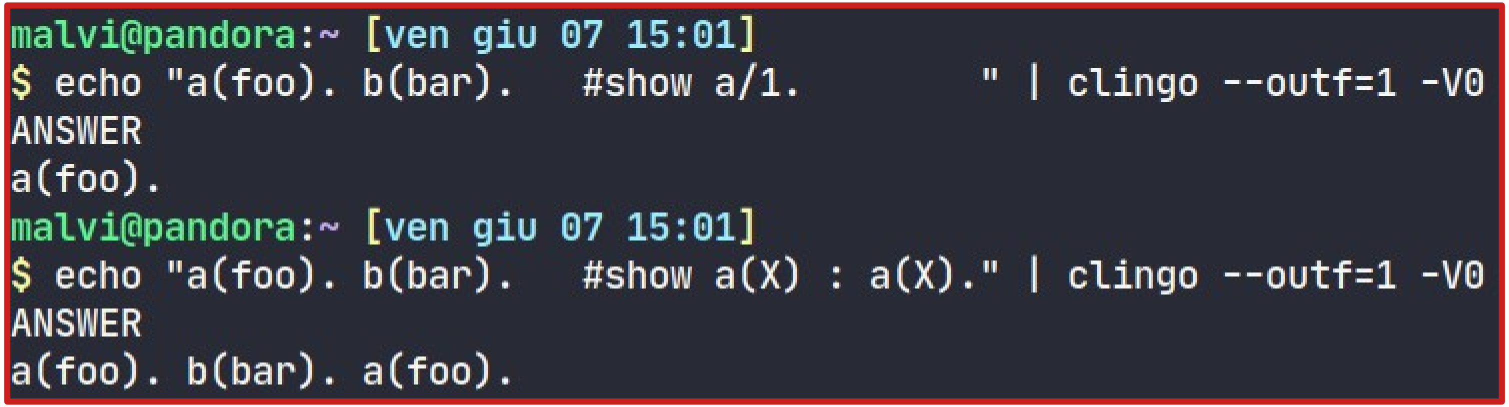
show *RecordNames*;

List of record names

We may include more expressive query instructions in the future, but for now… KISS!

By default, we only show SAT/UNSAT...

to avoid any ambiguity



```
malvi@pandora:~ [ven giu 07 15:01]
$ echo "a(foo). b(bar).   #show a/1.         " | clingo --outf=1 -V0
ANSWER
a(foo).
malvi@pandora:~ [ven giu 07 15:01]
$ echo "a(foo). b(bar).   #show a(X) : a(X)." | clingo --outf=1 -V0
ANSWER
a(foo). b(bar). a(foo).
```

Model Instructions (several)

Refer records by defining aliases (optional);
prepend **not** to refer the complement

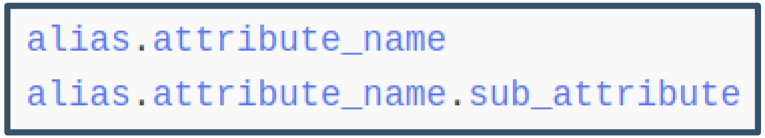
$$RecordName\ \mathbf{as}\ alias$$

Model Instructions (several)

Refer records by defining aliases (optional); prepend **not** to refer the complement

$$\mathit{RecordName}\ \textbf{as}\ \mathit{alias}$$

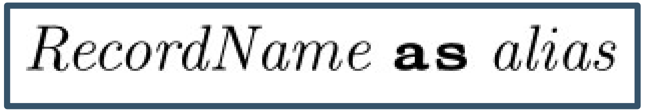
Attributes are accessed with dot operations, as in OOP languages



```
alias.attribute_name
alias.attribute_name.sub_attribute
```

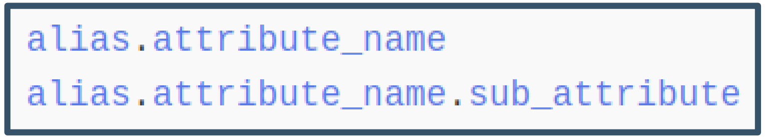
Model Instructions (several)

Refer records by defining aliases (optional);
prepend **not** to refer the complement



$RecordName$ **as** $alias$

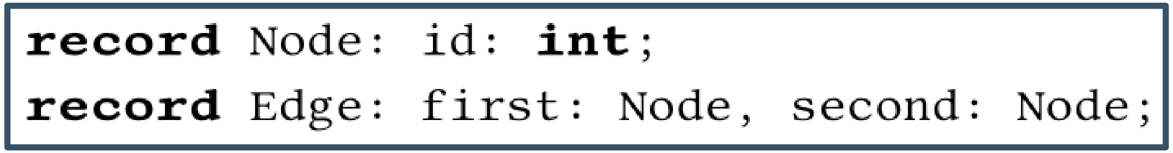
Attributes are accessed with dot operations,
as in OOP languages



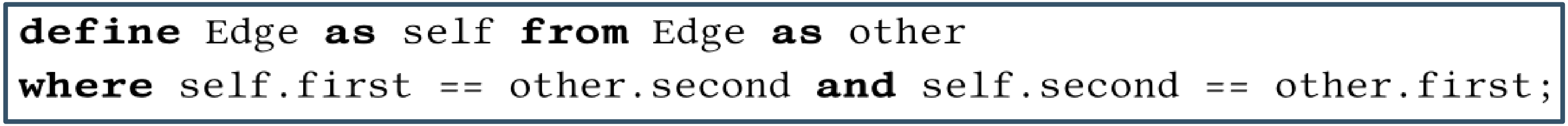
```
alias.attribute_name
alias.attribute_name.sub_attribute
```

Values (of attributes, or constants)
can be combined in expressions

Model Instructions: Definitions



```
record Node: id: int;
record Edge: first: Node, second: Node;
```

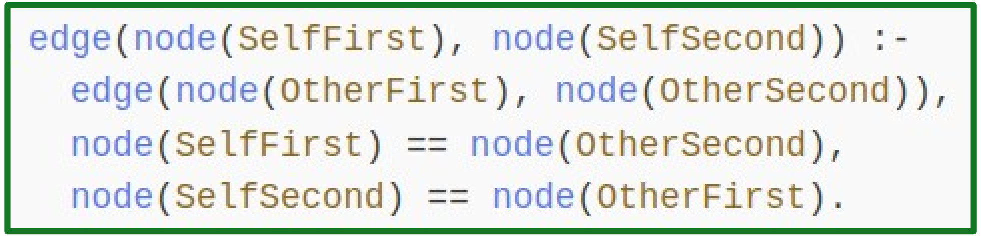


```
define Edge as self from Edge as other
where self.first == other.second and self.second == other.first;
```

Model Instructions: Definitions

```
record Node: id: int;
record Edge: first: Node, second: Node;
```

```
define Edge as self from Edge as other
where self.first == other.second and self.second == other.first;
```

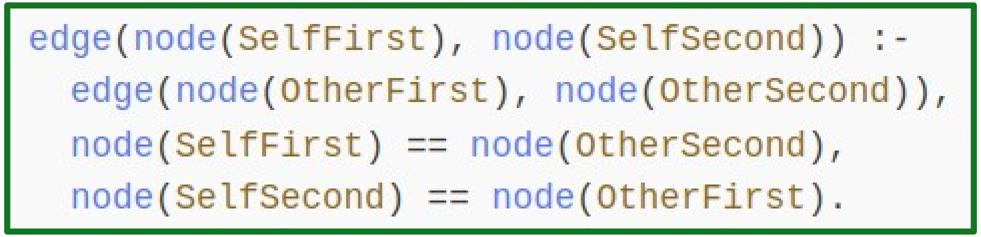


```
edge(node(SelfFirst), node(SelfSecond)) :-
  edge(node(OtherFirst), node(OtherSecond)),
  node(SelfFirst) == node(OtherSecond),
  node(SelfSecond) == node(OtherFirst).
```

Model Instructions: Definitions

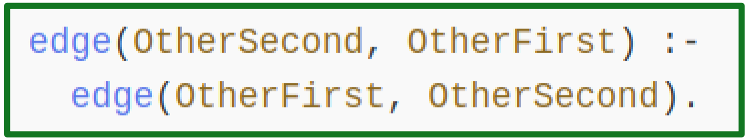
```
record Node: id: int;
record Edge: first: Node, second: Node;
```

```
define Edge as self from Edge as other
where self.first == other.second and self.second == other.first;
```



```
edge(node(SelfFirst), node(SelfSecond)) :-
  edge(node(OtherFirst), node(OtherSecond)),
  node(SelfFirst) == node(OtherSecond),
  node(SelfSecond) == node(OtherFirst).
```

Essentially, the same of



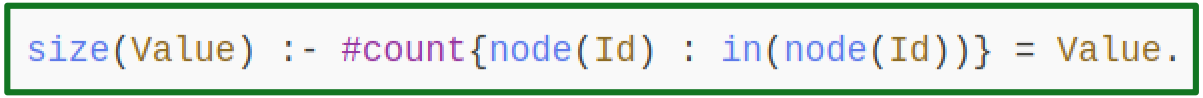
```
edge(OtherSecond, OtherFirst) :-
  edge(OtherFirst, OtherSecond).
```

```
record Node: id: int;
record Edge: first: Node, second: Node;
record In:   node: Node;
record Size: value: int;
```

```
define Size having count {In.node from In} == Size.value;
```

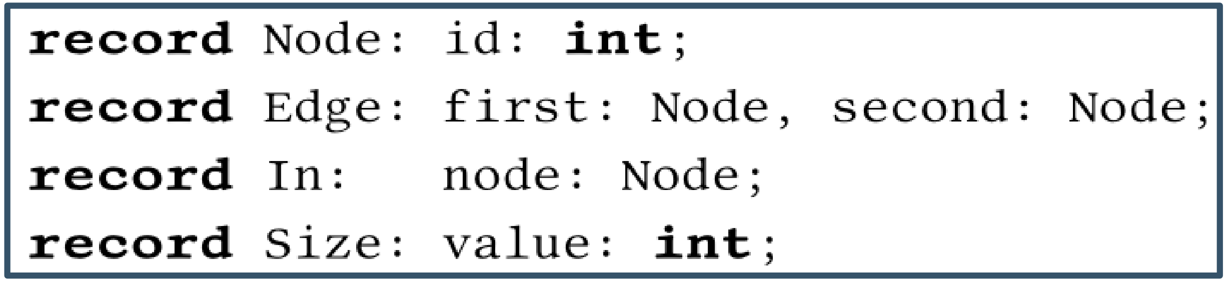
```
record Node: id: int;
record Edge: first: Node, second: Node;
record In:   node: Node;
record Size: value: int;
```

```
define Size having count {In.node from In} == Size.value;
```

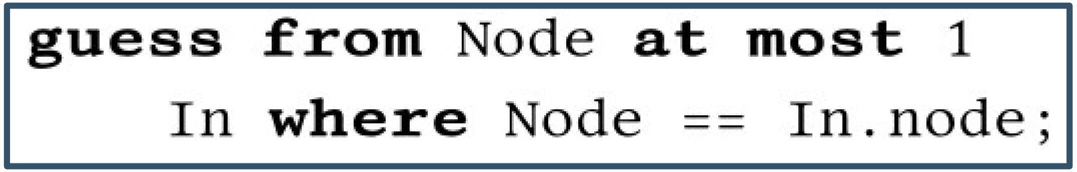


```
size(Value) :- #count{node(Id) : in(node(Id))} = Value.
```

Model Instructions: Guesses

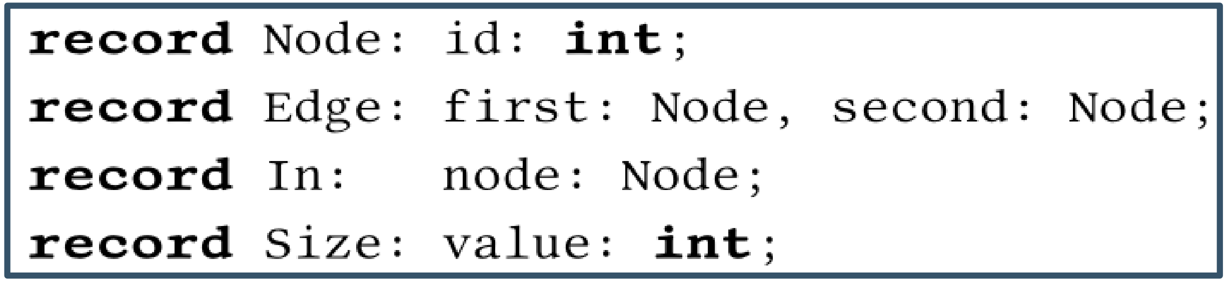


```
record Node: id: int;
record Edge: first: Node, second: Node;
record In:   node: Node;
record Size: value: int;
```

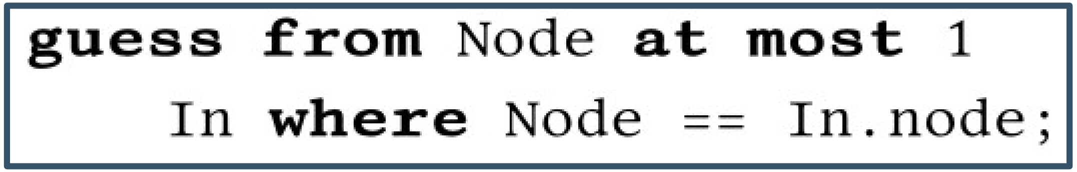


```
guess from Node at most 1
    In where Node == In.node;
```

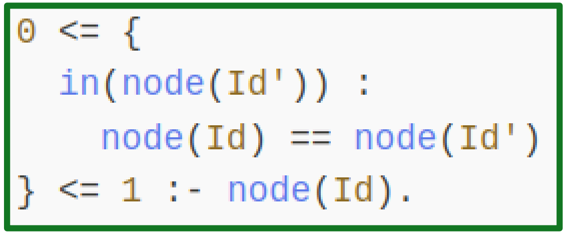
Model Instructions: Guesses



```
record Node: id: int;
record Edge: first: Node, second: Node;
record In:   node: Node;
record Size: value: int;
```

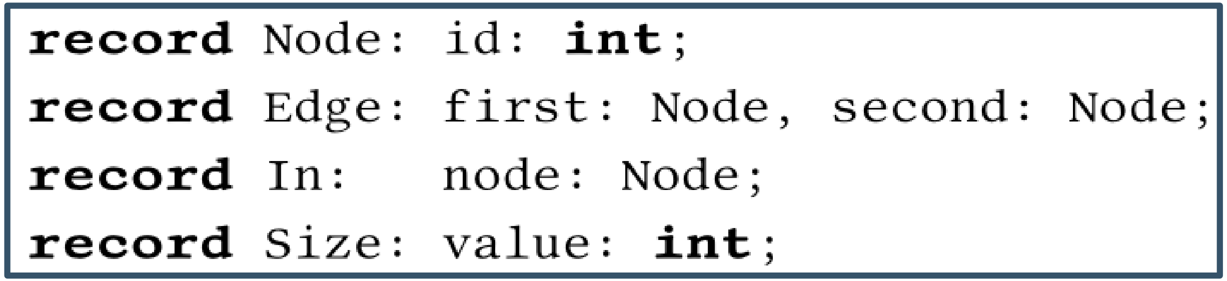


```
guess from Node at most 1
    In where Node == In.node;
```

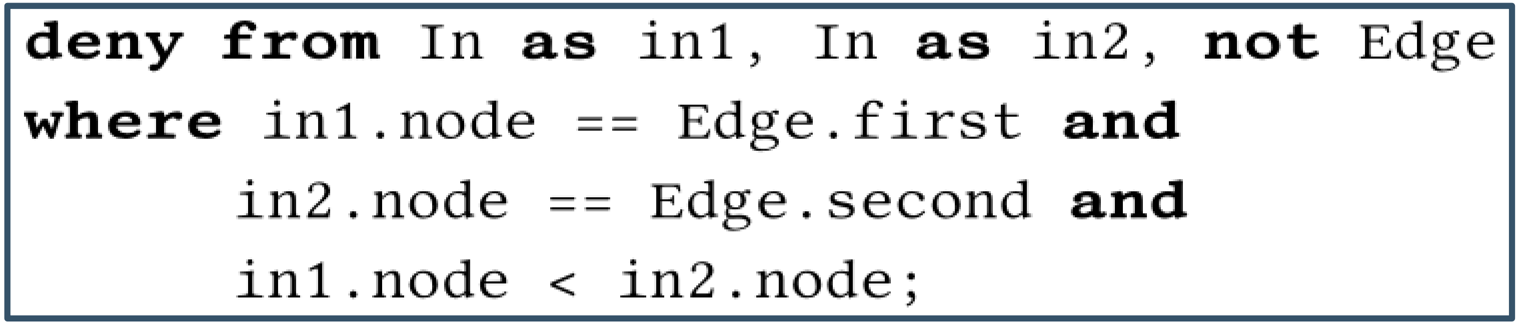


```
0 <= {
  in(node(Id')) :
    node(Id) == node(Id')
} <= 1 :- node(Id).
```

Model Instructions: Denies



```
record Node: id: int;
record Edge: first: Node, second: Node;
record In:   node: Node;
record Size: value: int;
```



```
deny from In as in1, In as in2, not Edge
where in1.node == Edge.first and
      in2.node == Edge.second and
      in1.node < in2.node;
```

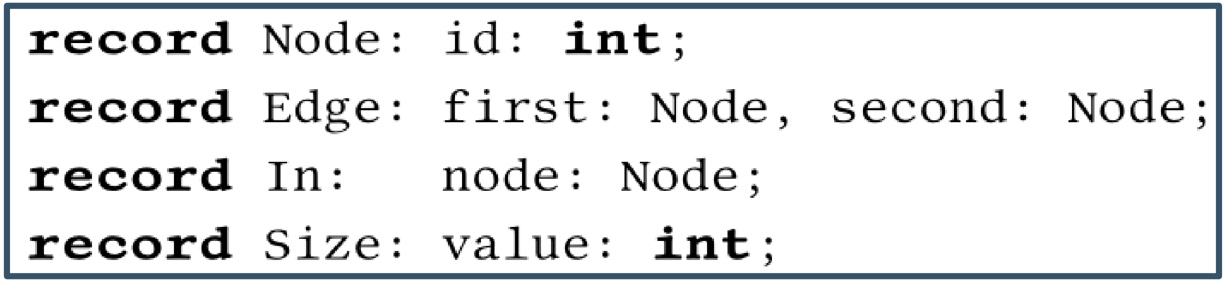
Model Instructions: Denies

```
record Node: id: int;
record Edge: first: Node, second: Node;
record In:   node: Node;
record Size: value: int;
```

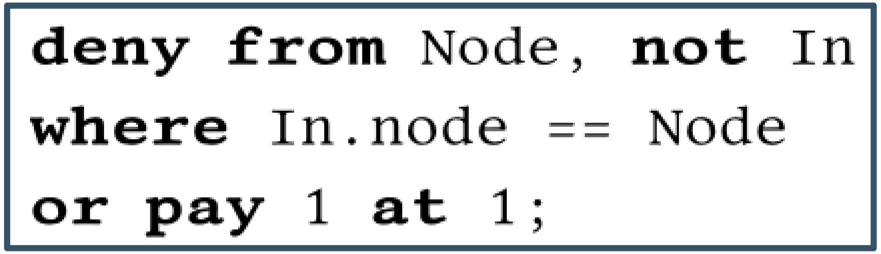
```
deny from In as in1, In as in2, not Edge
where in1.node == Edge.first and
      in2.node == Edge.second and
      in1.node < in2.node;
```

```
:- in(node(Id)), in(node(Id')), not edge(node(F), node(S)),
   node(Id) == node(F),
   node(Id') == node(S),
   node(Id) < node(Id').
```

Model Instructions: Denies with Penalty



```
record Node: id: int;
record Edge: first: Node, second: Node;
record In:   node: Node;
record Size: value: int;
```

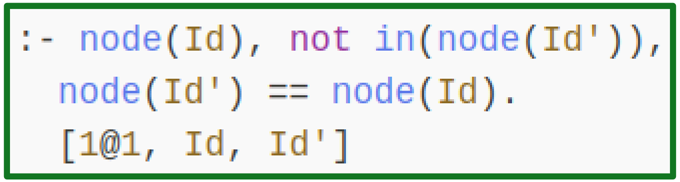


```
deny from Node, not In
where In.node == Node
or pay 1 at 1;
```

Model Instructions: Denies with Penalty

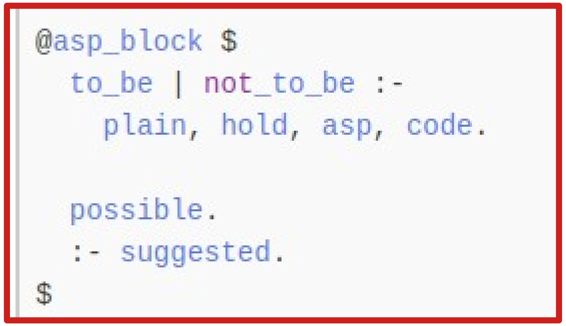
```
record Node: id: int;
record Edge: first: Node, second: Node;
record In:   node: Node;
record Size: value: int;
```

```
deny from Node, not In
where In.node == Node
or pay 1 at 1;
```



```
:- node(Id), not in(node(Id')),
  node(Id') == node(Id).
  [1@1, Id, Id']
```

And if you need “assembly”...



```
@asp_block $
  to_be | not_to_be :-
    plain, hold, asp, code.

  possible.
  :- suggested.
$
```

We don’t do any syntactic or semantic check in

@asp_block $ … $

Summing UP

ASP syntax is not suitable for long-standing codebases
(it doesn’t even look like a programming language)

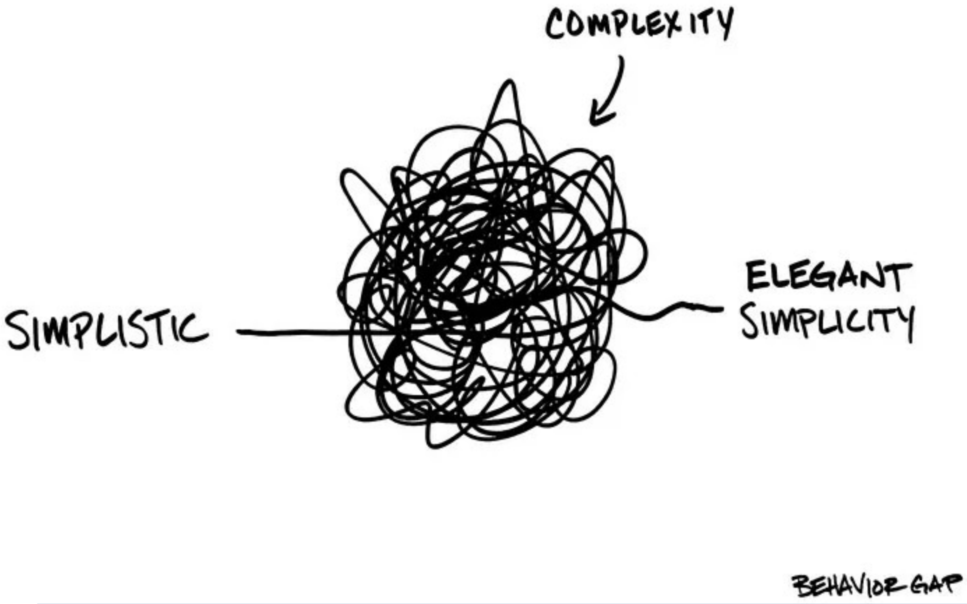


Image Credits to Hill Investment Group

Summing UP

ASP syntax is not suitable for long-standing codebases (it doesn't even look like a programming language)

SDL is a programming language of higher level, enforcing a proper structure of records, and introducing dependencies only if explicitly needed

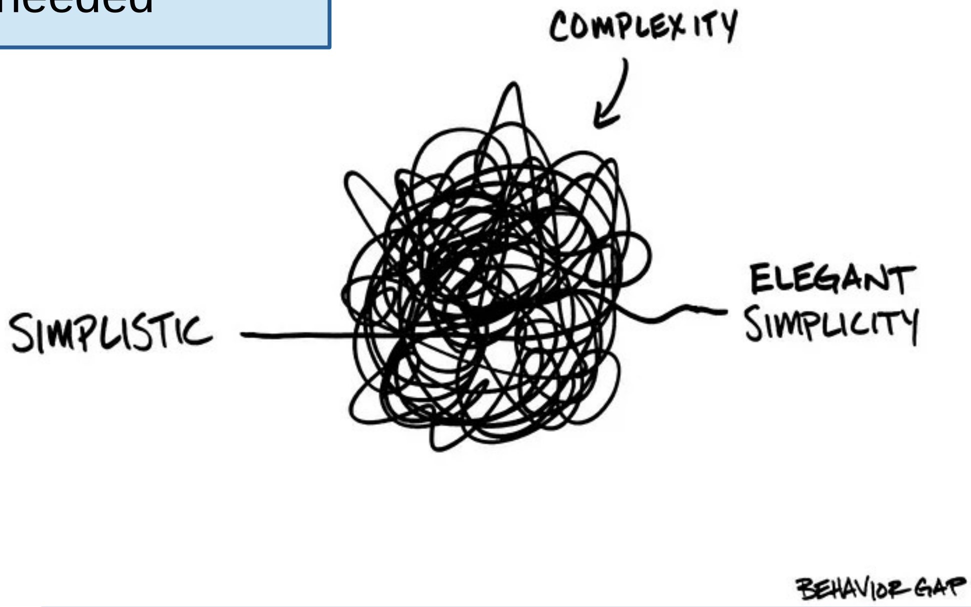


Image Credits to Hill Investment Group

Summing UP

ASP syntax is not suitable for long-standing codebases (it doesn’t even look like a programming language)

SDL is a programming language of higher level, enforcing a proper structure of records, and introducing dependencies only if explicitly needed

On the long run, programmers may code in SDL without thinking at all to ASP (as SQL programmers ~~never~~ rarely think to relational algebra)

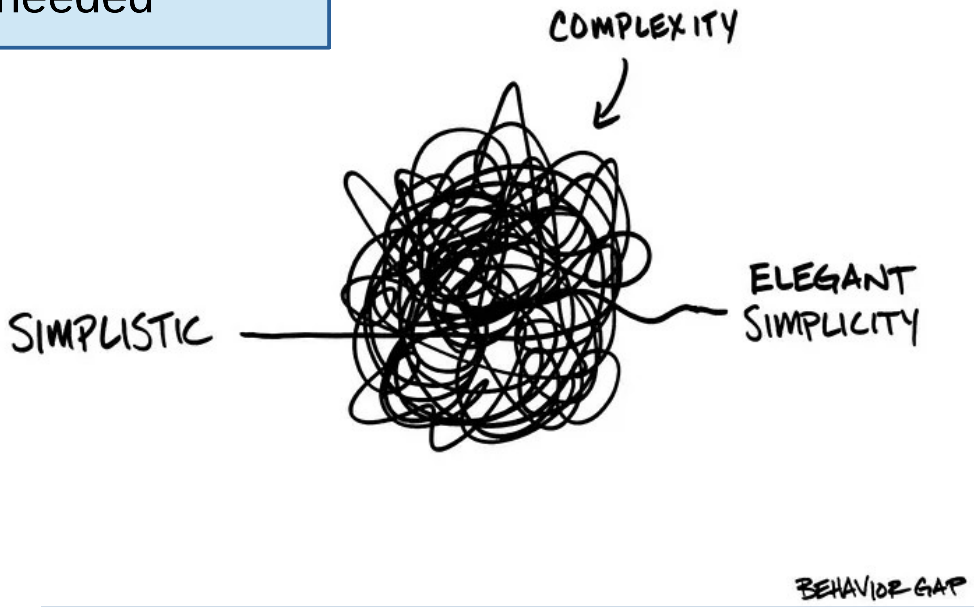


Image Credits to Hill Investment Group

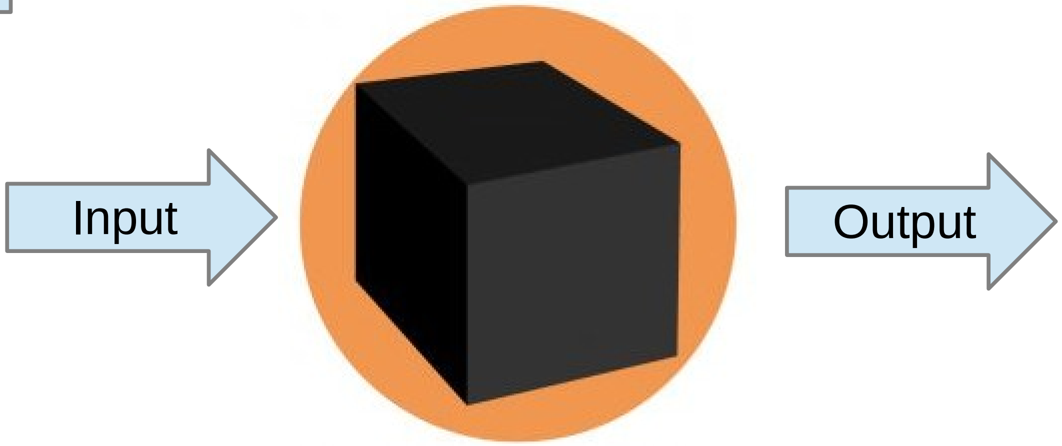
Explainability Issues

https://asp-chef.alviano.net/s/ucorexplain/

Outline

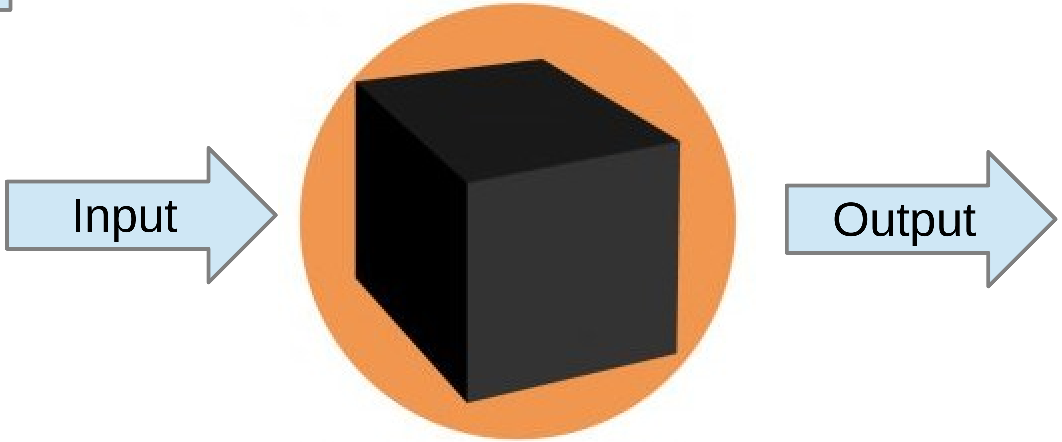
- Who has an issue with explainability
- What you are asked to believe in ASP
- How to ask for less faith
- ASP powered approach
- Interactive representation

Explainability: A ML issue?



Black-box ML algorithms for AI:
The link between input and output is unclear
(i.e., it cannot be explained)

Explainability: A ML issue?



Input

Output

Black-box ML algorithms for AI:
The link between input and output is unclear
(i.e., it cannot be explained)

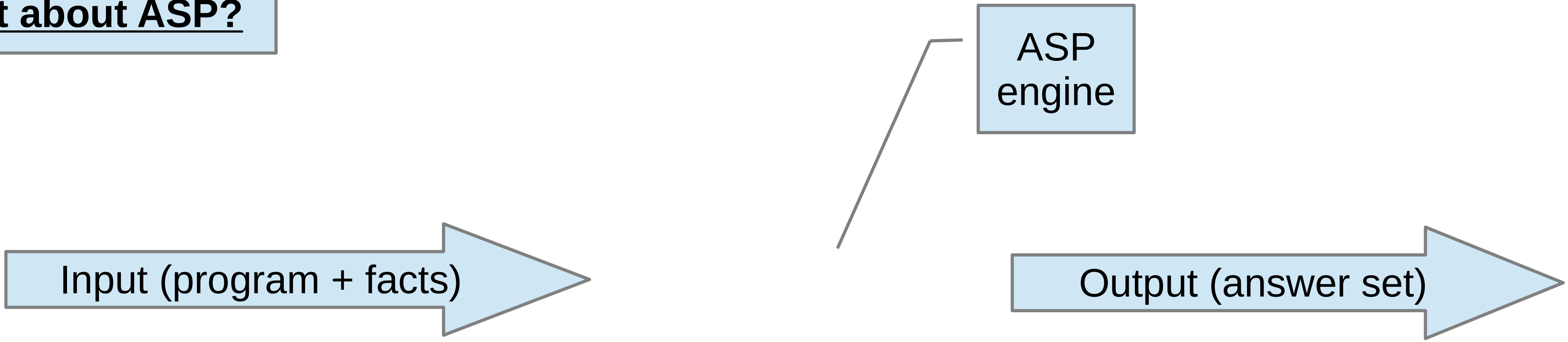
White-box ML algorithms for AI

Input

Output

The link between input and output is understandable by domain experts (e.g., a collection of features that influenced the result)

What about ASP?

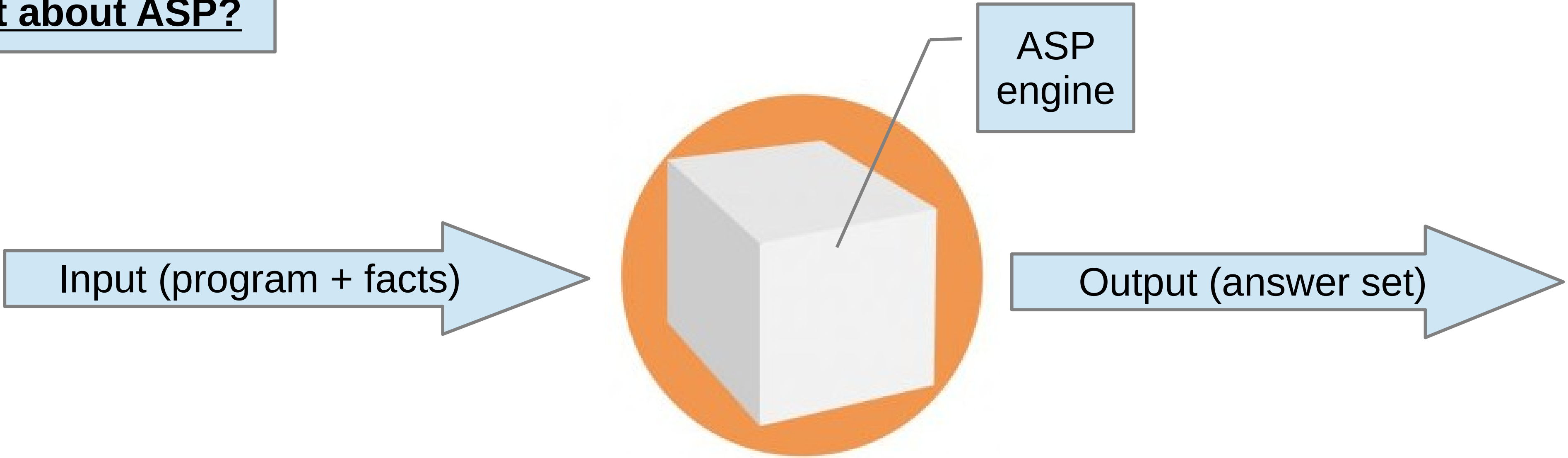


ASP engine

Input (program + facts)

Output (answer set)

What about ASP?



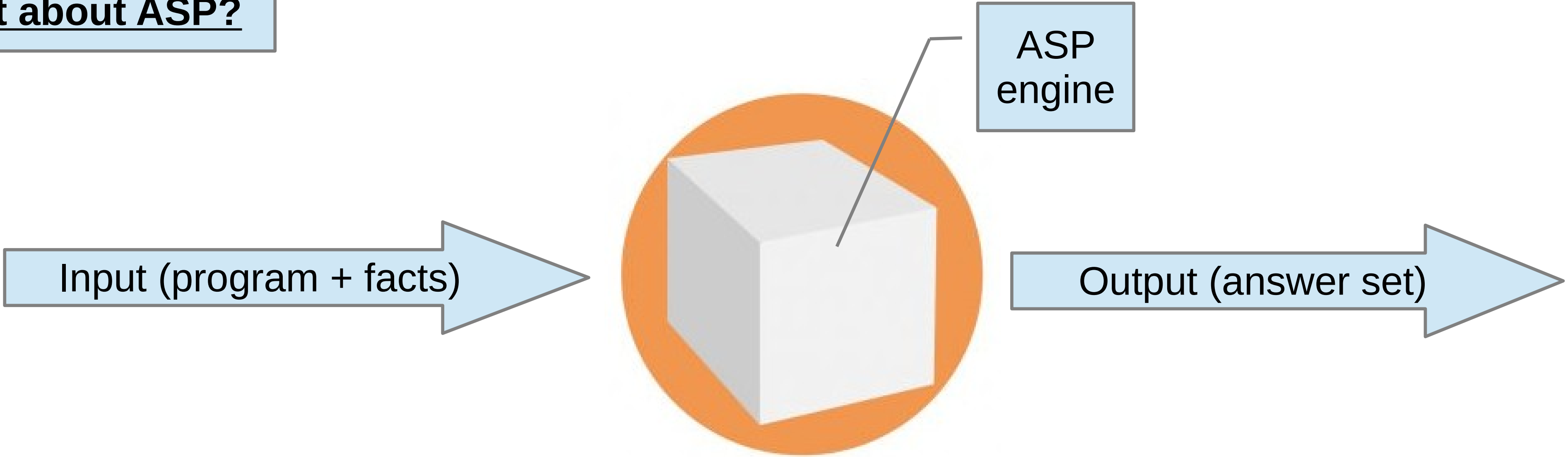
ASP engine

Input (program + facts)

Output (answer set)

The algorithm is known, the semantics is intuitive.
The answer set can be checked in polynomial time
(in the normal case, wrt. the ground program).

What about ASP?



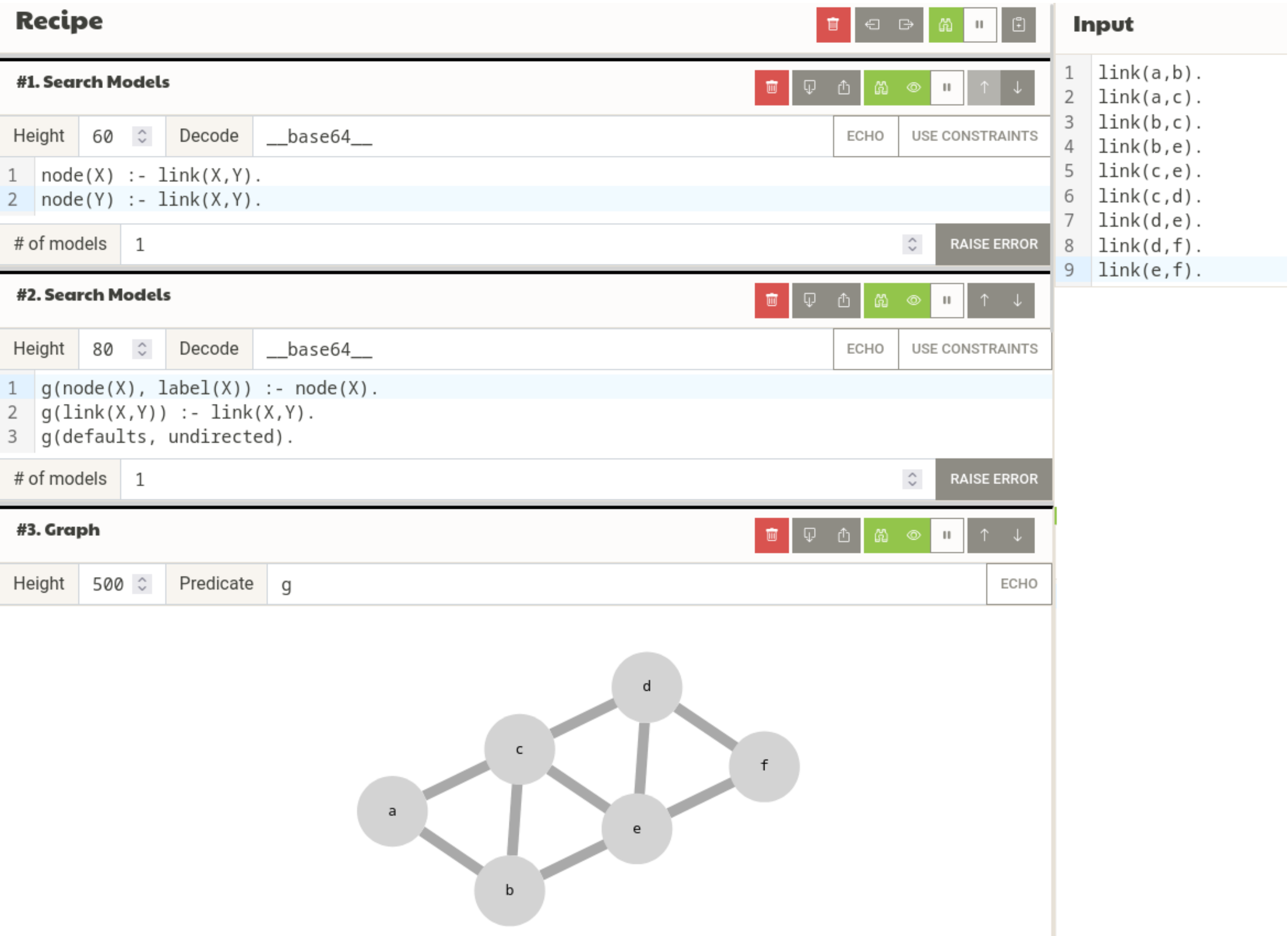
ASP engine

Input (program + facts)

Output (answer set)

**The algorithm is known, the semantics is intuitive.
The answer set can be checked in polynomial time
(in the normal case, wrt. the ground program).**

So… is everything perfect here?



Recipe

#1. Search Models

Height 60

Decode `__base64__`

```
node(X) :- link(X,Y).
node(Y) :- link(X,Y).
```

# of models 1

#2. Search Models

Height 80

Decode `__base64__`

```
g(node(X), label(X)) :- node(X).
g(link(X,Y)) :- link(X,Y).
g(defaults, undirected).
```

# of models 1

#3. Graph

Height 500

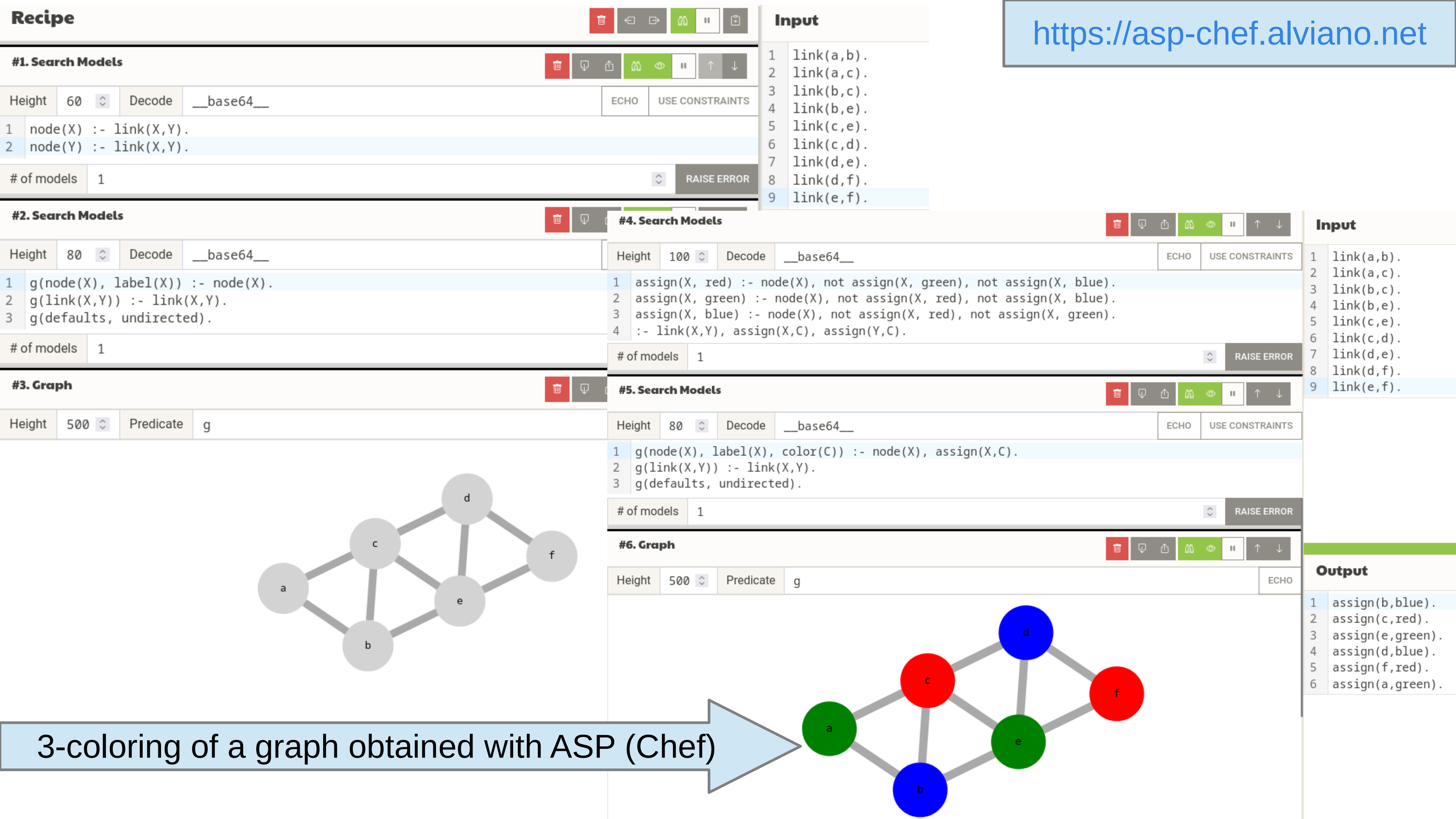
Predicate g

Input

```
link(a,b).
link(a,c).
link(b,c).
link(b,e).
link(c,e).
link(c,d).
link(d,e).
link(d,f).
link(e,f).
```

https://asp-chef.alviano.net

https://asp-chef.alviano.net



#1. Search Models

Height 60 Decode __base64__

```
node(X) :- link(X,Y).
node(Y) :- link(X,Y).
```

# of models 1

#2. Search Models

Height 80 Decode __base64__

```
g(node(X), label(X)) :- node(X).
g(link(X,Y)) :- link(X,Y).
g(defaults, undirected).
```

# of models 1

#3. Graph

Height 500 Predicate g

#4. Search Models

Height 100 Decode __base64__

```
assign(X, red) :- node(X), not assign(X, green), not assign(X, blue).
assign(X, green) :- node(X), not assign(X, red), not assign(X, blue).
assign(X, blue) :- node(X), not assign(X, red), not assign(X, green).
:- link(X,Y), assign(X,C), assign(Y,C).
```

# of models 1

#5. Search Models

Height 80 Decode __base64__

```
g(node(X), label(X), color(C)) :- node(X), assign(X,C).
g(link(X,Y)) :- link(X,Y).
g(defaults, undirected).
```

# of models 1

#6. Graph

Height 500 Predicate g

Input

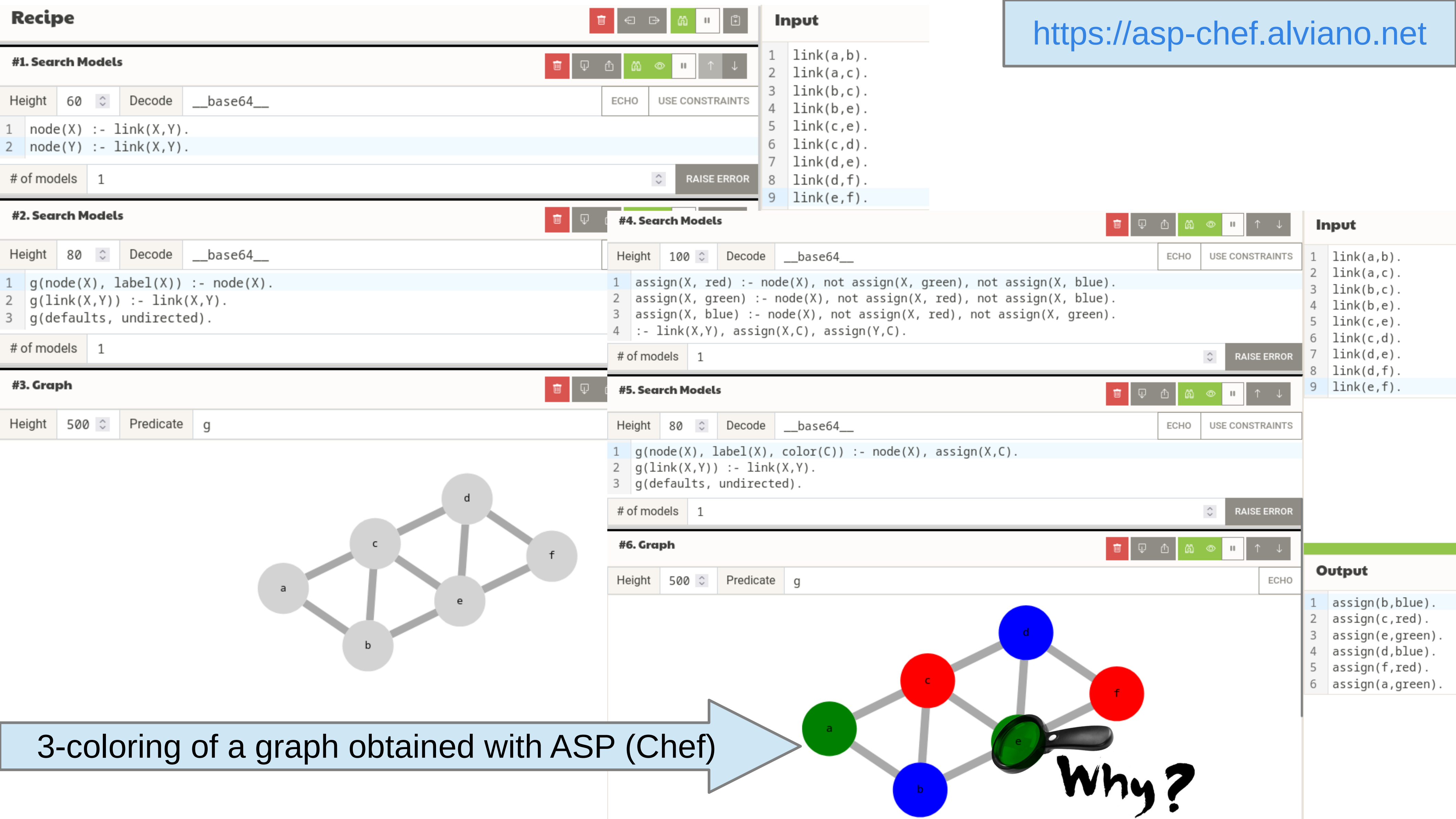
```
link(a,b).
link(a,c).
link(b,c).
link(b,e).
link(c,e).
link(c,d).
link(d,e).
link(d,f).
link(e,f).
```

Output

```
assign(b,blue).
assign(c,red).
assign(e,green).
assign(d,blue).
assign(f,red).
assign(a,green).
```

3-coloring of a graph obtained with ASP (Chef)

https://asp-chef.alviano.net



Recipe

#1. Search Models

Height 60 Decode __base64__

```
node(X) :- link(X,Y).
node(Y) :- link(X,Y).
```

# of models 1

#2. Search Models

Height 80 Decode __base64__

```
g(node(X), label(X)) :- node(X).
g(link(X,Y)) :- link(X,Y).
g(defaults, undirected).
```

# of models 1

#3. Graph

Height 500 Predicate g

Input

```
link(a,b).
link(a,c).
link(b,c).
link(b,e).
link(c,e).
link(c,d).
link(d,e).
link(d,f).
link(e,f).
```

#4. Search Models

Height 100 Decode __base64__

```
assign(X, red) :- node(X), not assign(X, green), not assign(X, blue).
assign(X, green) :- node(X), not assign(X, red), not assign(X, blue).
assign(X, blue) :- node(X), not assign(X, red), not assign(X, green).
:- link(X,Y), assign(X,C), assign(Y,C).
```

# of models 1

#5. Search Models

Height 80 Decode __base64__

```
g(node(X), label(X), color(C)) :- node(X), assign(X,C).
g(link(X,Y)) :- link(X,Y).
g(defaults, undirected).
```

# of models 1

#6. Graph

Height 500 Predicate g

Input

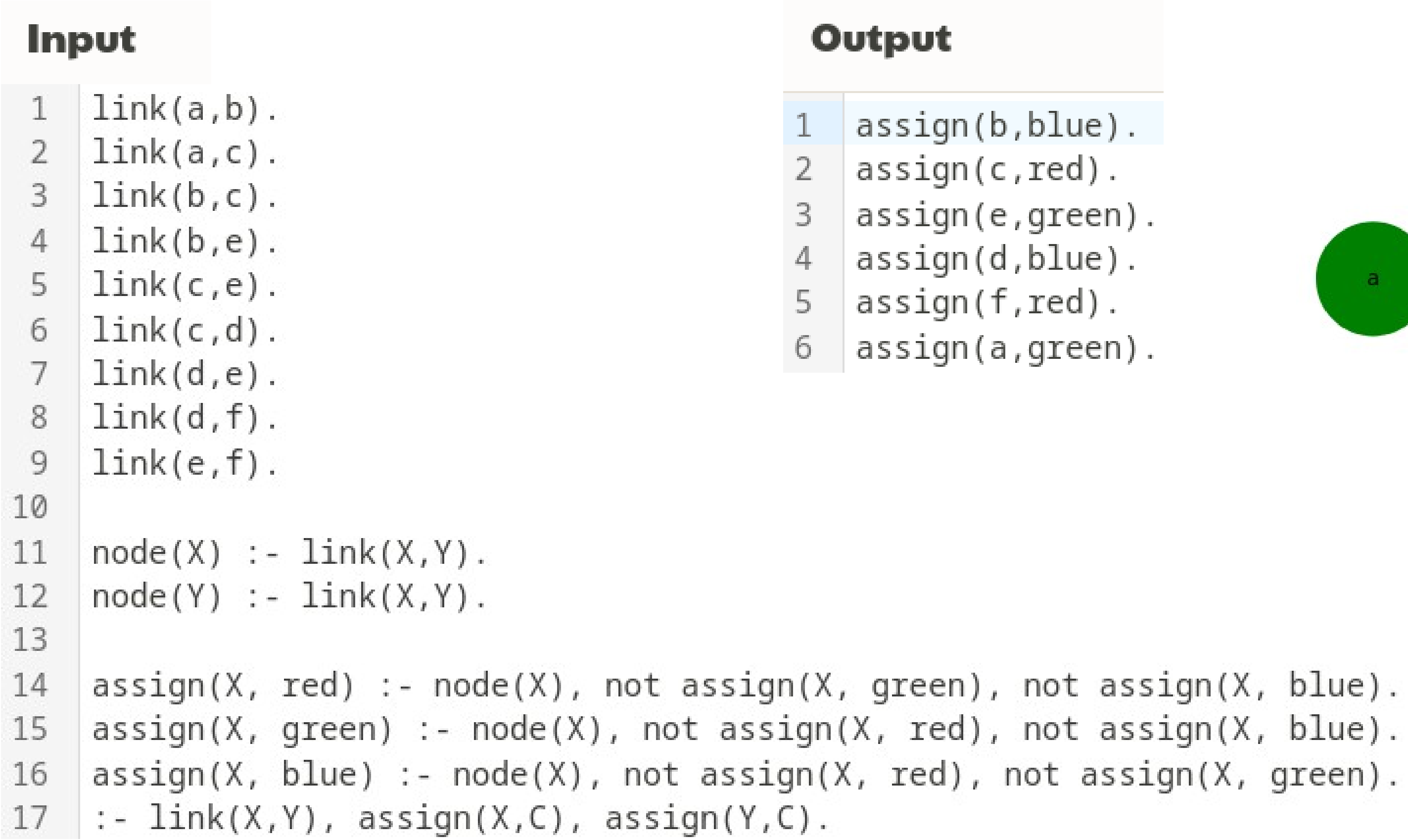
```
link(a,b).
link(a,c).
link(b,c).
link(b,e).
link(c,e).
link(c,d).
link(d,e).
link(d,f).
link(e,f).
```

Output

```
assign(b,blue).
assign(c,red).
assign(e,green).
assign(d,blue).
assign(f,red).
assign(a,green).
```

3-coloring of a graph obtained with ASP (Chef)

Why?

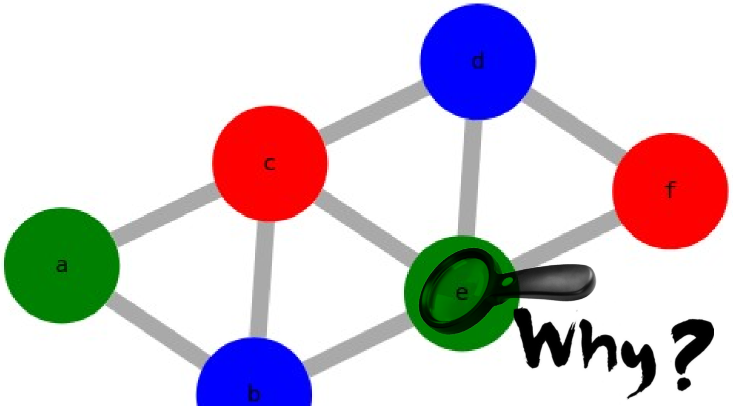


Input

```
link(a,b).
link(a,c).
link(b,c).
link(b,e).
link(c,e).
link(c,d).
link(d,e).
link(d,f).
link(e,f).

node(X) :- link(X,Y).
node(Y) :- link(X,Y).

assign(X, red) :- node(X), not assign(X, green), not assign(X, blue).
assign(X, green) :- node(X), not assign(X, red), not assign(X, blue).
assign(X, blue) :- node(X), not assign(X, red), not assign(X, green).
:- link(X,Y), assign(X,C), assign(Y,C).
```

Output

```
assign(b,blue).
assign(c,red).
assign(e,green).
assign(d,blue).
assign(f,red).
assign(a,green).
```

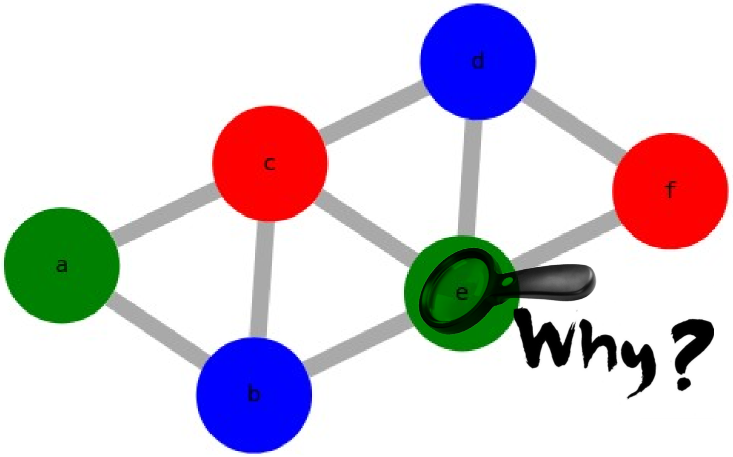


An answer set is a minimal model of its program reduct (a Datalog program).
Datalog models can be computed by support inference.
The computation gives a DAG. Extract a proof tree for the query atom.

The program reduct is obtained from
the ground version of the program in input
by fixing the interpretation of negative literals

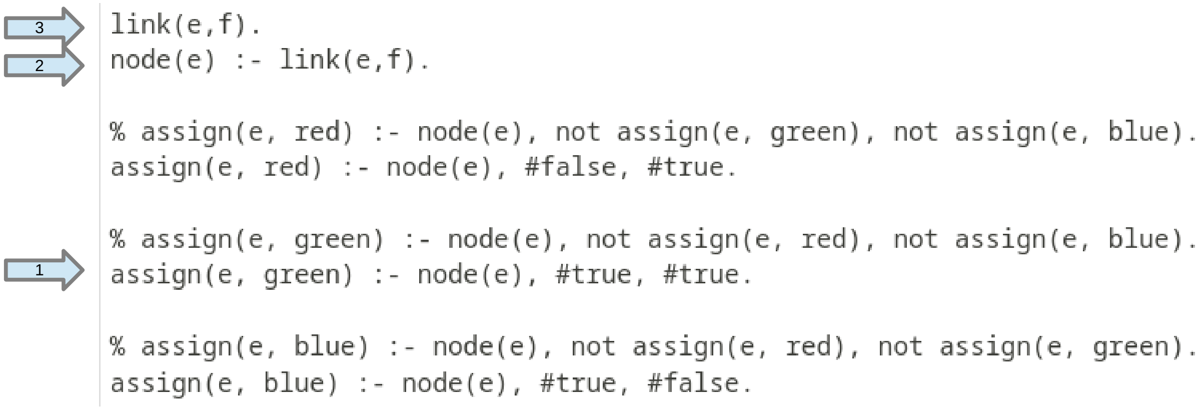
In the program reduct...

```
link(e,f).
node(e) :- link(e,f).

% assign(e, red) :- node(e), not assign(e, green), not assign(e, blue).
assign(e, red) :- node(e), #false, #true.

% assign(e, green) :- node(e), not assign(e, red), not assign(e, blue).
assign(e, green) :- node(e), #true, #true.

% assign(e, blue) :- node(e), not assign(e, red), not assign(e, green).
assign(e, blue) :- node(e), #true, #false.
```

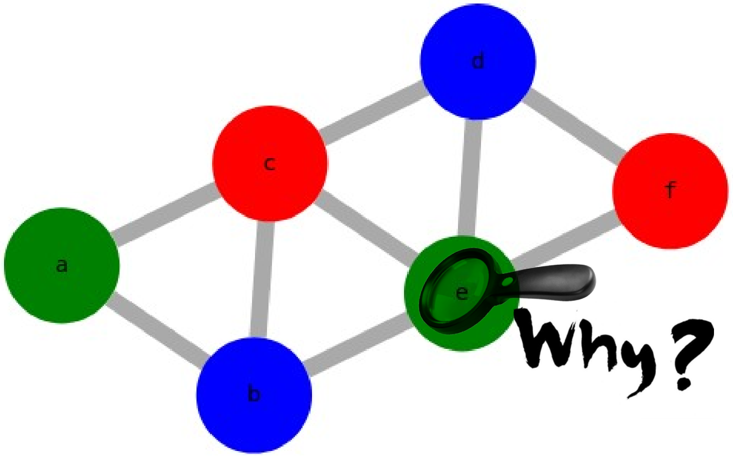


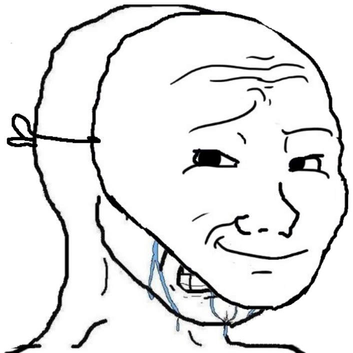
1) assign(e, green) is supported by node(e)
2) node(e) is supported by link(e,f)
3) link(e,f) is a fact

In the program reduct...



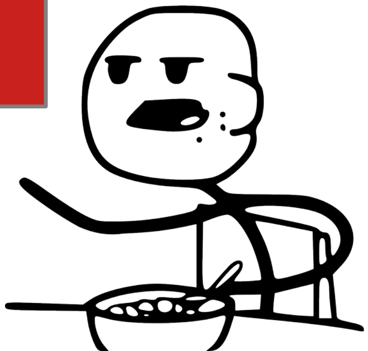
```
link(e,f).
node(e) :- link(e,f).

% assign(e, red) :- node(e), not assign(e, green), not assign(e, blue).
assign(e, red) :- node(e), #false, #true.

% assign(e, green) :- node(e), not assign(e, red), not assign(e, blue).
assign(e, green) :- node(e), #true, #true.

% assign(e, blue) :- node(e), not assign(e, red), not assign(e, green).
assign(e, blue) :- node(e), #true, #false.
```





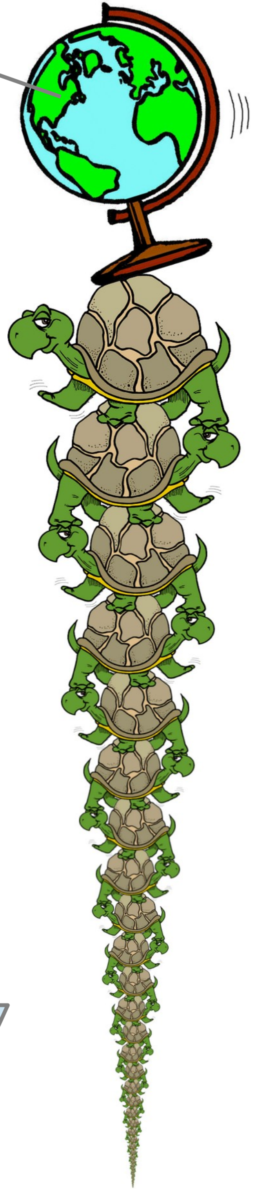
1) assign(e, green) is supported by node(e)
2) node(e) is supported by link(e,f)
3) link(e,f) is a fact

Why #true, #true ?



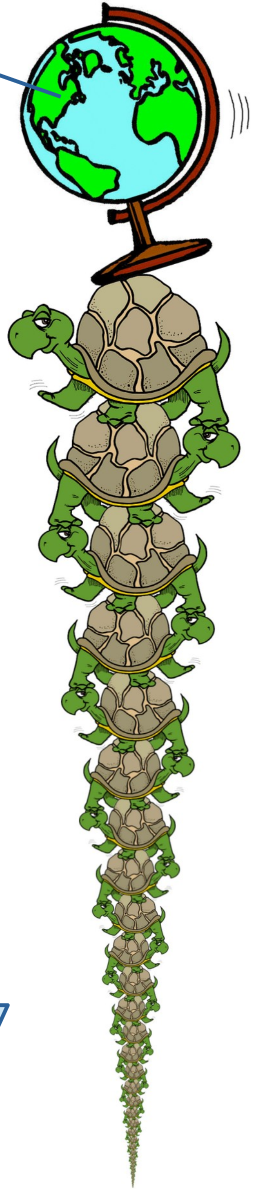
Because ALL false atoms are assumed false...

The answer set
(true atoms)



Infinitely many false atoms

The answer set
(true atoms)

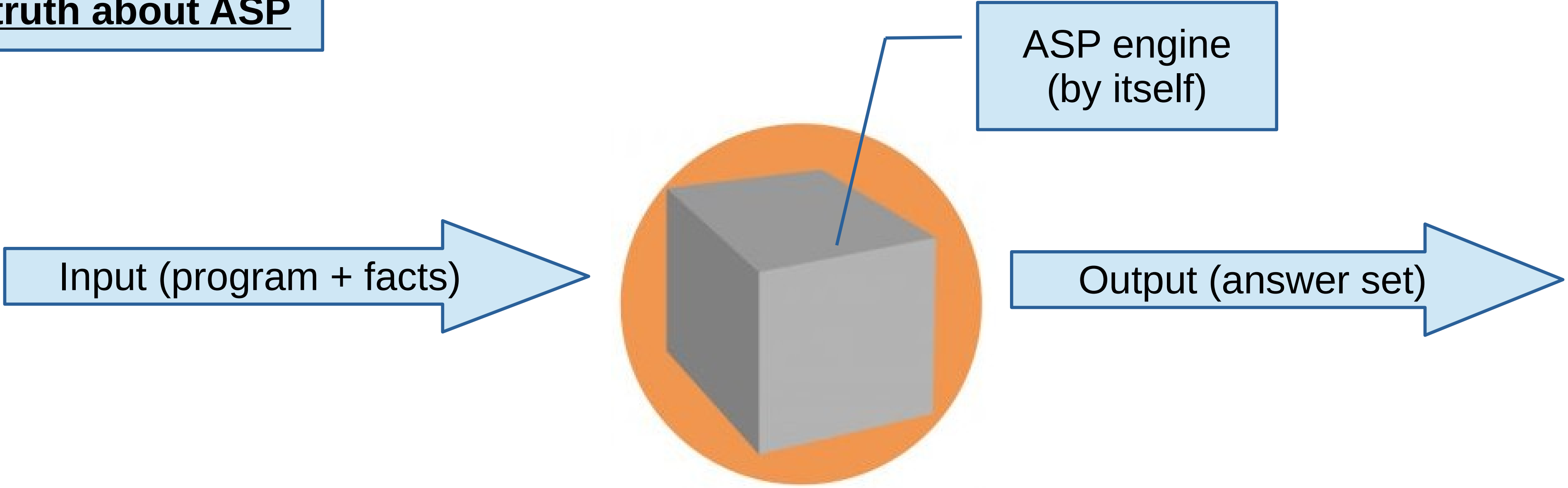


Infinitely many false atoms

Fixing here means (blindly) trusting!

How big must our faith be?

The truth about ASP



ASP engine
(by itself)

Input (program + facts)

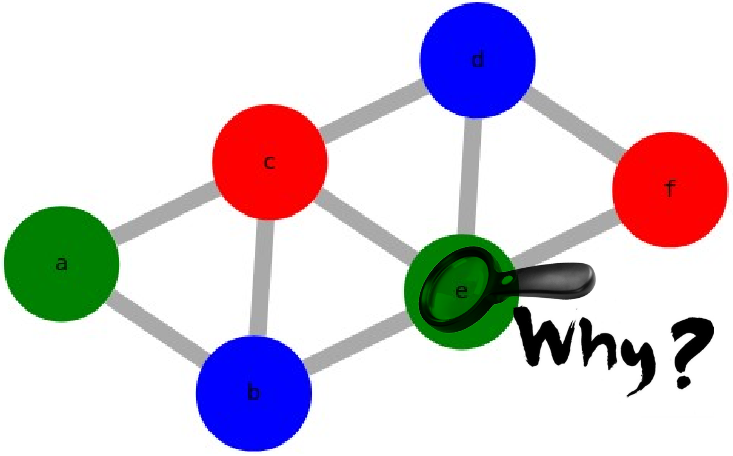
Output (answer set)

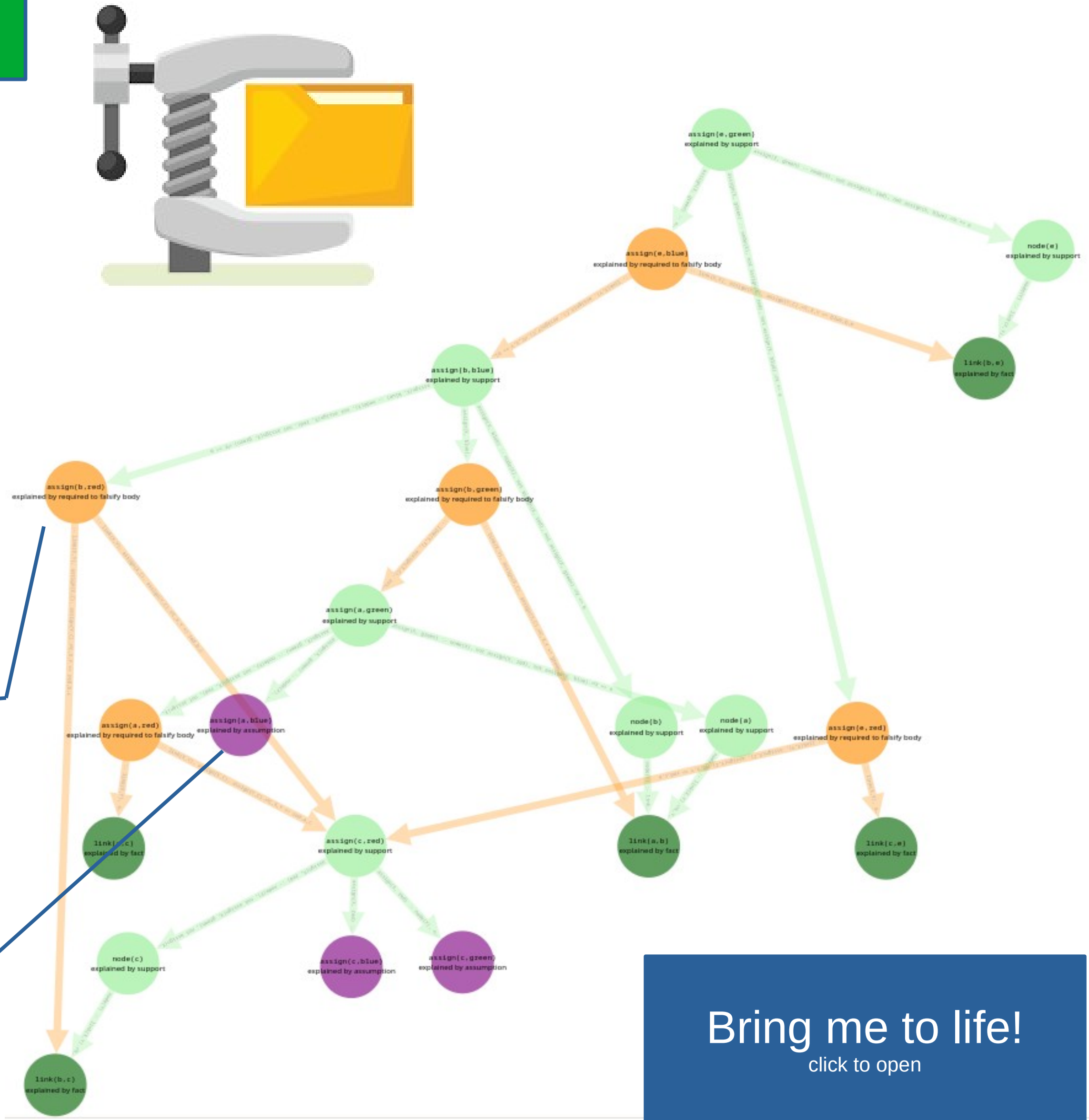
Let’s say it is a gray-box!

You can check the answer set in polynomial time,
but who want to do this?

Anyhow you must trust all false atoms,
a huge act of faith!

Let’s try to make the assumption smaller!





Rules with false heads
(e.g., constraints)
can infer falsity
of some atoms!

Some other
atoms are simply
assumed false

Bring me to life!
click to open

How? Step 1: Minimal Assumption Set!

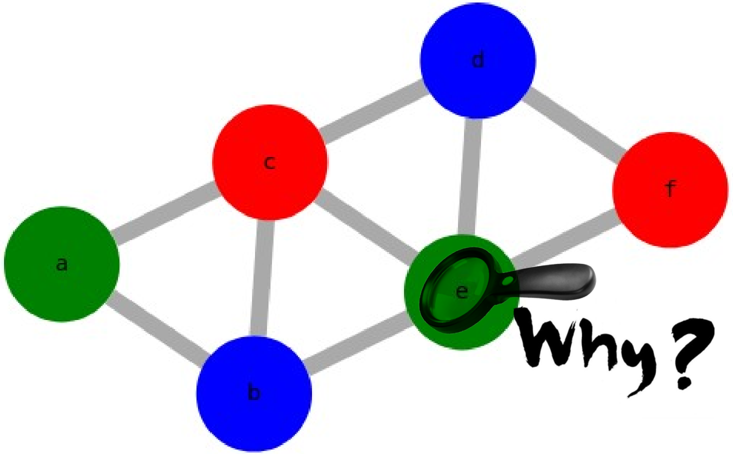
Assume falsity of some atoms

1) Initial well-founded simplification
2) Support inference
3) Lack of support inference
4) Inference via constraint-like rules

If the answer set is reconstructed,
it was a (good) assumption set!

Prefer subset-minimal assumption sets not containing the query atom

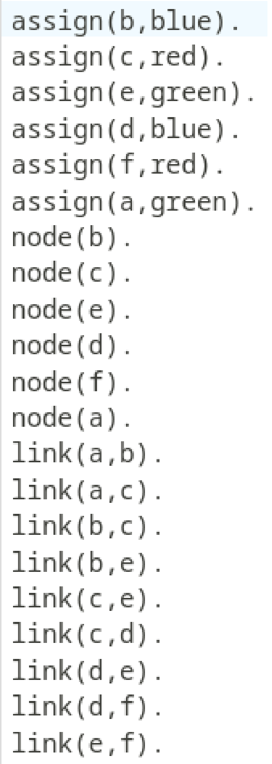
Example of Minimal Assumption Set



assign(a, blue)
assign(c, blue)
assign(c, green)

vs

\

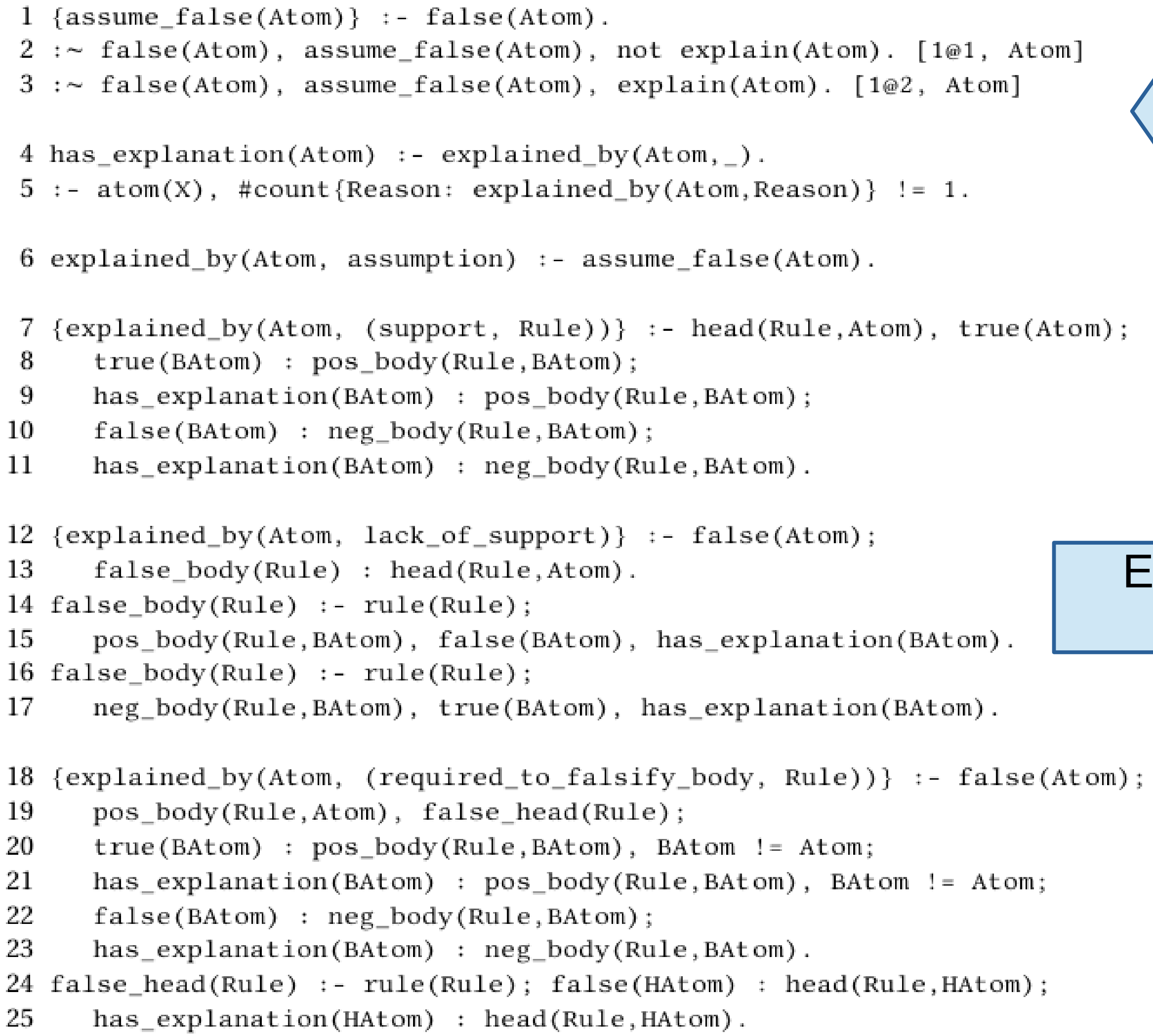


```
assign(b,blue).
assign(c,red).
assign(e,green).
assign(d,blue).
assign(f,red).
assign(a,green).
node(b).
node(c).
node(e).
node(d).
node(f).
node(a).
link(a,b).
link(a,c).
link(b,c).
link(b,e).
link(c,e).
link(c,d).
link(d,e).
link(d,f).
link(e,f).
```

MAS is a combinatorial optimization problem

Serialize input
(program, answer set,
query)
as facts

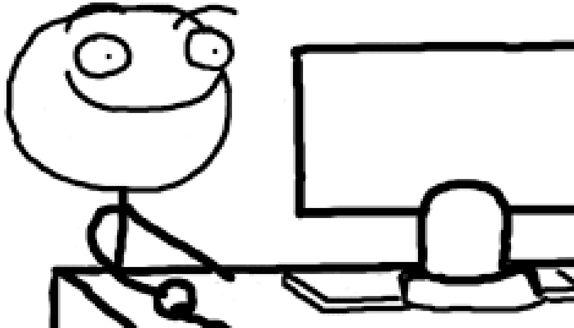
MAS is a combinatorial optimization problem



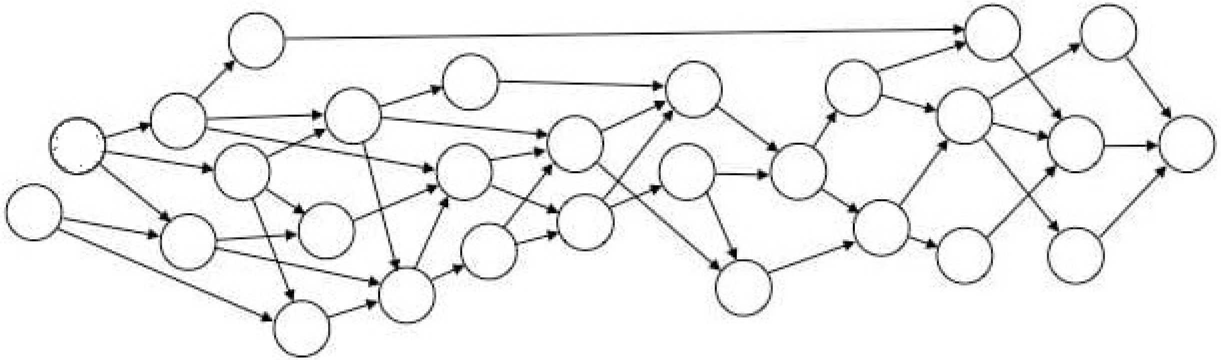
```
{assume_false(Atom)} :- false(Atom).
:~ false(Atom), assume_false(Atom), not explain(Atom). [1@1, Atom]
:~ false(Atom), assume_false(Atom), explain(Atom). [1@2, Atom]

has_explanation(Atom) :- explained_by(Atom,_).
:- atom(X), #count{Reason: explained_by(Atom,Reason)} != 1.

explained_by(Atom, assumption) :- assume_false(Atom).

{explained_by(Atom, (support, Rule))} :- head(Rule,Atom), true(Atom);
   true(BAtom) : pos_body(Rule,BAtom);
   has_explanation(BAtom) : pos_body(Rule,BAtom);
   false(BAtom) : neg_body(Rule,BAtom);
   has_explanation(BAtom) : neg_body(Rule,BAtom).

{explained_by(Atom, lack_of_support)} :- false(Atom);
   false_body(Rule) : head(Rule,Atom).
false_body(Rule) :- rule(Rule);
   pos_body(Rule,BAtom), false(BAtom), has_explanation(BAtom).
false_body(Rule) :- rule(Rule);
   neg_body(Rule,BAtom), true(BAtom), has_explanation(BAtom).

{explained_by(Atom, (required_to_falsify_body, Rule))} :- false(Atom);
   pos_body(Rule,Atom), false_head(Rule);
   true(BAtom) : pos_body(Rule,BAtom), BAtom != Atom;
   has_explanation(BAtom) : pos_body(Rule,BAtom), BAtom != Atom;
   false(BAtom) : neg_body(Rule,BAtom);
   has_explanation(BAtom) : neg_body(Rule,BAtom).
false_head(Rule) :- rule(Rule); false(HAtom) : head(Rule,HAtom);
   has_explanation(HAtom) : head(Rule,HAtom).
```

Serialize input
(program, answer set,
query)
as facts

Each optimal answer set is a MAS
(and comes with a derivation)

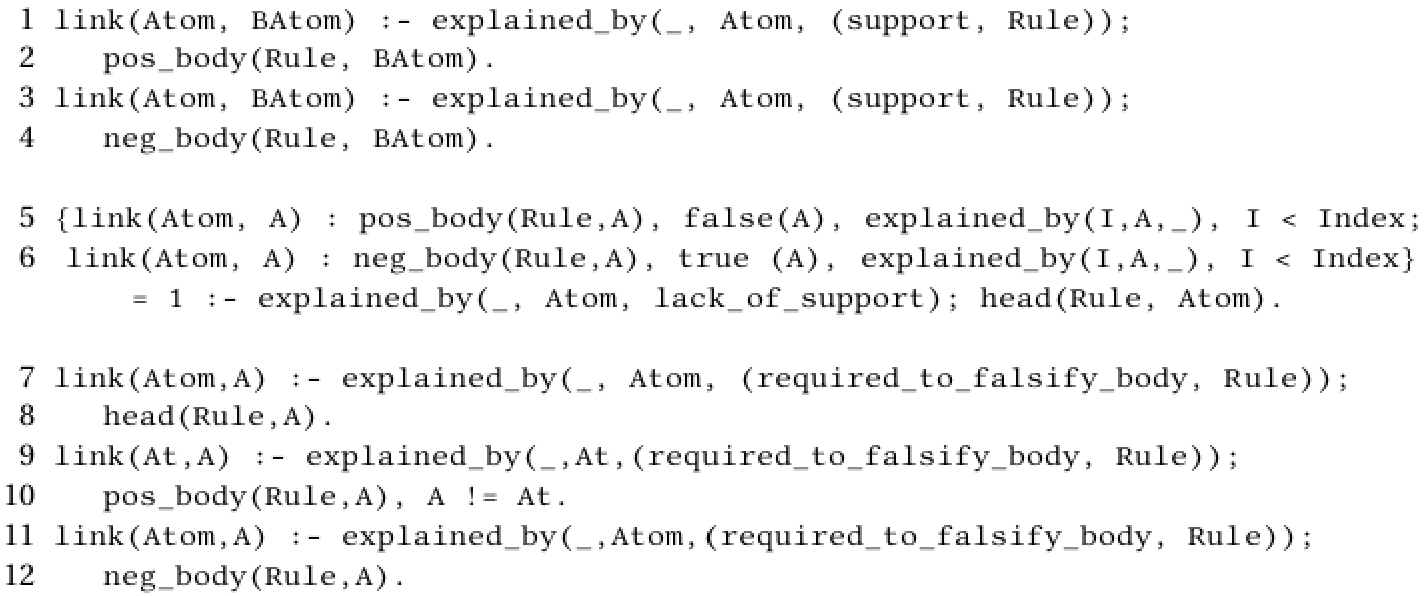


How? Step 2: Directed Acyclic Graph (DAG)



Program, answer set, and query serialized as facts…
derivation serialized (and indexed) as explained_by/3

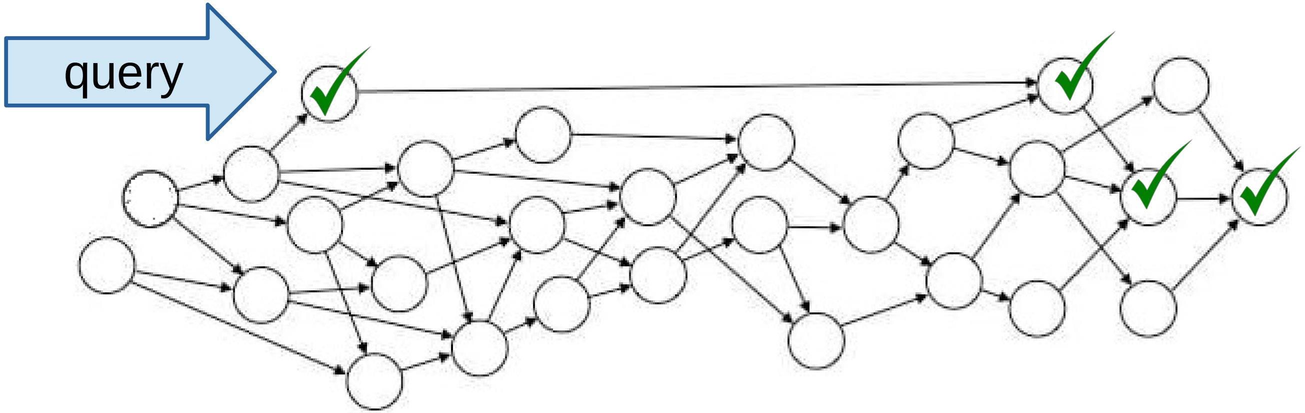
The DAG can be materialized by a Datalog program



```
link(Atom, BAtom) :- explained_by(_, Atom, (support, Rule));
   pos_body(Rule, BAtom).
link(Atom, BAtom) :- explained_by(_, Atom, (support, Rule));
   neg_body(Rule, BAtom).

{link(Atom, A) : pos_body(Rule,A), false(A), explained_by(I,A,_), I < Index;
 link(Atom, A) : neg_body(Rule,A), true (A), explained_by(I,A,_), I < Index}
    = 1 :- explained_by(_, Atom, lack_of_support); head(Rule, Atom).

link(Atom,A) :- explained_by(_, Atom, (required_to_falsify_body, Rule));
   head(Rule,A).
link(At,A) :- explained_by(_,At,(required_to_falsify_body, Rule));
   pos_body(Rule,A), A != At.
link(Atom,A) :- explained_by(_,Atom,(required_to_falsify_body, Rule));
   neg_body(Rule,A).
```

The DAG can be materialized by a Datalog program

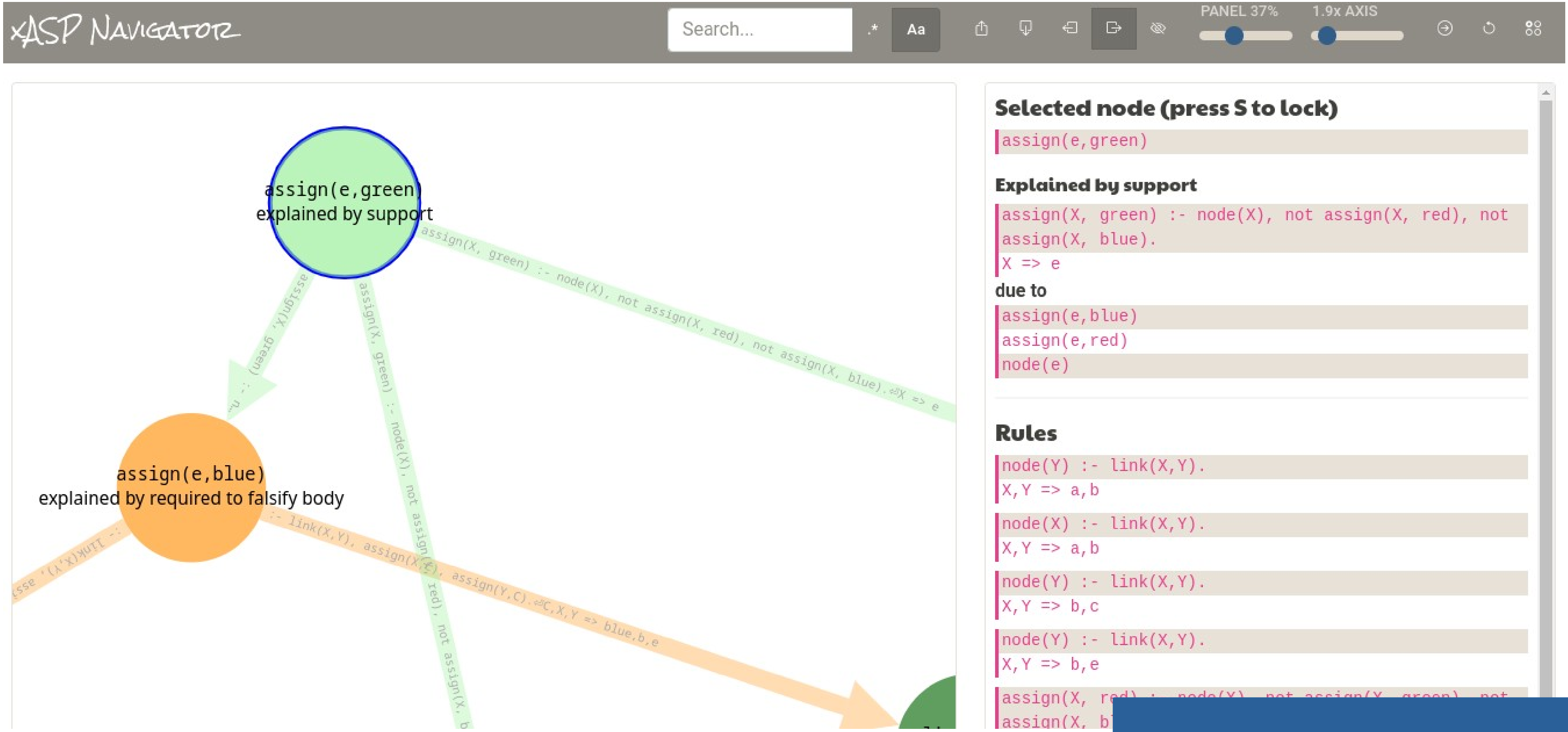
How? Step 3: Extract the relevant subDAG



Induced subgraph on nodes reachable from the query

Mix of ASP and igraph for better performances
(we don’t really compute the full DAG)

How? Step 4: Interactive Representation


xASP Navigator
Search...
.*
Aa
PANEL 37%
1.9x AXIS
assign(e,green)
explained by support
assign(e,blue)
explained by required to falsify body
assign(X, green) :- node(X), not assign(X, red), not assign(X, blue). X => e
:- link(X,Y), assign(X,C), assign(Y,C). C,X,Y => blue,b,e

Selected node (press S to lock)

```
assign(e,green)
```

Explained by support

```
assign(X, green) :- node(X), not assign(X, red), not
assign(X, blue).
X => e
```

due to

```
assign(e,blue)
assign(e,red)
node(e)
```

Rules

```
node(Y) :- link(X,Y).
X,Y => a,b
node(X) :- link(X,Y).
X,Y => a,b
node(Y) :- link(X,Y).
X,Y => b,c
node(Y) :- link(X,Y).
X,Y => b,e
assign(X, red) :- node(X), not assign(X, green), not
assign(X, bl
```

Everything is in the URL!

click to open

Assessment: Marcello had a program...

420 rules
651 facts

grounding

4261 ground rules
4468 ground atoms

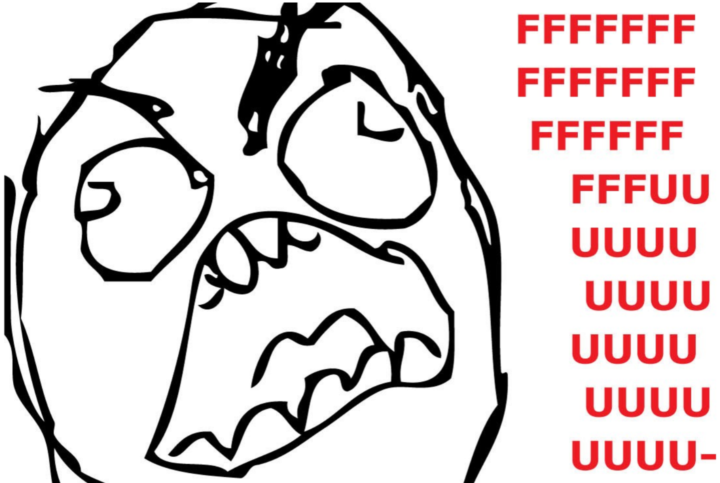
Assessment: Marcello had a program...

420 rules
651 facts

grounding

4261 ground rules
4468 ground atoms

Number of answer sets
- expected: 1
- actual: 2



Assessment: Marcello had a program...

420 rules
651 facts

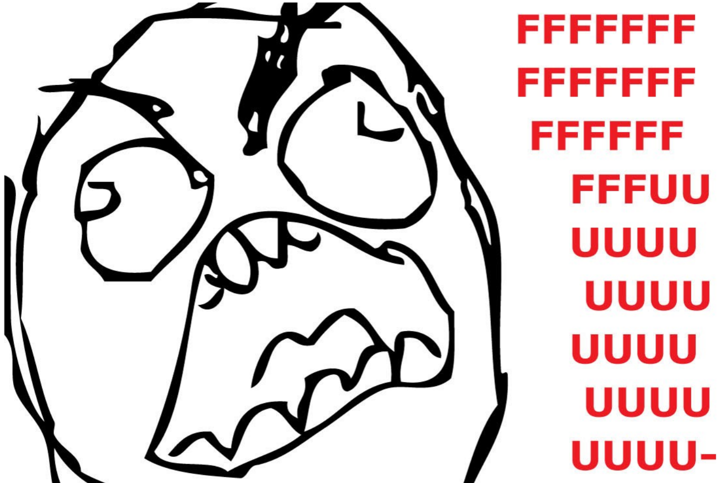
grounding

4261 ground rules
4468 ground atoms

Number of answer sets
- expected: 1
- actual: 2

The two answer sets differ in one atom

- DAG computed in ~15 seconds
- 87 links, 45 internal nodes
- 20 leaves (1 assumption)
- 30 symbolic rules, 11 facts
- 48 ground rules, 65 ground atoms



Bring me to life!
click to open

Summing up

- Shrink the set of atoms assumed false
  - as much as possible
  - answer set still reconstructible (with simple inferences)
  - computation powered by ASP
- Compute a DAG with ASP (and igraph)
  - open it in xASP Navigator
  - share it by sending a link
  - search in the graph (symbolic rules, substitutions, atoms)

ASP and LLMs interaction

Large Language Models

Computational models achieving **general-purpose language generation** and other natural language processing tasks.

Large Language Models

Computational models achieving **general-purpose language generation** and other natural language processing tasks.

We have General AI here!

Large Language Models

Computational models achieving **general-purpose language generation** and other natural language processing tasks.

We have General AI here!

False! It is good at NLP

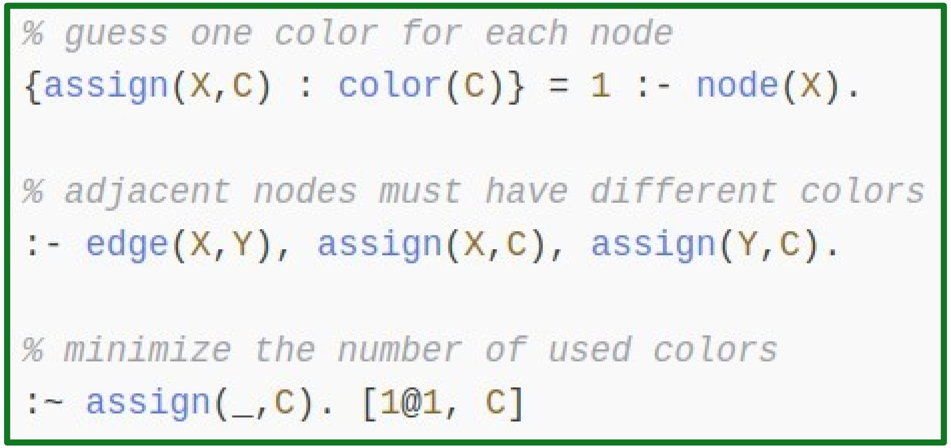
Logical reasoning is still our subject

Answer Set Programming

KRR language, good for combinatorial search and optimization

Answer Set Programming

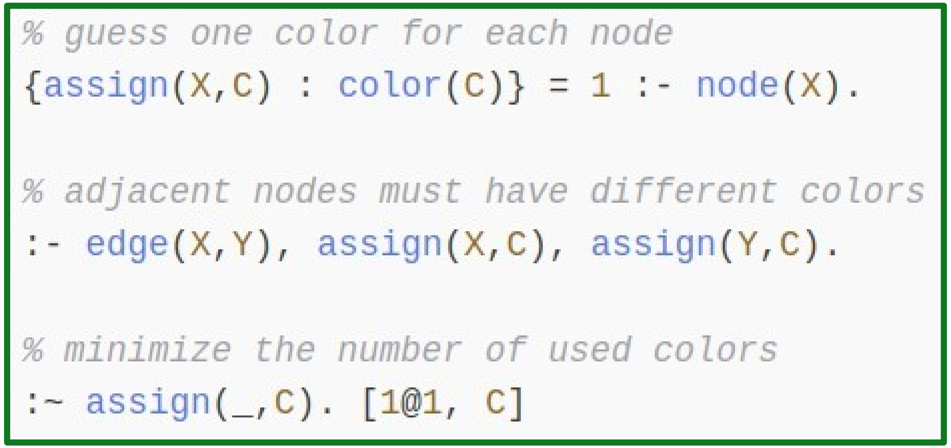
KRR language, good for combinatorial search and optimization



```
% guess one color for each node
{assign(X,C) : color(C)} = 1 :- node(X).

% adjacent nodes must have different colors
:- edge(X,Y), assign(X,C), assign(Y,C).

% minimize the number of used colors
:~ assign(_,C). [1@1, C]
```

Answer Set Programming

KRR language, good for combinatorial search and optimization



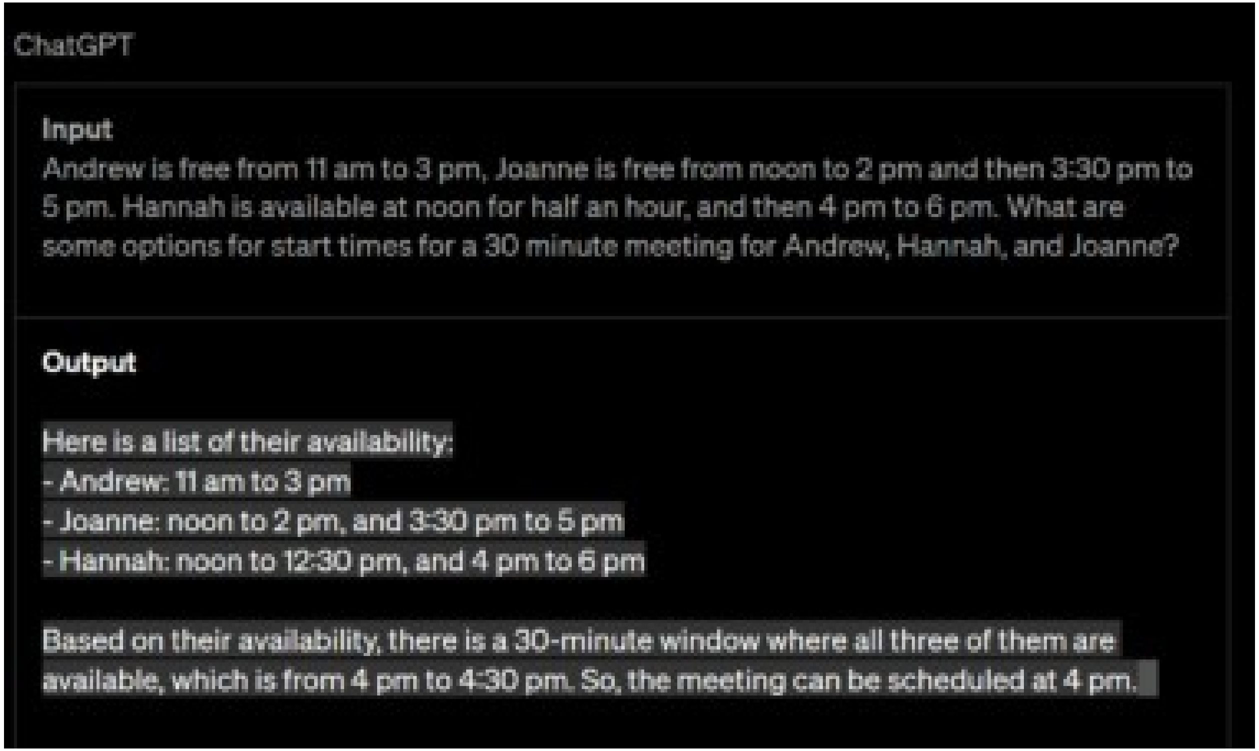
```
% guess one color for each node
{assign(X,C) : color(C)} = 1 :- node(X).

% adjacent nodes must have different colors
:- edge(X,Y), assign(X,C), assign(Y,C).

% minimize the number of used colors
:~ assign(_,C). [1@1, C]
```

Possibly (read it certainly)
there is a (huge) barrier to entry

Computer Programming

Computer Programming

People chatting and telling around they are programming



ChatGPT

Input

Andrew is free from 11 am to 3 pm, Joanne is free from noon to 2 pm and then 3:30 pm to 5 pm. Hannah is available at noon for half an hour, and then 4 pm to 6 pm. What are some options for start times for a 30 minute meeting for Andrew, Hannah, and Joanne?

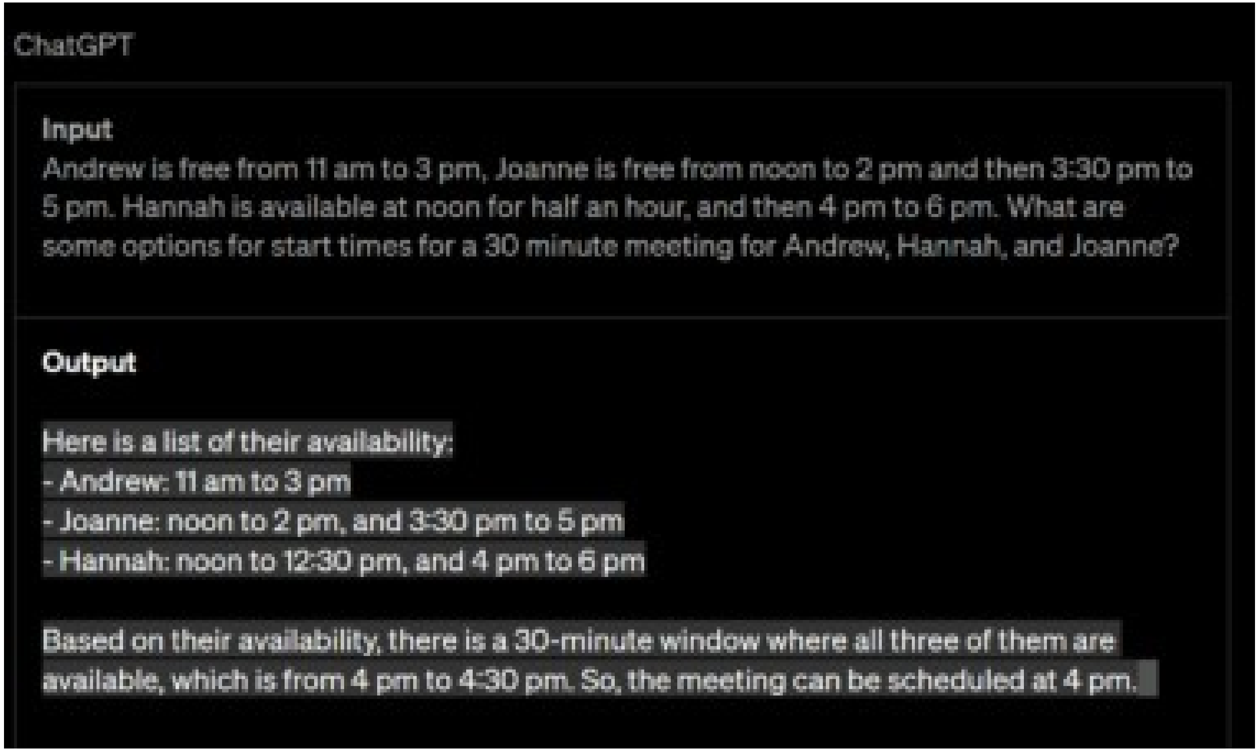
Output

Here is a list of their availability:
- Andrew: 11 am to 3 pm
- Joanne: noon to 2 pm, and 3:30 pm to 5 pm
- Hannah: noon to 12:30 pm, and 4 pm to 6 pm

Based on their availability, there is a 30-minute window where all three of them are available, which is from 4 pm to 4:30 pm. So, the meeting can be scheduled at 4 pm.

Computer Programming

People chatting and telling around they are programming


ChatGPT

Input
Andrew is free from 11 am to 3 pm, Joanne is free from noon to 2 pm and then 3:30 pm to 5 pm. Hannah is available at noon for half an hour, and then 4 pm to 6 pm. What are some options for start times for a 30 minute meeting for Andrew, Hannah, and Joanne?

Output

Here is a list of their availability:
- Andrew: 11 am to 3 pm
- Joanne: noon to 2 pm, and 3:30 pm to 5 pm
- Hannah: noon to 12:30 pm, and 4 pm to 6 pm

Based on their availability, there is a 30-minute window where all three of them are available, which is from 4 pm to 4:30 pm. So, the meeting can be scheduled at 4 pm.

As we are not a curch… **let's take the best of both worlds!**

Idea Overview

Start with natural language

Idea Overview

Start with natural language

Obtain a relational representation (facts)

Idea Overview

Start with natural language

Obtain a relational representation (facts)

Combine facts with a KB (an ASP program)

Idea Overview

Start with natural language

Obtain a relational representation (facts)

Combine facts with a KB (an ASP program)

Map the answer set to natural language

Idea Overview

Start with natural language

Obtain a relational representation (facts)

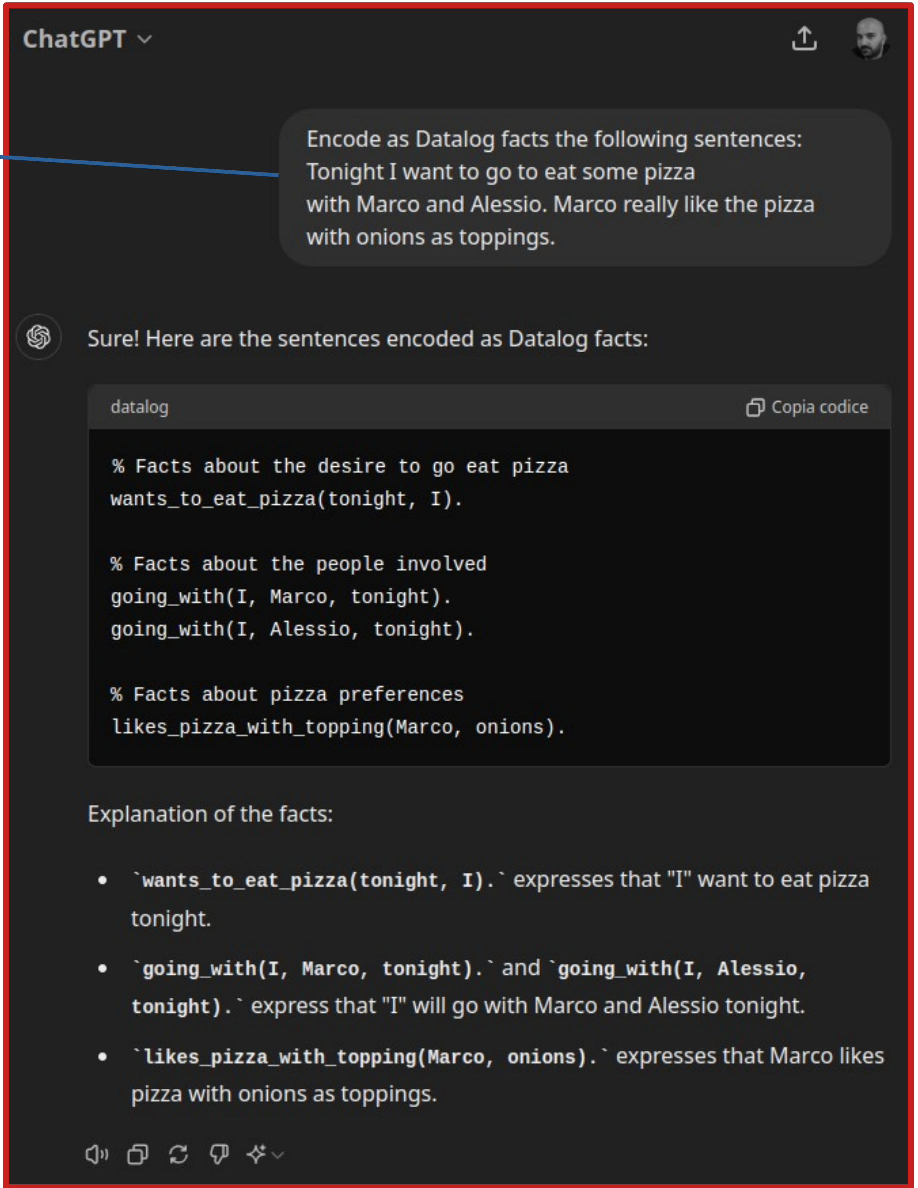
Combine facts with a KB (an ASP program)

Map the answer set to natural language

Users will not see ASP
(not that before we were showing them ASP)

LLMs themselves

A simple PROMPT asking for facts



Encode as Datalog facts the following sentences:
Tonight I want to go to eat some pizza
with Marco and Alessio. Marco really like the pizza
with onions as toppings.

Sure! Here are the sentences encoded as Datalog facts:

```
% Facts about the desire to go eat pizza
wants_to_eat_pizza(tonight, I).

% Facts about the people involved
going_with(I, Marco, tonight).
going_with(I, Alessio, tonight).

% Facts about pizza preferences
likes_pizza_with_topping(Marco, onions).
```

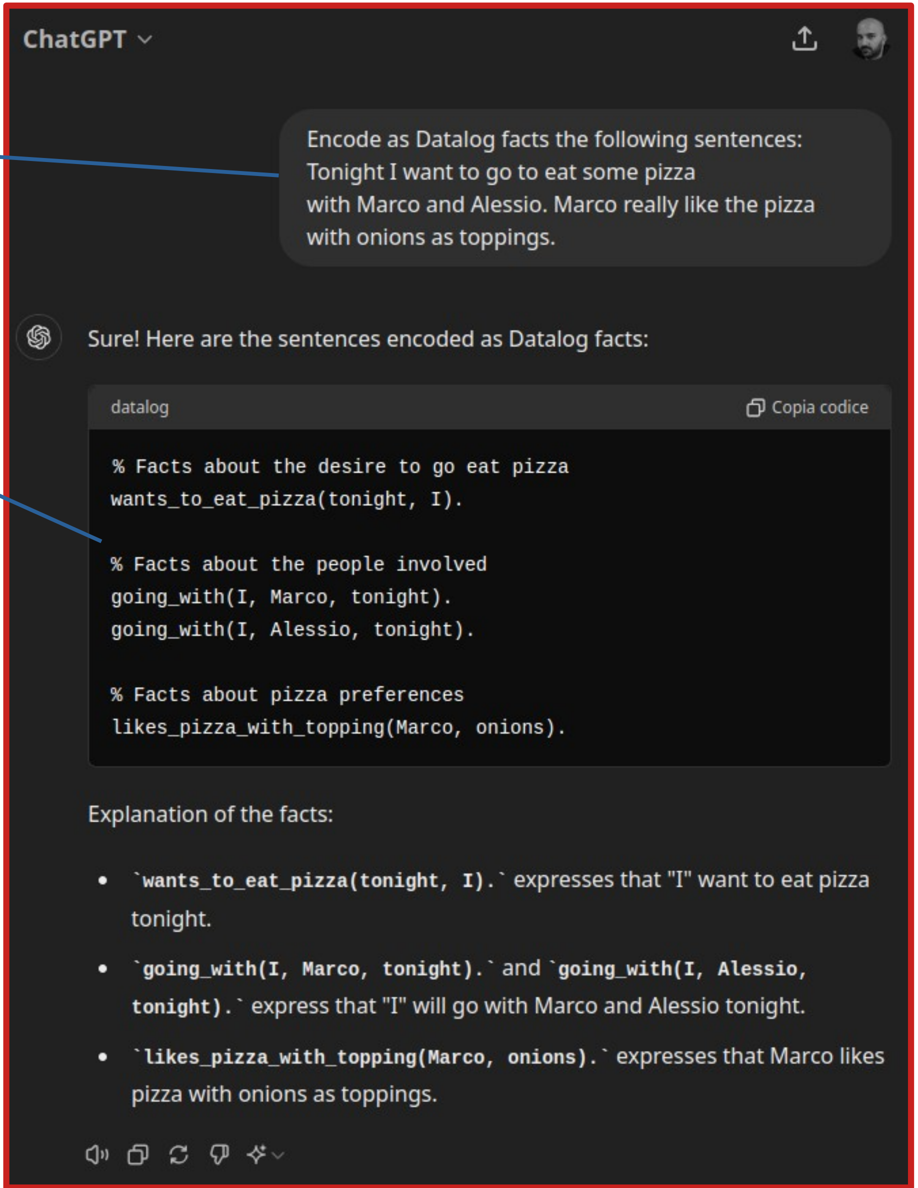
Explanation of the facts:

- `wants_to_eat_pizza(tonight, I).` expresses that "I" want to eat pizza tonight.
- `going_with(I, Marco, tonight).` and `going_with(I, Alessio, tonight).` express that "I" will go with Marco and Alessio tonight.
- `likes_pizza_with_topping(Marco, onions).` expresses that Marco likes pizza with onions as toppings.

LLMs themselves

A simple PROMPT asking for facts

We need **more control** on these facts!



Encode as Datalog facts the following sentences:
Tonight I want to go to eat some pizza
with Marco and Alessio. Marco really like the pizza
with onions as toppings.

Sure! Here are the sentences encoded as Datalog facts:

```datalog
% Facts about the desire to go eat pizza
wants_to_eat_pizza(tonight, I).

% Facts about the people involved
going_with(I, Marco, tonight).
going_with(I, Alessio, tonight).

% Facts about pizza preferences
likes_pizza_with_topping(Marco, onions).
```

Explanation of the facts:

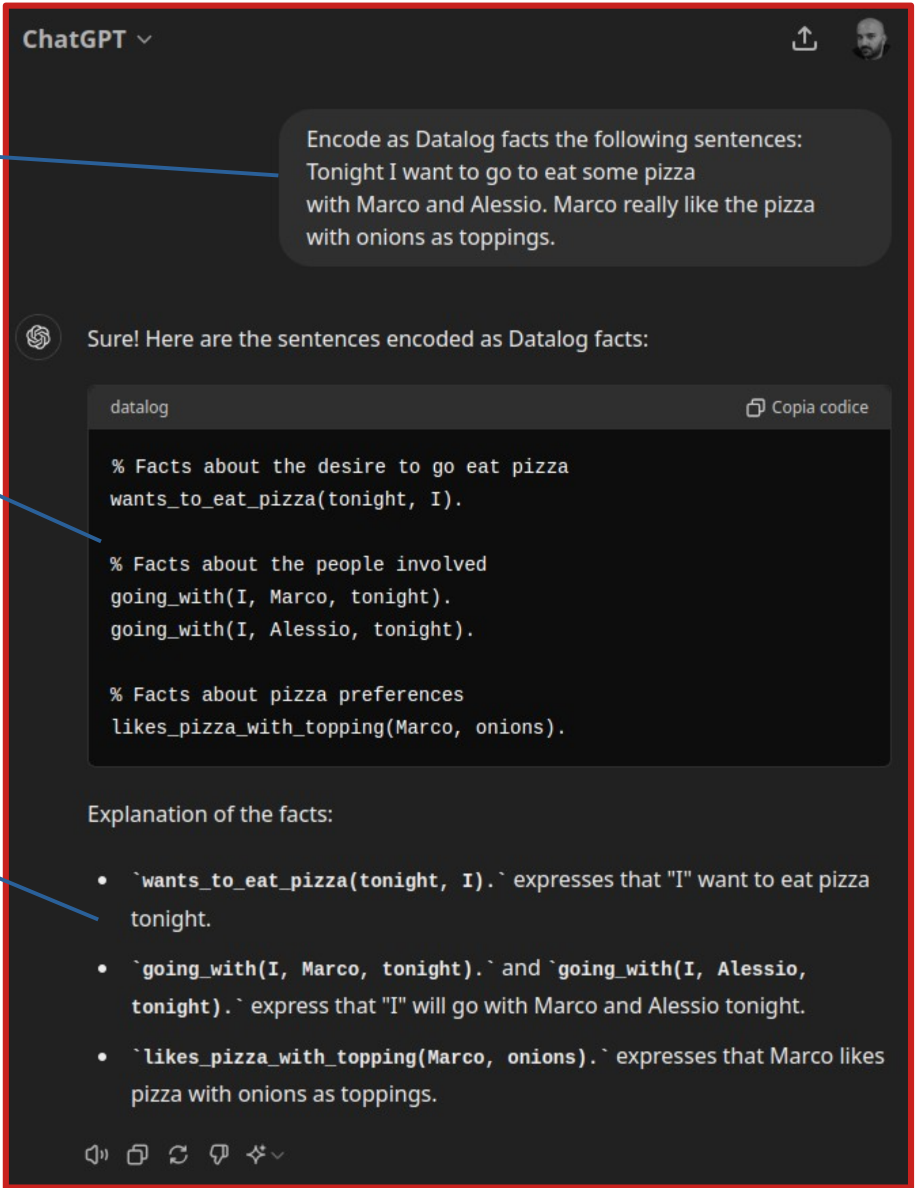
- `wants_to_eat_pizza(tonight, I).` expresses that "I" want to eat pizza tonight.
- `going_with(I, Marco, tonight).` and `going_with(I, Alessio, tonight).` express that "I" will go with Marco and Alessio tonight.
- `likes_pizza_with_topping(Marco, onions).` expresses that Marco likes pizza with onions as toppings.

LLMs themselves

A simple PROMPT asking for facts

We need **more control** on these facts!

And we don't need this garbage



Encode as Datalog facts the following sentences:
Tonight I want to go to eat some pizza
with Marco and Alessio. Marco really like the pizza
with onions as toppings.

Sure! Here are the sentences encoded as Datalog facts:

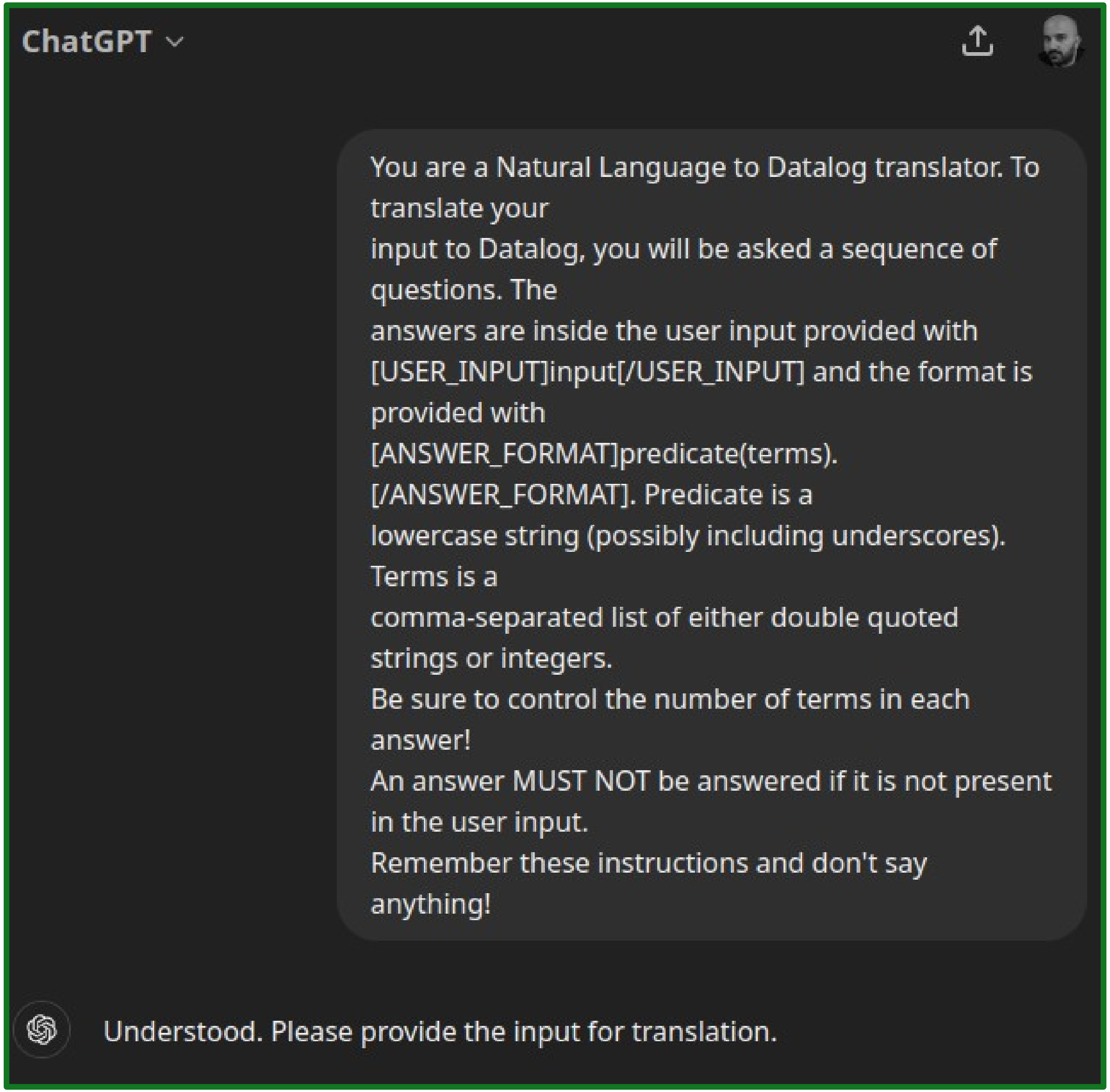
```datalog
% Facts about the desire to go eat pizza
wants_to_eat_pizza(tonight, I).

% Facts about the people involved
going_with(I, Marco, tonight).
going_with(I, Alessio, tonight).

% Facts about pizza preferences
likes_pizza_with_topping(Marco, onions).
```

Explanation of the facts:

- `wants_to_eat_pizza(tonight, I).` expresses that "I" want to eat pizza tonight.
- `going_with(I, Marco, tonight).` and `going_with(I, Alessio, tonight).` express that "I" will go with Marco and Alessio tonight.
- `likes_pizza_with_topping(Marco, onions).` expresses that Marco likes pizza with onions as toppings.

Prompt Engineering

Chat with the LLM…
give instructions
on the format and
task you want to achieve



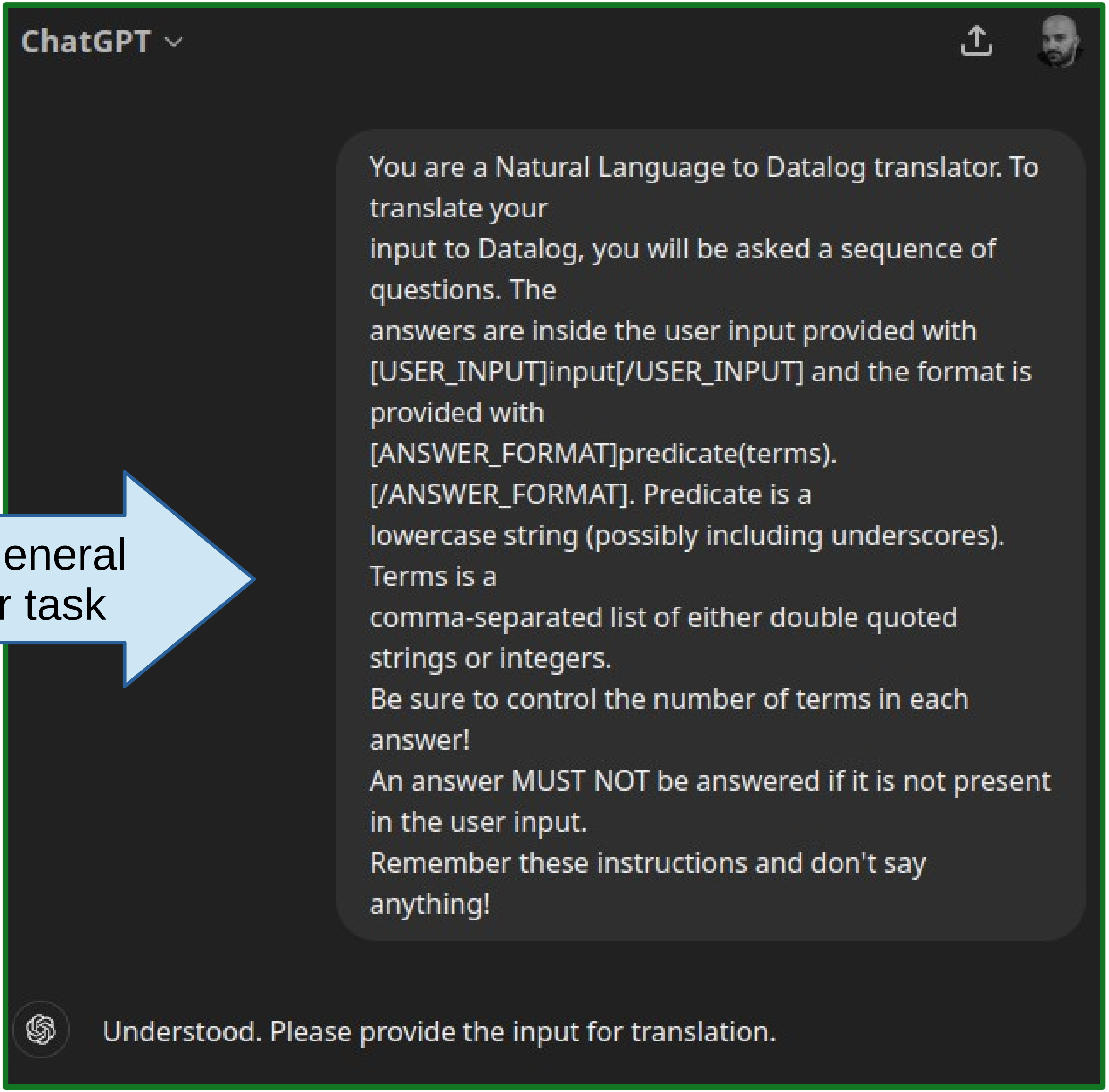
You are a Natural Language to Datalog translator. To translate your
input to Datalog, you will be asked a sequence of questions. The
answers are inside the user input provided with [USER_INPUT]input[/USER_INPUT] and the format is provided with
[ANSWER_FORMAT]predicate(terms).[/ANSWER_FORMAT]. Predicate is a
lowercase string (possibly including underscores). Terms is a
comma-separated list of either double quoted strings or integers.
Be sure to control the number of terms in each answer!
An answer MUST NOT be answered if it is not present in the user input.
Remember these instructions and don't say anything!

Understood. Please provide the input for translation.

Prompt Engineering

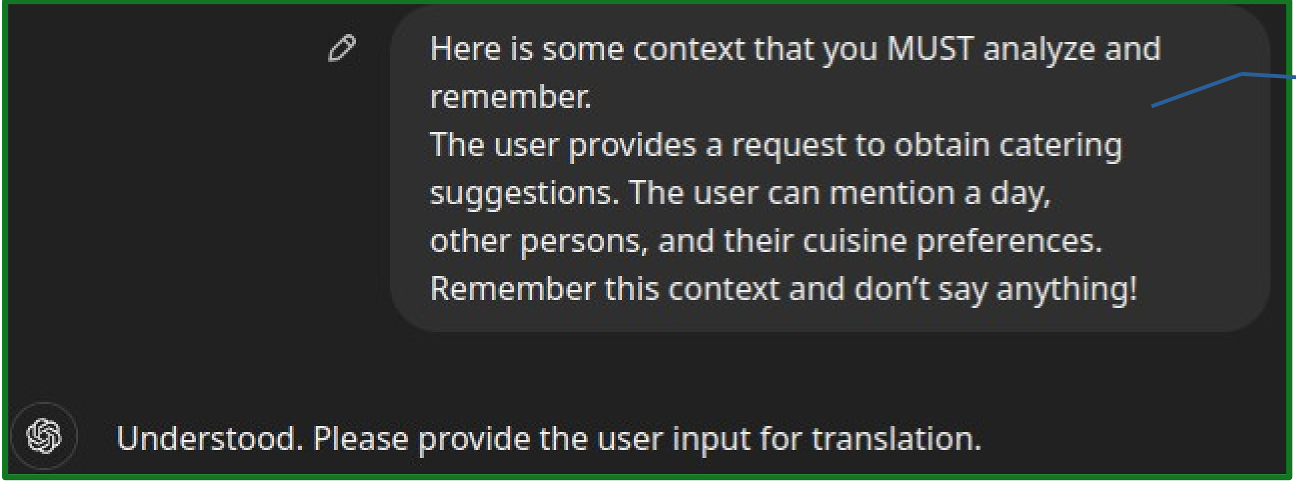
Chat with the LLM…
give instructions
on the format and
task you want to achieve

We can start with general
instructions for our task



You are a Natural Language to Datalog translator. To translate your input to Datalog, you will be asked a sequence of questions. The answers are inside the user input provided with [USER_INPUT]input[/USER_INPUT] and the format is provided with [ANSWER_FORMAT]predicate(terms). [/ANSWER_FORMAT]. Predicate is a lowercase string (possibly including underscores). Terms is a comma-separated list of either double quoted strings or integers.
Be sure to control the number of terms in each answer!
An answer MUST NOT be answered if it is not present in the user input.
Remember these instructions and don't say anything!

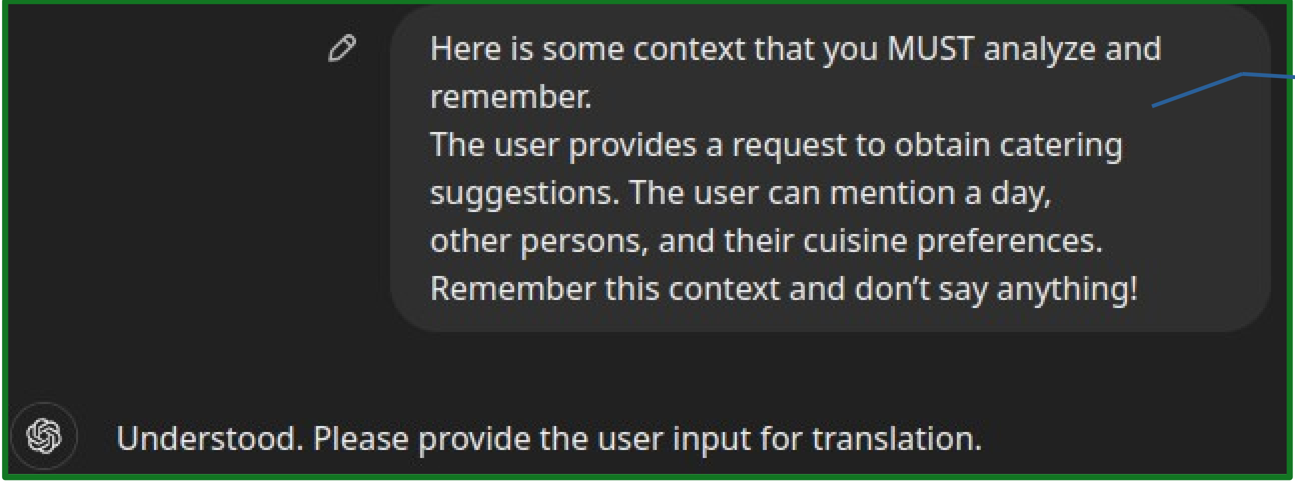
Understood. Please provide the input for translation.



Here is some context that you MUST analyze and remember.
The user provides a request to obtain catering suggestions. The user can mention a day, other persons, and their cuisine preferences.
Remember this context and don't say anything!

Understood. Please provide the user input for translation.

Something specific of the application we are working on

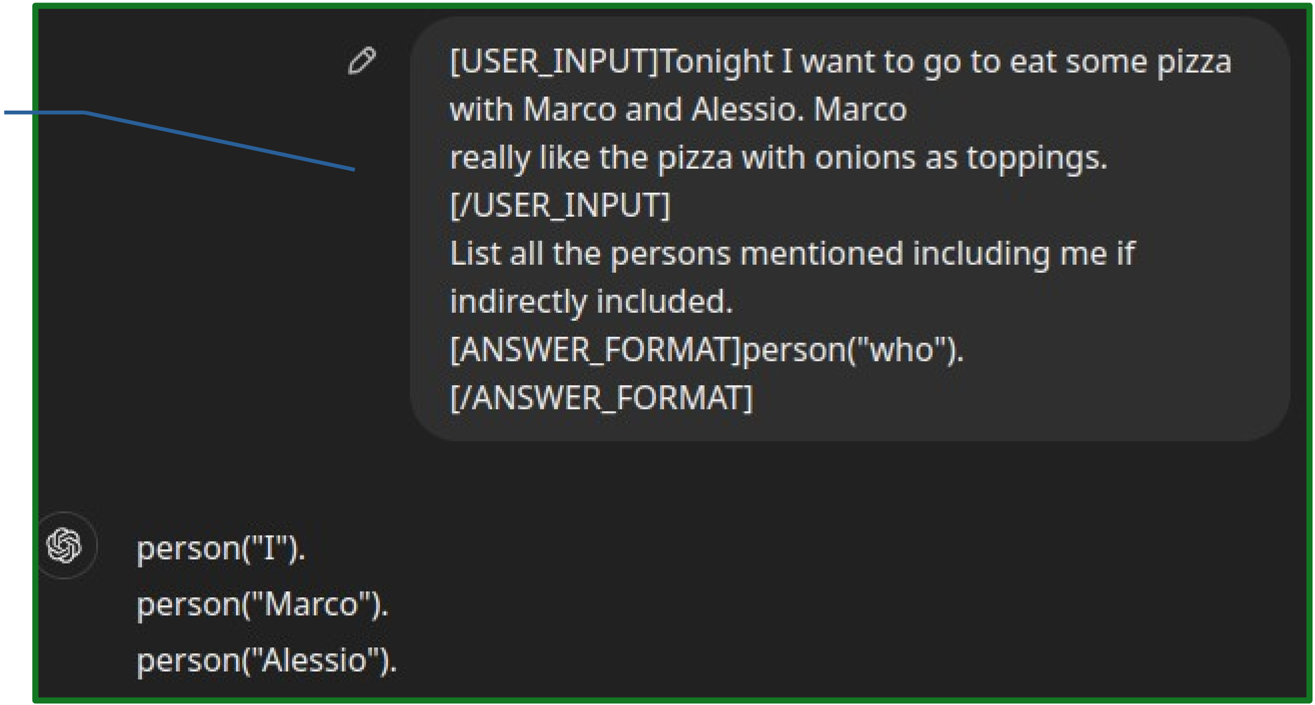


Here is some context that you MUST analyze and remember.
The user provides a request to obtain catering suggestions. The user can mention a day, other persons, and their cuisine preferences.
Remember this context and don’t say anything!

Understood. Please provide the user input for translation.

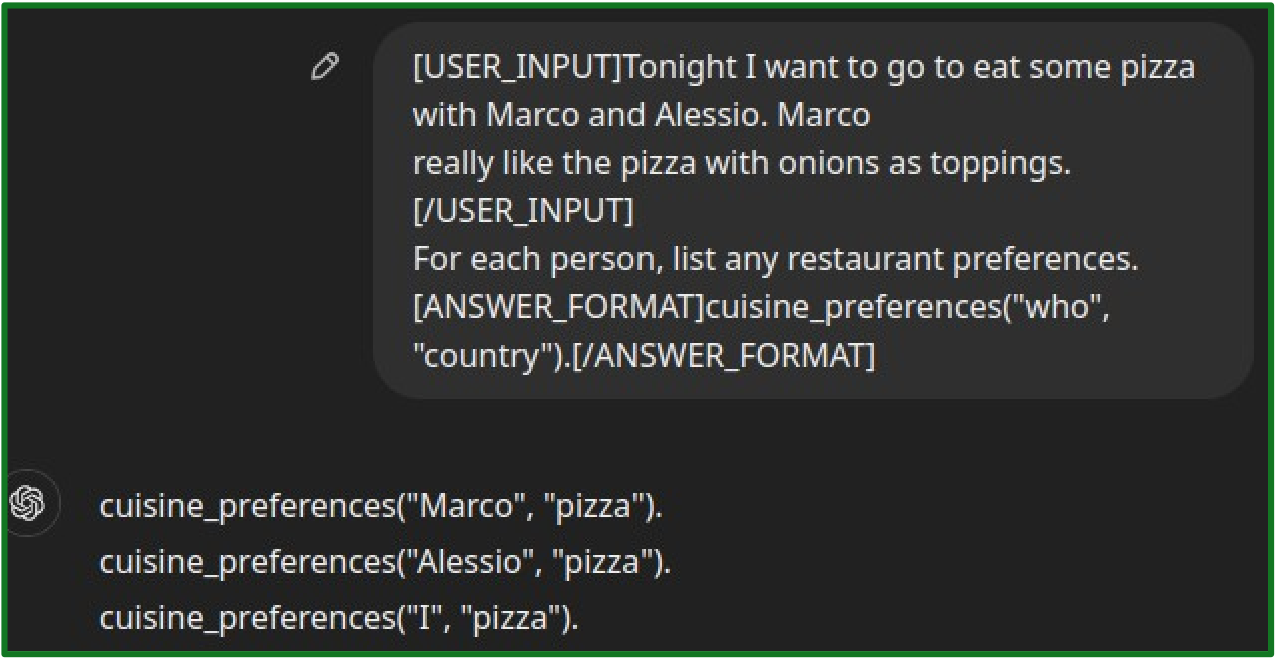
Something specific of the application we are working on

And we can start extracting facts!



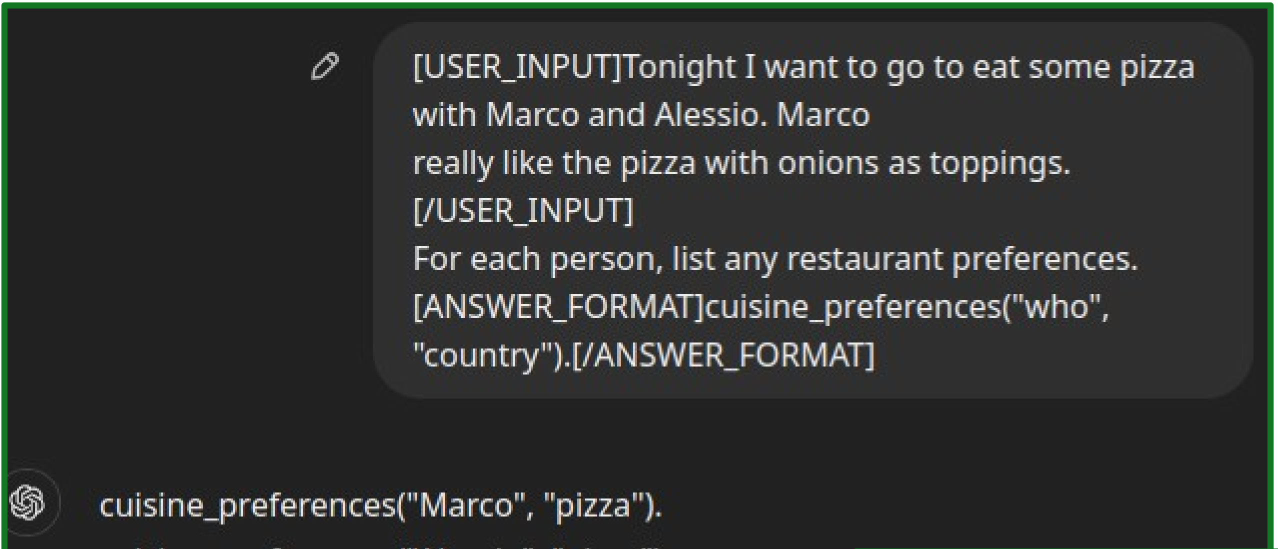
[USER_INPUT]Tonight I want to go to eat some pizza with Marco and Alessio. Marco
really like the pizza with onions as toppings.
[/USER_INPUT]
List all the persons mentioned including me if indirectly included.
[ANSWER_FORMAT]person("who").
[/ANSWER_FORMAT]

```
person("I").
person("Marco").
person("Alessio").
```



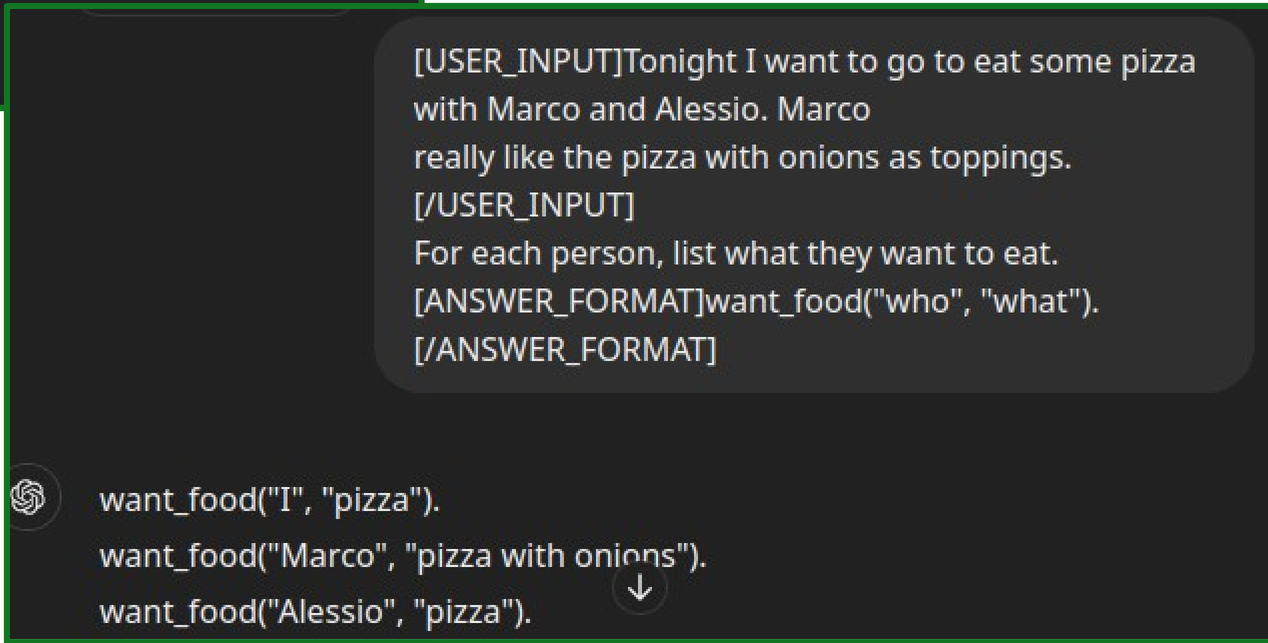
[USER_INPUT]Tonight I want to go to eat some pizza with Marco and Alessio. Marco
really like the pizza with onions as toppings.
[/USER_INPUT]
For each person, list any restaurant preferences.
[ANSWER_FORMAT]cuisine_preferences("who", "country").[/ANSWER_FORMAT]

```
cuisine_preferences("Marco", "pizza").
cuisine_preferences("Alessio", "pizza").
cuisine_preferences("I", "pizza").
```



[USER_INPUT]Tonight I want to go to eat some pizza with Marco and Alessio. Marco
really like the pizza with onions as toppings.
[/USER_INPUT]
For each person, list any restaurant preferences.
[ANSWER_FORMAT]cuisine_preferences("who", "country").[/ANSWER_FORMAT]

```
cuisine_preferences("Marco", "pizza").
cuisine_preferences("Alessio", "pizza").
cuisine_preferences("I", "pizza").
```

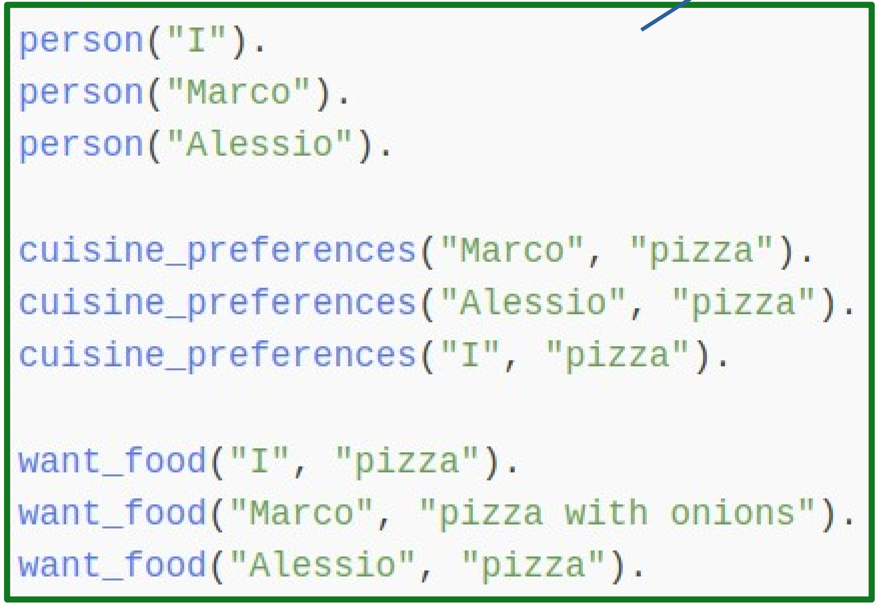


[USER_INPUT]Tonight I want to go to eat some pizza with Marco and Alessio. Marco
really like the pizza with onions as toppings.
[/USER_INPUT]
For each person, list what they want to eat.
[ANSWER_FORMAT]want_food("who", "what").
[/ANSWER_FORMAT]

```
want_food("I", "pizza").
want_food("Marco", "pizza with onions").
want_food("Alessio", "pizza").
```

Reasoning with ASP

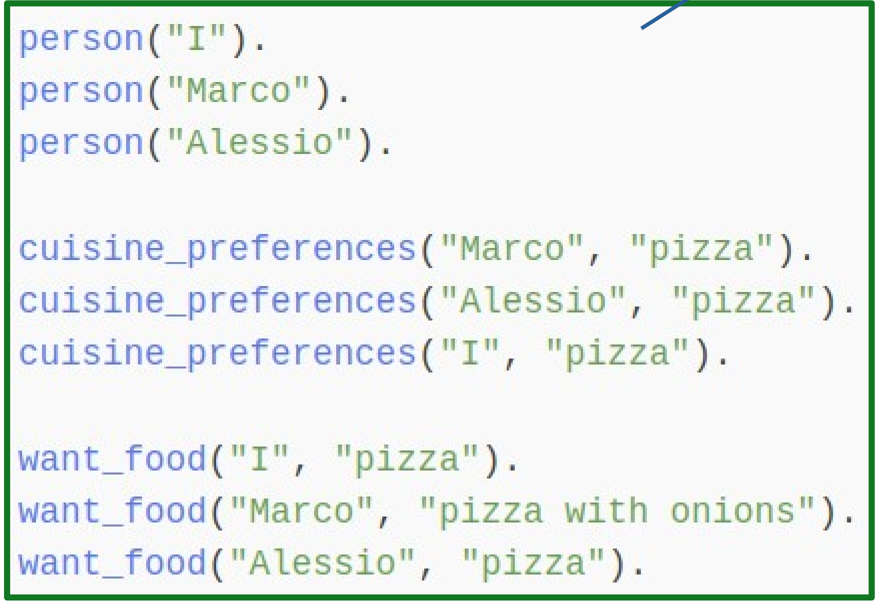
Extracted facts



```
person("I").
person("Marco").
person("Alessio").

cuisine_preferences("Marco", "pizza").
cuisine_preferences("Alessio", "pizza").
cuisine_preferences("I", "pizza").

want_food("I", "pizza").
want_food("Marco", "pizza with onions").
want_food("Alessio", "pizza").
```

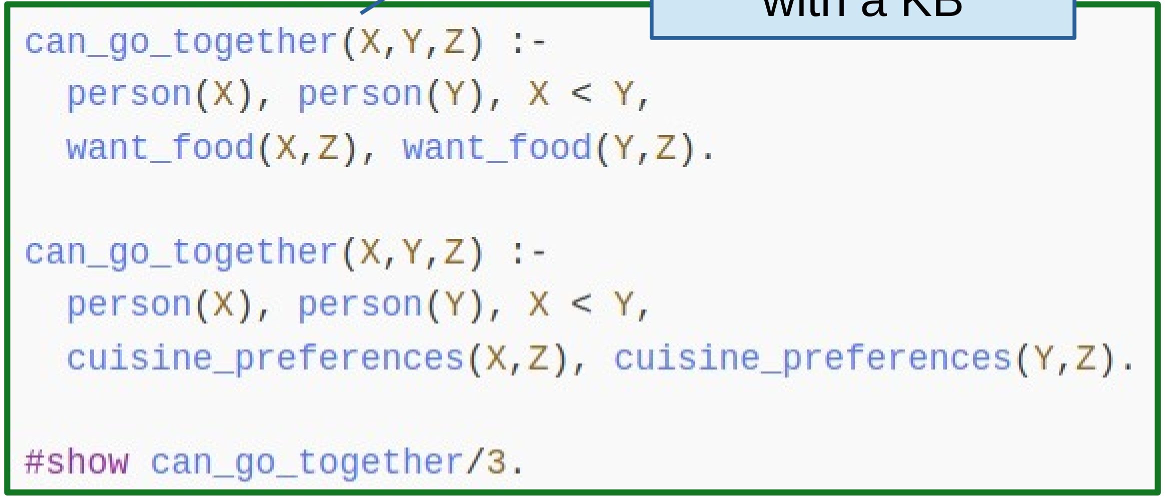
Reasoning with ASP

Extracted facts



```
person("I").
person("Marco").
person("Alessio").

cuisine_preferences("Marco", "pizza").
cuisine_preferences("Alessio", "pizza").
cuisine_preferences("I", "pizza").

want_food("I", "pizza").
want_food("Marco", "pizza with onions").
want_food("Alessio", "pizza").
```

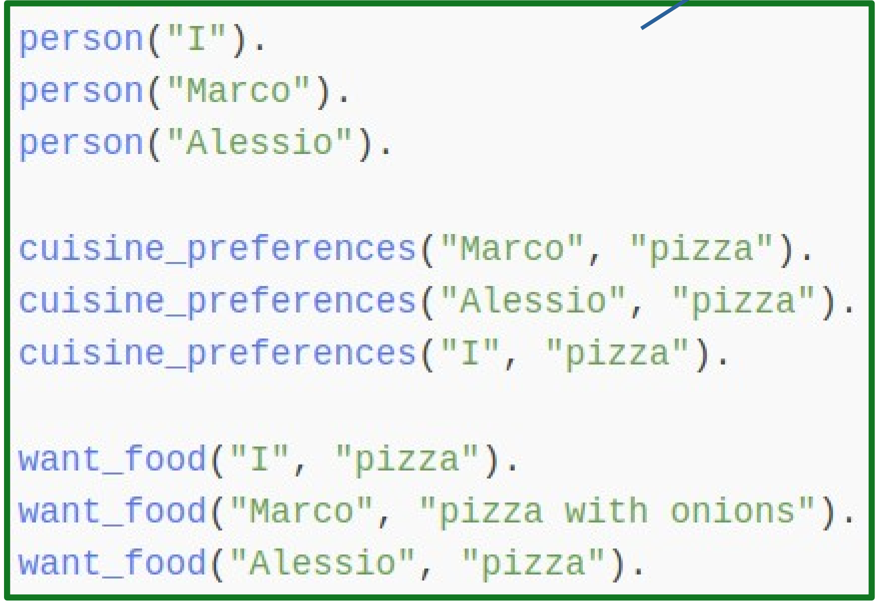
Combine facts with a KB



```
can_go_together(X,Y,Z) :-
  person(X), person(Y), X < Y,
  want_food(X,Z), want_food(Y,Z).

can_go_together(X,Y,Z) :-
  person(X), person(Y), X < Y,
  cuisine_preferences(X,Z), cuisine_preferences(Y,Z).

#show can_go_together/3.
```

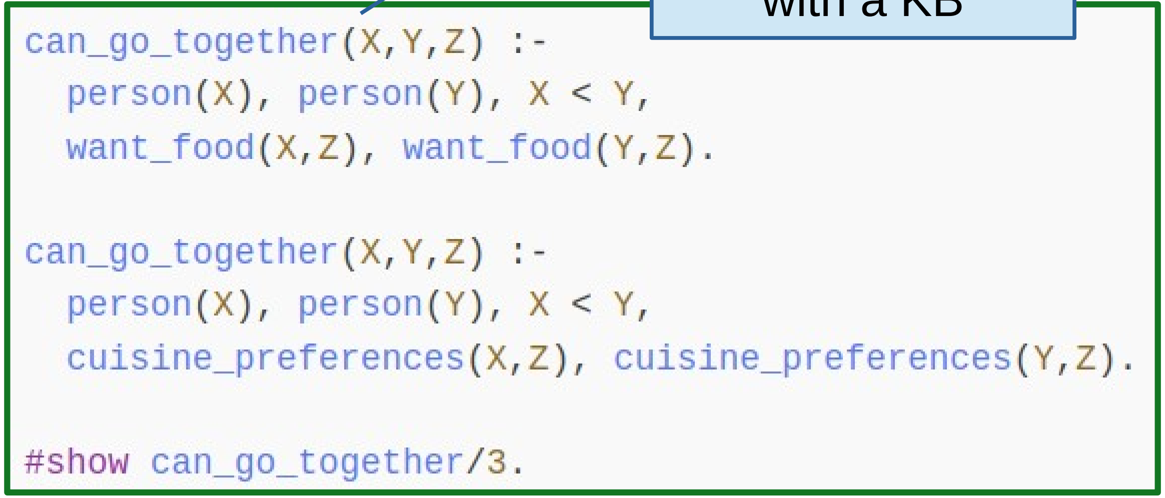
Reasoning with ASP

Extracted facts



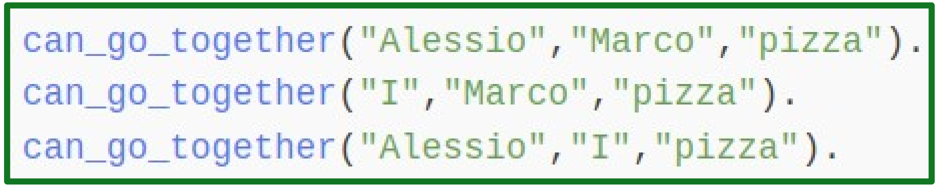
```
person("I").
person("Marco").
person("Alessio").

cuisine_preferences("Marco", "pizza").
cuisine_preferences("Alessio", "pizza").
cuisine_preferences("I", "pizza").

want_food("I", "pizza").
want_food("Marco", "pizza with onions").
want_food("Alessio", "pizza").
```

Combine facts
with a KB



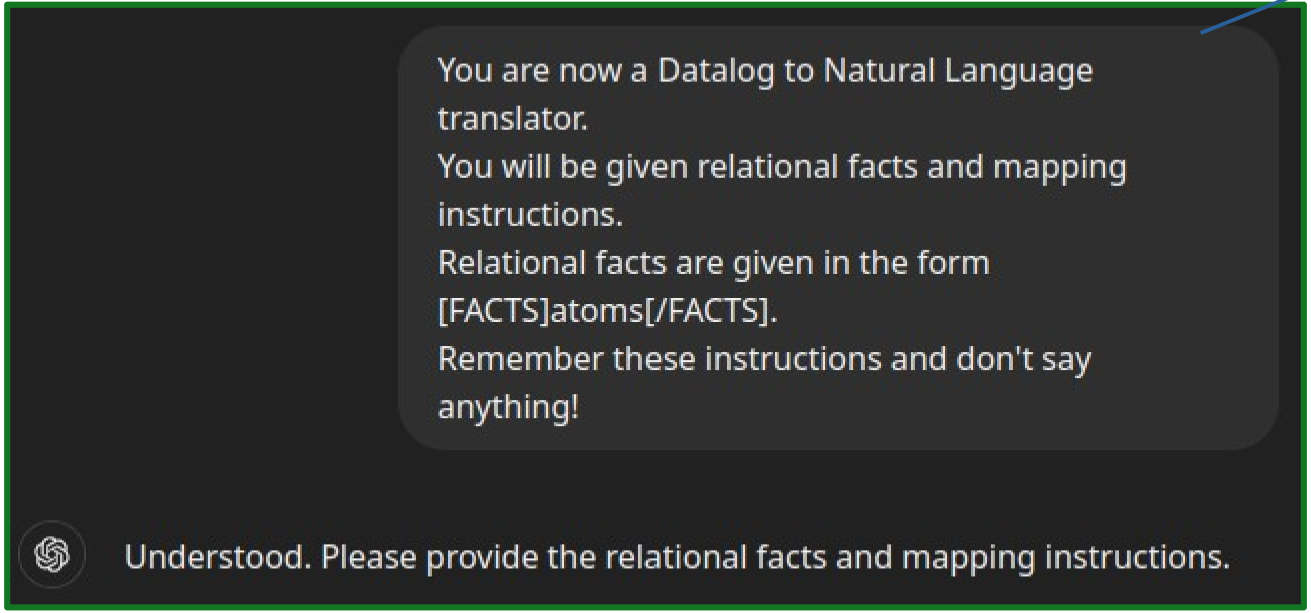
```
can_go_together(X,Y,Z) :-
  person(X), person(Y), X < Y,
  want_food(X,Z), want_food(Y,Z).

can_go_together(X,Y,Z) :-
  person(X), person(Y), X < Y,
  cuisine_preferences(X,Z), cuisine_preferences(Y,Z).

#show can_go_together/3.
```

Obtain an
answer set



```
can_go_together("Alessio","Marco","pizza").
can_go_together("I","Marco","pizza").
can_go_together("Alessio","I","pizza").
```

Back to Natural Language

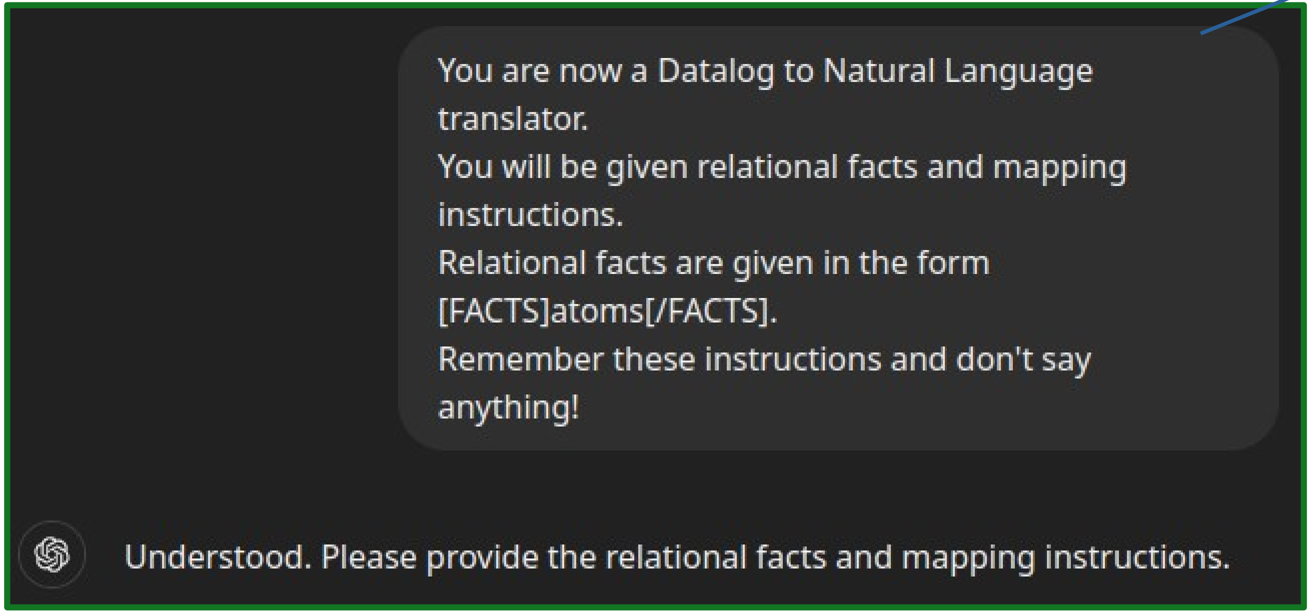


You are now a Datalog to Natural Language translator.
You will be given relational facts and mapping instructions.
Relational facts are given in the form [FACTS]atoms[/FACTS].
Remember these instructions and don't say anything!

Understood. Please provide the relational facts and mapping instructions.

General instructions on the task

Back to Natural Language

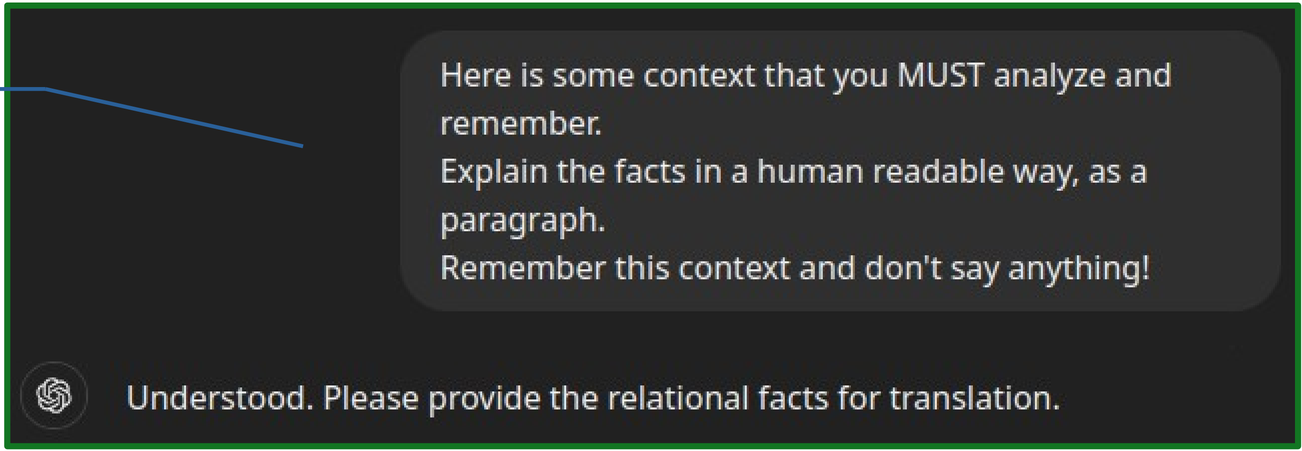


You are now a Datalog to Natural Language translator.
You will be given relational facts and mapping instructions.
Relational facts are given in the form [FACTS]atoms[/FACTS].
Remember these instructions and don't say anything!

Understood. Please provide the relational facts and mapping instructions.

General instructions on the task

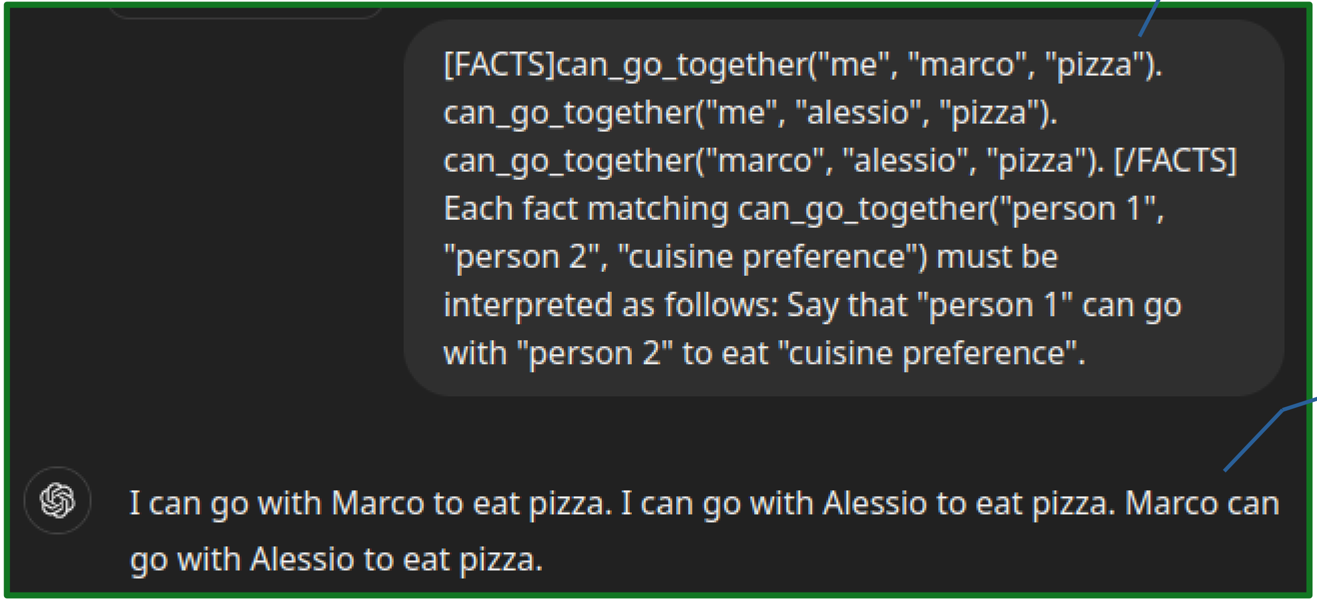
Something specific of the application we are working on



Here is some context that you MUST analyze and remember.
Explain the facts in a human readable way, as a paragraph.
Remember this context and don't say anything!

Understood. Please provide the relational facts for translation.

Provide facts and mapping instructions

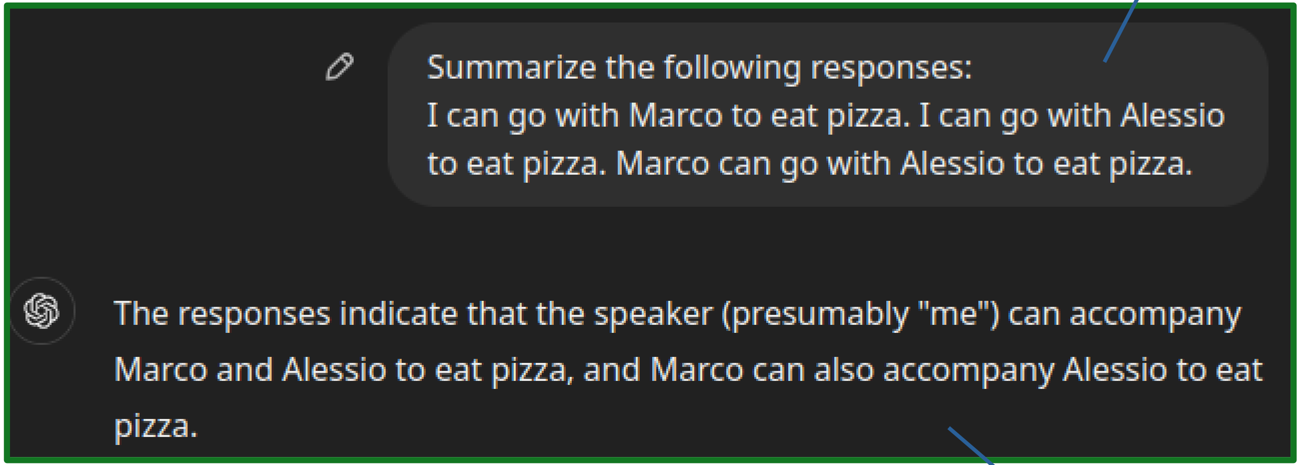


[FACTS]can_go_together("me", "marco", "pizza"). can_go_together("me", "alessio", "pizza"). can_go_together("marco", "alessio", "pizza"). [/FACTS] Each fact matching can_go_together("person 1", "person 2", "cuisine preference") must be interpreted as follows: Say that "person 1" can go with "person 2" to eat "cuisine preference".

I can go with Marco to eat pizza. I can go with Alessio to eat pizza. Marco can go with Alessio to eat pizza.

Collect answers

Ask to summarize the collected responses



Summarize the following responses:
I can go with Marco to eat pizza. I can go with Alessio to eat pizza. Marco can go with Alessio to eat pizza.

The responses indicate that the speaker (presumably "me") can accompany Marco and Alessio to eat pizza, and Marco can also accompany Alessio to eat pizza.

This is the output shown to the user

Let’s Automate IT!

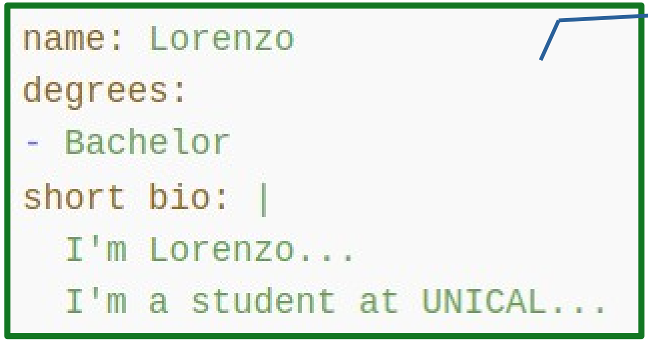
YAML (yet another markup language)

Like JSON… but more readable

Let’s Automate IT!

YAML (yet another markup language)

Like JSON… but more readable



```yaml
name: Lorenzo
degrees:
- Bachelor
short bio: |
  I'm Lorenzo...
  I'm a student at UNICAL...
```

Here is an object with properties **name**, **degrees** and **short bio**

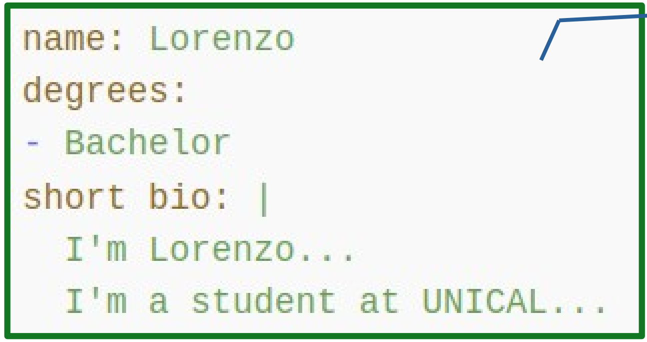
degrees is a list (of values)

short bio is a string (block notation)

Let’s Automate IT!

YAML (yet another markup language)

Like JSON… but more readable



```
name: Lorenzo
degrees:
- Bachelor
short bio: |
  I'm Lorenzo...
  I'm a student at UNICAL...
```

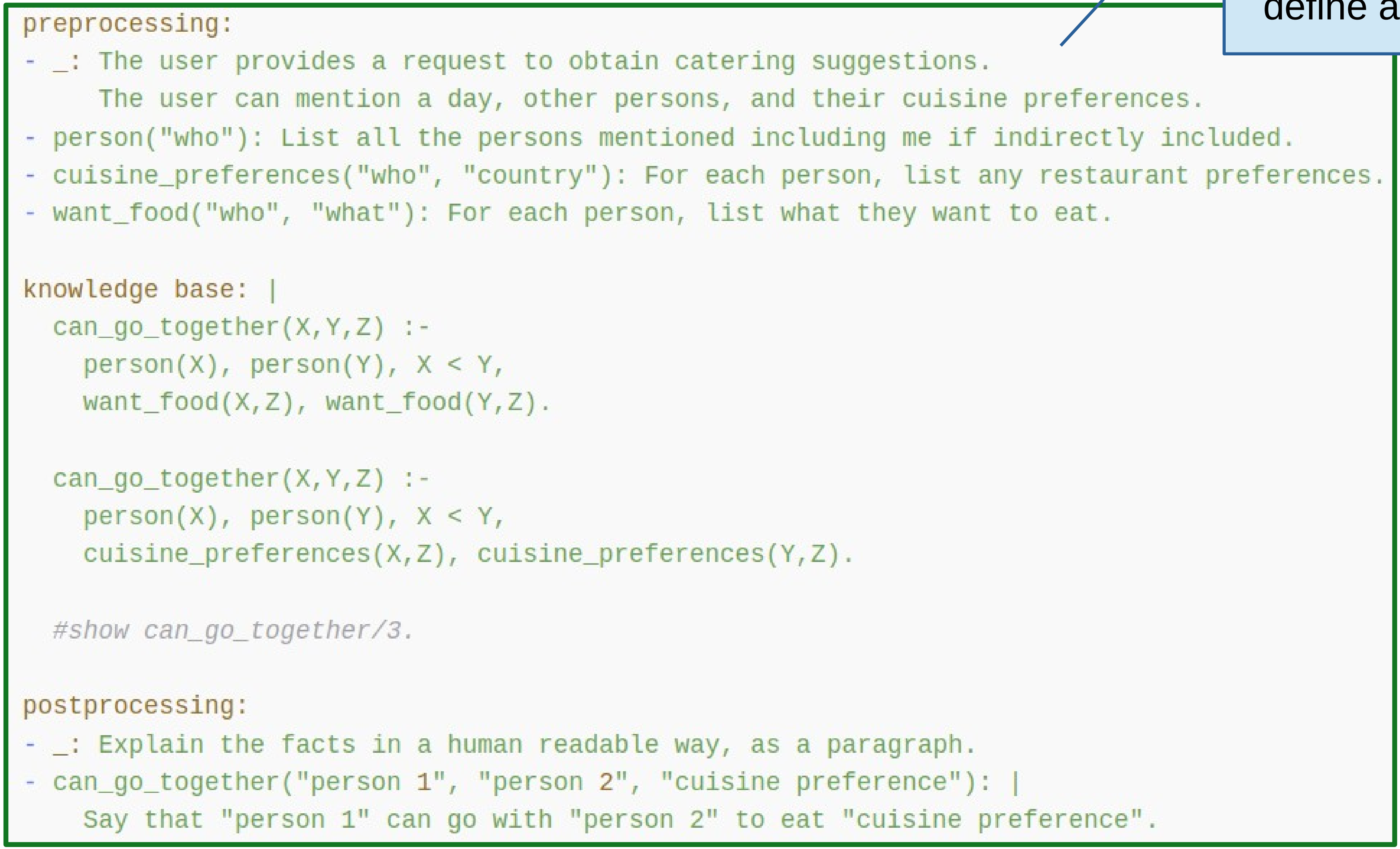
Here is an object with properties **name**, **degrees** and **short bio**

degrees is a list (of values)

short bio is a string (block notation)

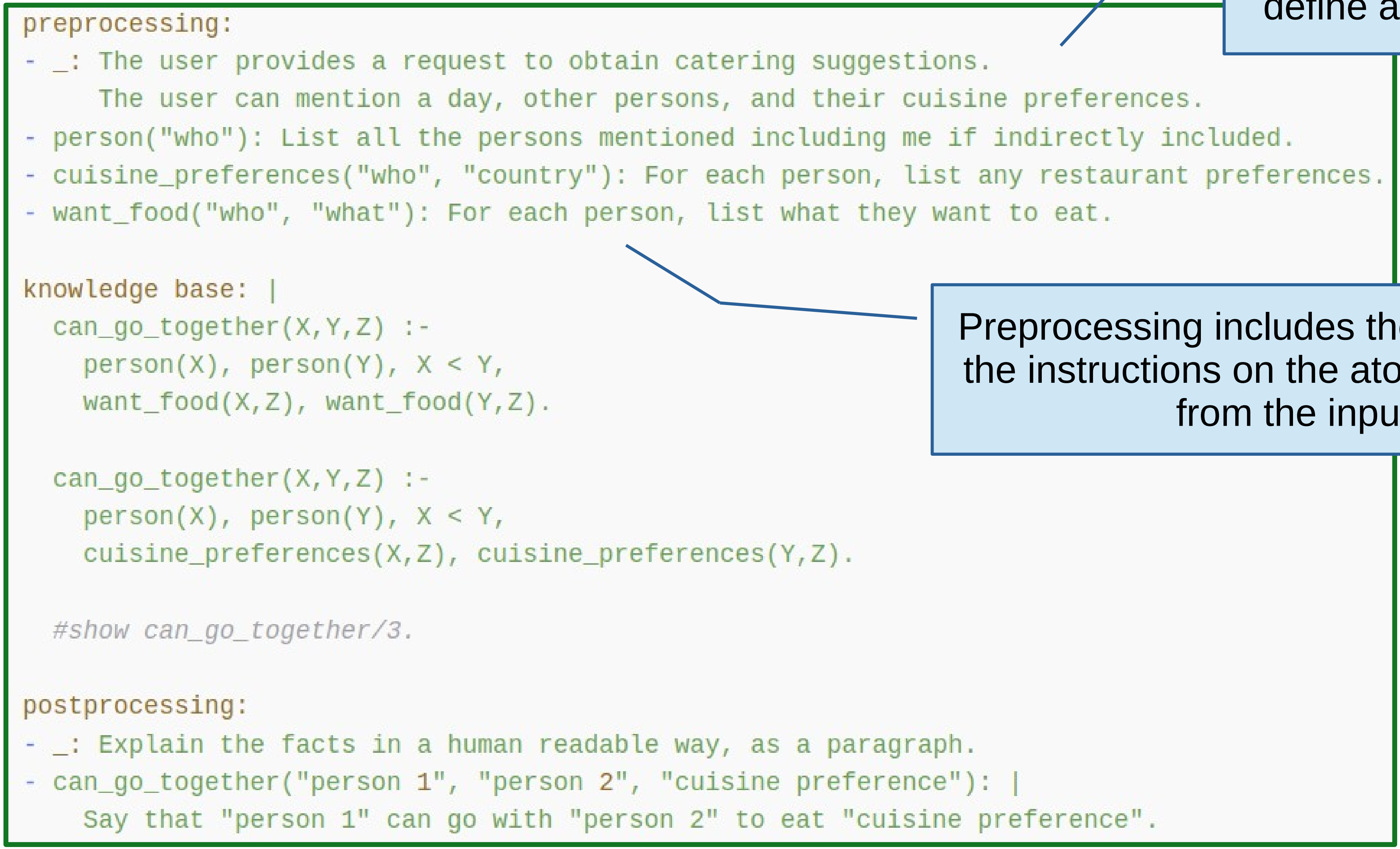
YAML is used in many applications, among them Kubernetes

We use YAML to define an application



```
preprocessing:
- _: The user provides a request to obtain catering suggestions.
     The user can mention a day, other persons, and their cuisine preferences.
- person("who"): List all the persons mentioned including me if indirectly included.
- cuisine_preferences("who", "country"): For each person, list any restaurant preferences.
- want_food("who", "what"): For each person, list what they want to eat.

knowledge base: |
  can_go_together(X,Y,Z) :-
    person(X), person(Y), X < Y,
    want_food(X,Z), want_food(Y,Z).

  can_go_together(X,Y,Z) :-
    person(X), person(Y), X < Y,
    cuisine_preferences(X,Z), cuisine_preferences(Y,Z).

  #show can_go_together/3.

postprocessing:
- _: Explain the facts in a human readable way, as a paragraph.
- can_go_together("person 1", "person 2", "cuisine preference"): |
    Say that "person 1" can go with "person 2" to eat "cuisine preference".
```

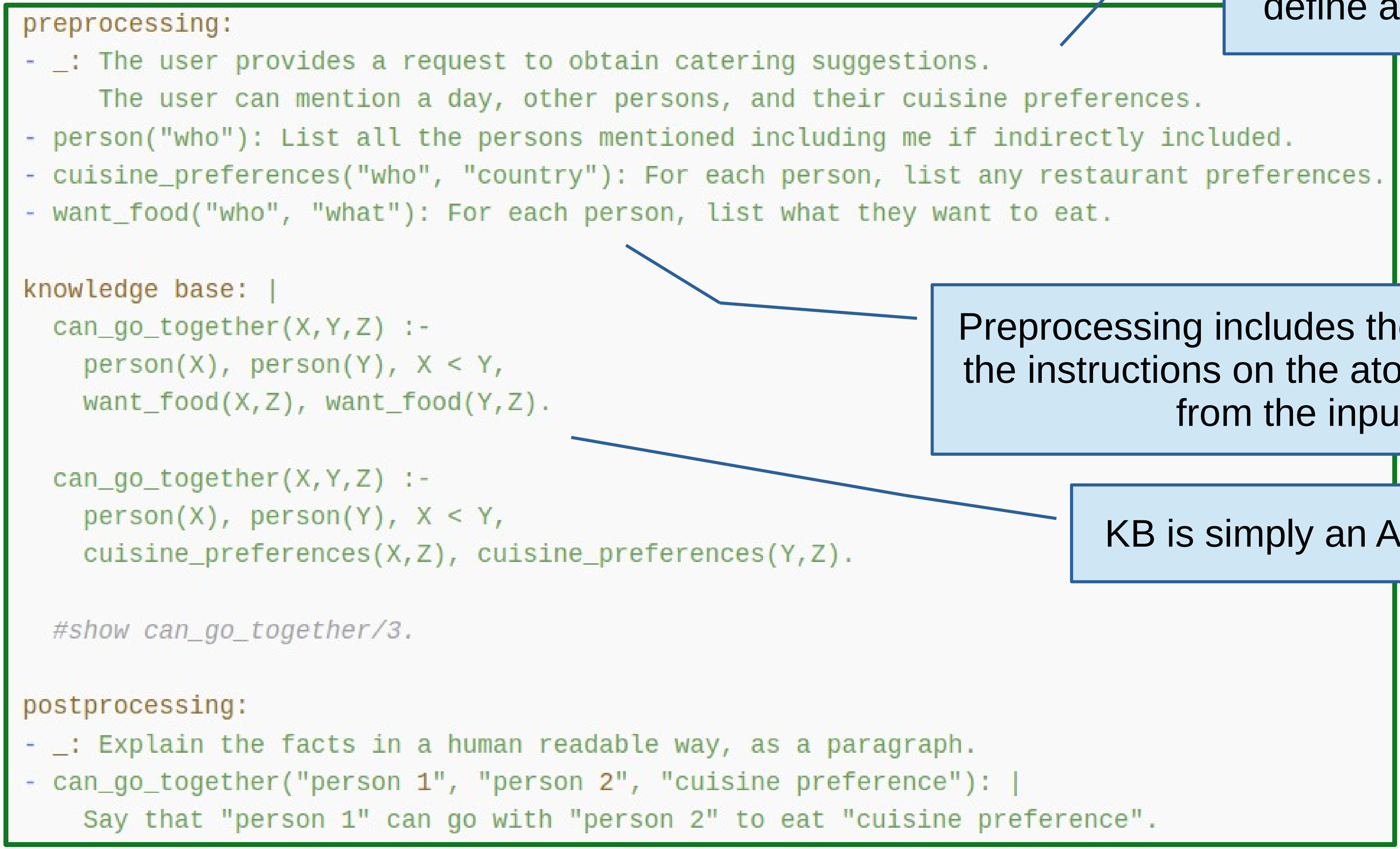
We use YAML to define an application



```
preprocessing:
- _: The user provides a request to obtain catering suggestions.
     The user can mention a day, other persons, and their cuisine preferences.
- person("who"): List all the persons mentioned including me if indirectly included.
- cuisine_preferences("who", "country"): For each person, list any restaurant preferences.
- want_food("who", "what"): For each person, list what they want to eat.

knowledge base: |
  can_go_together(X,Y,Z) :-
    person(X), person(Y), X < Y,
    want_food(X,Z), want_food(Y,Z).

  can_go_together(X,Y,Z) :-
    person(X), person(Y), X < Y,
    cuisine_preferences(X,Z), cuisine_preferences(Y,Z).

  #show can_go_together/3.

postprocessing:
- _: Explain the facts in a human readable way, as a paragraph.
- can_go_together("person 1", "person 2", "cuisine preference"): |
    Say that "person 1" can go with "person 2" to eat "cuisine preference".
```

Preprocessing includes the context and the instructions on the atoms to extract from the input

We use YAML to define an application

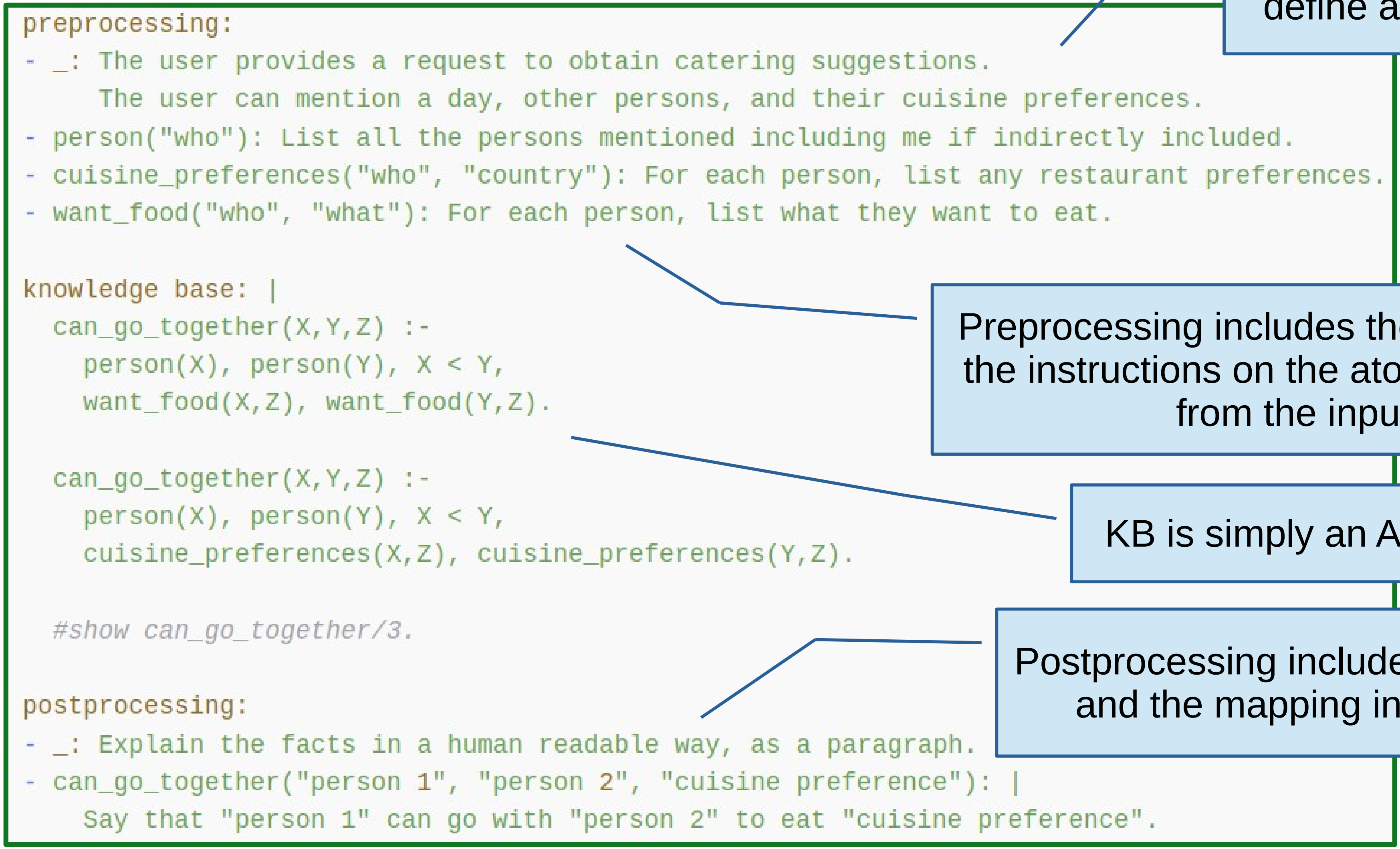


```
preprocessing:
- _: The user provides a request to obtain catering suggestions.
     The user can mention a day, other persons, and their cuisine preferences.
- person("who"): List all the persons mentioned including me if indirectly included.
- cuisine_preferences("who", "country"): For each person, list any restaurant preferences.
- want_food("who", "what"): For each person, list what they want to eat.

knowledge base: |
  can_go_together(X,Y,Z) :-
    person(X), person(Y), X < Y,
    want_food(X,Z), want_food(Y,Z).

  can_go_together(X,Y,Z) :-
    person(X), person(Y), X < Y,
    cuisine_preferences(X,Z), cuisine_preferences(Y,Z).

  #show can_go_together/3.

postprocessing:
- _: Explain the facts in a human readable way, as a paragraph.
- can_go_together("person 1", "person 2", "cuisine preference"): |
    Say that "person 1" can go with "person 2" to eat "cuisine preference".
```

Preprocessing includes the context and the instructions on the atoms to extract from the input

KB is simply an ASP program

We use YAML to define an application



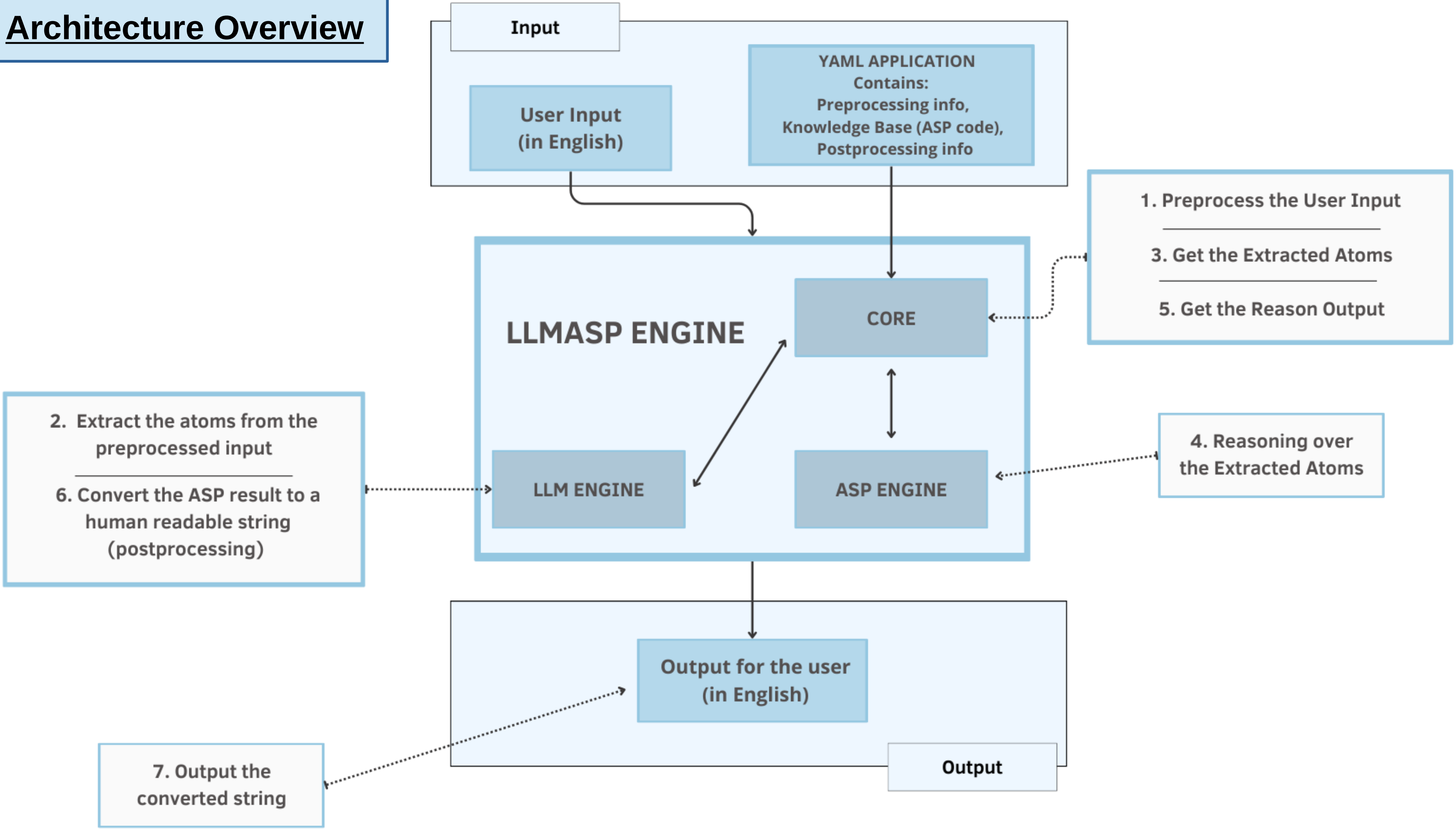
```
preprocessing:
- _: The user provides a request to obtain catering suggestions.
     The user can mention a day, other persons, and their cuisine preferences.
- person("who"): List all the persons mentioned including me if indirectly included.
- cuisine_preferences("who", "country"): For each person, list any restaurant preferences.
- want_food("who", "what"): For each person, list what they want to eat.

knowledge base: |
  can_go_together(X,Y,Z) :-
    person(X), person(Y), X < Y,
    want_food(X,Z), want_food(Y,Z).

  can_go_together(X,Y,Z) :-
    person(X), person(Y), X < Y,
    cuisine_preferences(X,Z), cuisine_preferences(Y,Z).

  #show can_go_together/3.

postprocessing:
- _: Explain the facts in a human readable way, as a paragraph.
- can_go_together("person 1", "person 2", "cuisine preference"): |
    Say that "person 1" can go with "person 2" to eat "cuisine preference".
```

Preprocessing includes the context and the instructions on the atoms to extract from the input

KB is simply an ASP program

Postprocessing includes the context and the mapping instructions

Architecture Overview



Input

User Input
(in English)

YAML APPLICATION
Contains:
Preprocessing info,
Knowledge Base (ASP code),
Postprocessing info

LLMASP ENGINE

CORE

LLM ENGINE

ASP ENGINE

1. Preprocess the User Input

3. Get the Extracted Atoms

5. Get the Reason Output

2. Extract the atoms from the preprocessed input

6. Convert the ASP result to a human readable string (postprocessing)

4. Reasoning over the Extracted Atoms

Output for the user
(in English)

Output

7. Output the converted string

Summing UP

LLMASP combines
NLP capabilities of LLMs
with the deep reasoning of ASP

Summing UP

LLMASP combines
NLP capabilities of LLMs
with the deep reasoning of ASP

We automated the interaction with LLM and ASP engines

Summing UP

LLMASP combines
NLP capabilities of LLMs
with the deep reasoning of ASP

**We automated the interaction
with LLM and ASP engines**

Still a lot to do!

**Why those prompts and not something different?
We have to build a dataset.
We have to try different prompts and measure their quality.**

We can start with the preprocessing and then evaluate the postprocessing by extracting facts from the produced output.

Questions