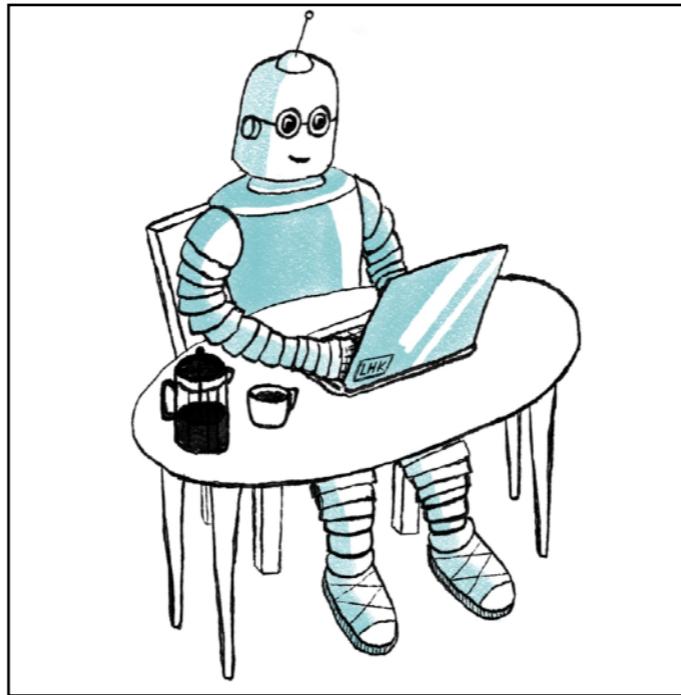


Automating Popper's logic of scientific discovery



Andrew Cropper

University of Oxford

Work with Rolf Morel, David Cerna, Andreas Niskanen, Matti Järvisalo, and
Céline Hocquette

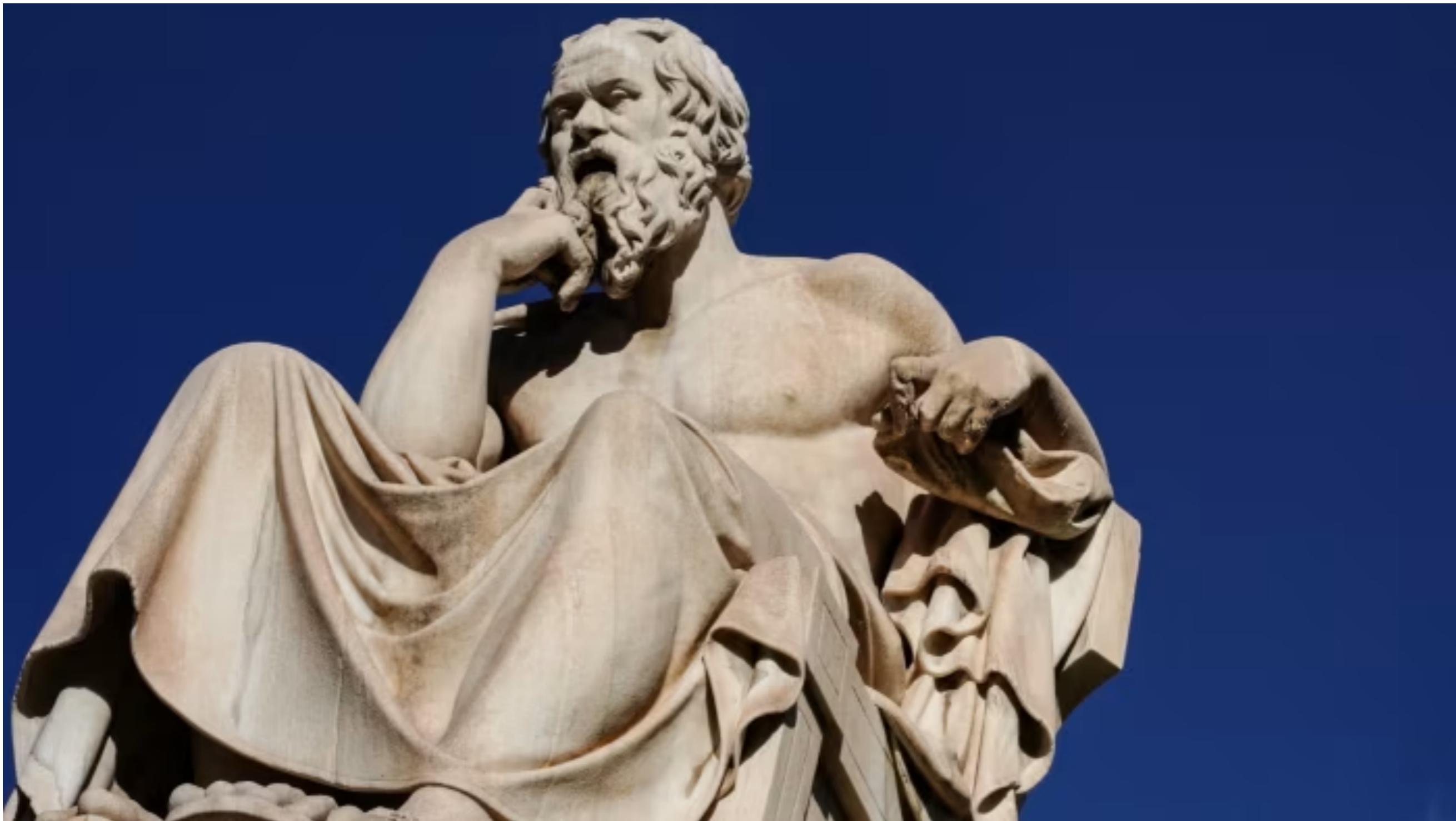
What is this talk about?

Inductive logic programming

What is this talk about?

ILP approach *inspired* by
Karl Popper's logic of discovery

What is this talk not about?



Why care?

Why care?

Simple

Why care?

Good performance

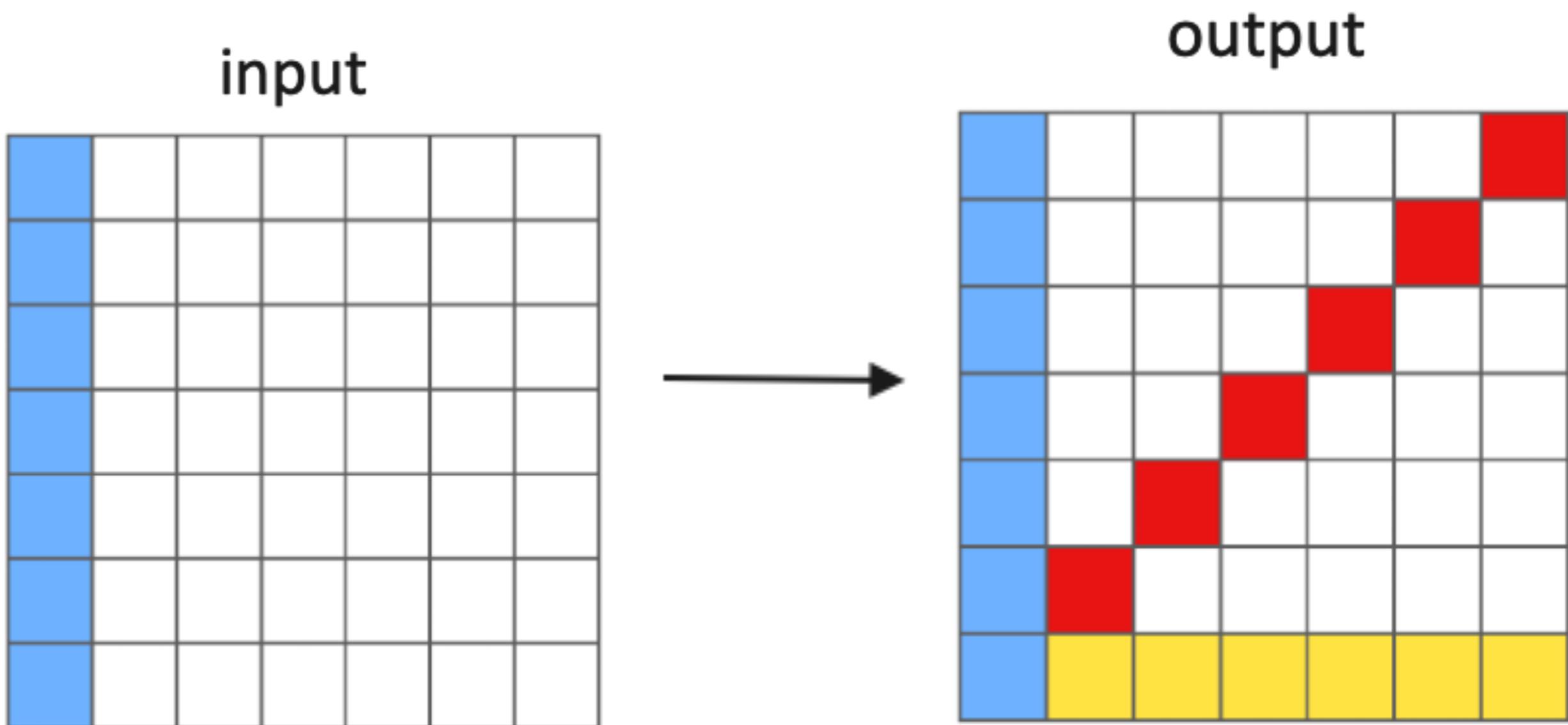
Why care?

New ideas

Why care?



Why care?



Inductive logic programming

Inductive logic programming

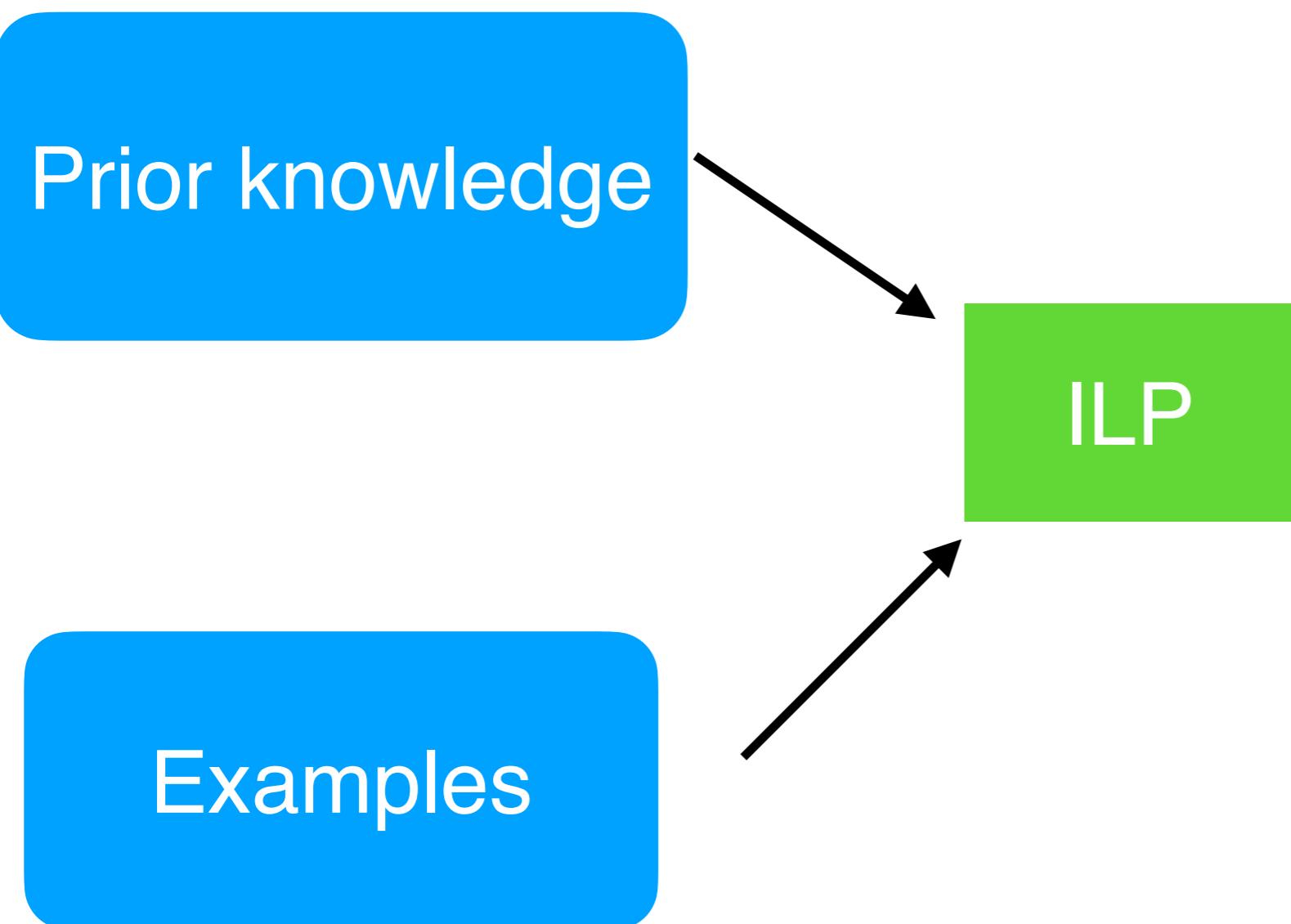
Prior knowledge

Inductive logic programming

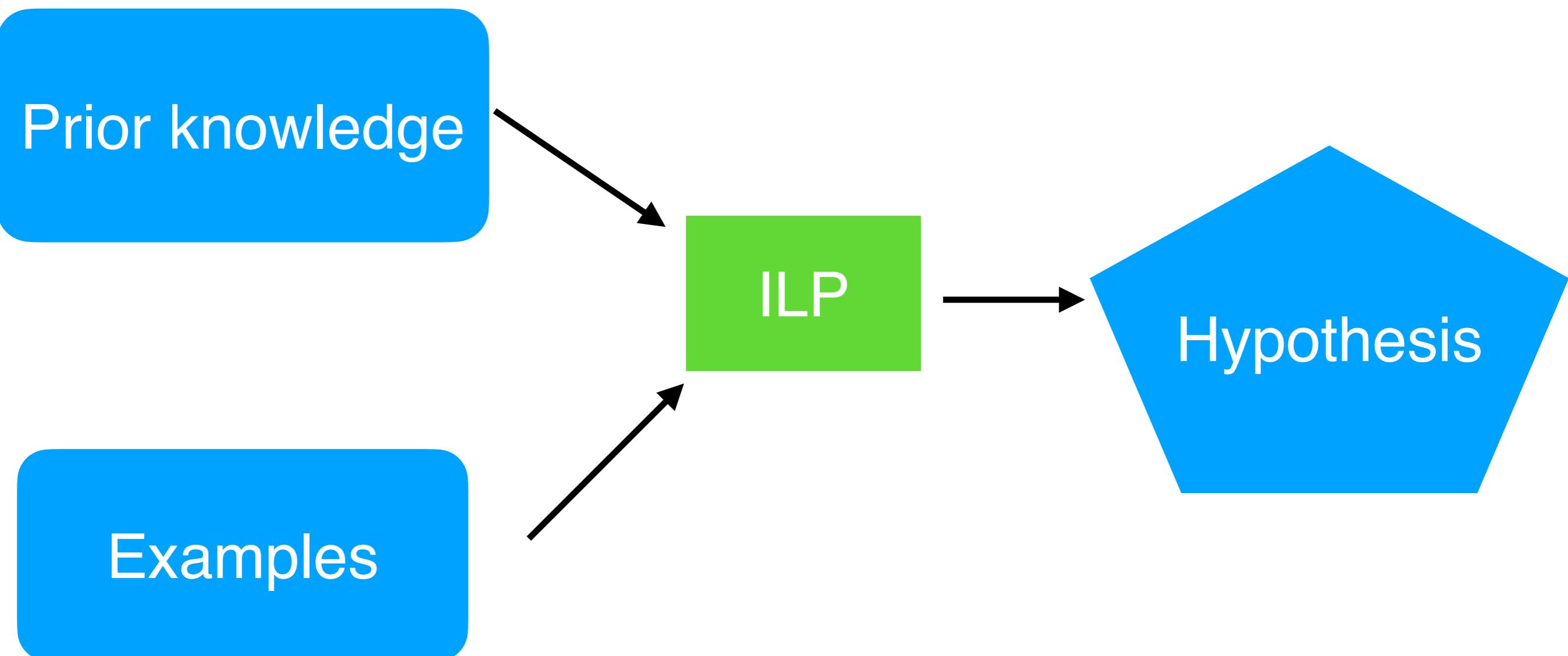
Prior knowledge

Examples

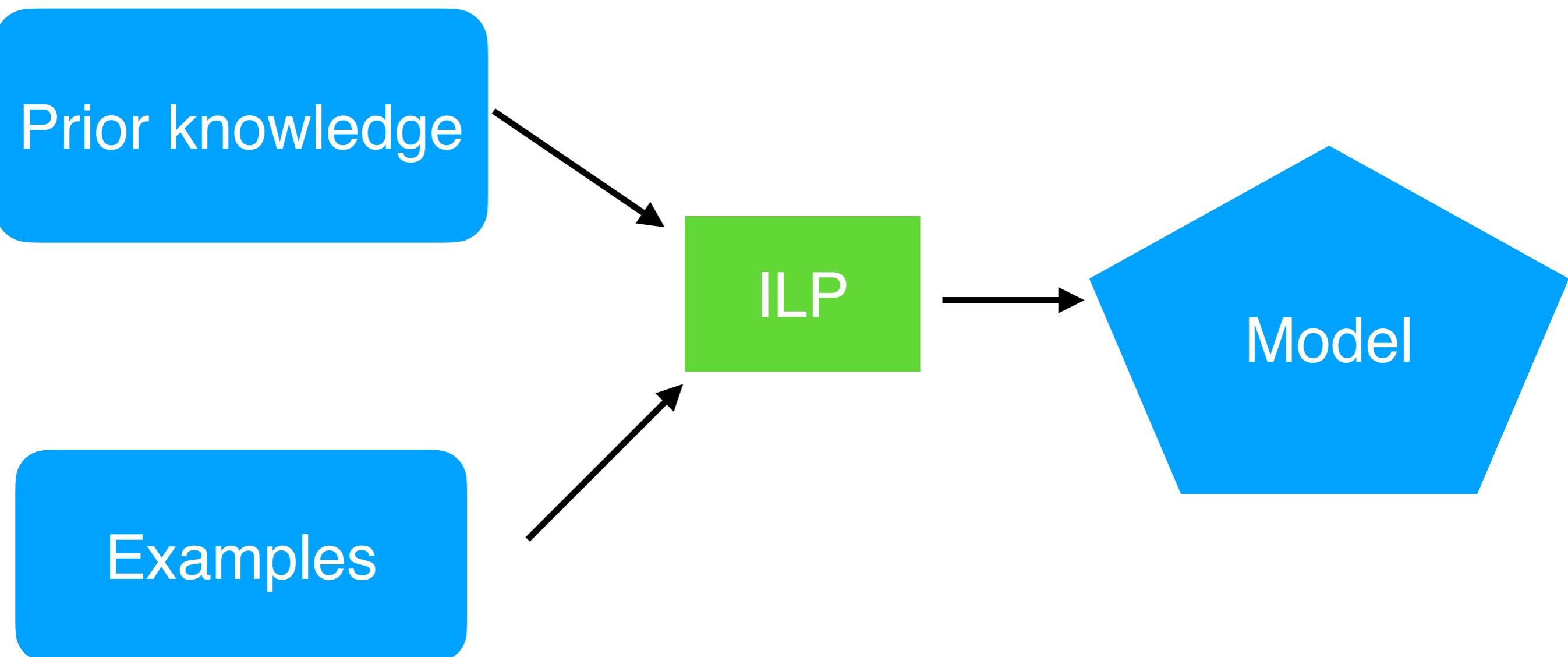
Inductive logic programming



Inductive logic programming

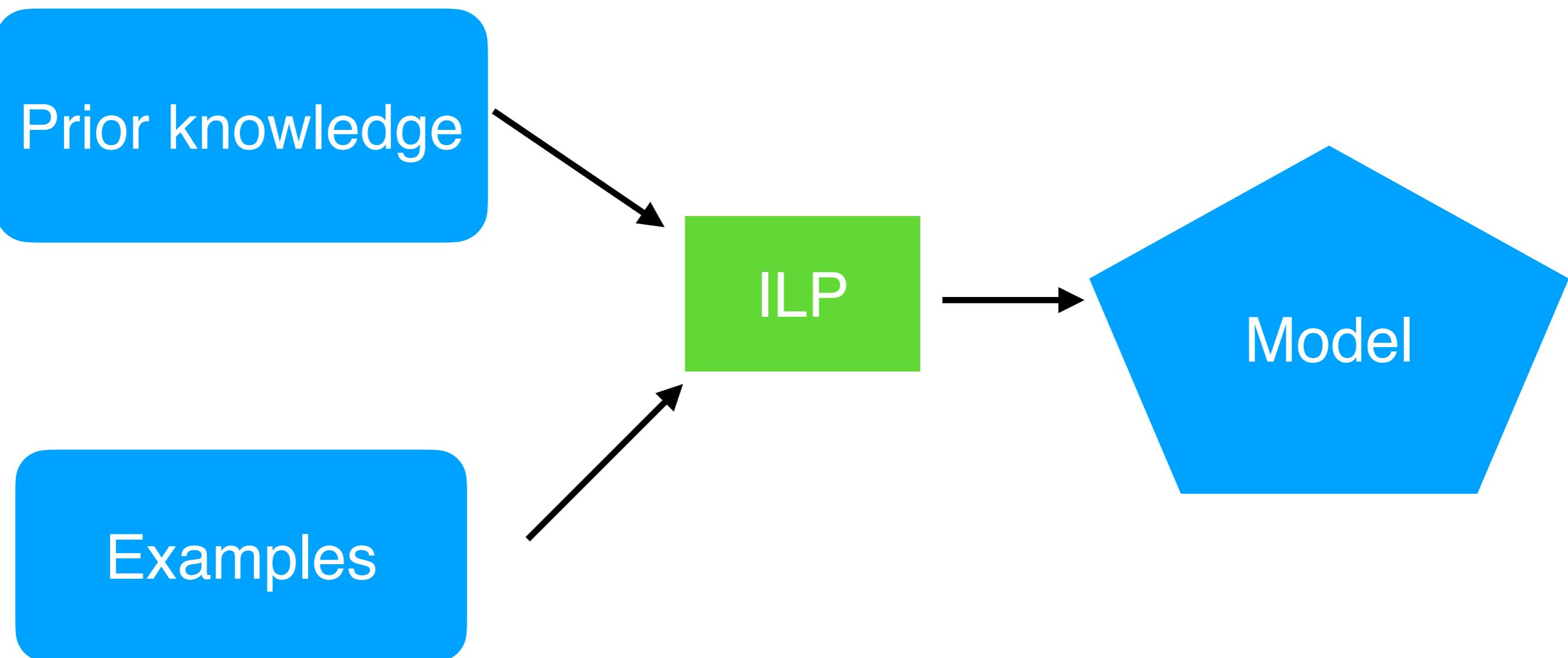


Inductive logic programming



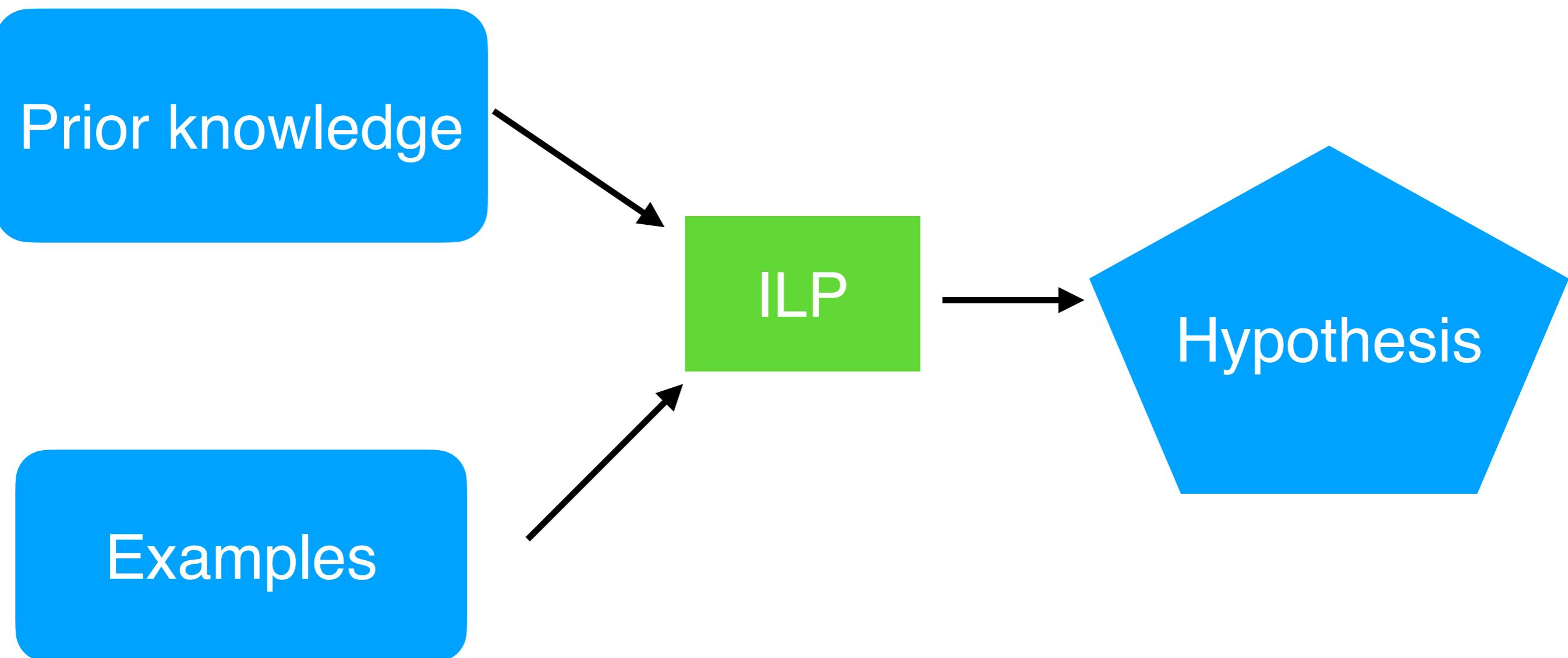
Supervised machine learning

Inductive logic programming



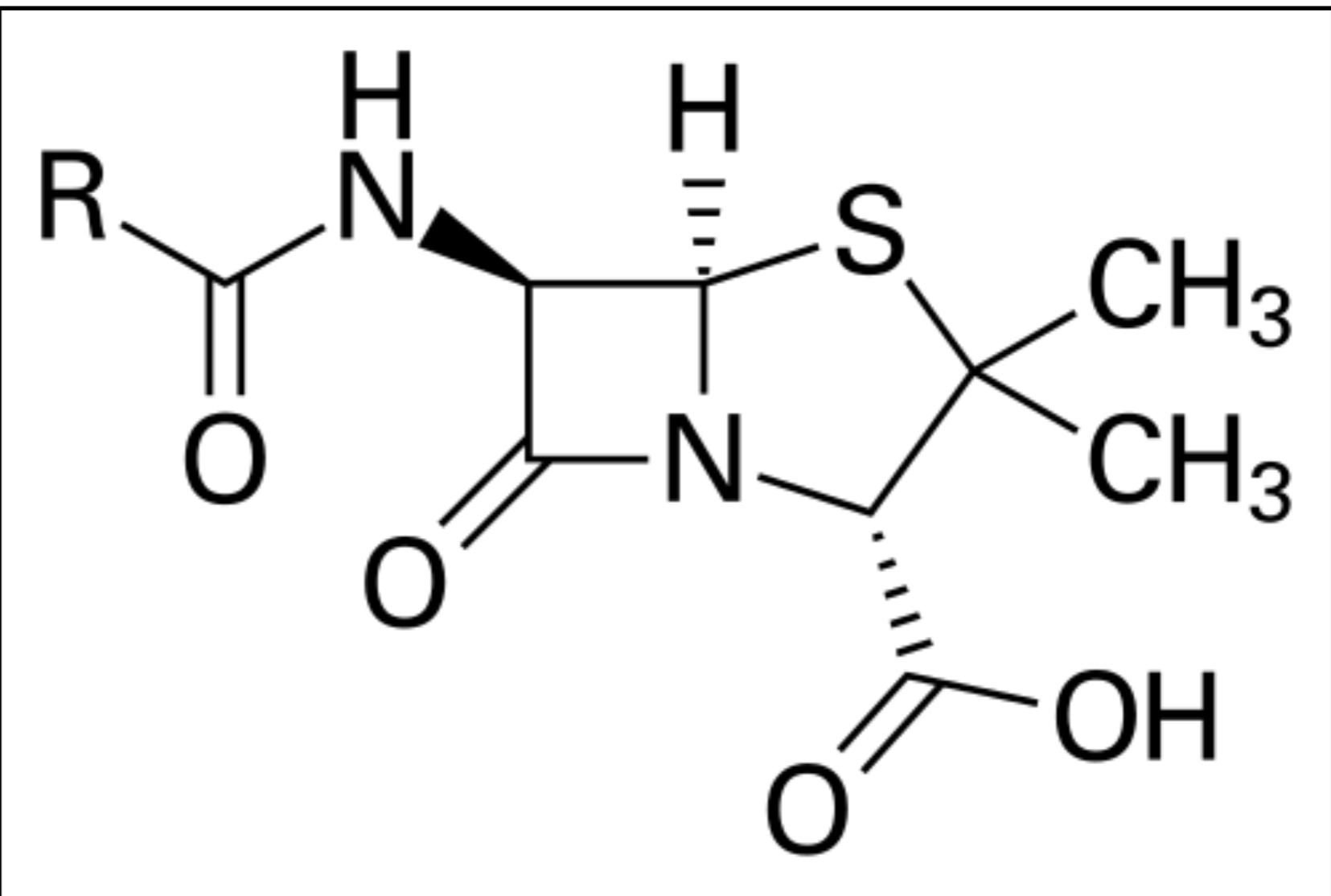
Uses **logic** to represent data

Inductive logic programming



Uses **logic** to represent data

Prior
knowledge



Prior knowledge

```
atom(7, o).  
atom(8, o).  
atom(9, h).  
atom(10, h).  
atom(11, h).  
atom(12, h).  
  
...  
bond(1, 2, single).  
bond(2, 3, single).  
bond(3, 4, single).  
bond(4, 5, single).  
  
...
```



Observations

Hypothesis

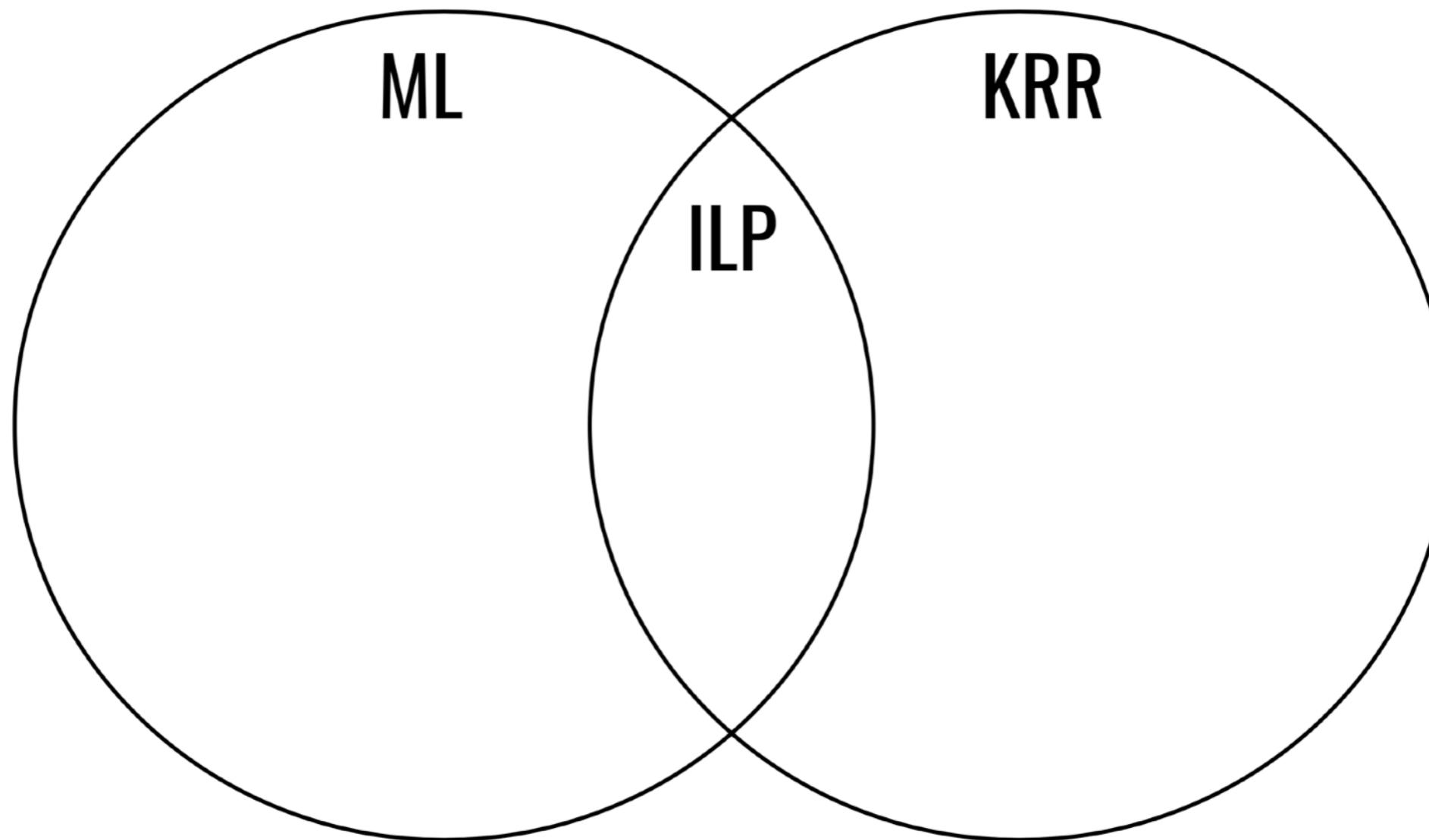
```
∀Mol((has_hydroxyl(Mol)
      ∧ has_amine(Mol)
      ∧ ∃A1,A2(bond(A1,A2,Mol)
                  ∧ atom(Mol,A1,oxygen)
                  ∧ atom(Mol,A2,nitrogen)))
      → active(Mol))
```

Hypothesis

A drug works if it has both a hydroxyl group and an amine group and the oxygen atom of there is a bond between the hydroxyl group and the nitrogen atom of the amine group

Inductive logic programming

Combines machine learning with logical reasoning



Learn from small a number of examples 

Interpretable solutions 

Learn from highly relational data 

How does ILP usually work?



Hypothesis

```
happy(A):- knows(A,B), enjoys_lego(B), lego_builder(B).
```

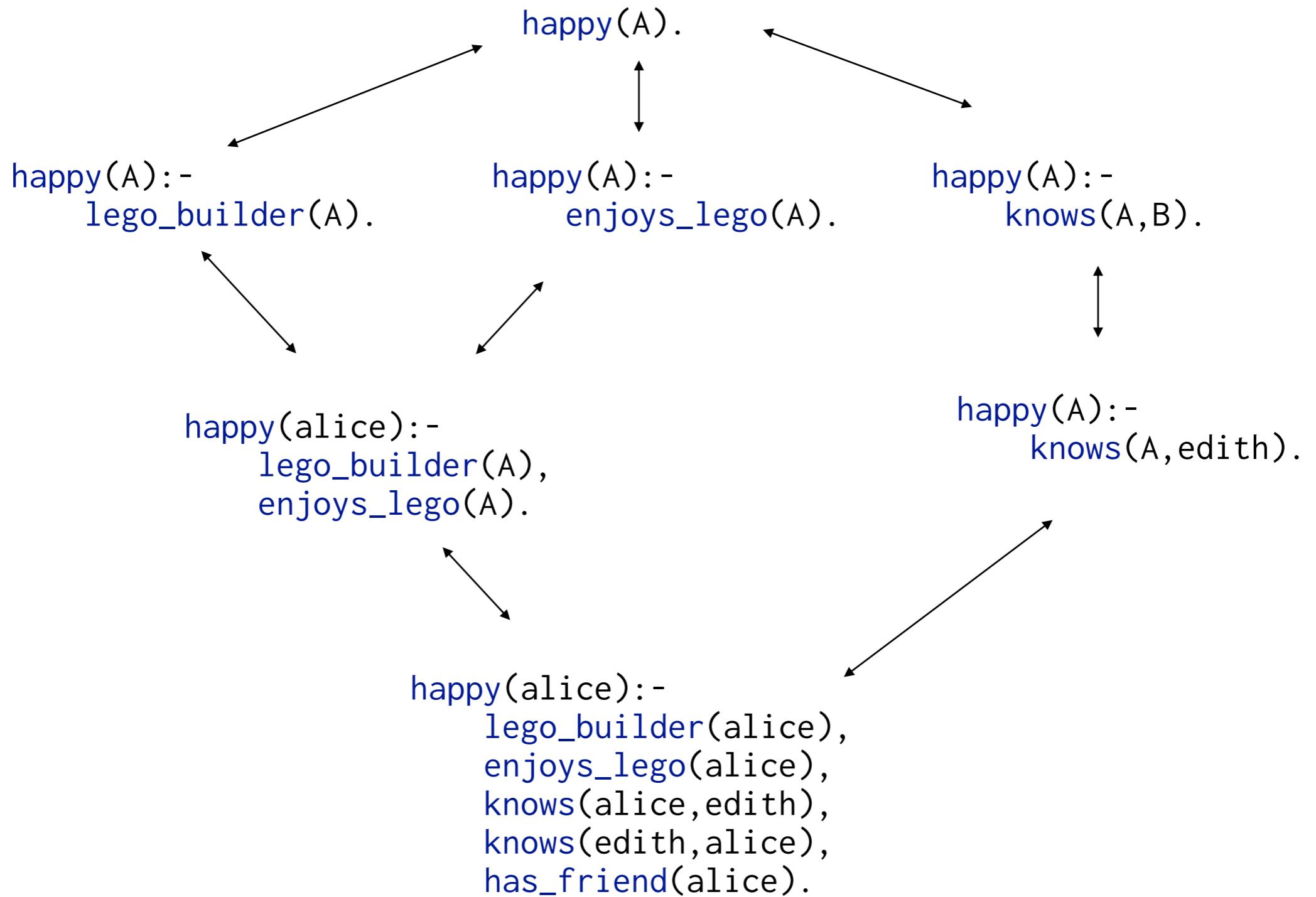
Hypothesis

```
happy(A) :- knows(A,B), enjoys_lego(B), lego_builder(B).
```

Someone (A) is happy if they know someone (B)
who loves lego and is a professional lego builder

Search

Search



Top-down

Start with an overly general hypothesis and iteratively specialise it

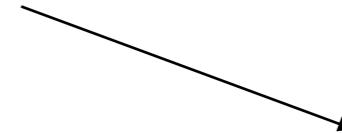
happy(A).



Top-down

Start with an overly general hypothesis and iteratively specialise it

happy(A).



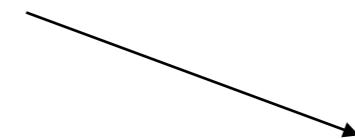
happy(A):-
knows(A,B).



Top-down

Start with an overly general hypothesis and iteratively specialise it

happy(A).



happy(A):-
knows(A,B).

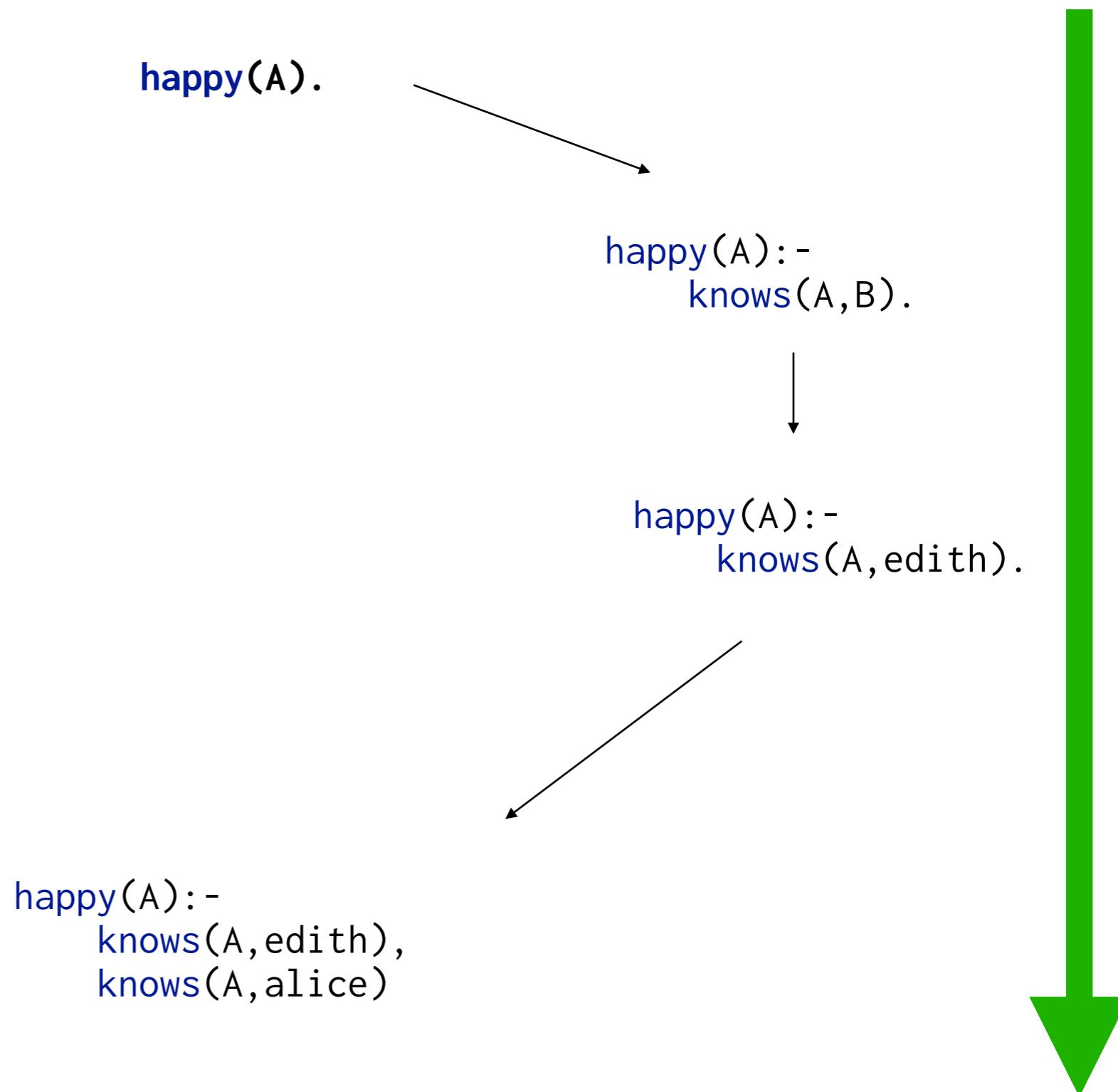


happy(A):-
knows(A,edith).



Top-down

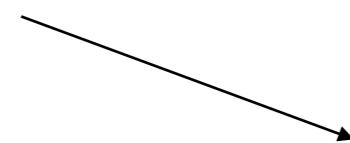
Start with an overly general hypothesis and iteratively specialise it



Top-down

Start with an overly general hypothesis and iteratively specialise it

happy(A).



happy(A):-
knows(A,B).



happy(A):-
knows(A,edith).



happy(A):-
knows(A,edith),
knows(A,peter)



Top-down

Start with an overly general hypothesis and iteratively specialise it

happy(A).

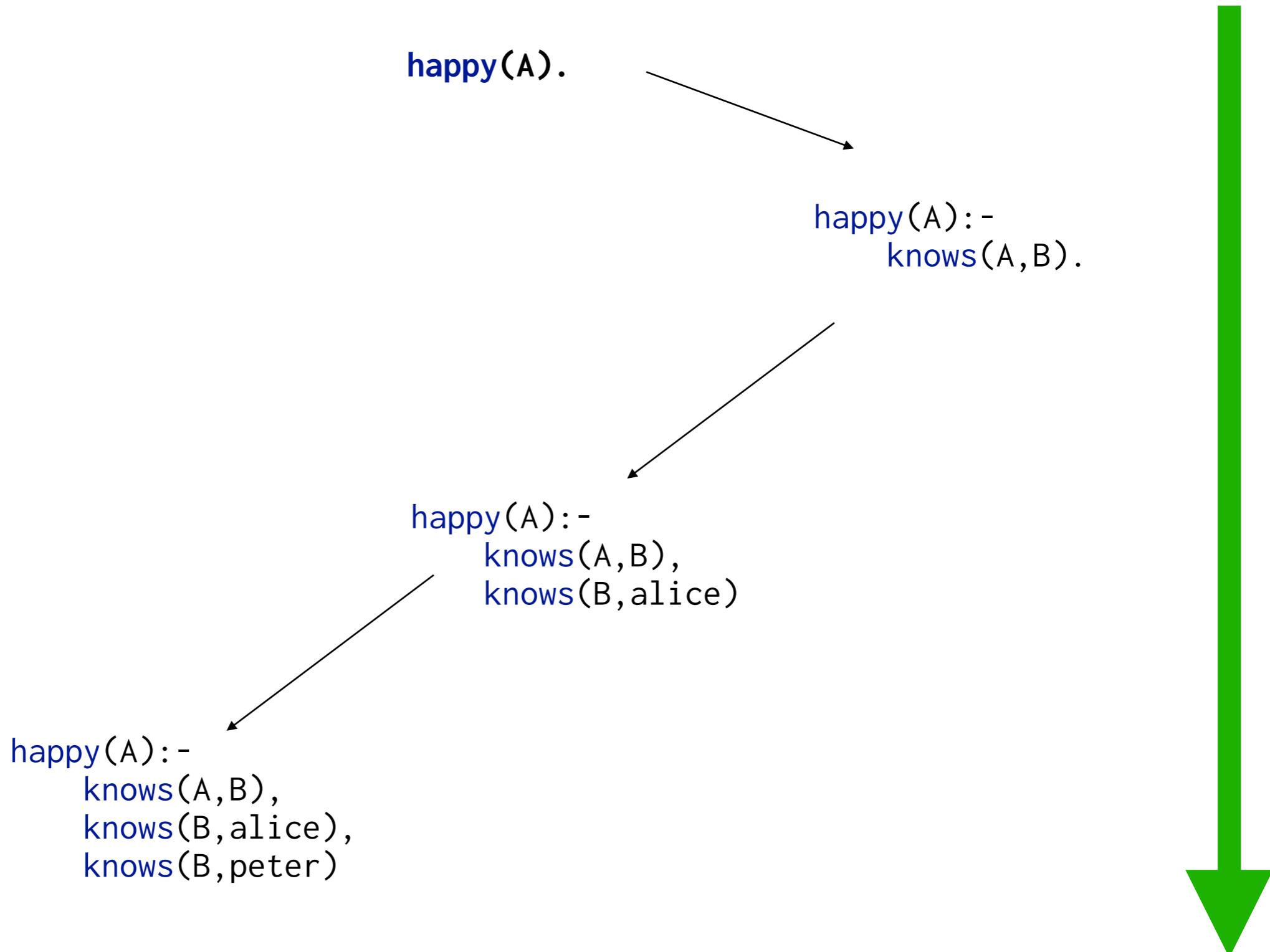
happy(A):-
knows(A,B).

happy(A):-
knows(A,B),
knows(B,alice)



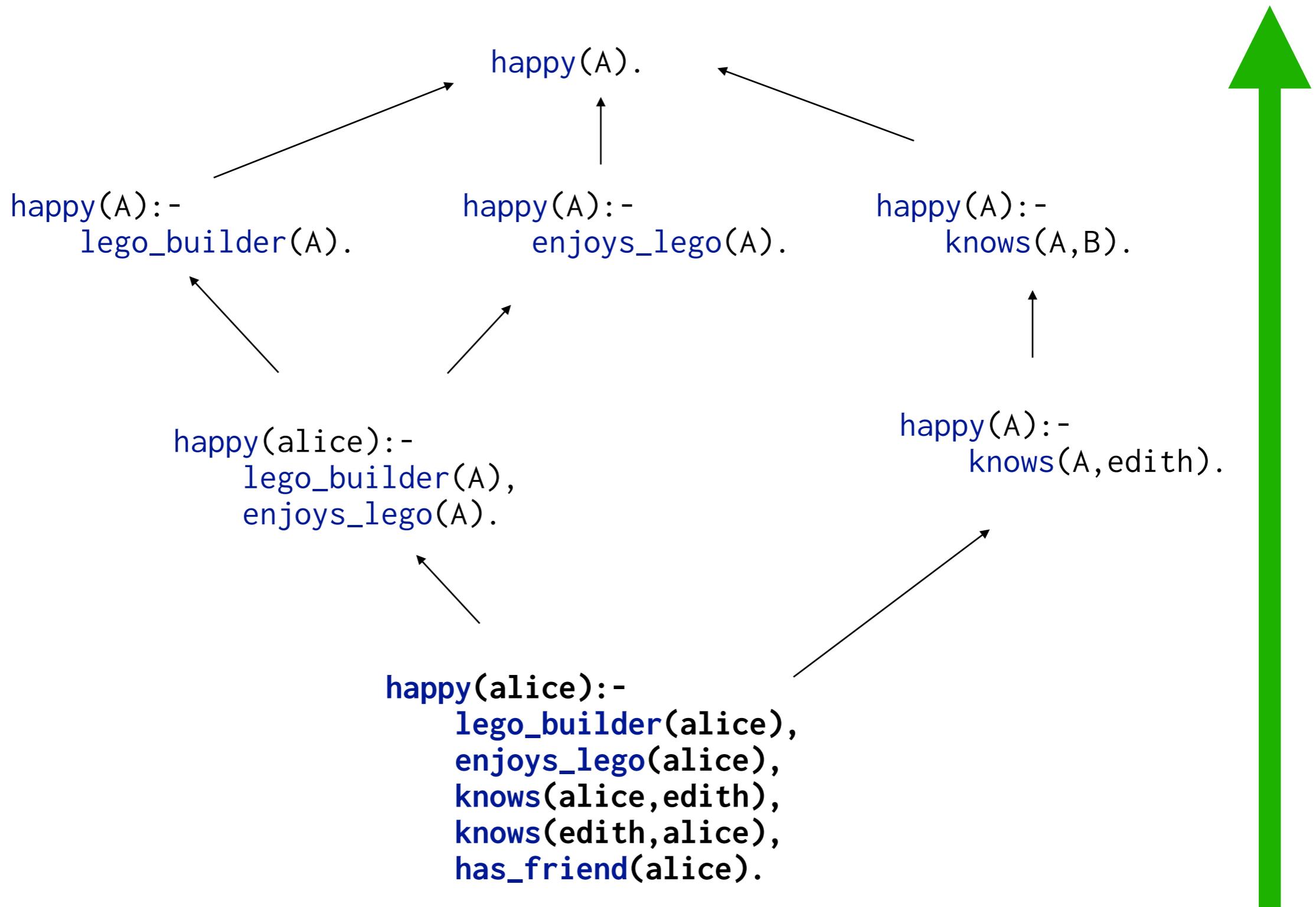
Top-down

Start with an overly general hypothesis and iteratively specialise it

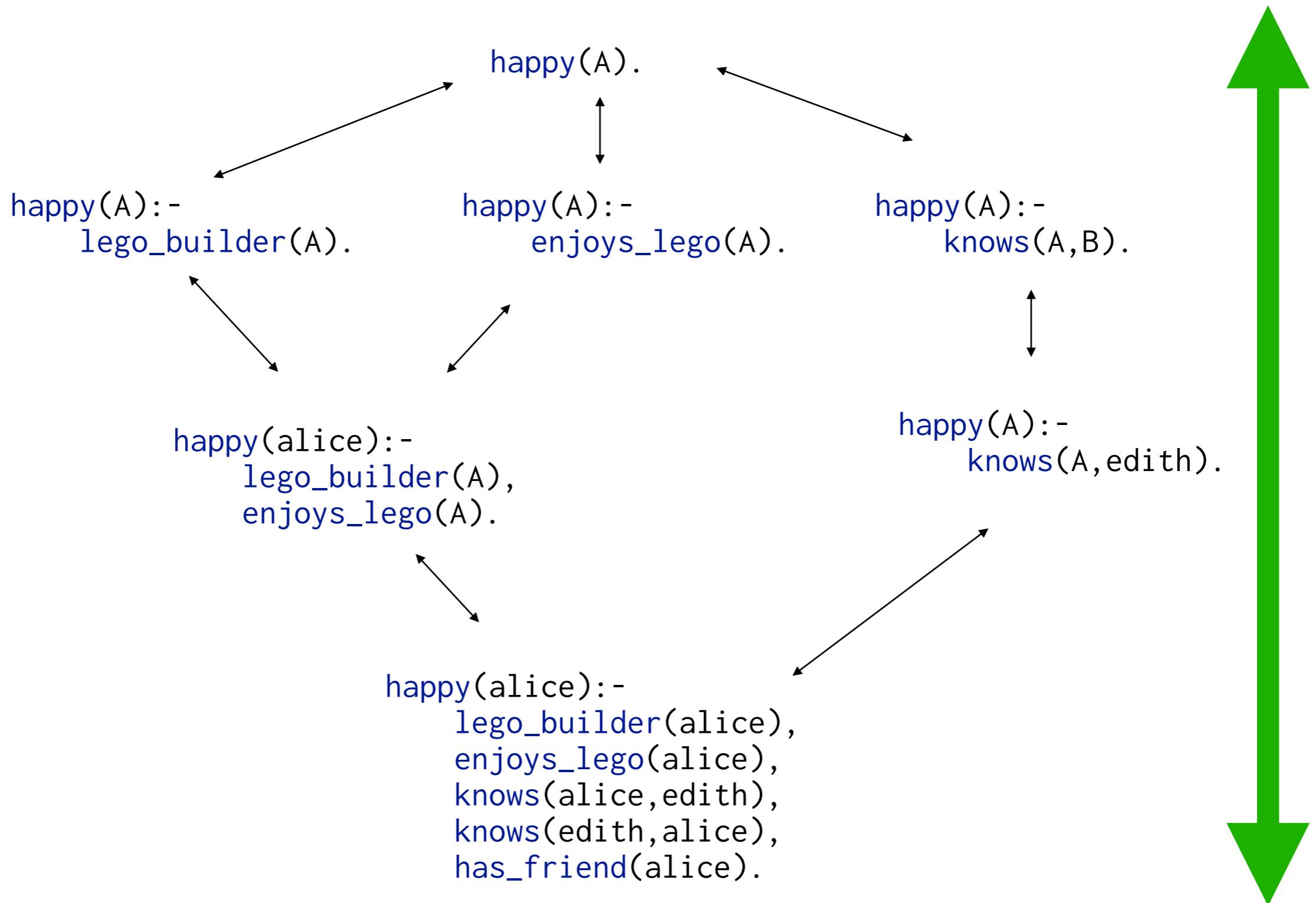


Bottom-up

Start with an overly specific hypothesis and iteratively generalise it



Top-down + bottom-up



Standard approaches

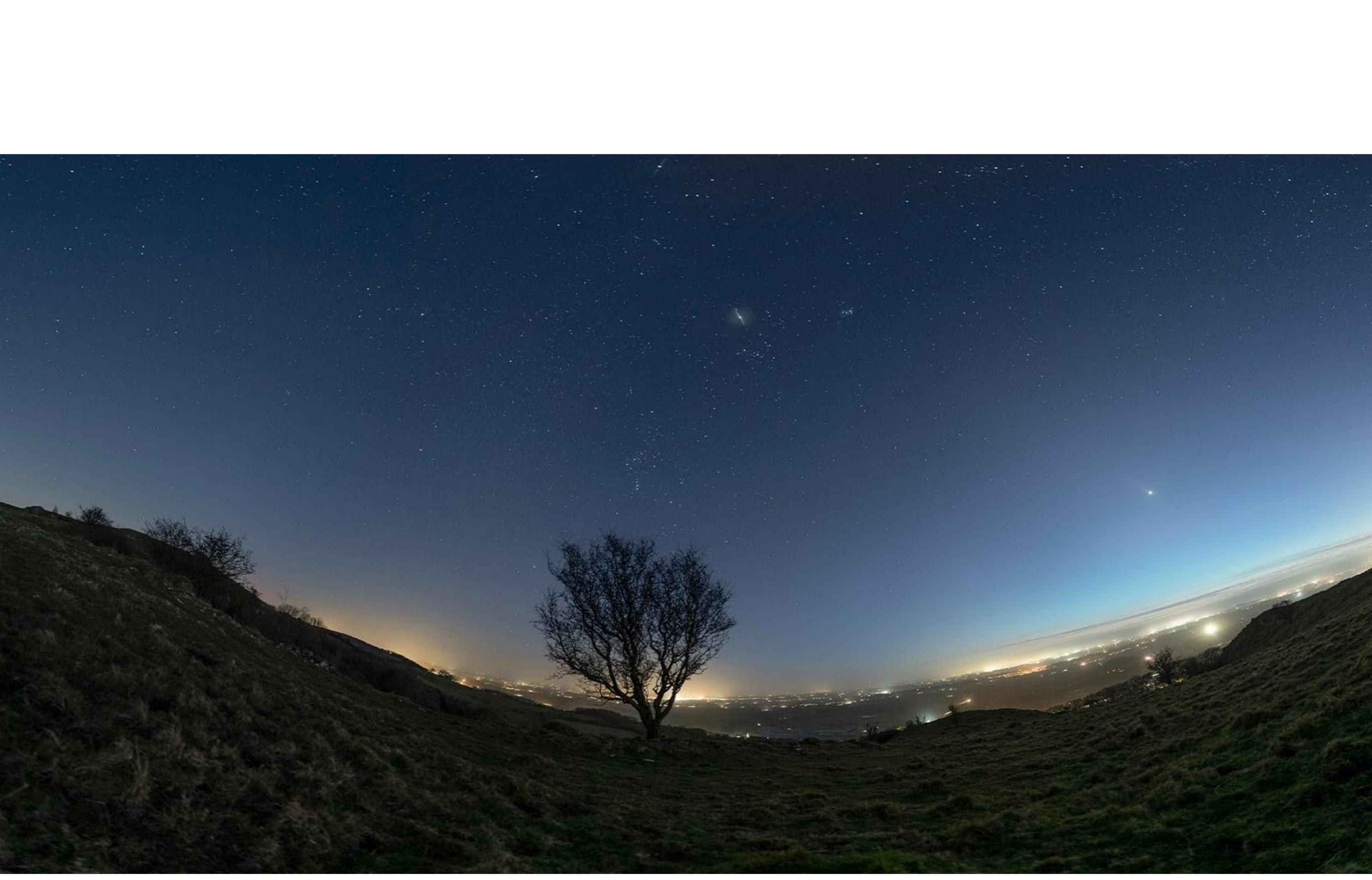
Revise a **hypothesis** to fit the data

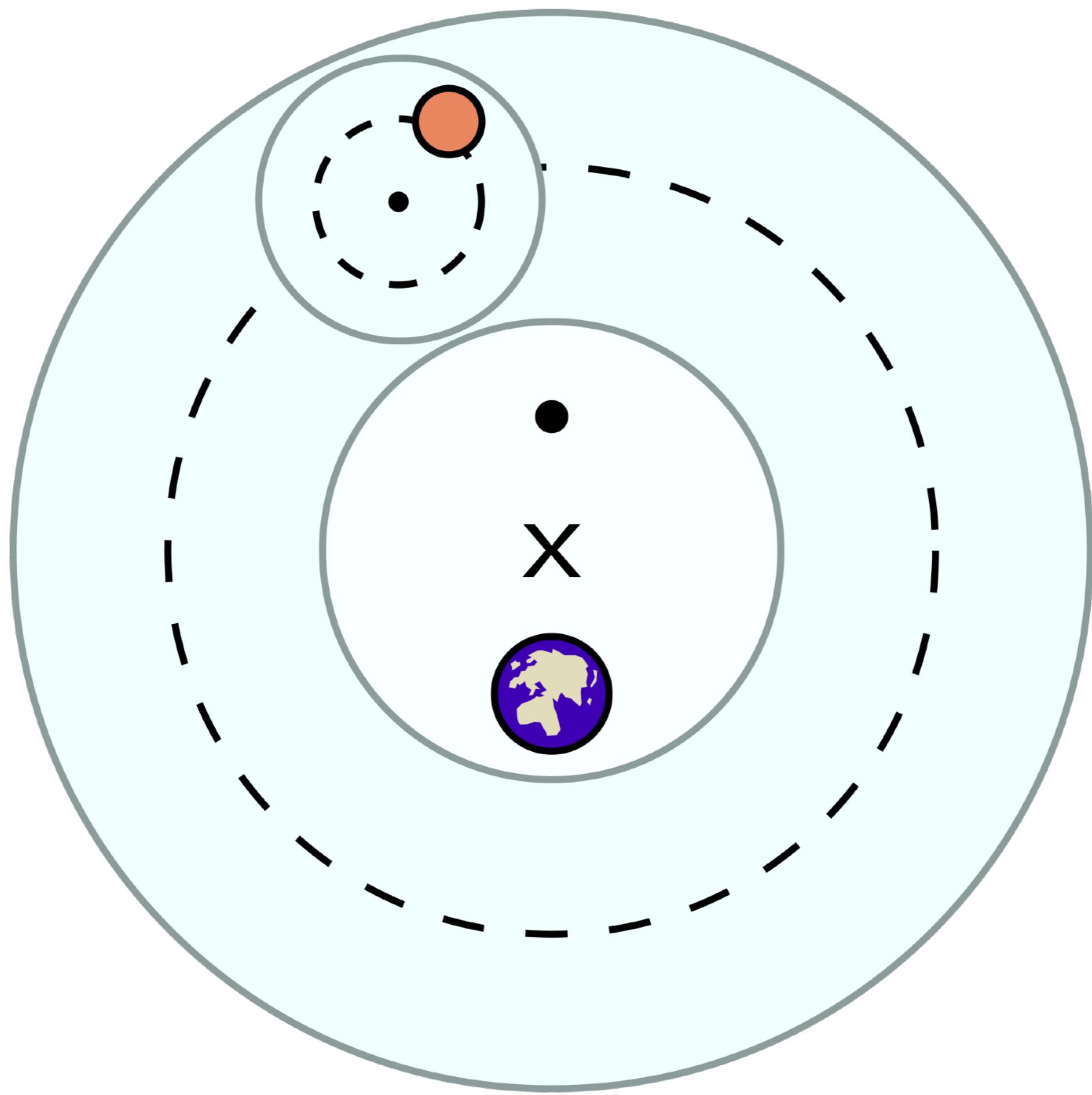
MIS, FOIL, Hyper, TILDE, Progol, Aleph, FORTE, XHAIL, Atom, QuickFOIL,
Metagol

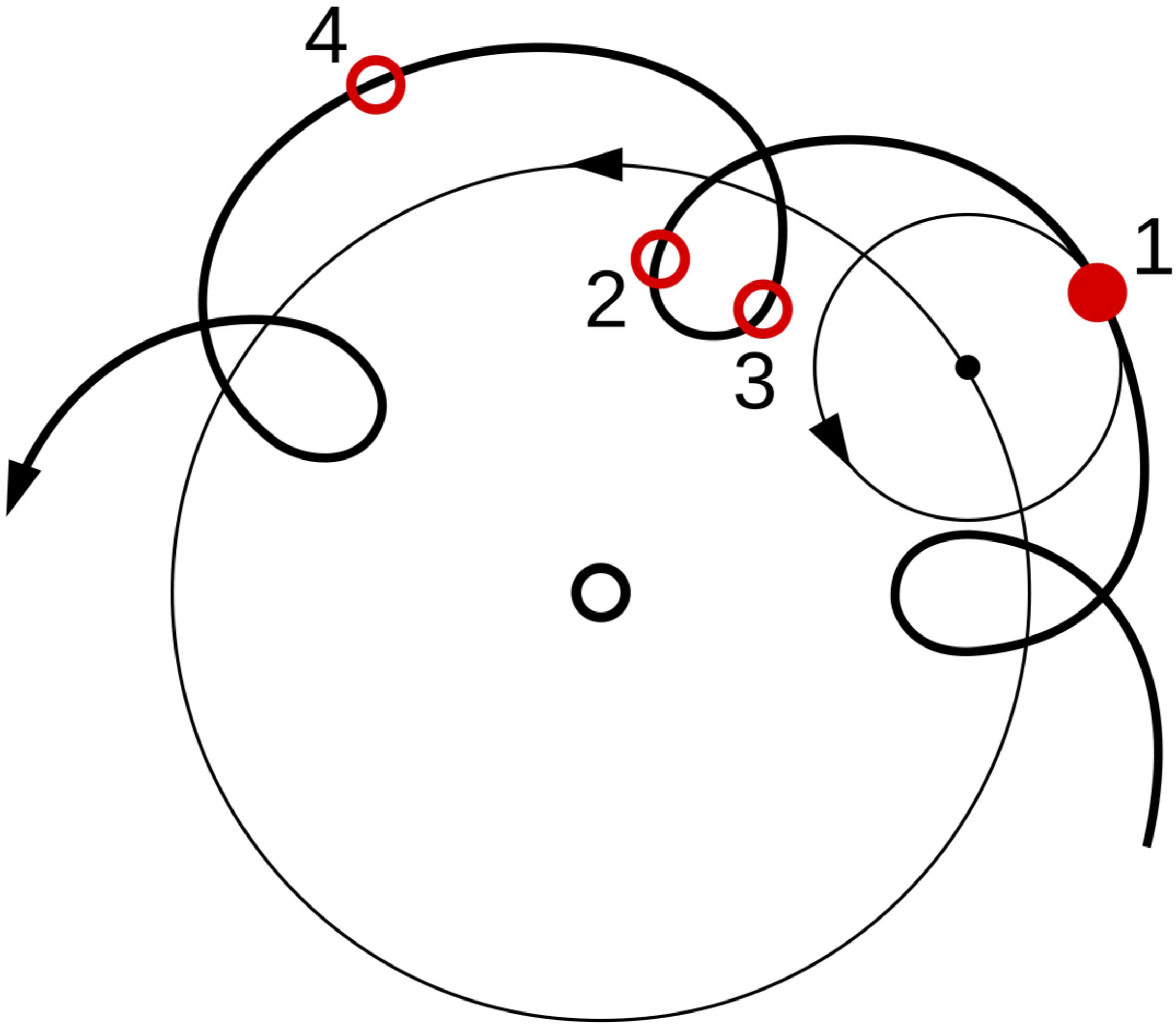
Standard approaches

Search for a hypothesis that fits the data

MIS, FOIL, Hyper, TILDE, Progol, Aleph, FORTE, XHAIL, Atom, QuickFOIL,
Metagol







Our contribution

Automate Karl Popper's
logic of scientific discovery



Science progresses by eliminating false theories

Popper

Popper

0. Start with a hypothesis space

Popper

0. Start with a hypothesis space
1. Select a hypothesis

Popper

0. Start with a hypothesis space
1. Select a hypothesis
2. Empirically try to refute it

Popper

0. Start with a hypothesis space
1. Select a hypothesis
2. Empirically try to refute it
3. If the hypothesis is falsified, determine **why**

Popper

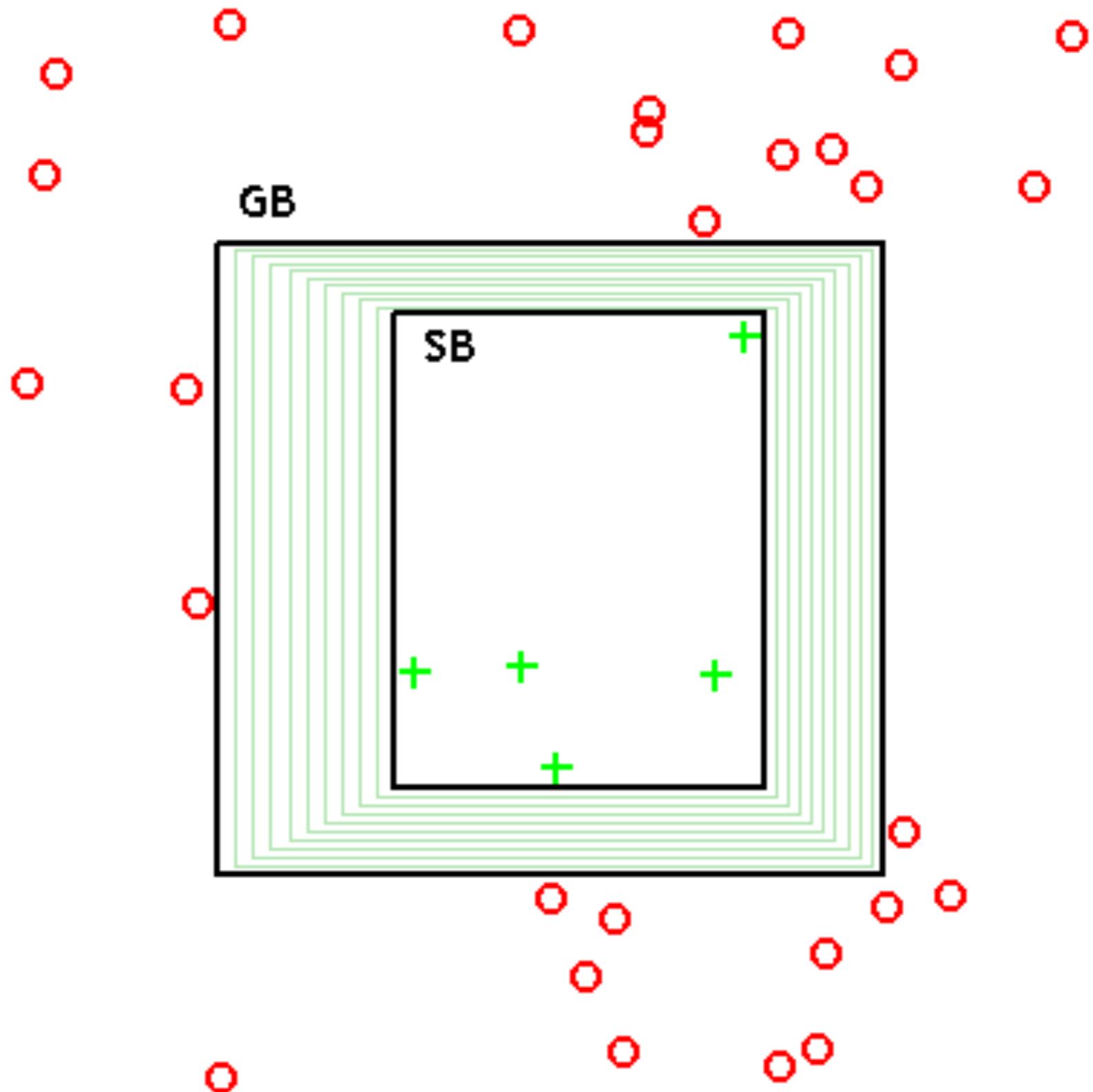
0. Start with a hypothesis space
1. Select a hypothesis
2. Empirically try to refute it
3. If the hypothesis is falsified, determine **why**
4. Use the **explanation** to prune the hypothesis space

Popper

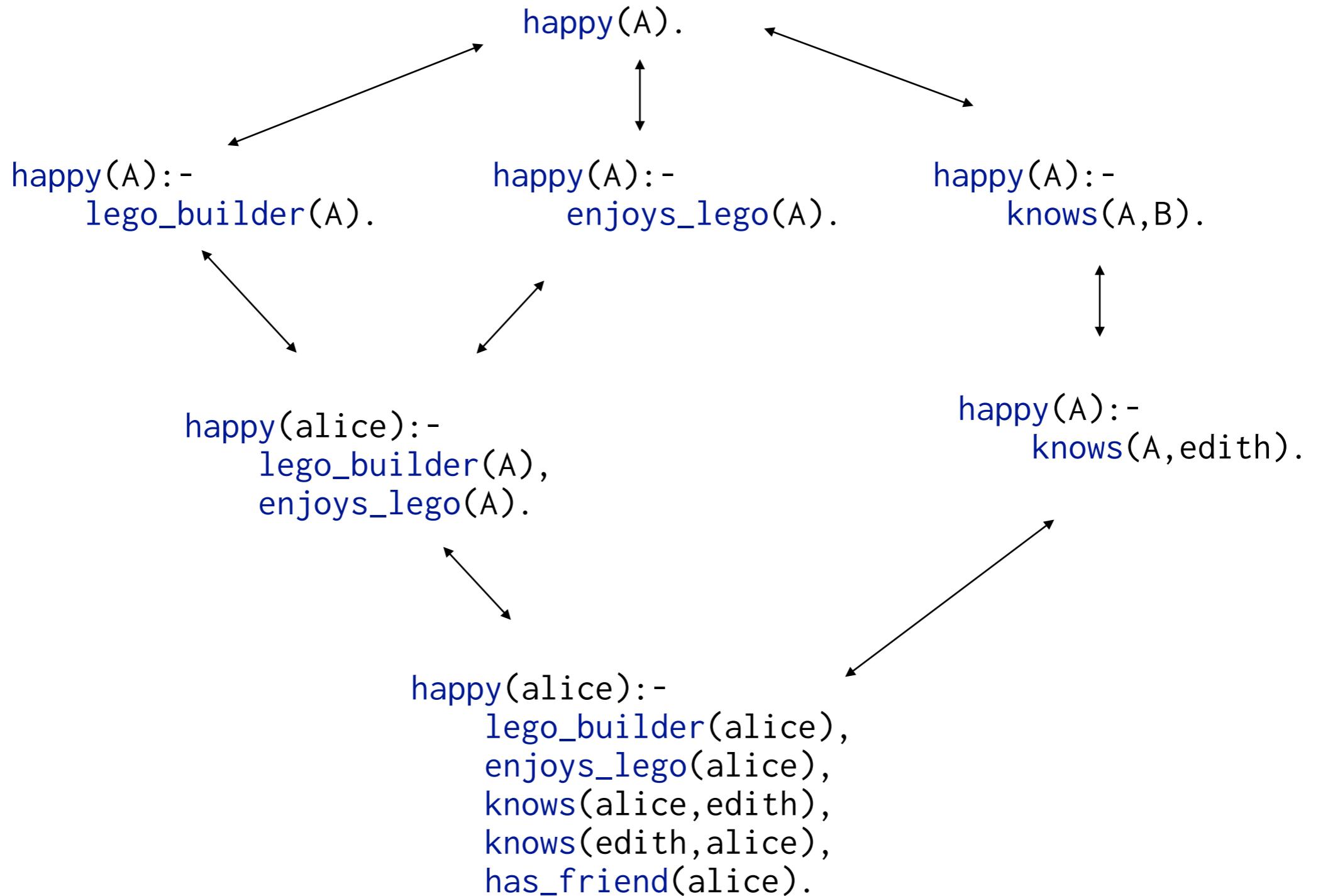
0. Start with a hypothesis space
1. Select a hypothesis
2. Empirically try to refute it
3. If the hypothesis is falsified, determine **why**
4. Use the **explanation** to prune the hypothesis space
5. Go to 1

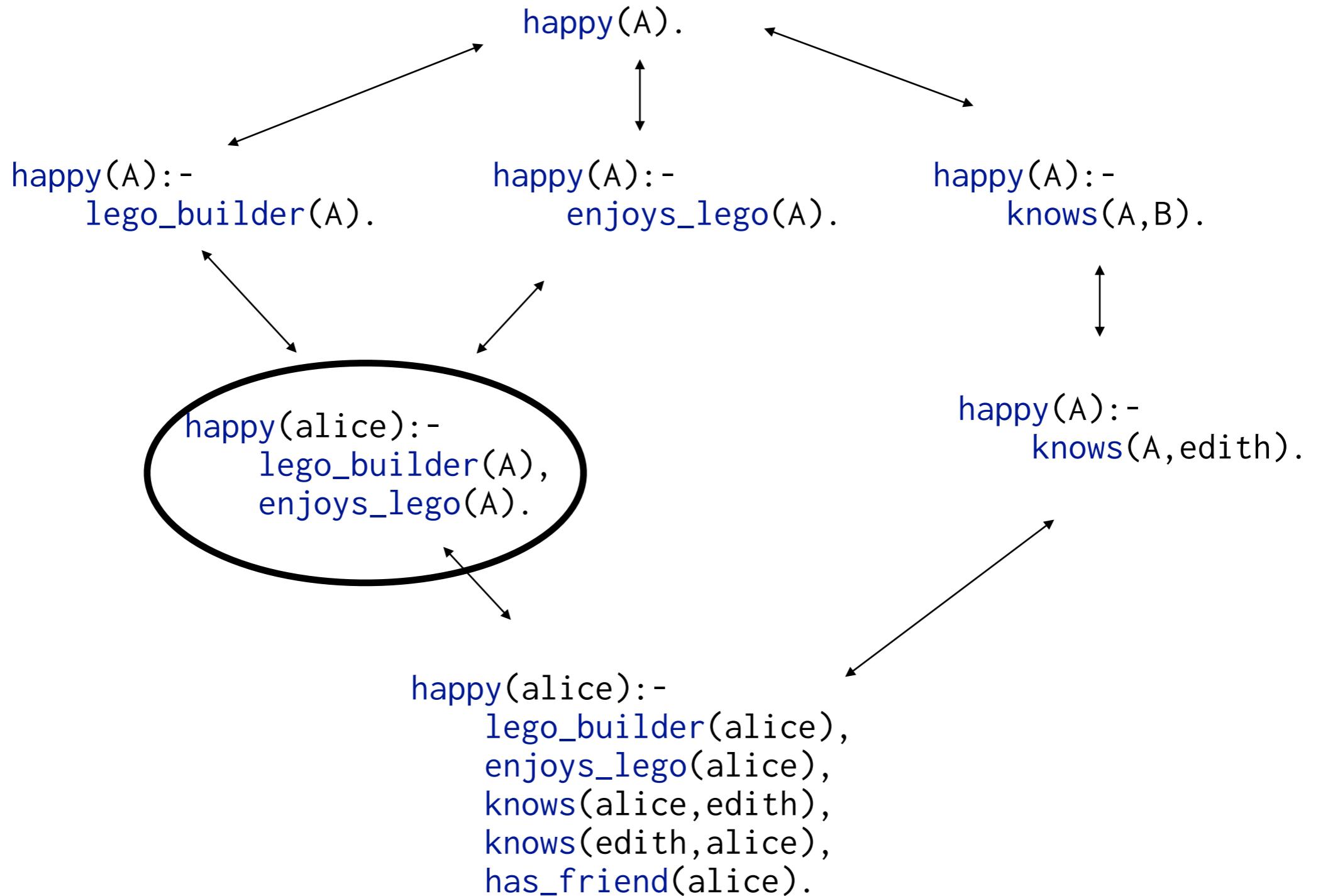
Popper

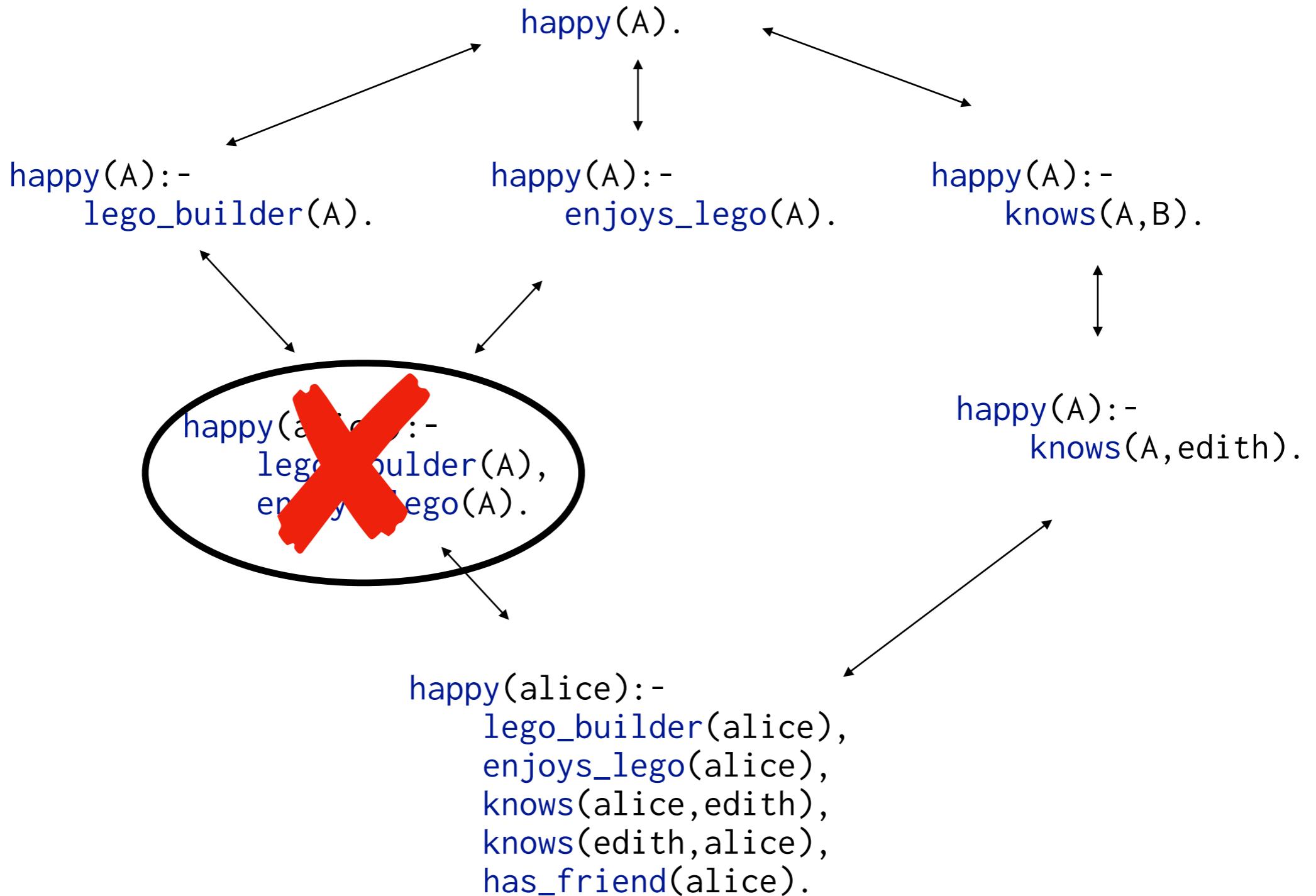
Search to eliminate hypotheses

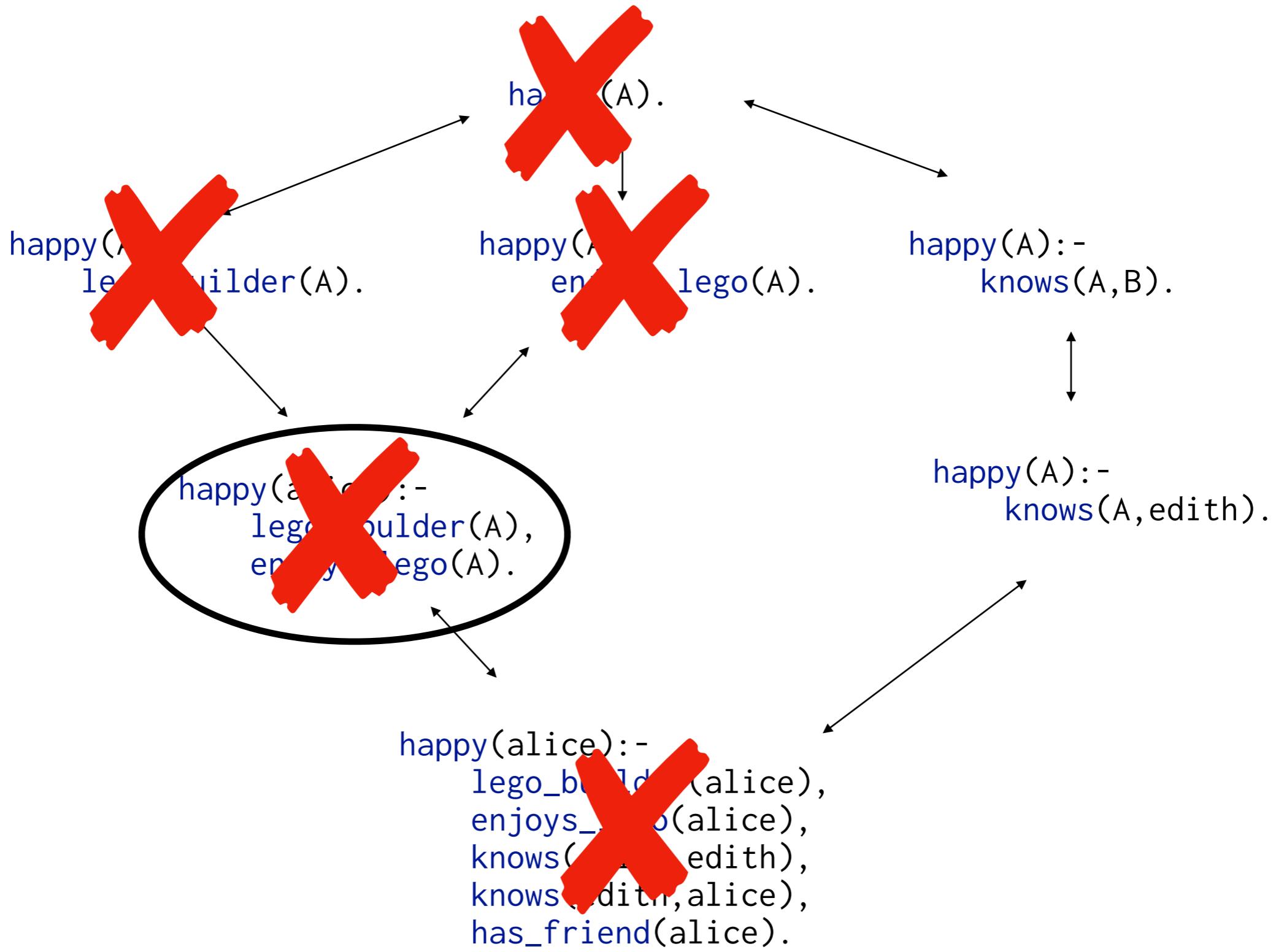


Tom Mitchell's version space learning and
candidate elimination

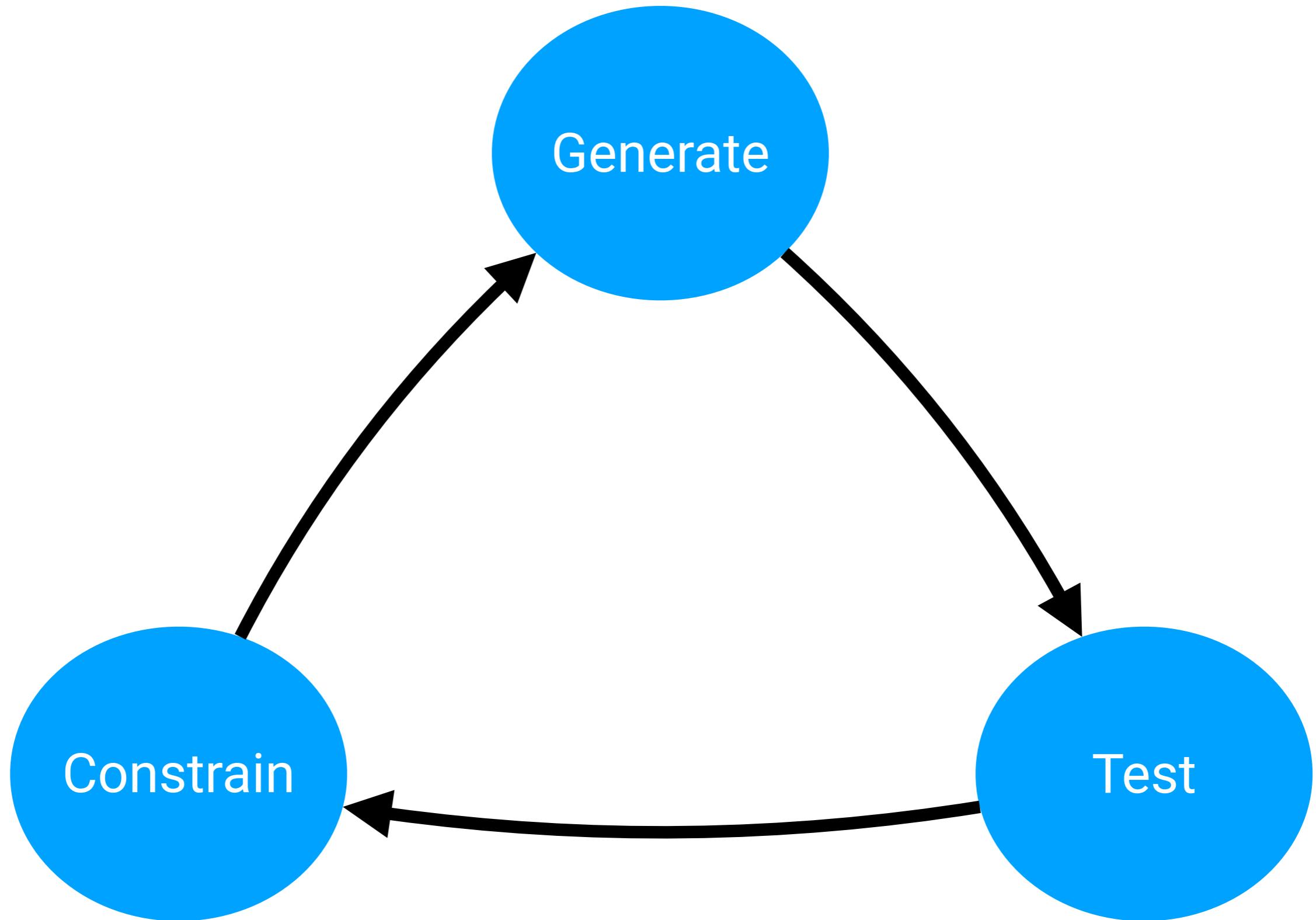




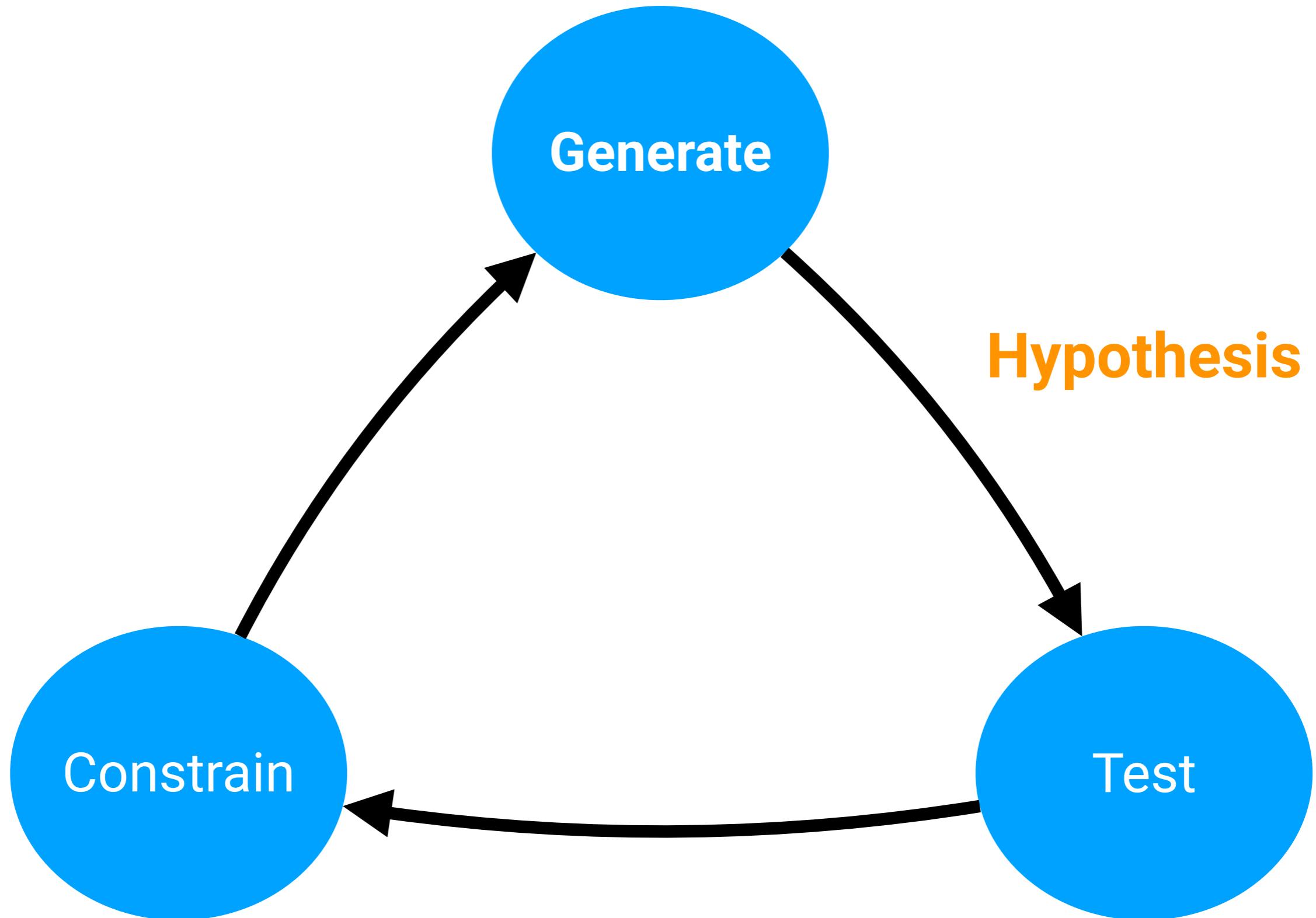




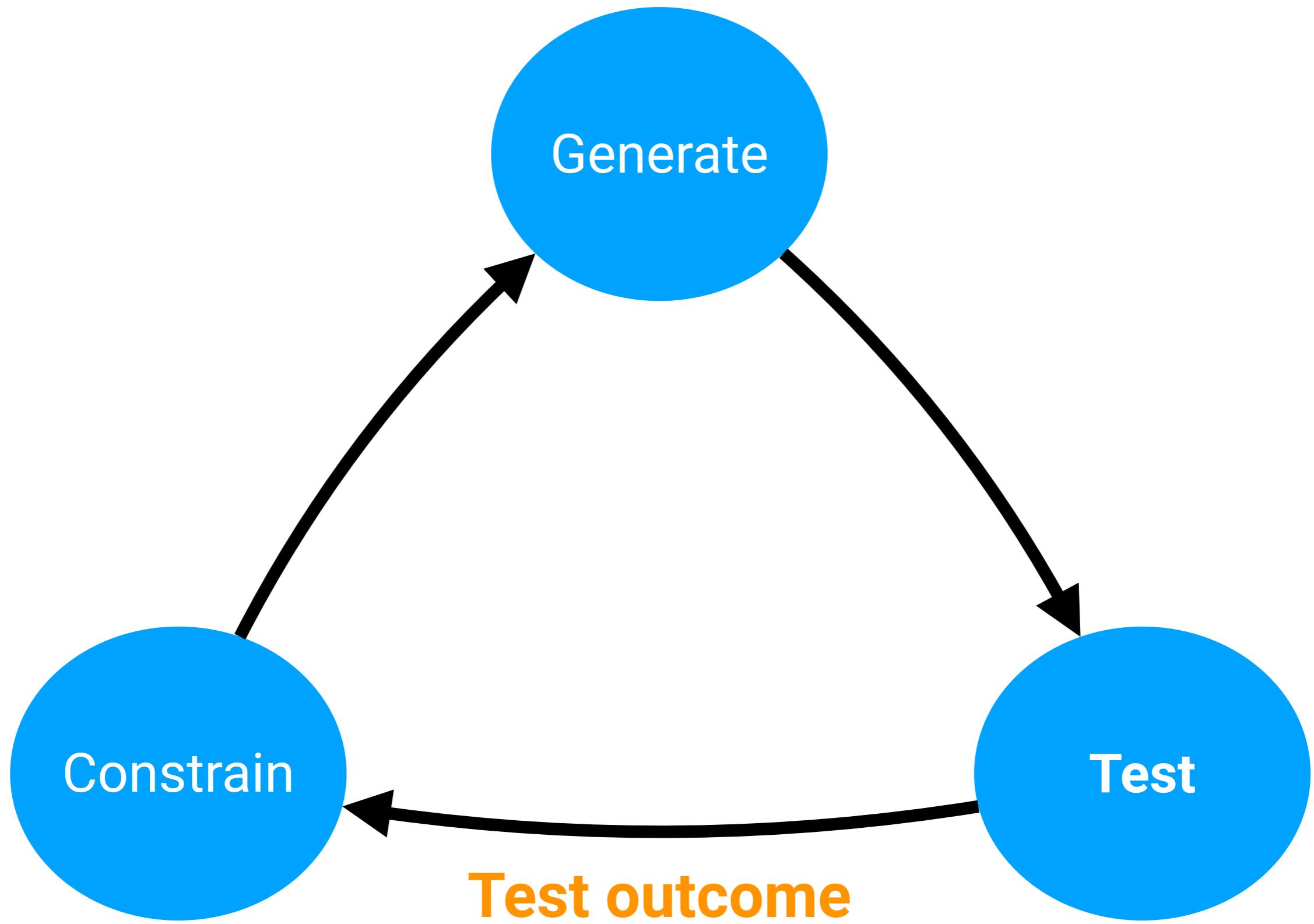
Popper



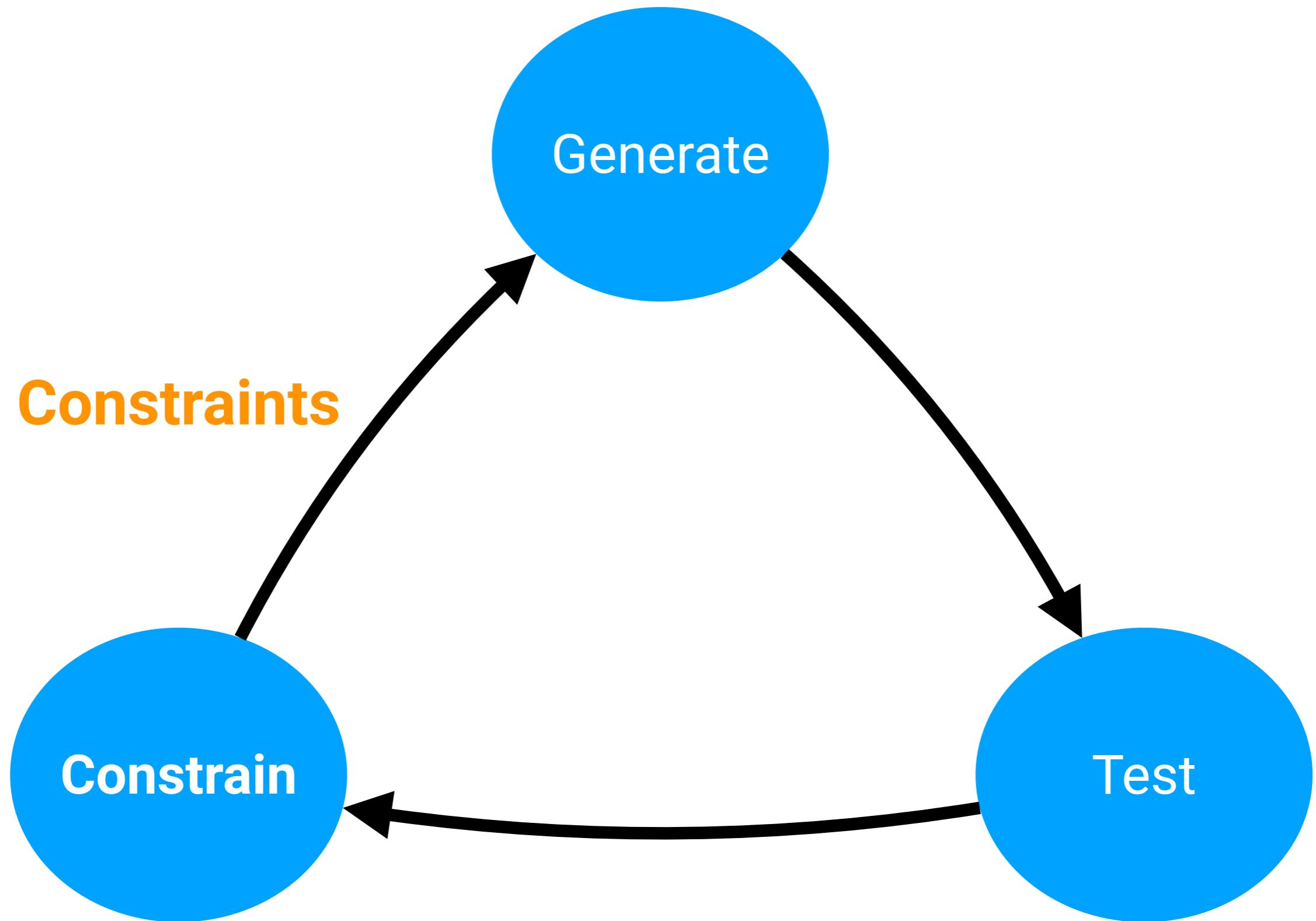
Popper



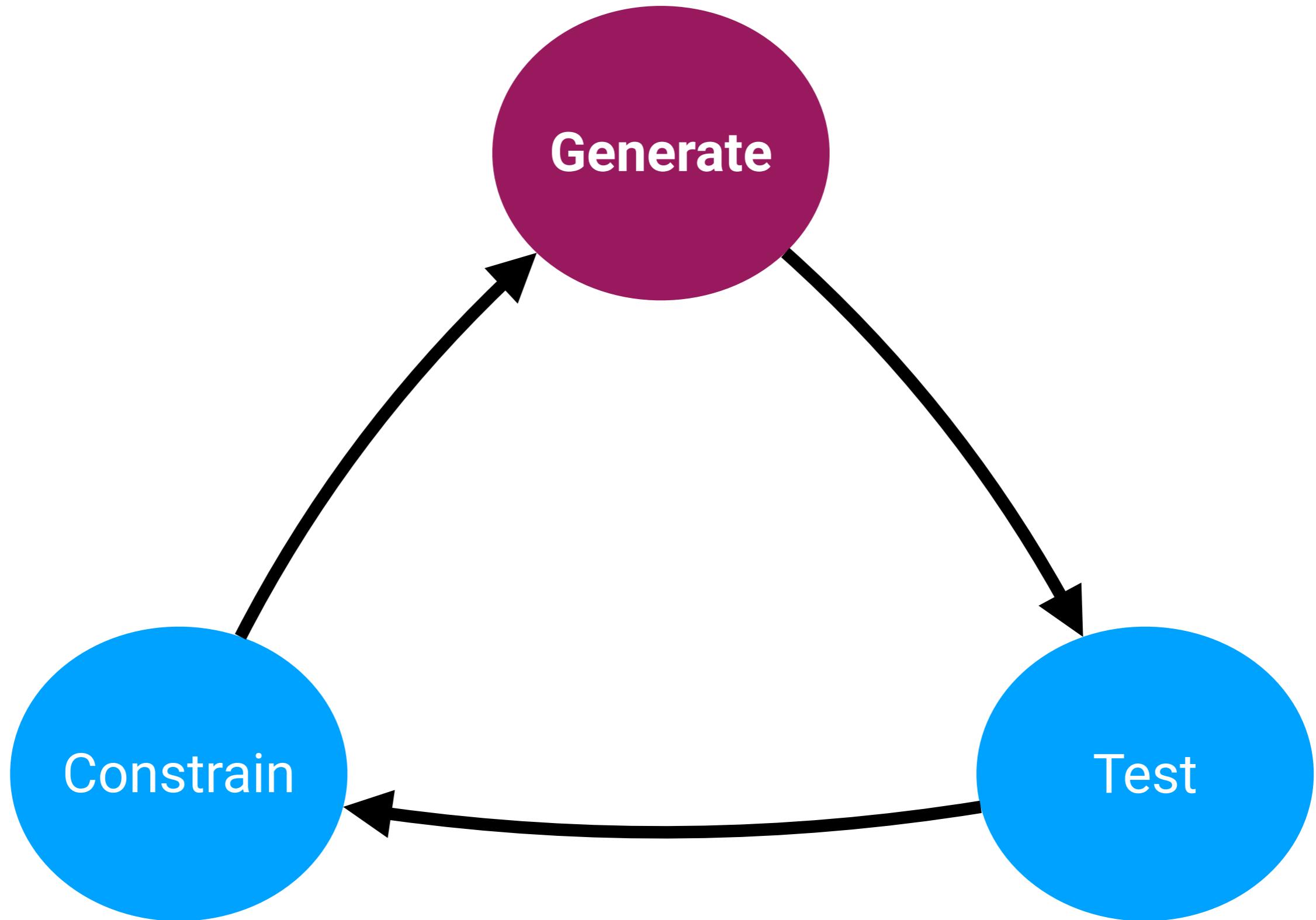
Popper



Popper



Popper

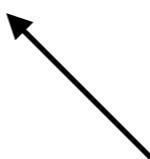


Generate

Generate

Given:

- a language bias



What is a valid rule/hypothesis

Generate

Given:

- a language bias
- constraints (initially empty)

Generate

Given:

- a language bias
- constraints (initially empty)

Find:

- a syntactically valid hypothesis that does not violate the constraints

Generate

Given:

- a language bias
- constraints (initially empty)

No examples!



Find:

- a syntactically valid hypothesis that does not violate the constraints

Generate

Given:

- a language bias
- constraints (initially empty)

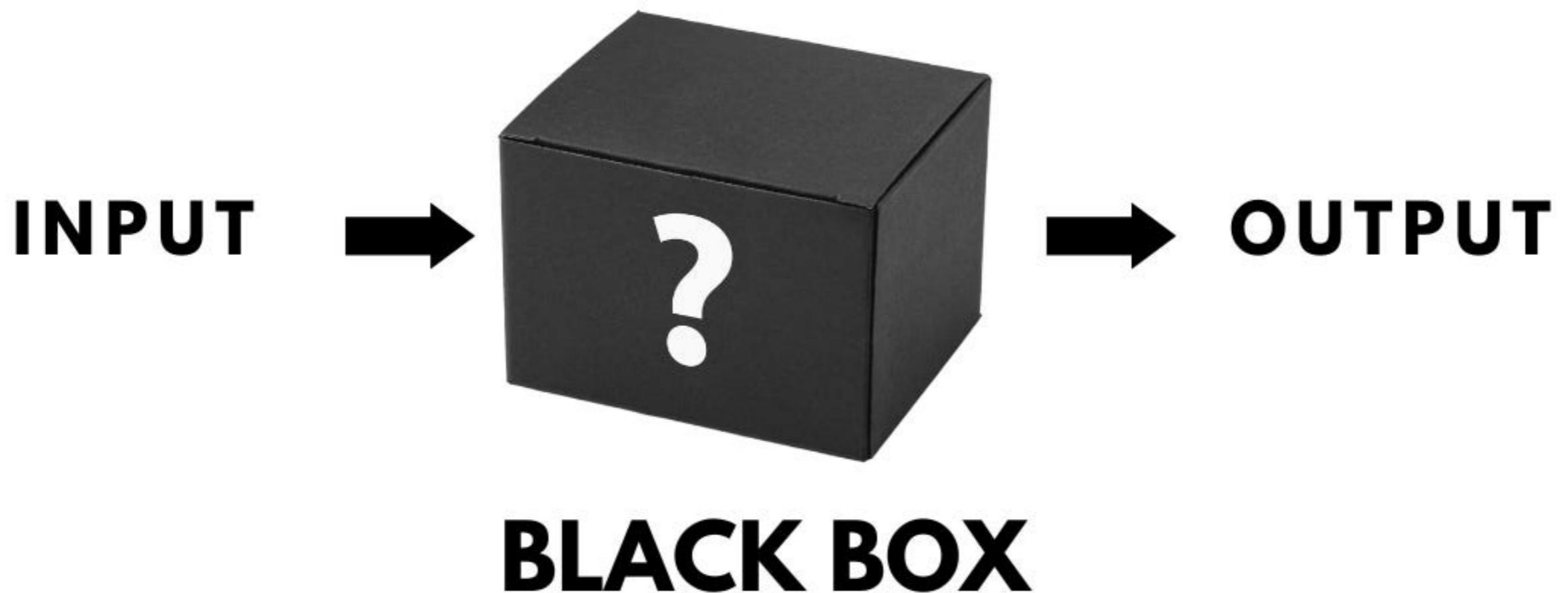
No BK!



Find:

- a syntactically valid hypothesis that does not violate the constraints

Generate



Satisfiability problem

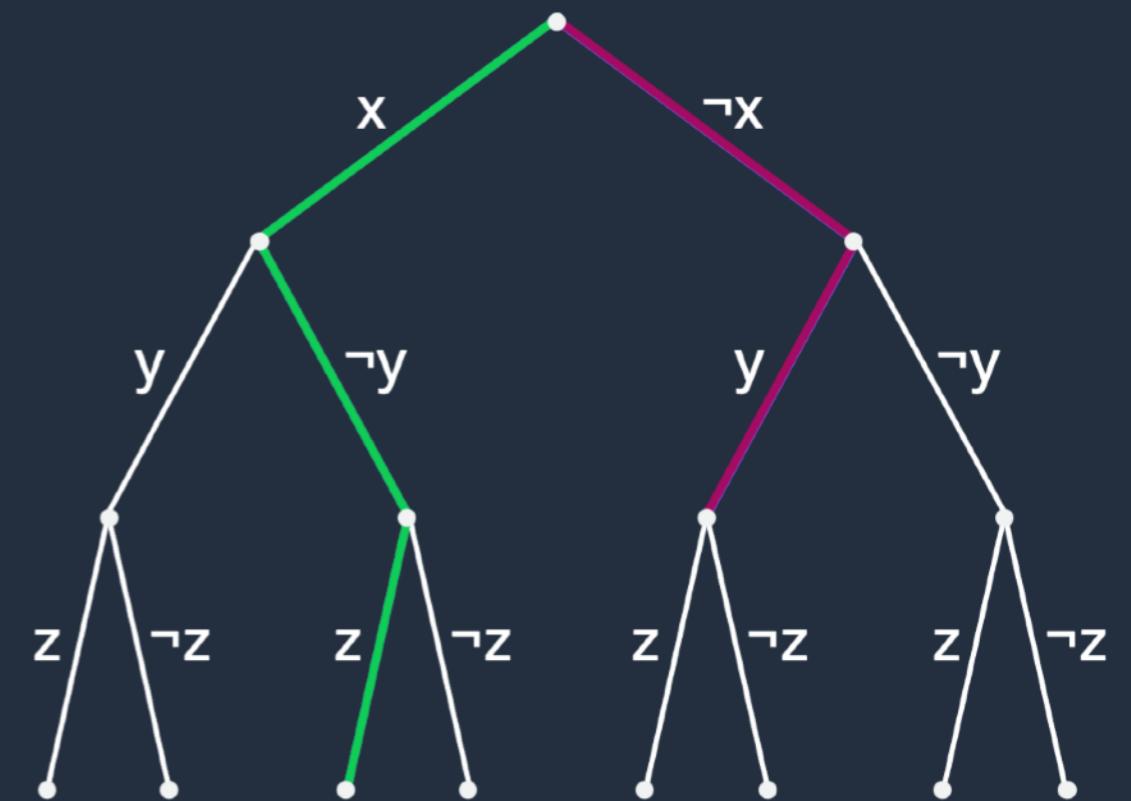
We build a formulae that represents the hypothesis space

Satisfiability problem

We ask a SAT solver to find a model of the formulae

SAT Solving

$(x \vee y) \wedge (x \vee \neg y) \wedge (y \vee z) \wedge (\neg x)$



Solver



UNSAT

+

Proof

Satisfiability problem

Every model is a hypothesis

Satisfiability problem

$p(A)$

$p(B)$

$q(A)$

$q(B)$

$r(A)$

$r(B)$

$\text{odd}(A)$

$\text{odd}(B)$

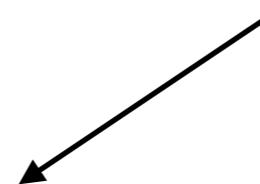
$\text{even}(A)$

$\text{even}(B)$

$\text{succ}(A, B)$

$\text{succ}(B, A)$

All possible literals



Satisfiability problem

p_a
p_b

q_a
q_b

r_a
r_b

odd_a

odd_b

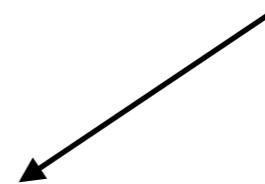
even_a

even_b

succ_ab

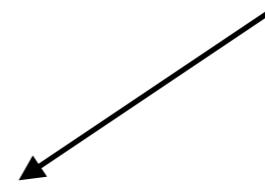
succ_ba

Propositional variables



Satisfiability problem

A single clause


$$F1 = (p_a \vee p_b \vee q_a \vee q_b \vee r_a \vee r_b \vee \\ \text{odd_}a \vee \text{odd_}b \vee \text{even_}a \vee \text{even_}b \vee \text{succ_}ab \vee \\ \text{succ_}ba)$$

Satisfiability problem

$F_1 = (p_a \vee p_b \vee q_a \vee q_b \vee r_a \vee r_b \vee$
 $\text{odd}_a \vee \text{odd}_b \vee \text{even}_a \vee \text{even}_b \vee \text{succ}_{ab} \vee$
 $\text{succ}_{ba})$

$$\begin{aligned} p_a &= T \\ r_b &= T \end{aligned}$$

Satisfiability problem

$F_1 = (p_a \vee p_b \vee q_a \vee q_b \vee r_a \vee r_b \vee$
 $\text{odd}_a \vee \text{odd}_b \vee \text{even}_a \vee \text{even}_b \vee \text{succ}_{ab} \vee$
 $\text{succ}_{ba})$

$$\begin{aligned} p_a &= T \\ r_b &= T \end{aligned}$$

$f(A) :- p(A), r(B)$

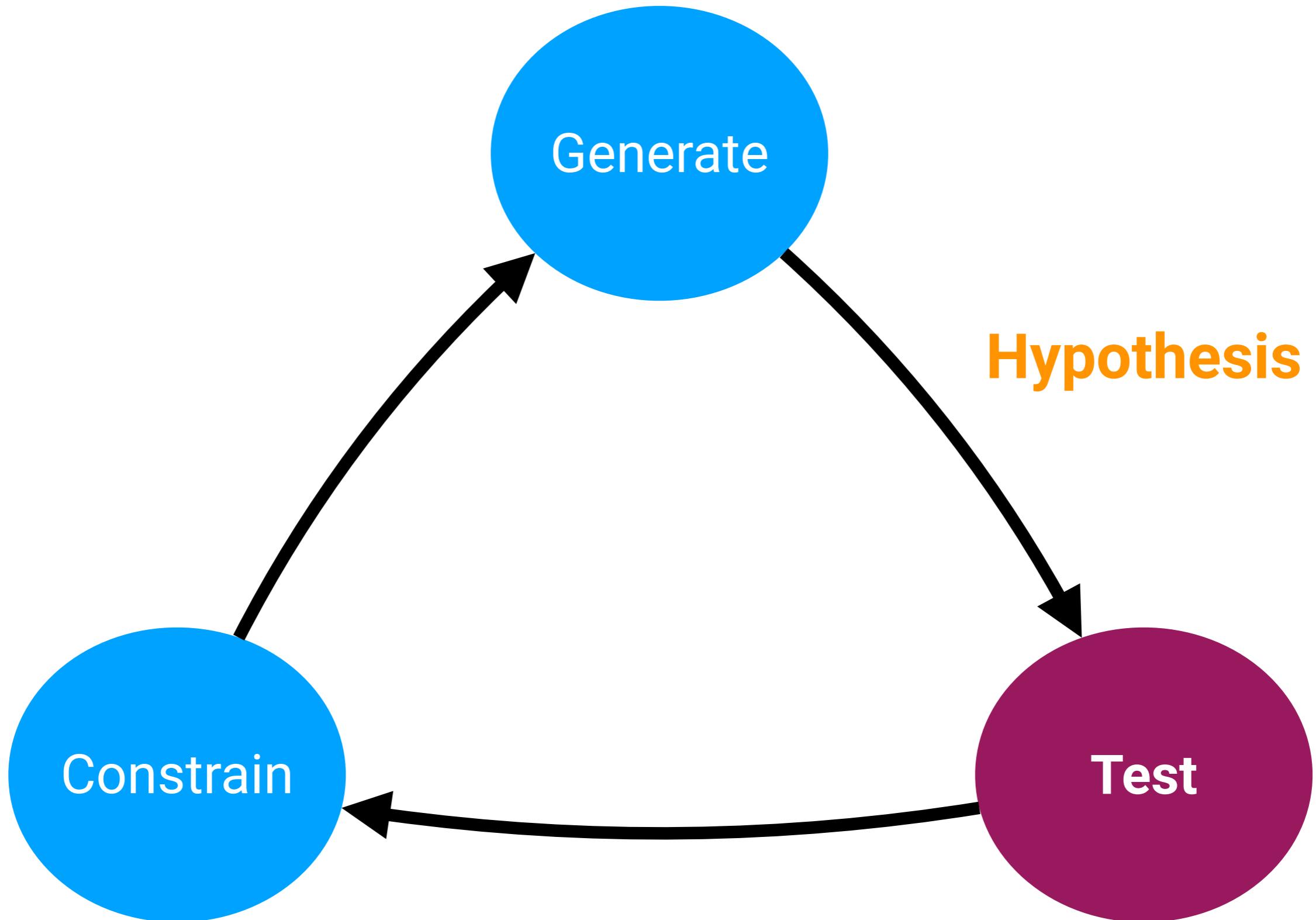
Why SAT?

Easy

Why SAT?

Fast

Popper



Test



SWI Prolog

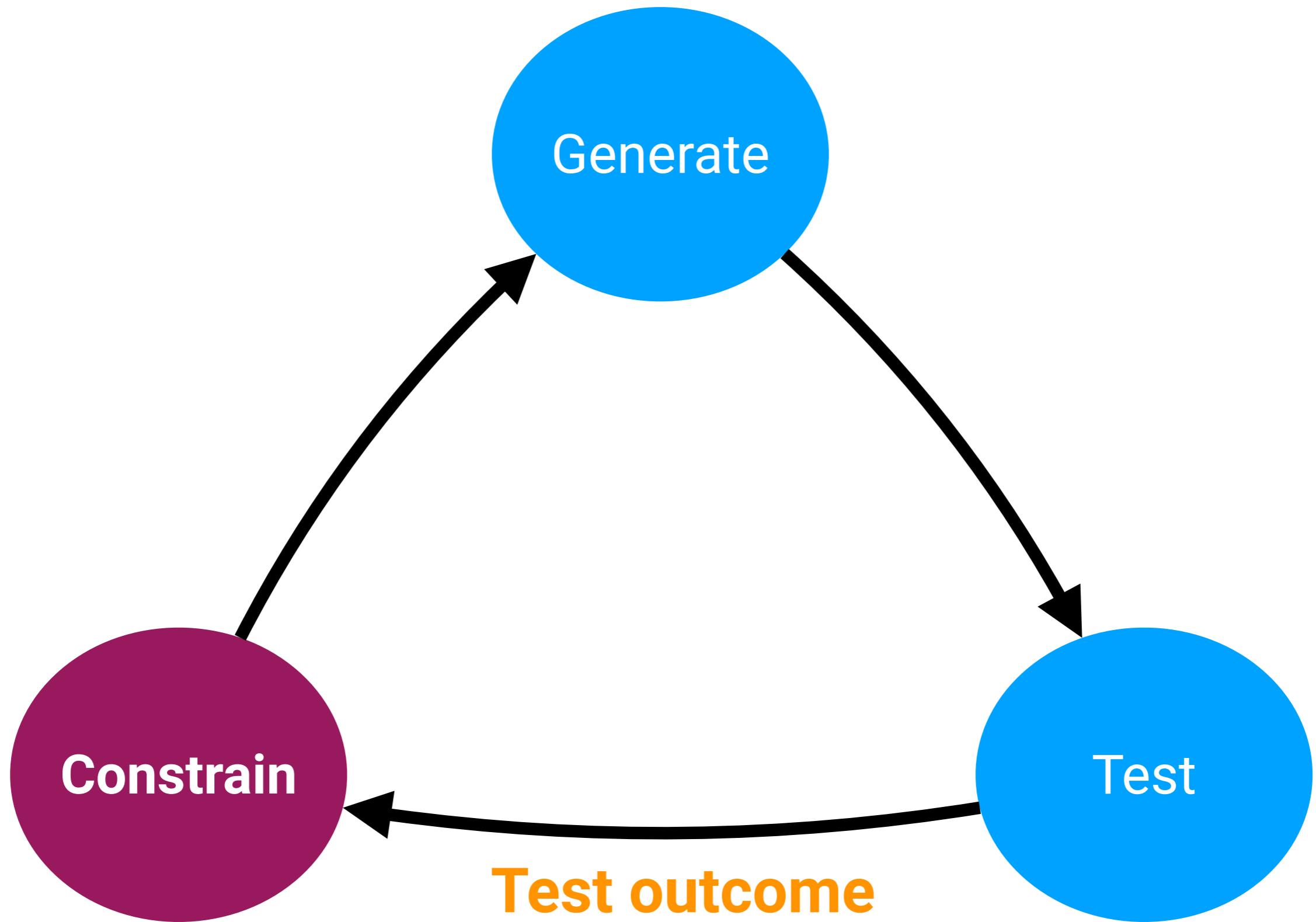
Test



SWI Prolog

- Fast
- Infinite domains
 - Continuous numbers
 - Lists

Popper



Constrain

Build constraints to prune hypotheses

Noiseless scenario

*for the next few slides, assume no noise

Specialisations

If:

a hypothesis does not entail any positive examples

Specialisations

If:

a hypothesis does not entail any positive examples

Then:

it is too specific

Specialisations

If:

a hypothesis does not entail any positive examples

Then:

it is too specific

Therefore:

prune all hypotheses that syntactically specialise it

Specialisations

If this rule is too specific:

```
f(A) :- p(A).
```

Specialisations

If this rule is too specific:

$f(A) :- p(A).$

Then we prune all supersets of it:

$f(A) :- p(A), q(A), \dots$

$f(A) :- p(A), r(A), \dots$

$f(A) :- p(A), s(A), \dots$

Specialisations

If this rule is too specific:

```
f(A) :- p(A).
```

Then we prune all supersets of it:

```
not p_a.
```

Specialisations

If this rule is too specific:

$f(A) :- p(A).$

Then we prune all supersets of it:

$$F2 = F1 \wedge (\text{not } p_a)$$



Original formula

Specialisations

If this rule is too specific:

$f(A) :- p(A).$

Then we prune all supersets of it:

$$F2 = F1 \wedge (\text{not } p_a)$$



New clause

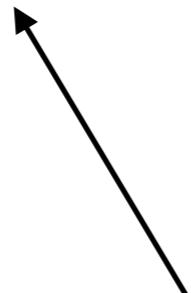
Specialisations

If this rule is too specific:

$f(A) :- p(A).$

Then we prune all supersets of it:

$$F2 = F1 \wedge (\text{not } p_a)$$



New formula

Generalisations

If a hypothesis entails a negative example then it is too general

Generalisations

If a hypothesis entails a negative example then it is too general

Therefore, we prune all hypotheses that syntactically generalise it

Generalisations

If this rule is too general:

```
f(A):- p(A).
```

Generalisations

If this rule is too general:

$f(A) :- p(A).$

Then we prune all generalisations of it:

{ $f(A) :- p(A).$
 $f(A) :- q(A).$

Explanations

We analyse hypotheses to explain them

UNSAT explanations

If we see this rule:

```
f(A):- p(A), ... , odd(A), even(A), ...
```

UNSAT explanations

If we see this rule:

f(A) :- p(A), ..., odd(A), even(A), ...

Then we prune all hypotheses that include:

odd(A), even(A)

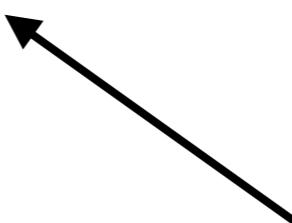
UNSAT explanations

If we see this rule:

$f(A) :- p(A), \dots, \text{odd}(A), \text{even}(A), \dots$

Then we prune all hypotheses that include:

$\text{odd}(A), \text{even}(A)$



Never select both

UNSAT explanations

If we see this rule:

$f(A) :- p(A), \dots, \text{odd}(A), \text{even}(A), \dots$

Then we prune all hypotheses that include:

$\text{not } (\text{odd}_a \wedge \text{even}_a)$

UNSAT explanations

If we see this rule:

$f(A) :- p(A), \dots, \text{odd}(A), \text{even}(A), \dots$

Then we prune all hypotheses that include:

$(\text{not odd}_a \vee \text{not even}_a)$

UNSAT explanations

If we see this rule:

$f(A) :- p(A), \dots, \text{odd}(A), \text{even}(A), \dots$

Then we prune all hypotheses that include:

$$F_3 = F_2 \wedge (\text{not odd}_a \vee \text{not even}_a)$$

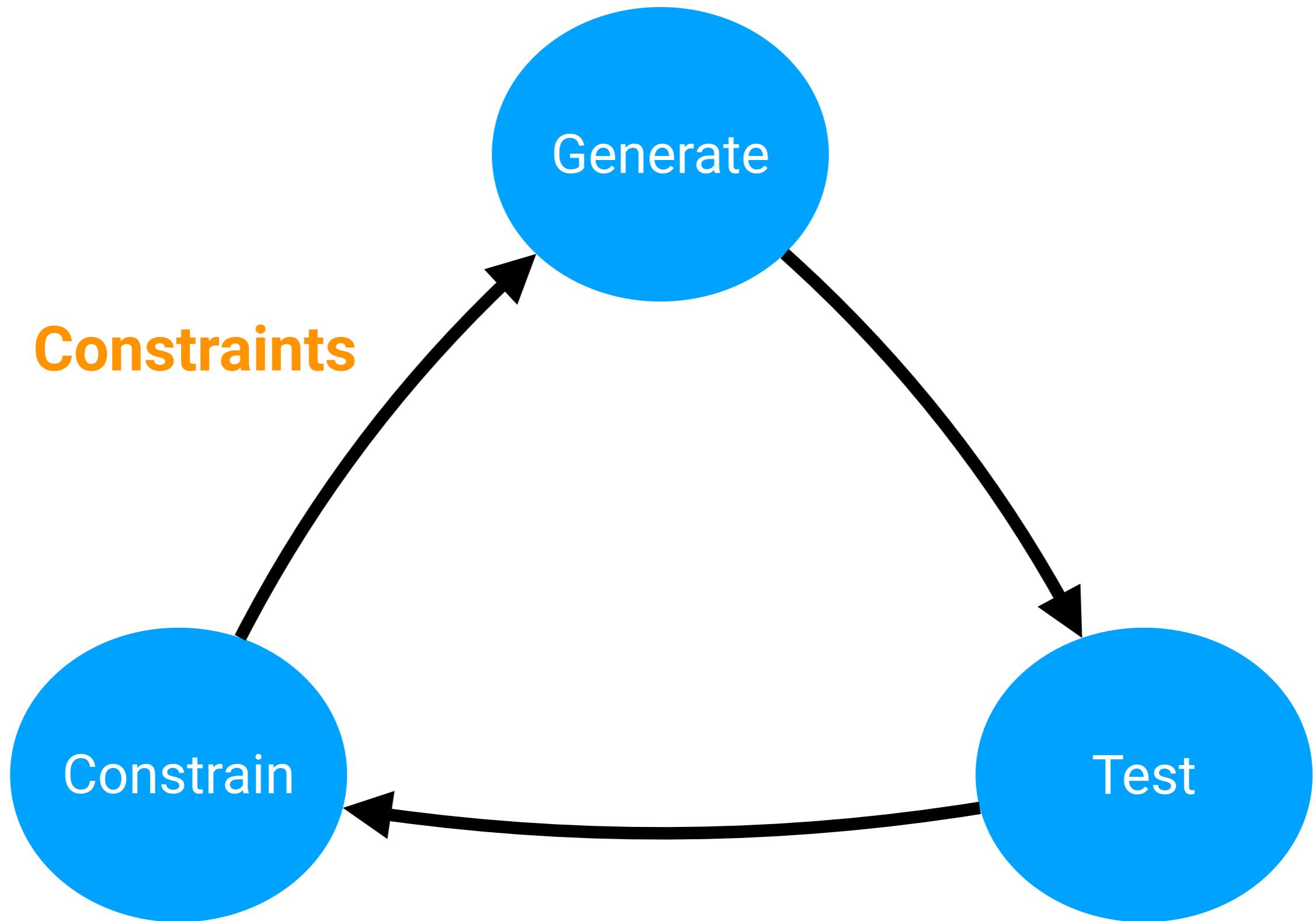
Key point

These constraints are easy to build
(we reason only about syntax)

Key point

Each constraint prunes at least one hypothesis

Popper



We return the best hypothesis not yet falsified

Key point

We learn from failures

Minor points

No bottom clauses

Minor points

No metarules

Minor points

No clause refinement operators

Popper (V1)

Cropper and Morel, MLJ21

Popper (V1)

+ Recursion

Popper (V1)

+ Recursion

```
f(V0,V1,V2) :- one(V1), tail(V0,V2).
```

```
f(V0,V1,V2) :- tail(V0,V3), decrement(V1,V4), f(V3,V4,V2).
```

Popper (V1)

+ Recursion

```
f(V0,V1):- empty(V0),empty(V1).  
f(V0,V1):- head(V0,V3),odd(V3),tail(V0,V2),f(V2,V1).  
f(V0,V1):- head(V0,V2),tail(V0,V4),even(V2),f(V4,V3),prepend(V2,V3,V1).
```

Popper (V1)

- + Recursion
- + Predicate invention*

Popper (V1)

- + Recursion
- + Predicate invention
- + Optimality

Popper (V1)

- + Recursion
- + Predicate invention
- + Optimality
- Small hypotheses (<5 rules)

Popper (V1)

- + Recursion
- + Predicate invention
- + Optimality

- Small hypotheses (<5 rules)
- Cannot handle noise

Popper (V1)

- + Recursion
- + Predicate invention
- + Optimality
- Small hypotheses (<5 rules)
- Cannot handle noise
- Small rules (< 10 literals)

Combo (Popper V2)

Learns optimal hypotheses with many (100+) rules

Cropper, AAAI22
Cropper and Hocquette, ECAI23

Combo (Popper V2)

Combine *non-seperable* hypotheses

Separable hypothesis

H { happy(A) :- rich(A).
 happy(A) :- friend(A,B), famous(B).
 happy(A) :- married(A,B), beautiful(B).

Separable hypothesis

h1 = { happy(A):- rich(A).

h2 = { happy(A):- friend(A,B), famous(B).

h3 = { happy(A):- married(A,B), beautiful(B).

Separable hypothesis

$h_1 = \{ \text{happy}(A) :- \text{rich}(A).$

$h_2 = \{ \text{happy}(A) :- \text{friend}(A, B), \text{famous}(B).$

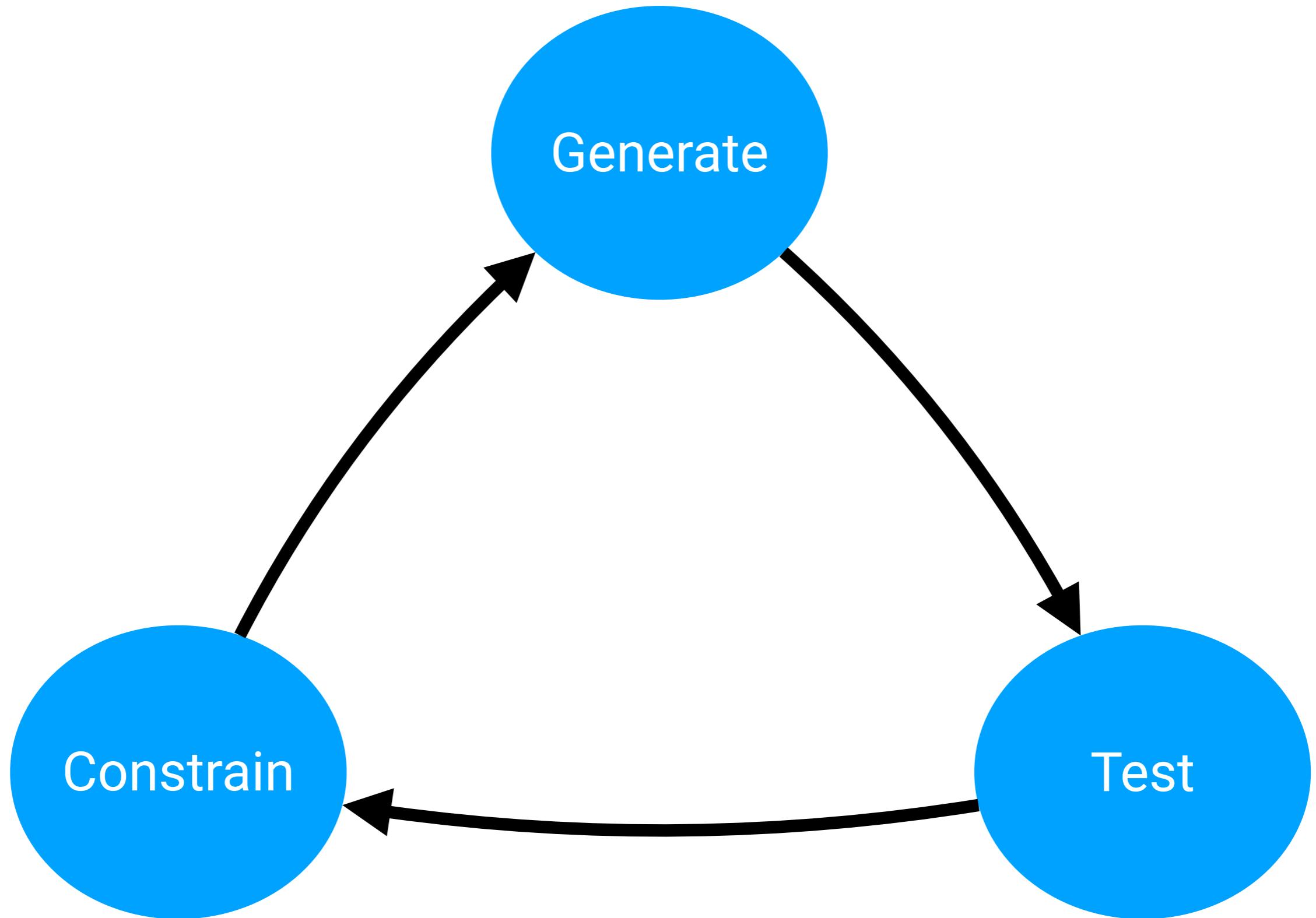
$h_3 = \{ \text{happy}(A) :- \text{married}(A, B), \text{beautiful}(B).$

$$H = h_1 \cup h_2 \cup h_3$$

Non-separable hypothesis

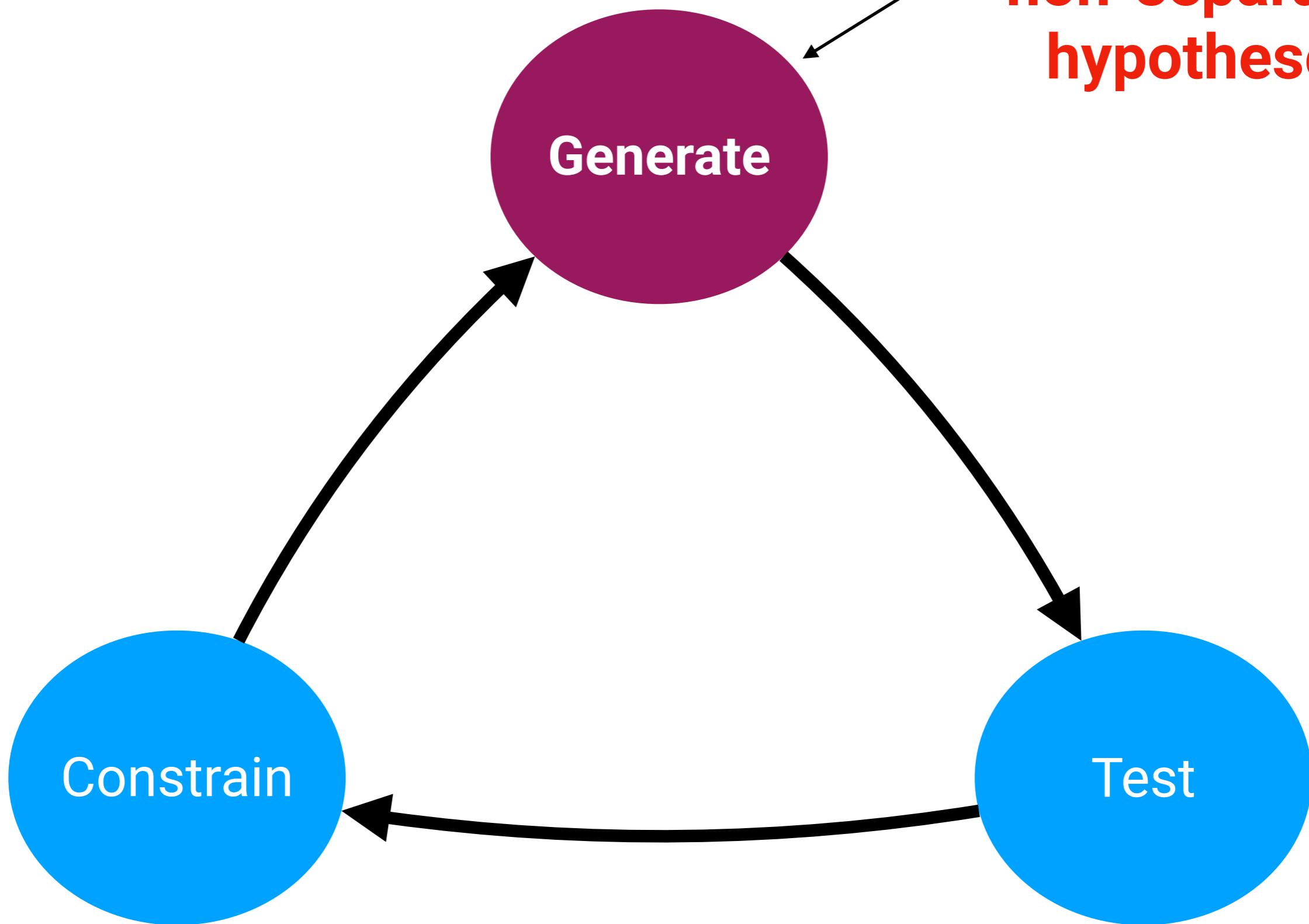
H {
 happy(A) :- rich(A).
 happy(A) :- married(A,B), happy(B).

Popper (V1)



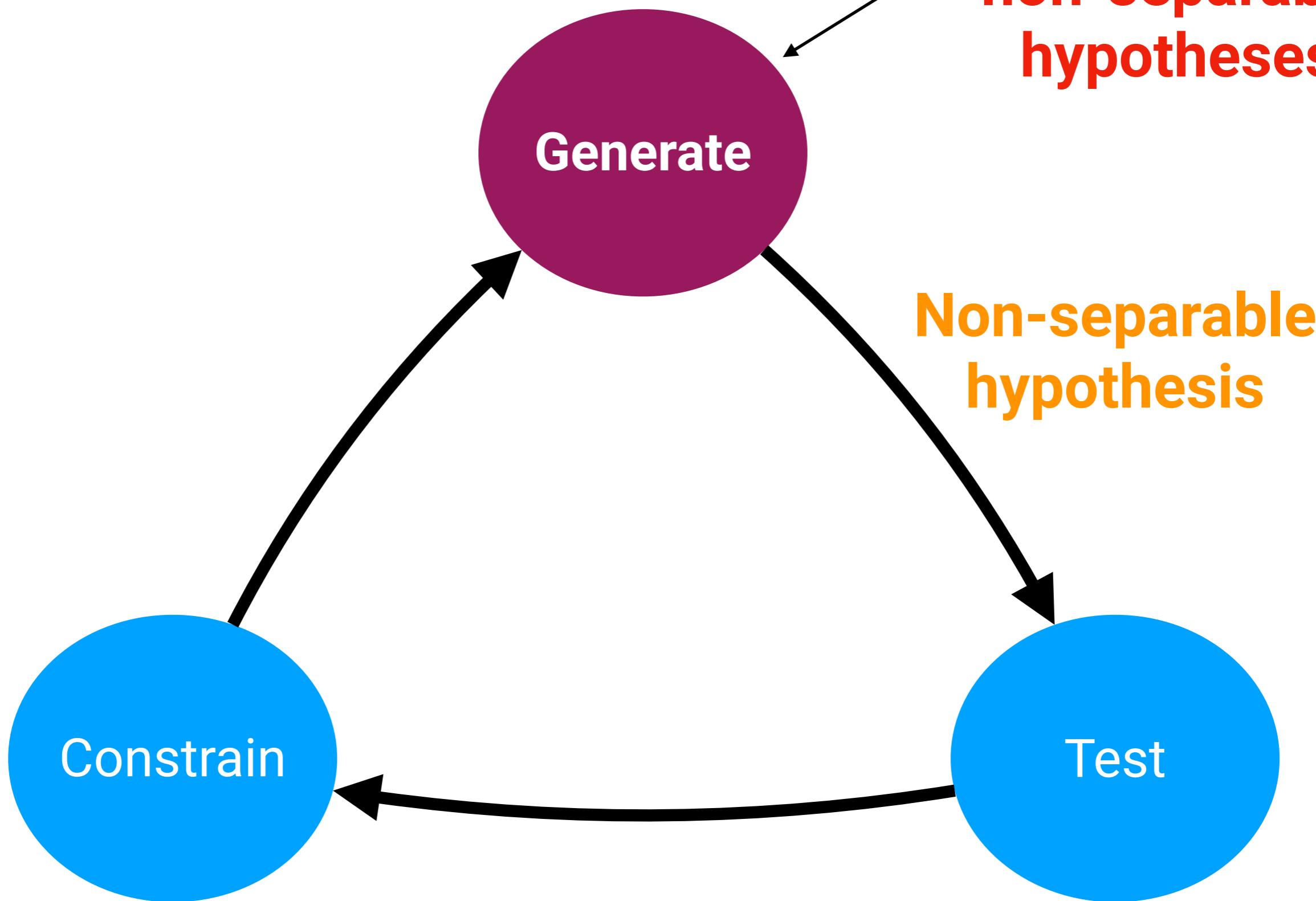
Combo (Popper V2)

Only generate
non-separable
hypotheses



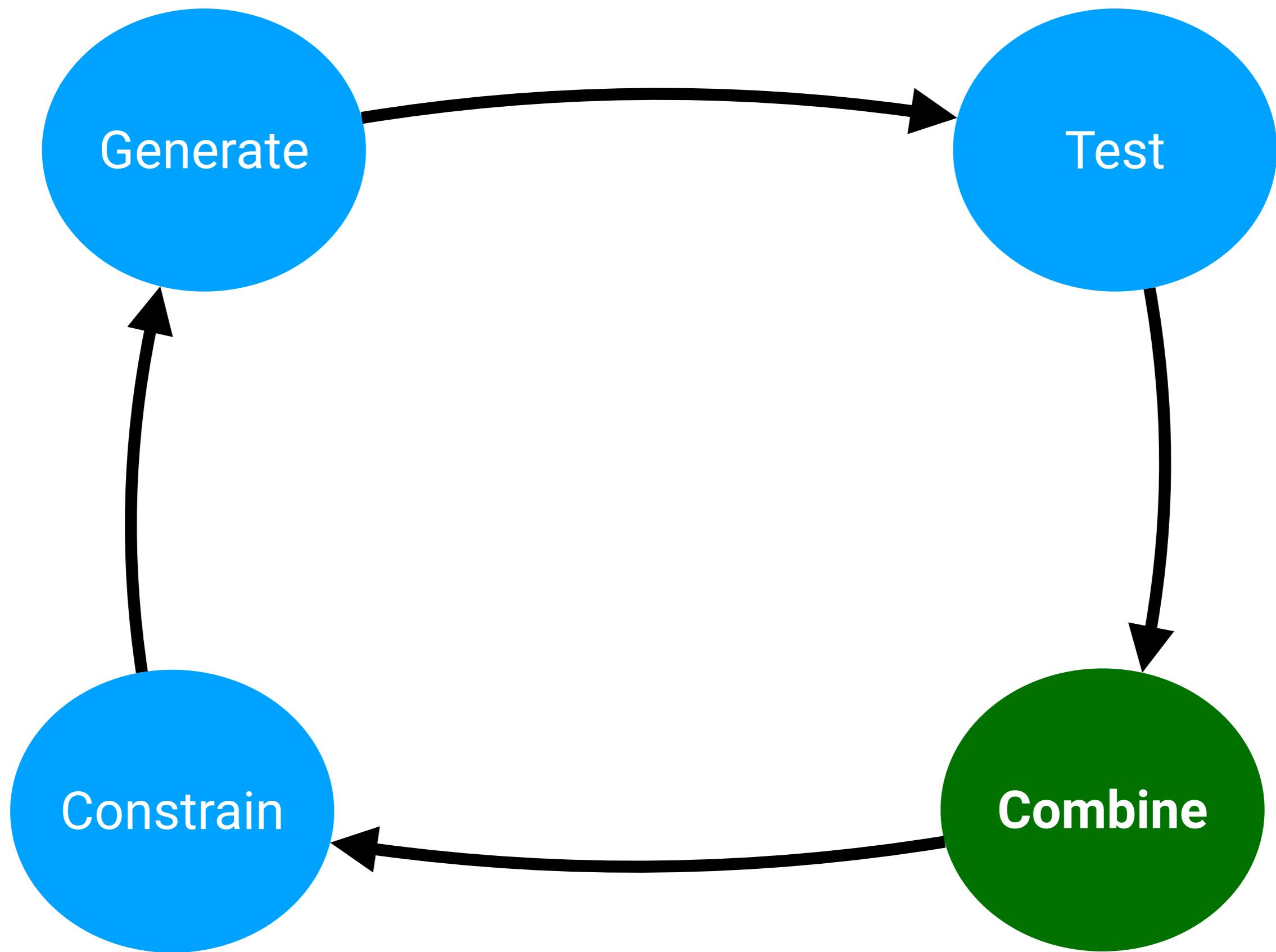
Combo (Popper V2)

Only generate
non-separable
hypotheses

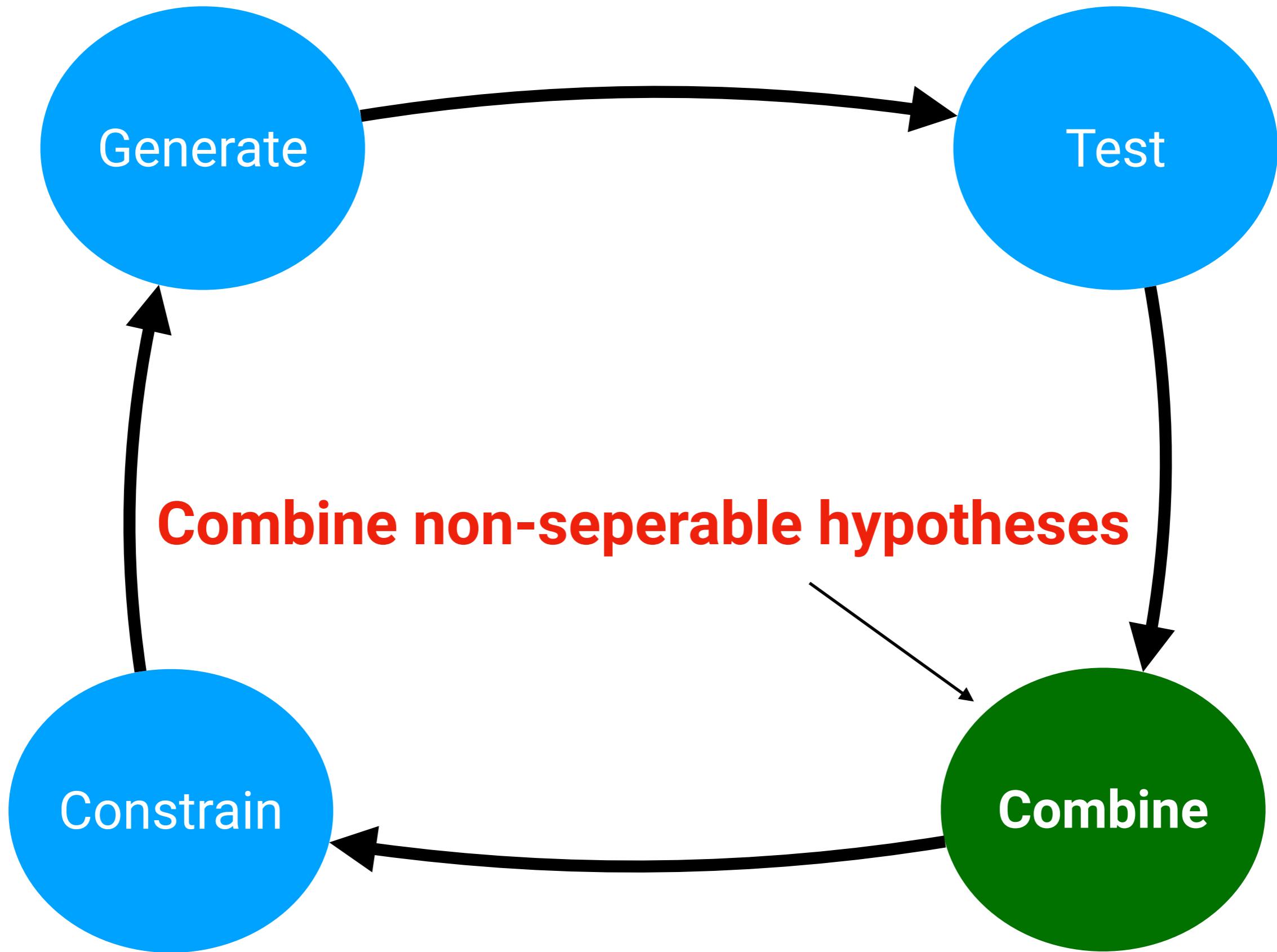


How to learn a seperable hypothesis?

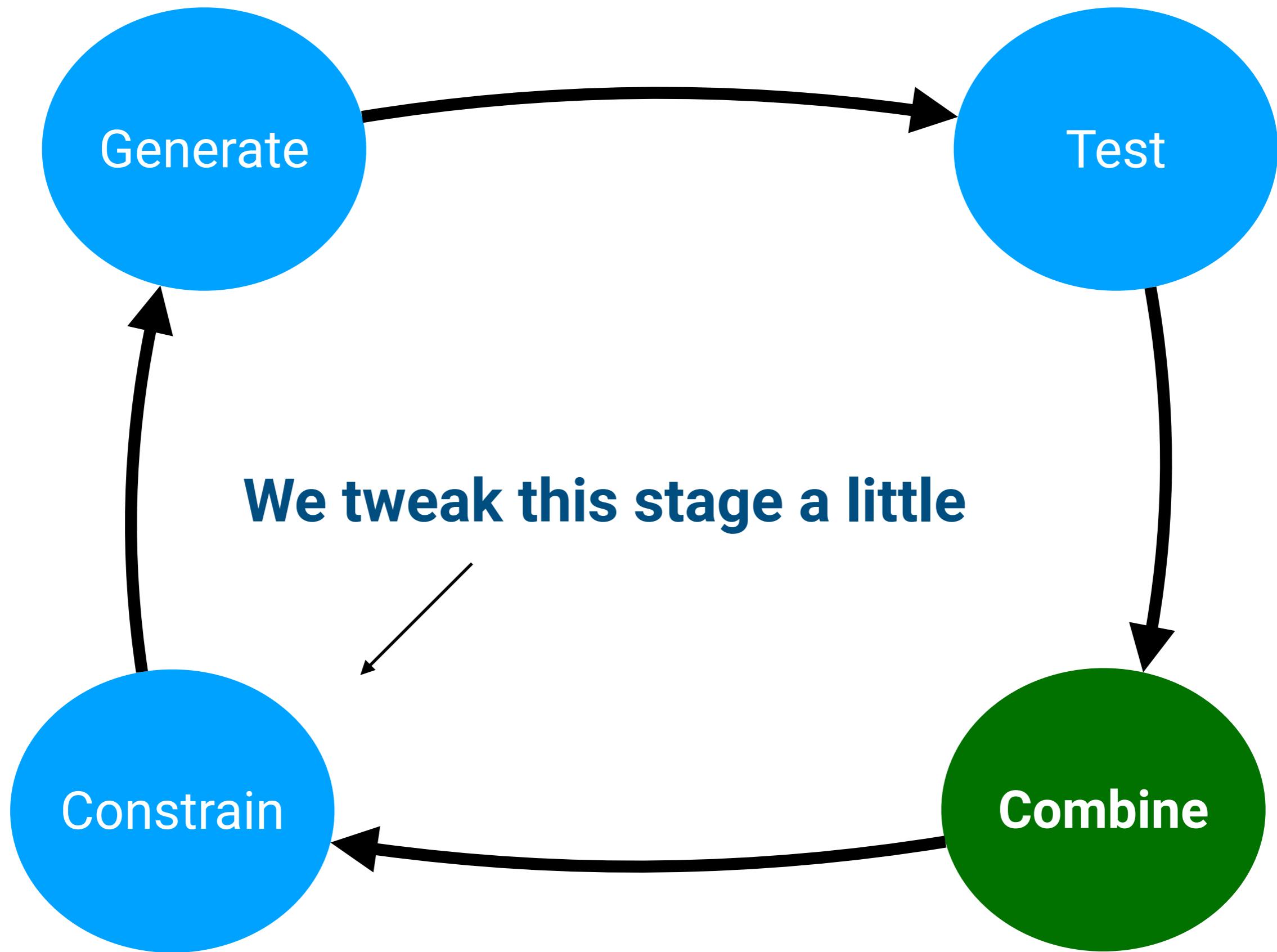
Combo (Popper V2)



Combo (Popper V2)



Combo (Popper V2)



Combine

Combine

Given

- a set of rules where each rule covers at least one positive and no negative examples

Combine

Given

- a set of rules where each rule covers at least one positive and no negative examples
- coverage of each rule

Combine

Given

- a set of rules where each rule covers at least one positive and no negative examples
- coverage of each rule
- size of each rule

Combine

Given

- a set of rules where each rule covers at least one positive and no negative examples
- coverage of each rule
- size of each rule

Find

- A subset of rules that covers all the positive examples and is minimal in size

Combine

Given

- a set of rules where each rule covers at least one positive and no negative examples
- coverage of each rule
- size of each rule

Find

- A subset of rules that covers all the positive examples and is minimal in size

Similar to ASPAL/ILASP but combine is not given all possible rules nor BK nor negative examples

MaxSAT

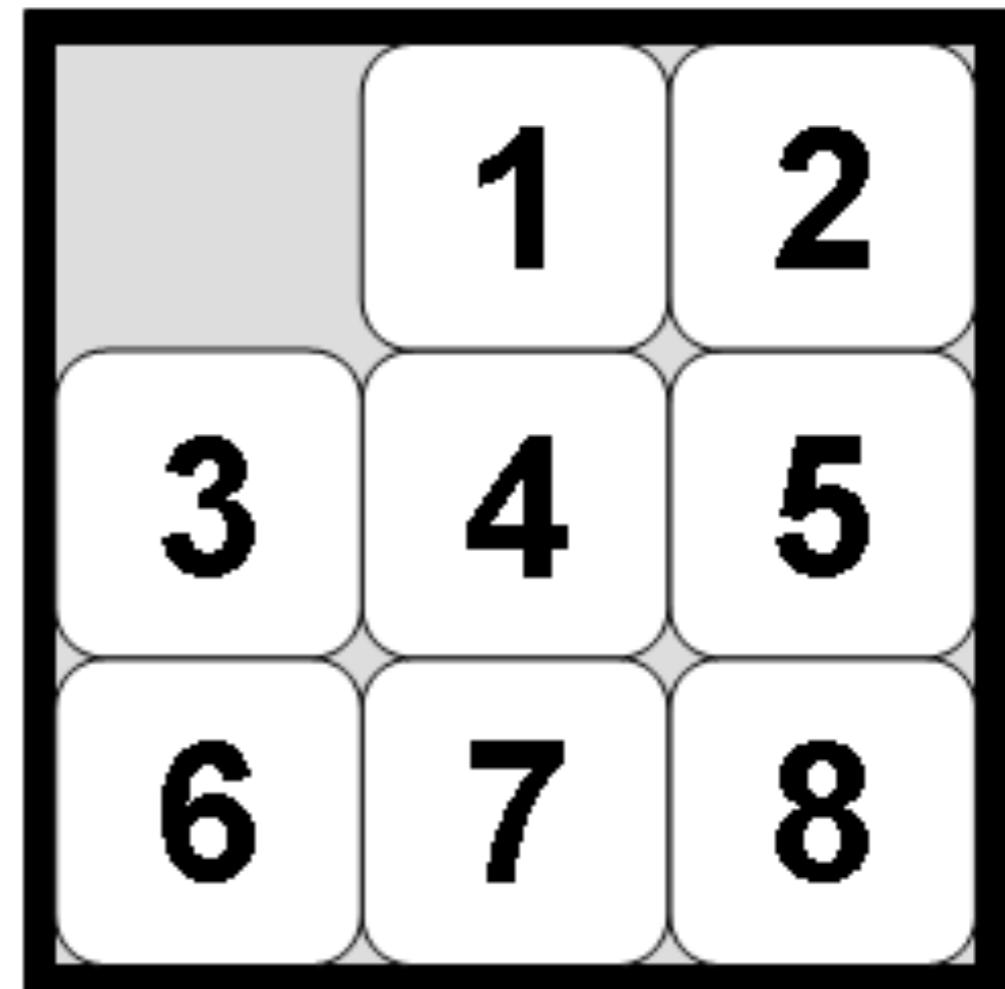
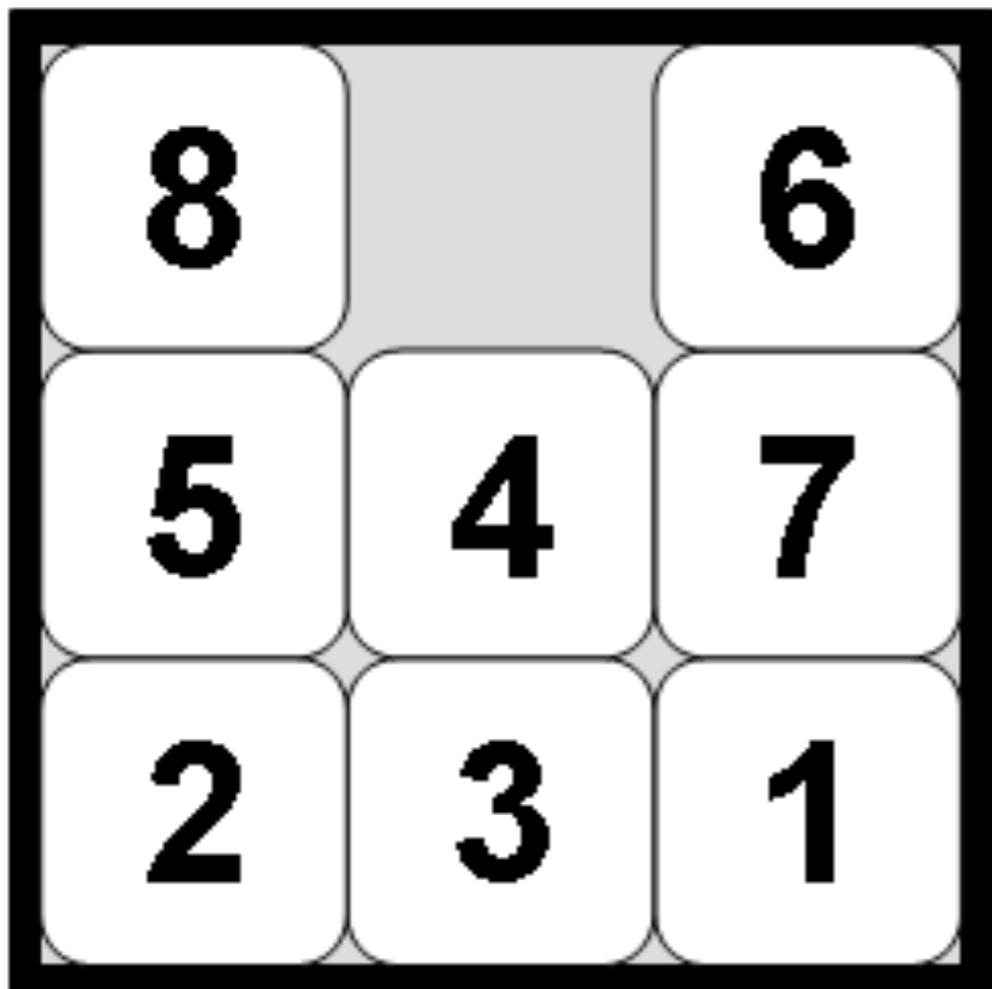
Combo impact

Can learn hypothesis with 100s of rules and
100s of literals

Combo impact

Guaranteed to learn the smallest hypothesis

Learing game rules



Learn the legal moves a game from game traces

modeh(legal_move,4).

modeb(cell,4).

modeb(step,2).

modeb(input_move,3).

modeb(role,1).

modeb(index,1).

modeb(succ,2).

modeb(scoremap,2).

modeb(successor,2).

modeb(tile,1).

modeb(int_1,1).

modeb(int_2,1).

modeb(int_3,1).

...

modeb(int_100,1).

What symbol in the head



modeh(legal_move, 4).

modeb(cell, 4).

modeb(step, 2).

modeb(input_move, 3).

modeb(role, 1).

modeb(index, 1).

modeb(succ, 2).

modeb(scoremap, 2).

modeb(successor, 2).

modeb(tile, 1).

modeb(int_1, 1).

modeb(int_2, 1).

modeb(int_3, 1).

...

modeb(int_100, 1).

`modeh(legal_move, 4).`

`modeb(cell, 4).`

`modeb(step, 2).`

`modeb(input_move, 3).`

`modeb(role, 1).`

`modeb(index, 1). ←`

`modeb(succ, 2).`

`modeb(scoremap, 2).`

`modeb(successor, 2).`

`modeb(tile, 1).`

`modeb(int_1, 1).`

`modeb(int_2, 1).`

`modeb(int_3, 1).`

`...`

`modeb(int_100, 1).`

**What symbols
can go in a body**

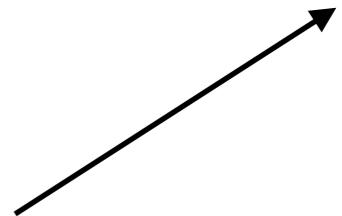
```
popper iggp-eight-puzzle-legal-move --max-vars=8 --max-body=50
```

```
popper iggp-eight-puzzle-legal-move --max-vars=8 --max-body=50
```



Rules can have at most 8 variables

```
popper iggp-eight-puzzle-legal-move --max-vars=8 --max-body=50
```



Rules can have at most 50 body literals

There are at least 10^{146} rules

The hypothesis space contains all rule combinations

```
3.8 s INFO: Generating programs of size: 4
4.1 s INFO: Generating programs of size: 5
4.5 s INFO: ****
4.5 s INFO: New best hypothesis:
4.5 s INFO: tp:344 fn:1039 tn:3117 fp:0 size:5
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V2,V4),input_move(V1,V2,V4),true_cell(V0,V4,V3,V5),cell_type_b(V5).
4.5 s INFO: ****
4.5 s INFO: ****
4.5 s INFO: New best hypothesis:
4.5 s INFO: tp:691 fn:692 tn:3117 fp:0 size:10
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V4,V2),input_move(V1,V2,V4),true_cell(V0,V4,V3,V5),cell_type_b(V5).
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V2,V4),input_move(V1,V2,V4),true_cell(V0,V4,V3,V5),cell_type_b(V5).
4.5 s INFO: ****
4.5 s INFO: ****
4.5 s INFO: New best hypothesis:
4.5 s INFO: tp:691 fn:692 tn:3117 fp:0 size:10
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V4,V2),input_move(V1,V2,V4),true_cell(V0,V4,V3,V5),cell_type_b(V5).
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V2,V4),input_move(V1,V2,V4),true_cell(V0,V4,V3,V5),cell_type_b(V5).
4.5 s INFO: ****
4.5 s INFO: ****
4.5 s INFO: New best hypothesis:
4.5 s INFO: tp:1034 fn:349 tn:3117 fp:0 size:15
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V4,V3),input_move(V1,V2,V4),true_cell(V0,V2,V4,V5),cell_type_b(V5).
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V4,V2),input_move(V1,V2,V4),true_cell(V0,V4,V3,V5),cell_type_b(V5).
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V2,V4),input_move(V1,V2,V4),true_cell(V0,V4,V3,V5),cell_type_b(V5).
4.5 s INFO: ****
4.5 s INFO: ****
4.5 s INFO: New best hypothesis:
4.5 s INFO: tp:1383 fn:0 tn:3117 fp:0 size:20
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V4,V3),input_move(V1,V2,V4),true_cell(V0,V2,V4,V5),cell_type_b(V5).
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V4,V2),input_move(V1,V2,V4),true_cell(V0,V4,V3,V5),cell_type_b(V5).
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V3,V4),input_move(V1,V2,V4),true_cell(V0,V2,V4,V5),cell_type_b(V5).
4.5 s INFO: legal_move(V0,V1,V2,V3):- succ(V2,V4),input_move(V1,V2,V4),true_cell(V0,V4,V3,V5),cell_type_b(V5).
4.5 s INFO: ****
6.6 s INFO: Generating programs of size: 6
15.1 s INFO: Generating programs of size: 7
17.0 s INFO: Generating programs of size: 8
17.6 s INFO: Generating programs of size: 9
***** SOLUTION *****
Precision:1.00 Recall:1.00 TP:1383 FN:0 TN:3117 FP:0 Size:20
legal_move(V0,V1,V2,V3):- succ(V4,V3),input_move(V1,V2,V4),true_cell(V0,V2,V4,V5),cell_type_b(V5).
legal_move(V0,V1,V2,V3):- succ(V4,V2),input_move(V1,V2,V4),true_cell(V0,V4,V3,V5),cell_type_b(V5).
legal_move(V0,V1,V2,V3):- succ(V3,V4),input_move(V1,V2,V4),true_cell(V0,V2,V4,V5),cell_type_b(V5).
legal_move(V0,V1,V2,V3):- succ(V2,V4),input_move(V1,V2,V4),true_cell(V0,V4,V3,V5),cell_type_b(V5).
*****
```

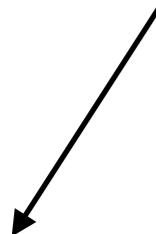
17.6s Generating programs of size: 9

***** SOLUTION *****

Precision:1.00 Recall:1.00 TP:1383 FN:0 TN:3117 FP:0 Size:20

```
legal_move(V0,V1,V2,V3):- succ(V4,V3),input_move(V1,V2,V4),cell(V0,V2,V4,V5),cell_type_b(V5).  
legal_move(V0,V1,V2,V3):- succ(V4,V2),input_move(V1,V2,V4),cell(V0,V4,V3,V5),cell_type_b(V5).  
legal_move(V0,V1,V2,V3):- succ(V3,V4),input_move(V1,V2,V4),cell(V0,V2,V4,V5),cell_type_b(V5).  
legal_move(V0,V1,V2,V3):- succ(V2,V4),input_move(V1,V2,V4),cell(V0,V4,V3,V5),cell_type_b(V5).
```

Terminates after around 20s



```
17.6s Generating programs of size: 9
***** SOLUTION *****
Precision:1.00 Recall:1.00 TP:1383 FN:0 TN:3117 FP:0 Size:20
legal_move(V0,V1,V2,V3):- succ(V4,V3),input_move(V1,V2,V4),cell(V0,V2,V4,V5),cell_type_b(V5).
legal_move(V0,V1,V2,V3):- succ(V4,V2),input_move(V1,V2,V4),cell(V0,V4,V3,V5),cell_type_b(V5).
legal_move(V0,V1,V2,V3):- succ(V3,V4),input_move(V1,V2,V4),cell(V0,V2,V4,V5),cell_type_b(V5).
legal_move(V0,V1,V2,V3):- succ(V2,V4),input_move(V1,V2,V4),cell(V0,V4,V3,V5),cell_type_b(V5).
```

```
popper iggp-coins-next-cell --max-vars=8 --max-body=50
```

```
1.5s Generating programs of size: 5
3.1s ****
3.1s New best hypothesis:
3.1s tp:576 fn:0 tn:1968 fp:0 size:35
3.1s next_cell(V0,V1,V2):- zerocoins(V2),does_jump(V0,V4,V1,V3).
3.1s next_cell(V0,V1,V2):- twocoins(V2),does_jump(V0,V3,V4,V1).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),succ(V3,V1),does_jump(V0,V5,V4,V3).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V4,V3,V5),succ(V1,V3).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),twocoins(V2),does_jump(V0,V4,V3,V5).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V4,V5,V3),succ(V1,V3).
3.1s next_cell(V0,V1,V2):- zerocoins(V2),cell(V0,V1,V2),does_jump(V0,V5,V4,V3).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V3,V4),succ(V3,V1).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V4,V3),different(V1,V3),different(V1,V4).
3.1s ****
3.1s ****
3.1s New best hypothesis:
3.1s tp:576 fn:0 tn:1968 fp:0 size:11
3.1s next_cell(V0,V1,V2):- twocoins(V2),does_jump(V0,V3,V4,V1).
3.1s next_cell(V0,V1,V2):- zerocoins(V2),does_jump(V0,V4,V1,V3).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V4,V3),different(V1,V3),different(V1,V4).
3.1s ****
3.1s Generating programs of size: 6
3.6s Generating programs of size: 7
3.6s Generating programs of size: 8
***** SOLUTION *****

Precision:1.00 Recall:1.00 TP:576 FN:0 TN:1968 FP:0 Size:11
next_cell(V0,V1,V2):- twocoins(V2),does_jump(V0,V3,V4,V1).
next_cell(V0,V1,V2):- zerocoins(V2),does_jump(V0,V4,V1,V3).
next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V4,V3),different(V1,V3),different(V1,V4).
```

Learns a hypothesis of size 35

```
1.5s Generating programs of size: 5
```

```
3.1s *****
```

```
3.1s New best hypothesis:
```

```
3.1s tp:576 fn:0 tn:1968 fp:0 size:35
```

```
3.1s next_cell(V0,V1,V2):- zerocoins(V2),does_jump(V0,V4,V1,V3).
```

```
3.1s next_cell(V0,V1,V2):- twocoins(V2),does_jump(V0,V3,V4,V1).
```

```
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),succ(V3,V1),does_jump(V0,V5,V4,V3).
```

```
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V4,V3,V5),succ(V1,V3).
```

```
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),twocoins(V2),does_jump(V0,V4,V3,V5).
```

```
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V4,V5,V3),succ(V1,V3).
```

```
3.1s next_cell(V0,V1,V2):- zerocoins(V2),cell(V0,V1,V2),does_jump(V0,V5,V4,V3).
```

```
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V3,V4),succ(V3,V1).
```

```
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V4,V3),different(V1,V3),different(V1,V4).
```

```
3.1s *****
```

```
3.1s *****
```

```
3.1s New best hypothesis:
```

```
3.1s tp:576 fn:0 tn:1968 fp:0 size:11
```

```
3.1s next_cell(V0,V1,V2):- twocoins(V2),does_jump(V0,V3,V4,V1).
```

```
3.1s next_cell(V0,V1,V2):- zerocoins(V2),does_jump(V0,V4,V1,V3).
```

```
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V4,V3),different(V1,V3),different(V1,V4).
```

```
3.1s *****
```

```
3.1s Generating programs of size: 6
```

```
3.6s Generating programs of size: 7
```

```
3.6s Generating programs of size: 8
```

```
***** SOLUTION *****
```

```
Precision:1.00 Recall:1.00 TP:576 FN:0 TN:1968 FP:0 Size:11
```

```
next_cell(V0,V1,V2):- twocoins(V2),does_jump(V0,V3,V4,V1).
```

```
next_cell(V0,V1,V2):- zerocoins(V2),does_jump(V0,V4,V1,V3).
```

```
next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V4,V3),different(V1,V3),different(V1,V4).
```

```
1.5s Generating programs of size: 5
3.1s *****
3.1s New best hypothesis:
3.1s tp:576 fn:0 tn:1968 fp:0 size:35
3.1s next_cell(V0,V1,V2):- zerocoins(V2),does_jump(V0,V4,V1,V3).
3.1s next_cell(V0,V1,V2):- twocoins(V2),does_jump(V0,V3,V4,V1).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),succ(V3,V1),does_jump(V0,V5,V4,V3).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V4,V3,V5),succ(V1,V3).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),twocoins(V2),does_jump(V0,V4,V3,V5).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V4,V5,V3),succ(V1,V3).
3.1s next_cell(V0,V1,V2):- zerocoins(V2),cell(V0,V1,V2),does_jump(V0,V5,V4,V3).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V3,V4),succ(V3,V1).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V4,V3),different(V1,V3),different(V1,V4).
3.1s *****
3.1s *****
3.1s New best hypothesis:
3.1s tp:576 fn:0 tn:1968 fp:0 size:11
```

```
3.1s next_cell(V0,V1,V2):- twocoins(V2),does_jump(V0,V3,V4,V1).
3.1s next_cell(V0,V1,V2):- zerocoins(V2),does_jump(V0,V4,V1,V3).
3.1s next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V4,V3),different(V1,V3),different(V1,V4).
```

```
3.1s *****
3.1s Generating programs of size: 6
3.6s Generating programs of size: 7
3.6s Generating programs of size: 8
***** SOLUTION *****
```



Improves to size 11

```
Precision:1.00 Recall:1.00 TP:576 FN:0 TN:1968 FP:0 Size:11
next_cell(V0,V1,V2):- twocoins(V2),does_jump(V0,V3,V4,V1).
next_cell(V0,V1,V2):- zerocoins(V2),does_jump(V0,V4,V1,V3).
next_cell(V0,V1,V2):- cell(V0,V1,V2),does_jump(V0,V5,V4,V3),different(V1,V3),different(V1,V4).
```

Why does it work?

Problem decomposition

Why does it work?

Generate stage is easier because the space
of non-separable hypotheses is smaller

Why does it work?

Combine stage is efficient because it only
considers promising hypotheses

Combo (Popper V2)

- + Recursion
- + Predicate invention
- + Optimality
- + Big hypotheses (>100 rules)

Combo (Popper V2)

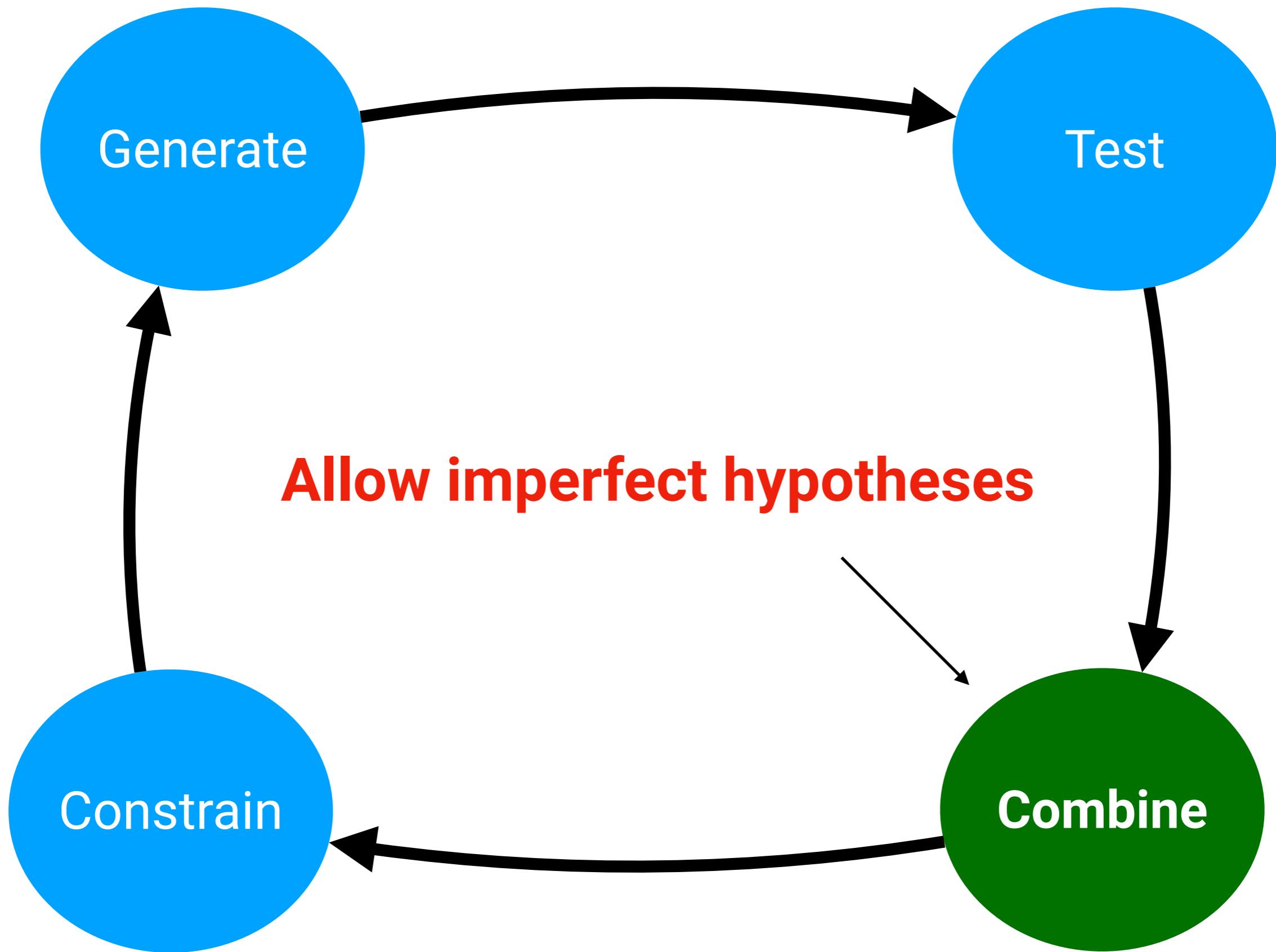
- + Recursion
- + Predicate invention
- + Optimality
- + Big hypotheses (>100 rules)

- Cannot handle noise
- Small rules (< 10 literals)

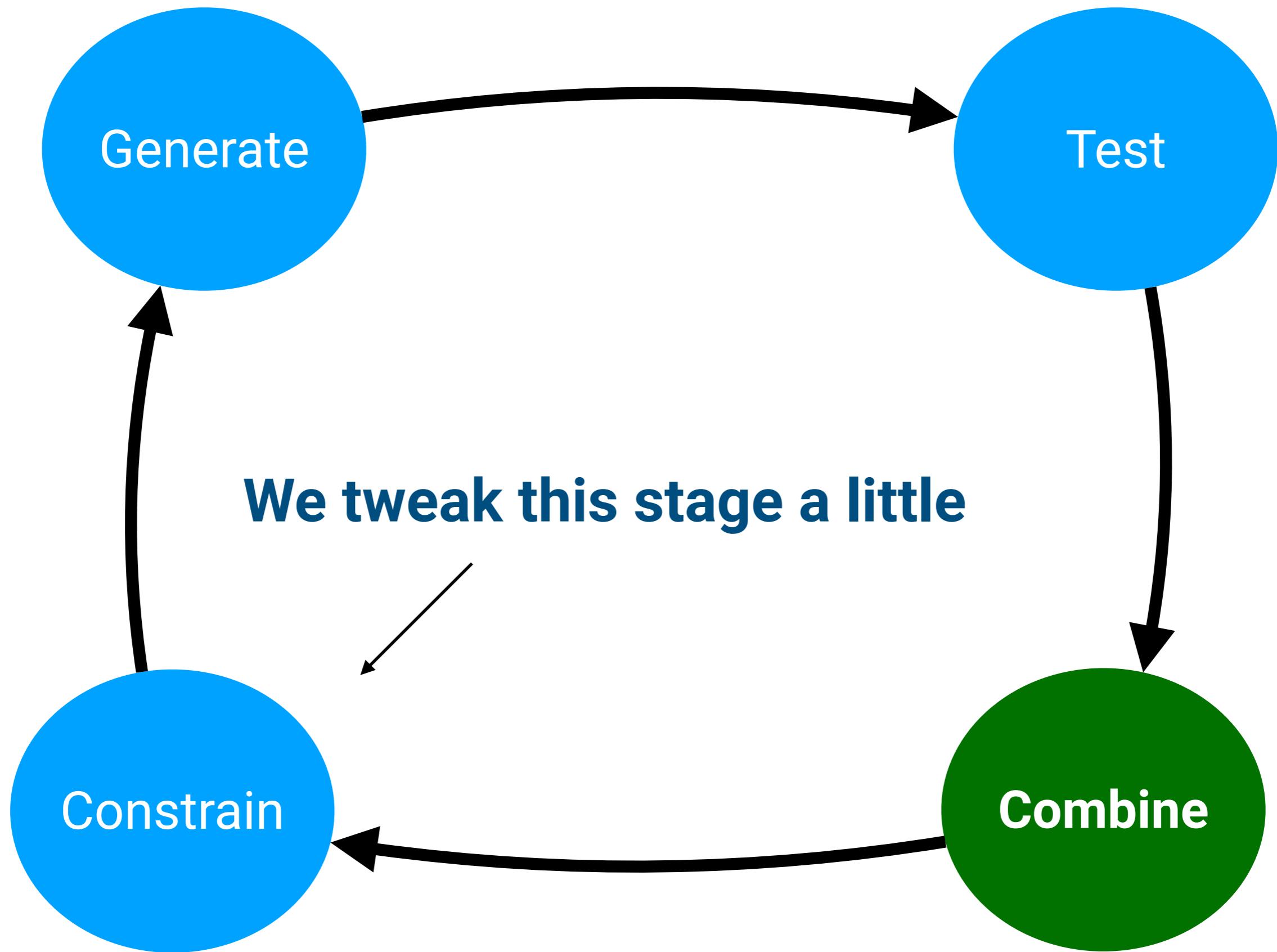
MaxSynth (Popper V3)

Noisy data

MaxSynth (Popper V3)



MaxSynth (Popper V3)



MaxSynth (Popper V3)

Given

- a set of rules where each rule covers **at least one positive example**

MaxSynth (Popper V3)

Given

- a set of rules where each rule covers **at least one positive example**
- coverage of each rule

MaxSynth (Popper V3)

Given

- a set of rules where each rule covers **at least one positive example**
- coverage of each rule
- size of each rule

MaxSynth (Popper V3)

Given

- a set of rules where each rule covers at least one positive example
- coverage of each rule
- size of each rule

Find

- A subset of rules that minimises errors + size

MaxSynth (Popper V3)

Given

- a set of rules where each rule covers at least one positive example
- coverage of each rule
- size of each rule

Find

- An MDL hypothesis

MaxSynth (Popper V3)

Exact + anytime MaxSAT

MaxSynth (Popper V3)

Many hidden details about constraints

MaxSynth (Popper V3)

- + Recursion
- + Predicate invention
- + Optimality
- + Big hypotheses (>100 rules)
- + Handles noise

- Small rules (< 10 literals)

MaxSynth (Popper V3)

- + Recursion
 - + Predicate invention
 - + Optimality
 - + Big hypotheses (>100 rules)
 - + Handles noise
-
- Small rules (< 10 literals)

Joiner (Popper V4)

Only generate non-splittable rules then join them

Joiner (Popper V4)

Learns hypotheses with big rules (100+ literals)

Why does it work?

Problem decomposition

Popper

Recursion [MLJ21]

Noisy data (MDL) [AAAI24]

Many (100+) rules [AAAI22, ECAI23]

Popper

Recursion [MLJ21]

Noisy data (MDL) [AAAI24]

Many (100+) rules [AAAI22, ECAI23]

Big rules (100+ literals) [IJCAI24]

Popper

Recursion [MLJ21]

Noisy data (MDL) [AAAI24]

Many (100+) rules [AAAI22, ECAI23]

Big rules (100+ literals) [IJCAI24]

Predicate invention [AAAI24, IJCAI24]

Negation as failure [AAAI24]

Popper

Recursion [MLJ21]

Noisy data (MDL) [AAAI24]

Many (100+) rules [AAAI22, ECAI23]

Big rules (100+ literals) [IJCAI24]

Predicate invention [AAAI24, IJCAI24]

Negation as failure [AAAI24]

Popper

Recursion [MLJ21]

Noisy data (MDL) [AAAI24]

Many (100+) rules [AAAI22, ECAI23]

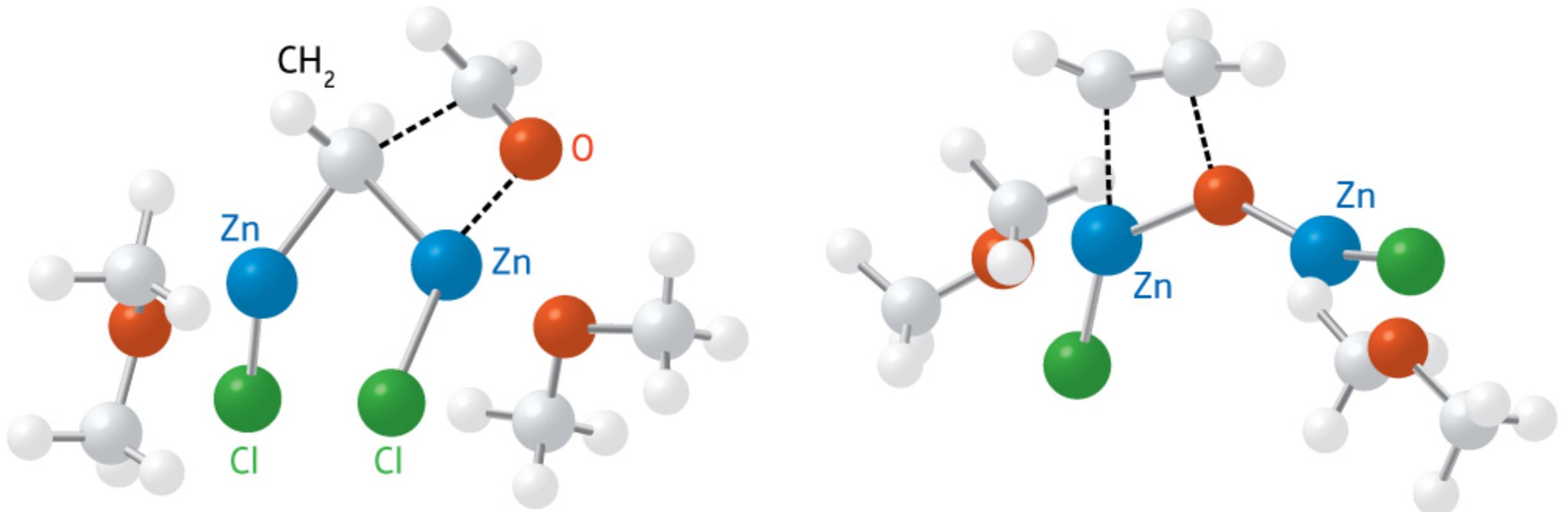
Big rules (100+ literals) [IJCAI24]

Predicate invention [AAAI24, IJCAI24]

Negation as failure [AAAI24]

Numerical reasoning [MLJ23,AAAI23]

Numerical reasoning



```
pharma4(A):- zinc(A,B), hacc(A,C), dist(A,B,C,D), leq(D,4.18), geq(D,2.22).  
pharma4(A):- hacc(A,C), hacc(A,E), dist(A,B,C,D), geq(D,1.23), leq(D,3.41).  
pharma4(A):- zinc(A,C), zinc(A,B), bond(B,C,du), dist(A,B,C,D), leq(D,1.23).
```

Popper

Recursion [MLJ21]

Noisy data (MDL) [AAAI24]

Many (100+) rules [AAAI22, ECAI23]

Big rules (100+ literals) [IJCAI24]

Predicate invention [AAAI24, IJCAI24]

Negation as failure [AAAI24]

Numerical reasoning [MLJ23,AAAI23]

Preprocessing [AAAI23]

Popper

Recursion [MLJ21]

Noisy data (MDL) [AAAI24]

Many (100+) rules [AAAI22, ECAI23]

Big rules (100+ literals) [IJCAI24]

Predicate invention [AAAI24, IJCAI24]

Negation as failure [AAAI24]

Numerical reasoning [MLJ23,AAAI23]

Preprocessing [AAAI23]

Probabilistic programs [?]

Popper

Recursion [MLJ21]

Noisy data (MDL) [AAAI24]

Many (100+) rules [AAAI22, ECAI23]

Big rules (100+ literals) [IJCAI24]

Predicate invention [AAAI24, IJCAI24]

Negation as failure [AAAI24]

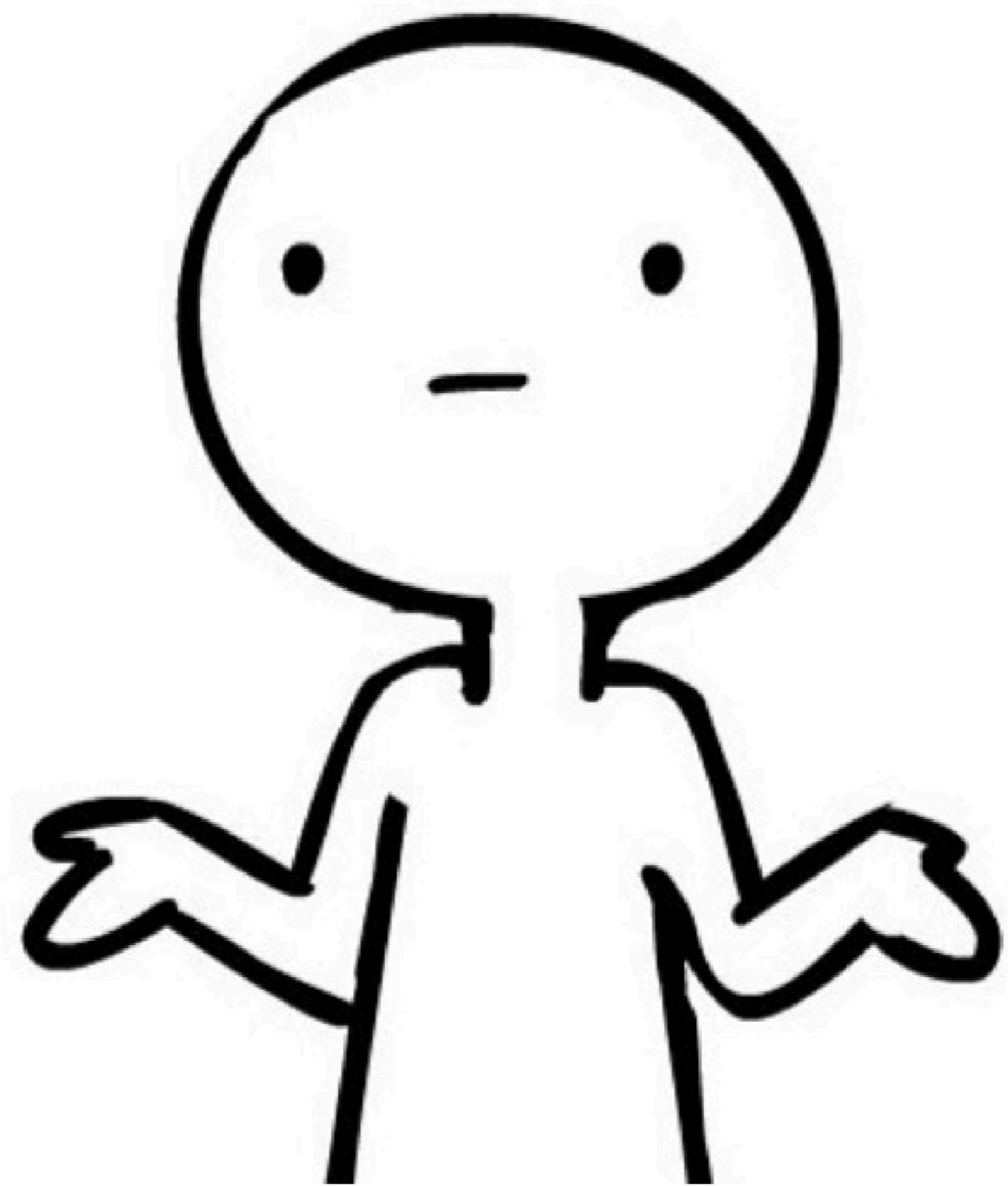
Numerical reasoning [MLJ23,AAAI23]

Preprocessing [AAAI23]

Probabilistic programs [?]

Symbolic ensembles [?]

So what?



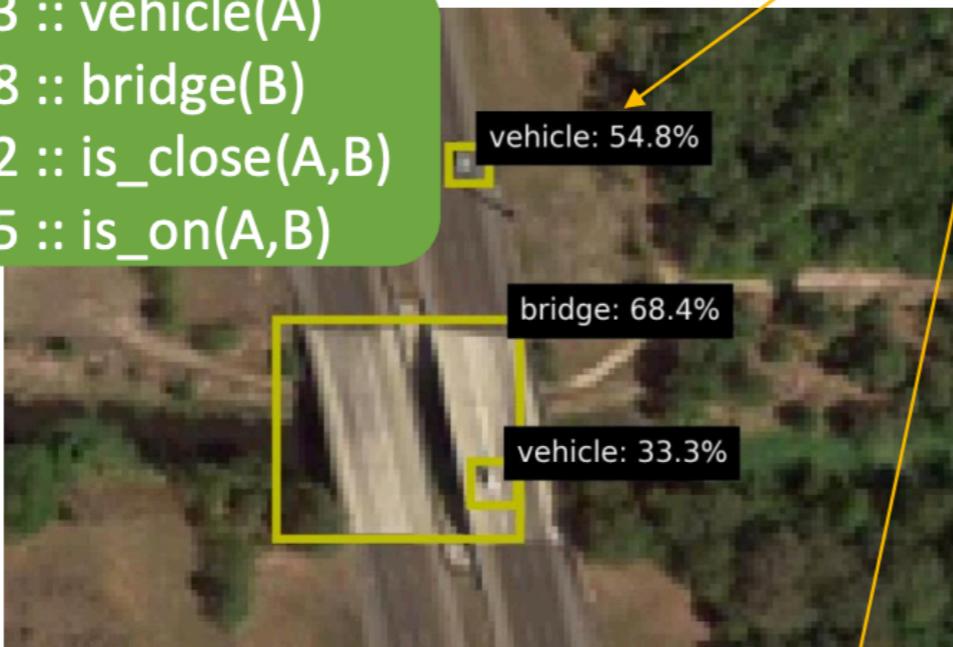
What is the actual impact?

People use it

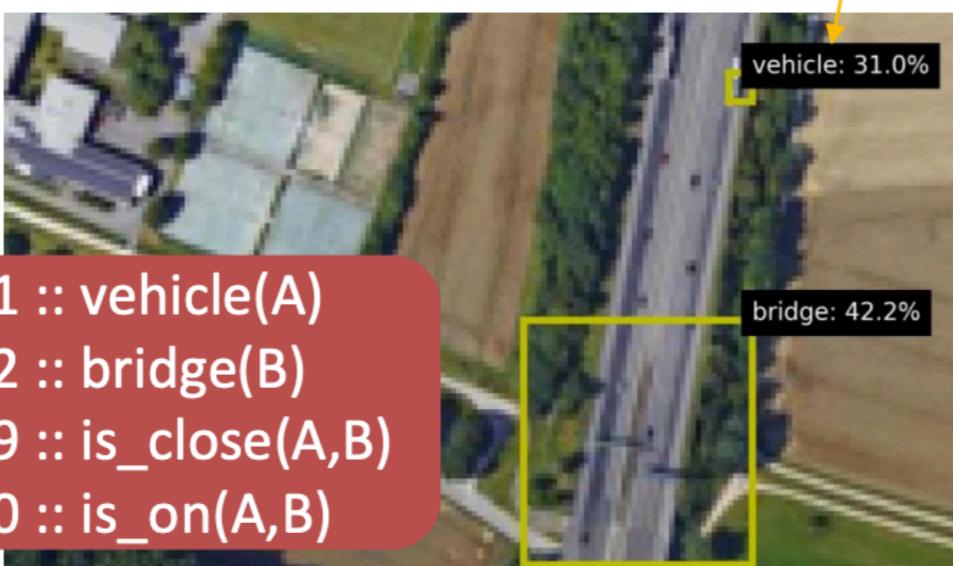
Image explanations

Examples

0.33 :: vehicle(A)
0.68 :: bridge(B)
0.92 :: is_close(A,B)
0.95 :: is_on(A,B)



positives



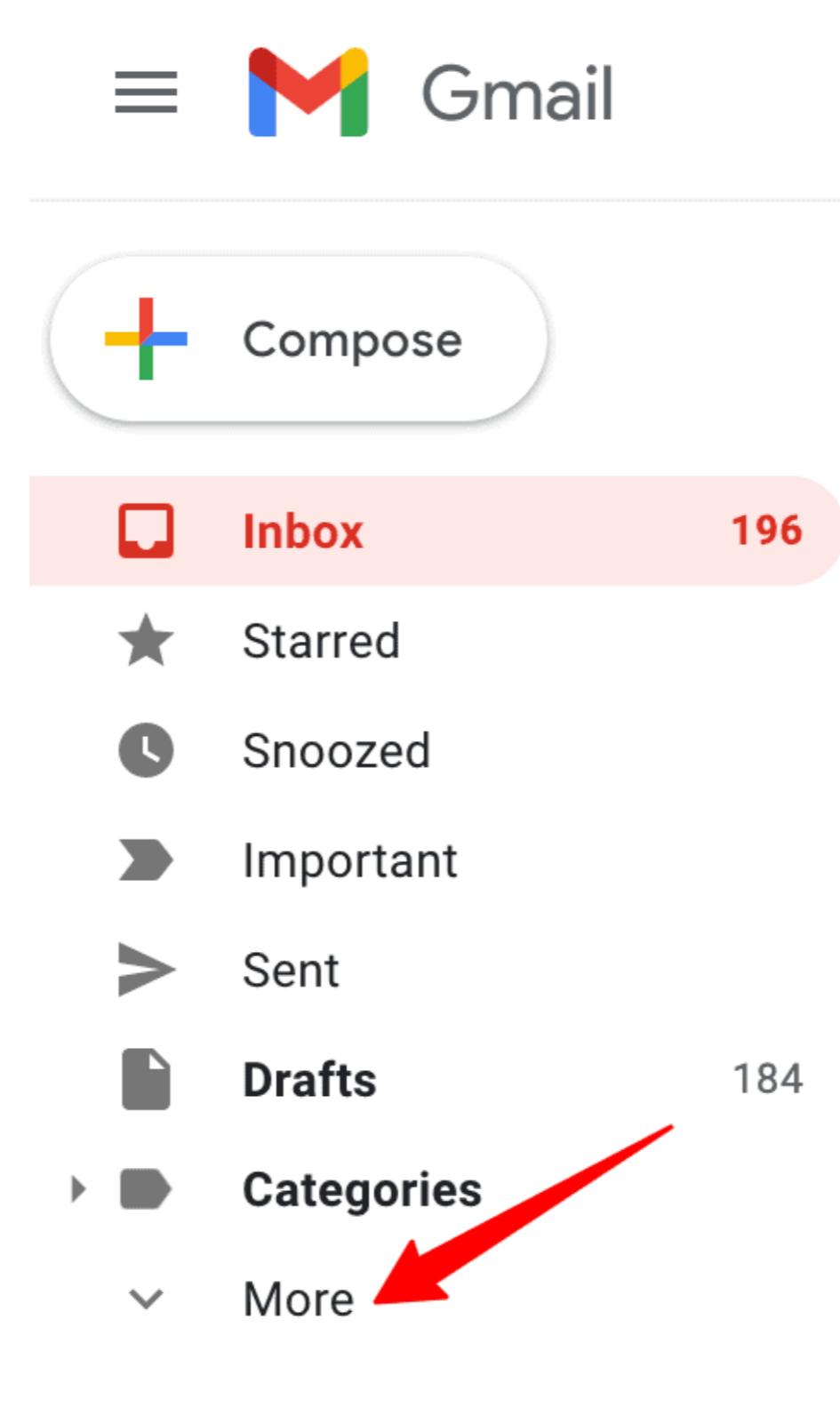
on(vehicle, bridge)
on(vehicle, bridge)

negatives

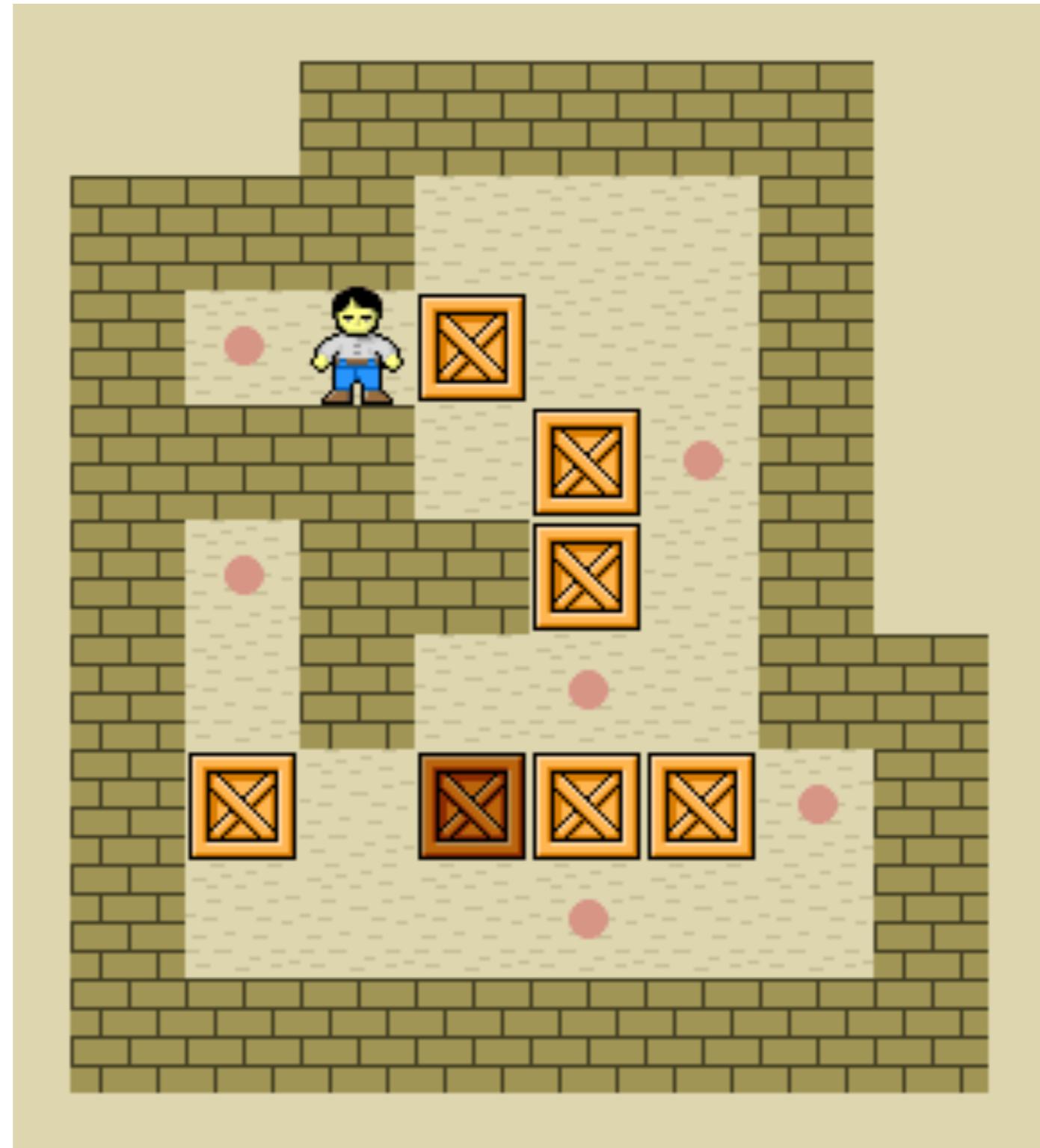
0.31 :: vehicle(A)
0.42 :: bridge(B)
0.29 :: is_close(A,B)
0.00 :: is_on(A,B)



Learn email rules

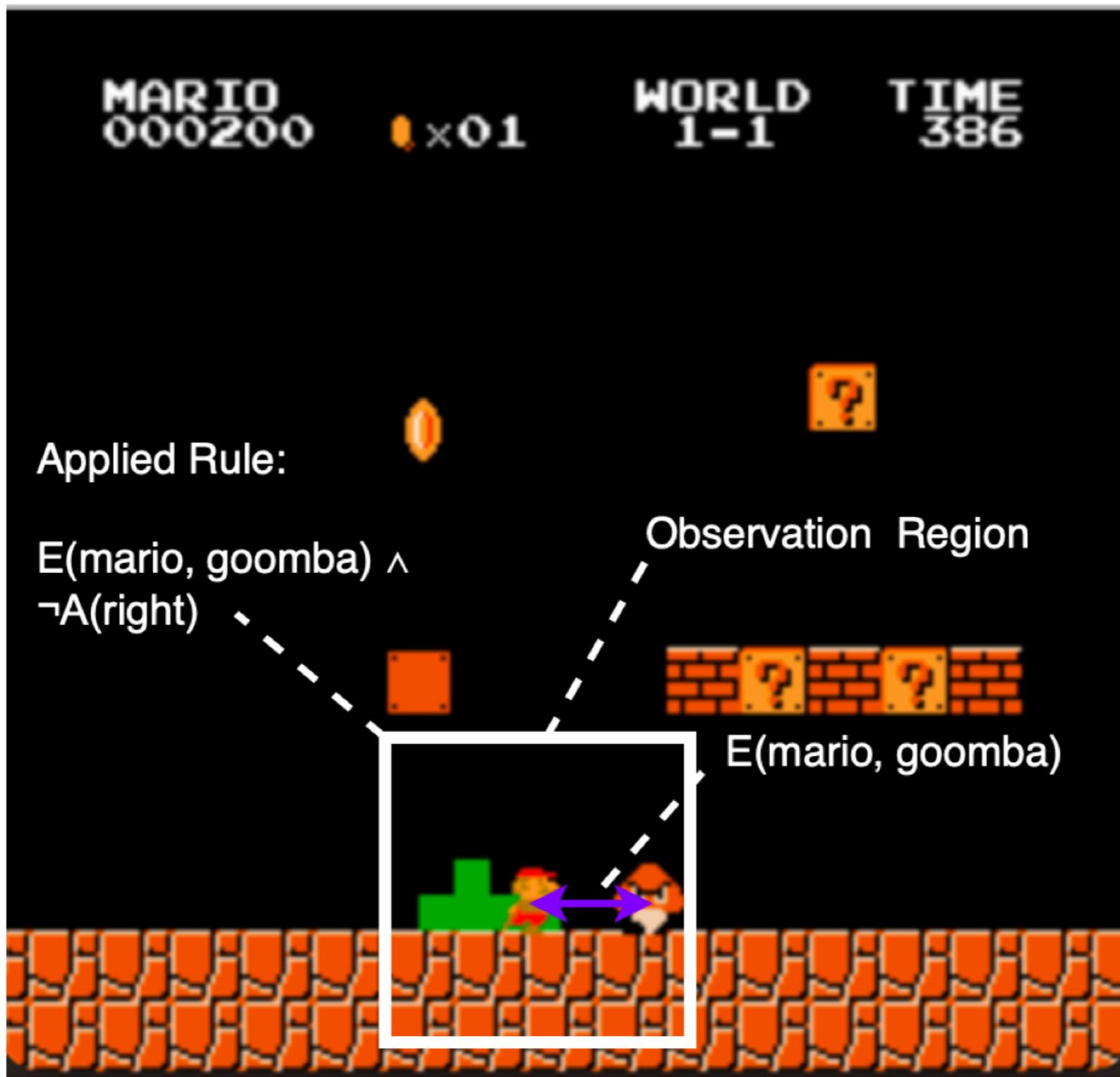


Learn game semantics

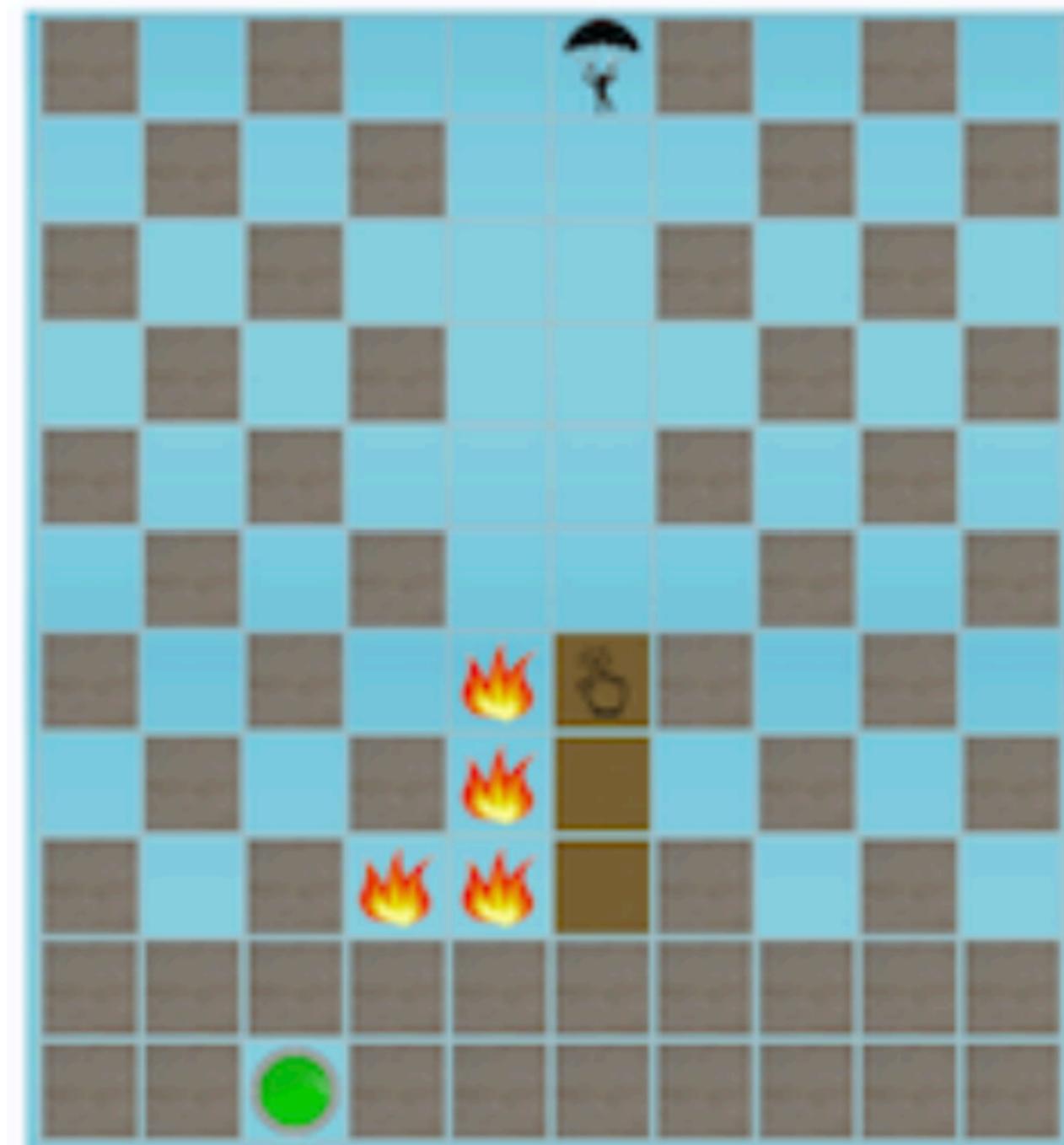
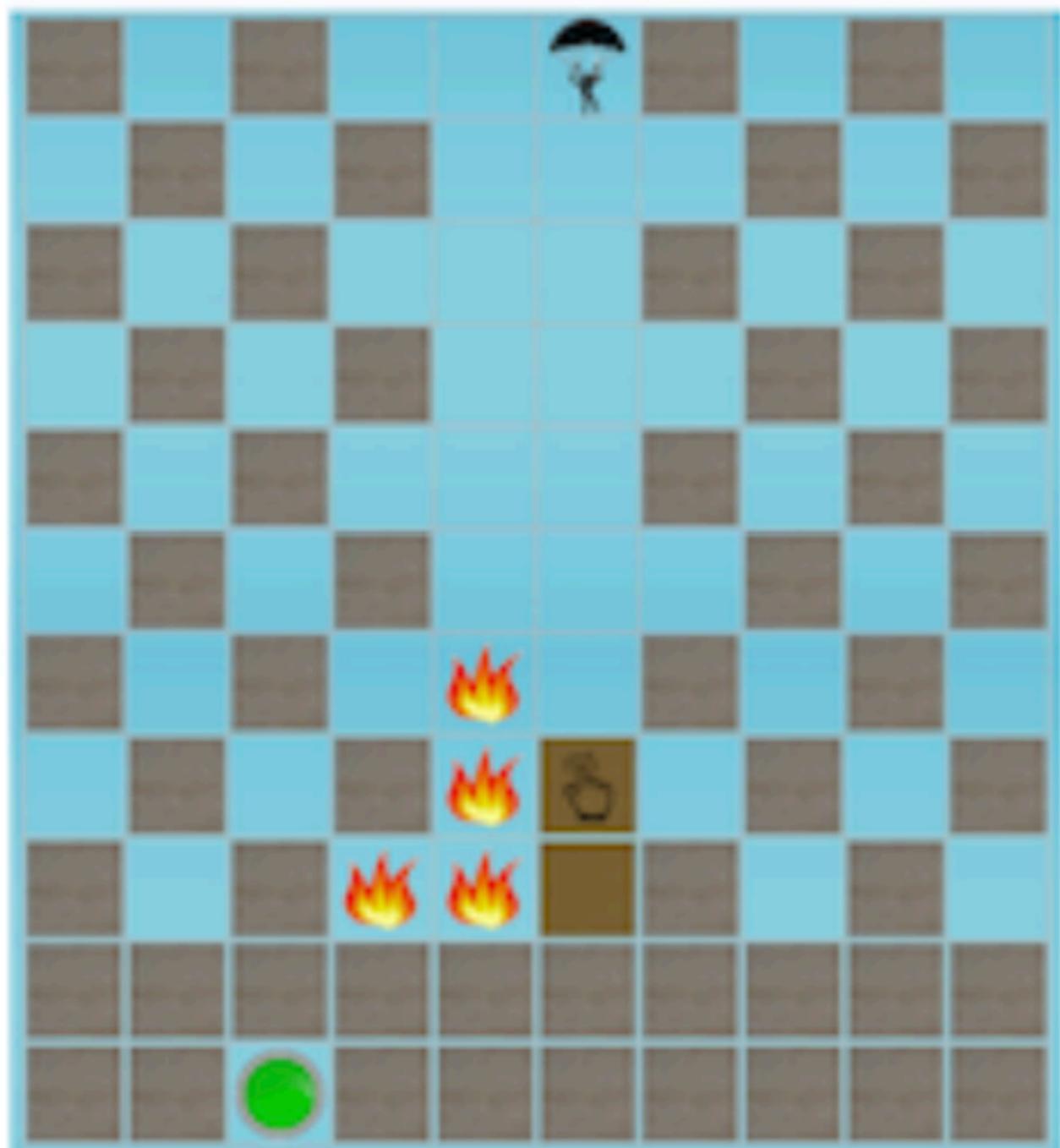


Cropper et al, MLJ21

Explain negative observations in RL

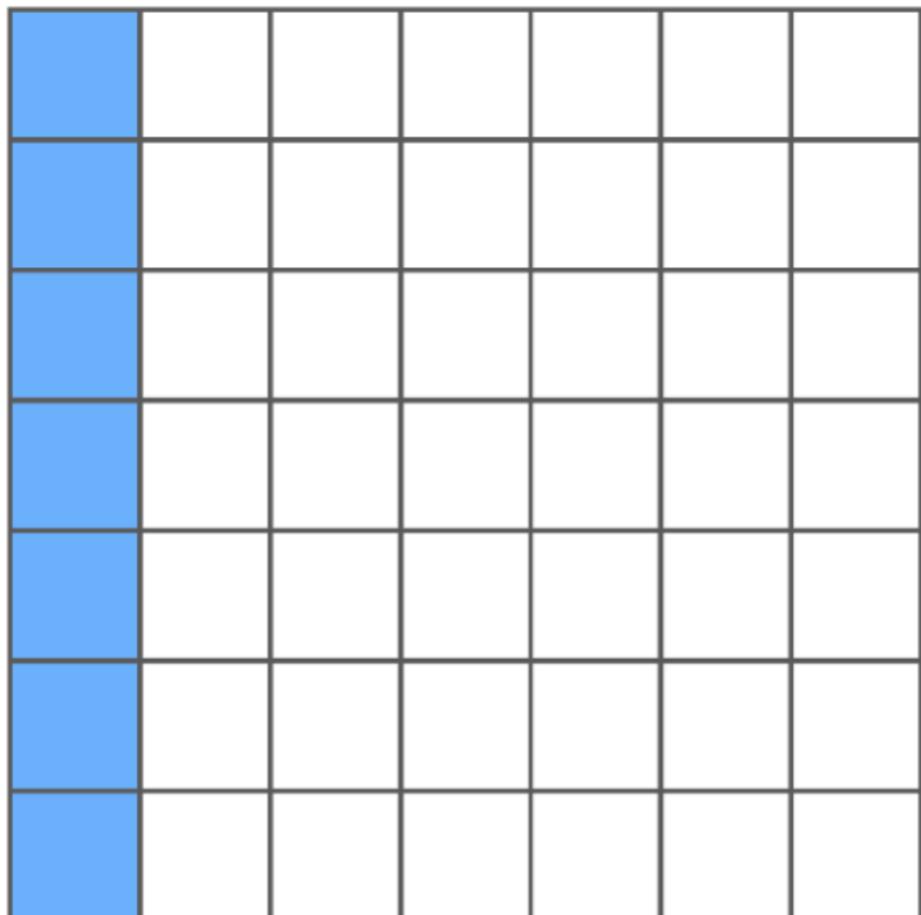


Playing games

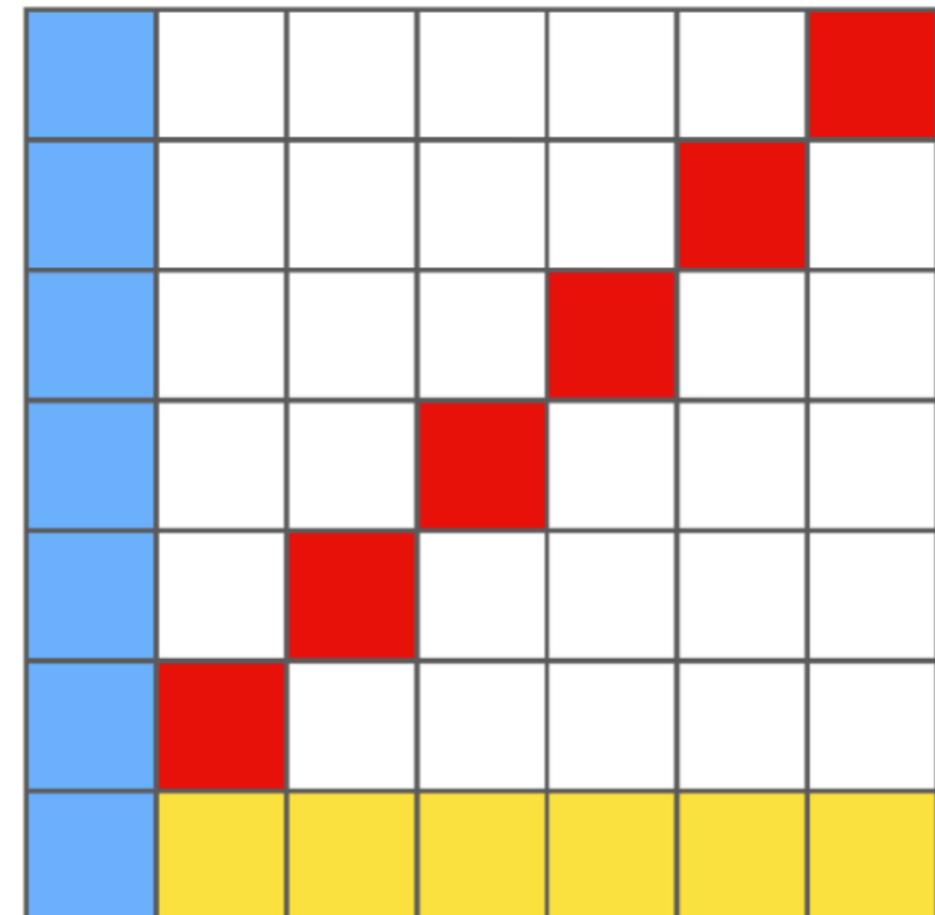


Abstraction and reasoning corpus

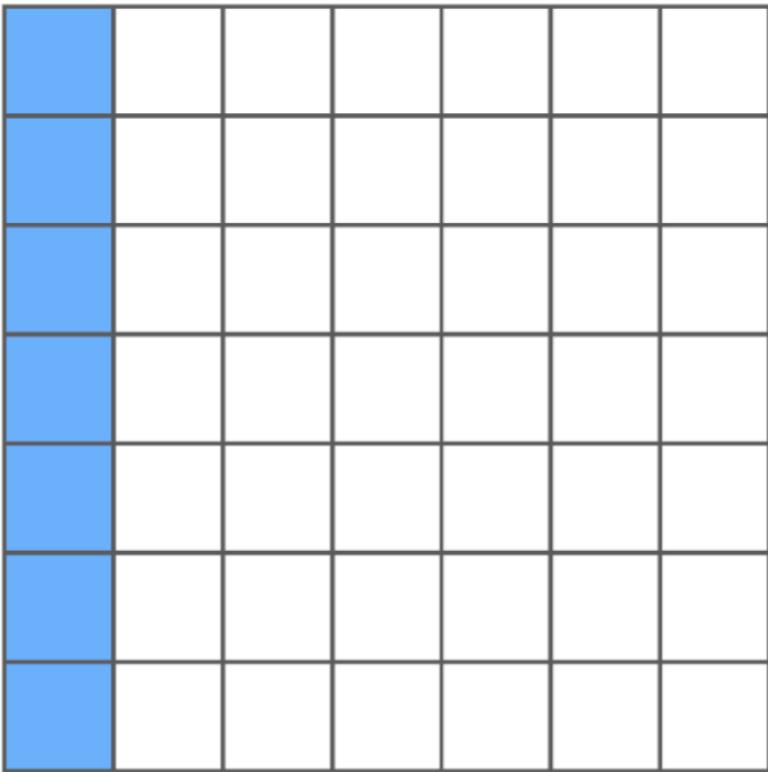
Input



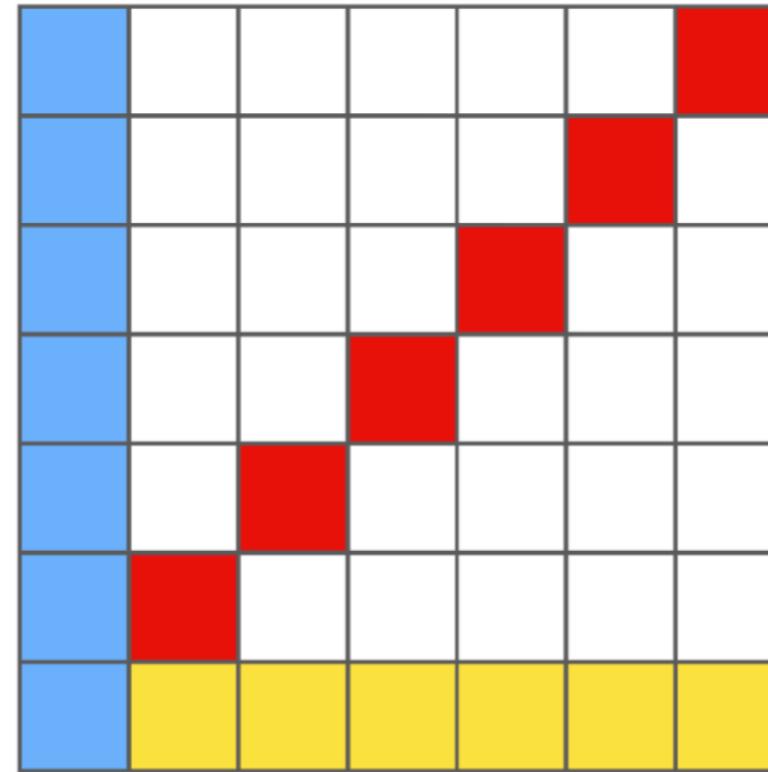
Output



Input



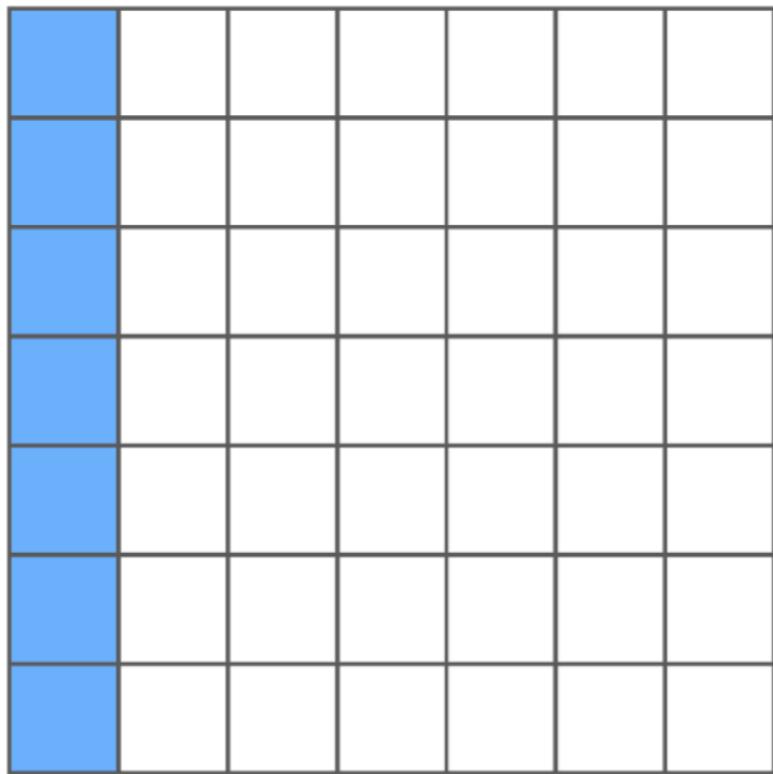
Output



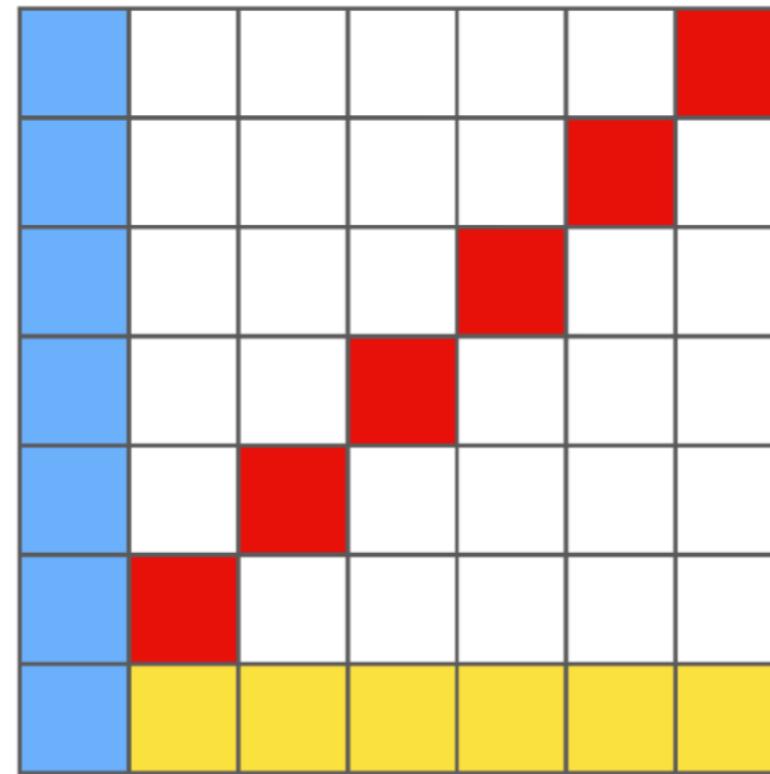
Standard approach:

learn a sequence of functions from
the input state to the output state

Input



Output



Standard approach:

design a set of functions, e.g. rotate, flip, scale,

Input

Blue						
Blue						
Blue						
Blue						
Blue						
Blue						
Blue						

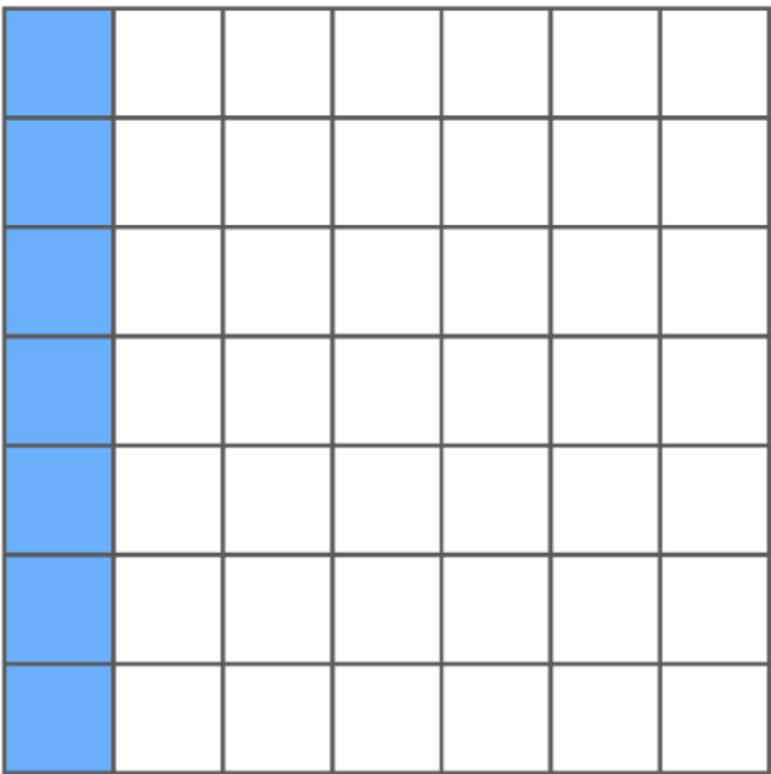
Output

Blue						Red
Blue						Red
Blue						Red
Blue						Red
Blue						Red
Blue						Yellow
Blue						Yellow

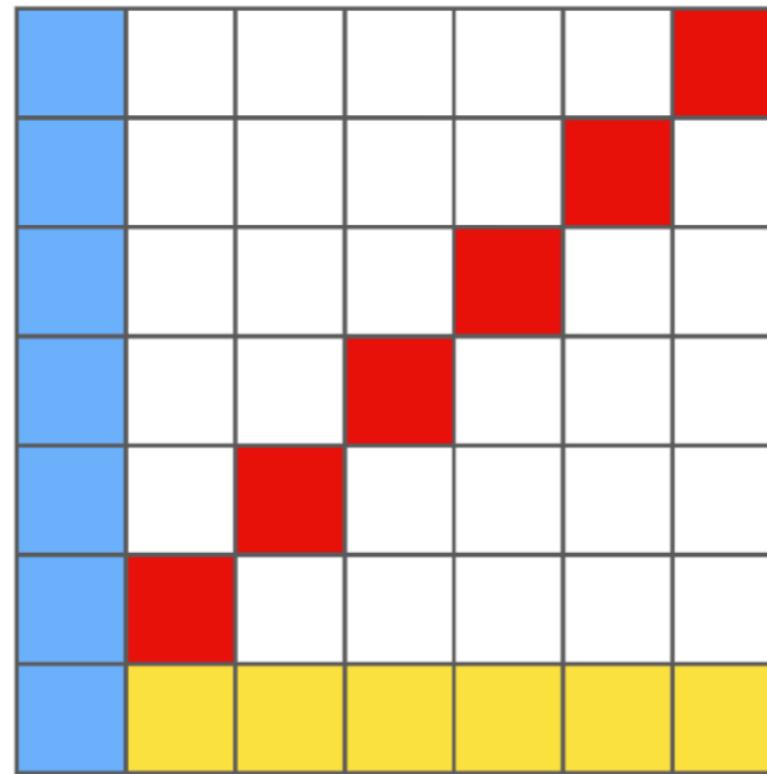
Our idea:

learn relations not functions

Input

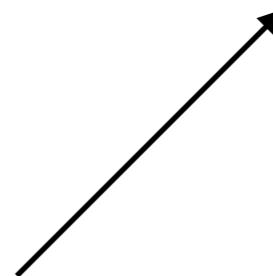


Output

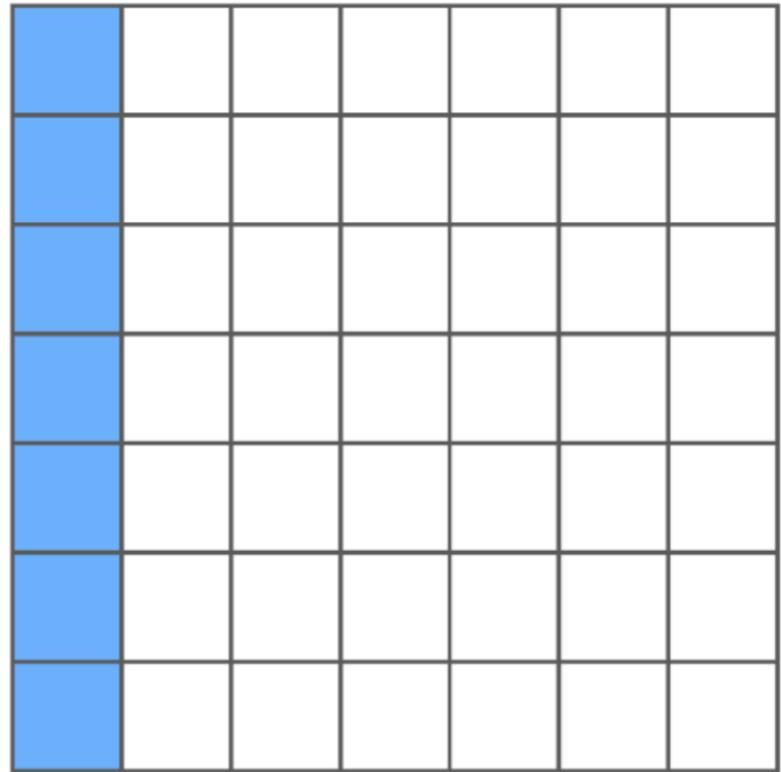


Our idea:

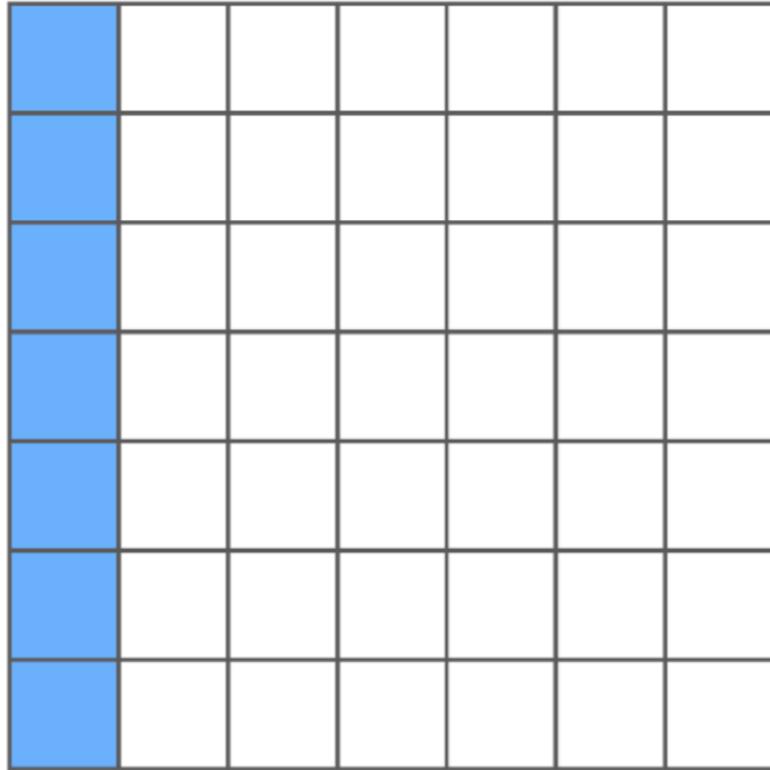
learn relations between **raw pixels**



Input

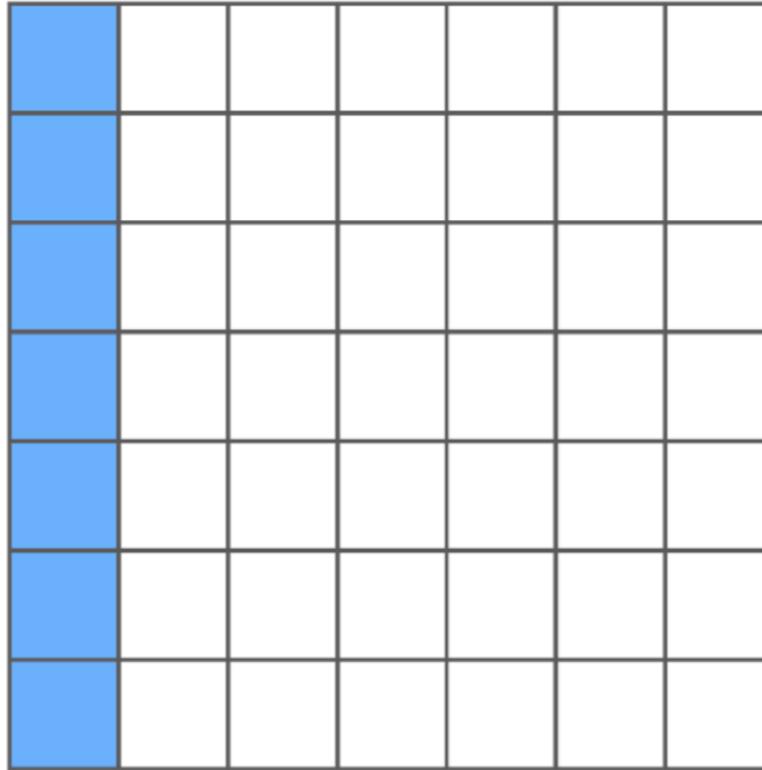


Input



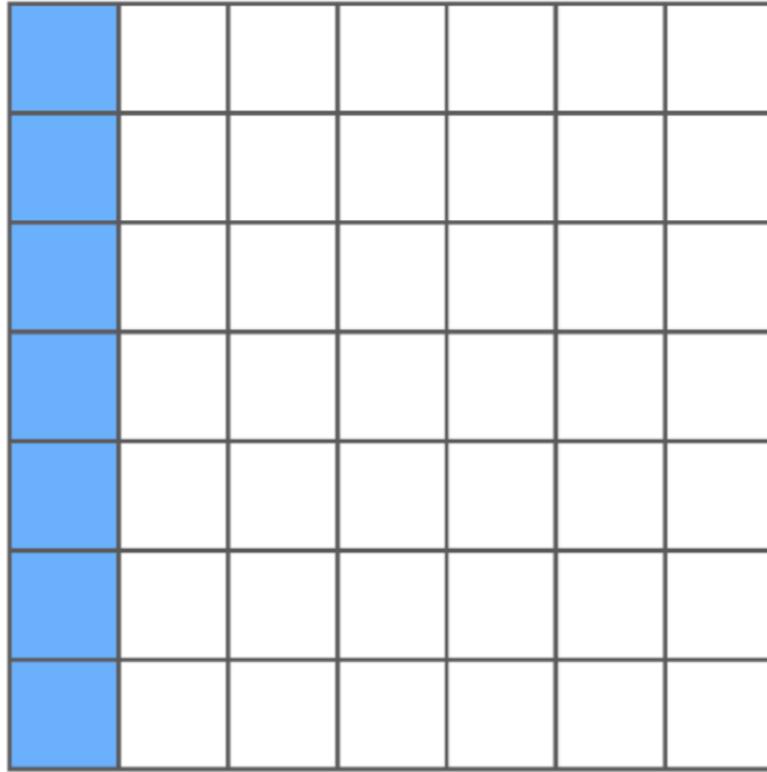
```
% bk  
in(1,1,blue).  
in(1,2,blue).  
in(1,3,blue).  
  
...
```

Input



```
% bk  
in(1,1,blue).  
in(1,2,blue).  
in(1,3,blue).  
empty(2,1).  
empty(2,2).  
empty(2,3).  
...  
...
```

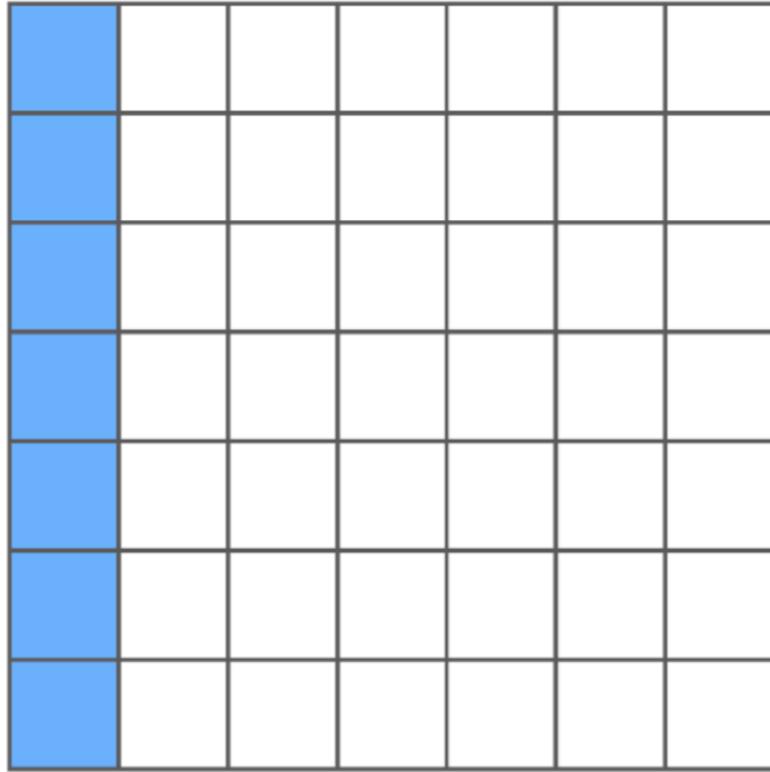
Input



```
% bk  
in(1,1,blue).  
in(1,2,blue).  
in(1,3,blue).  
empty(2,1).  
empty(2,2).  
empty(2,3).  
...  
...
```

```
% bk  
succ(1,2).  
succ(2,3).  
...
```

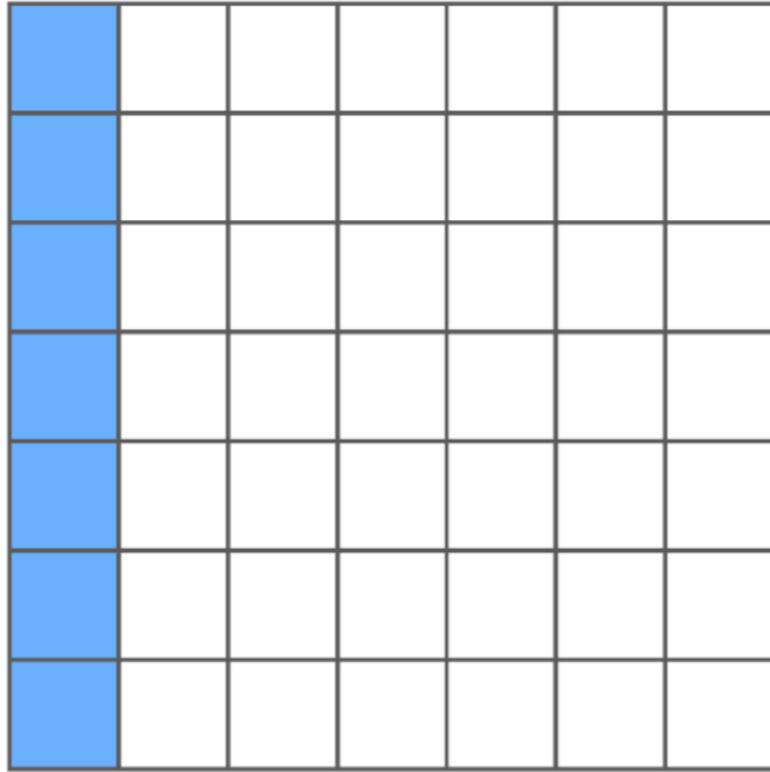
Input



```
% bk  
in(1,1,blue).  
in(1,2,blue).  
in(1,3,blue).  
empty(2,1).  
empty(2,2).  
empty(2,3).  
  
...
```

```
% bk  
succ(1,2).  
succ(2,3).  
add(1,1,2).  
add(1,2,3).  
...
```

Input

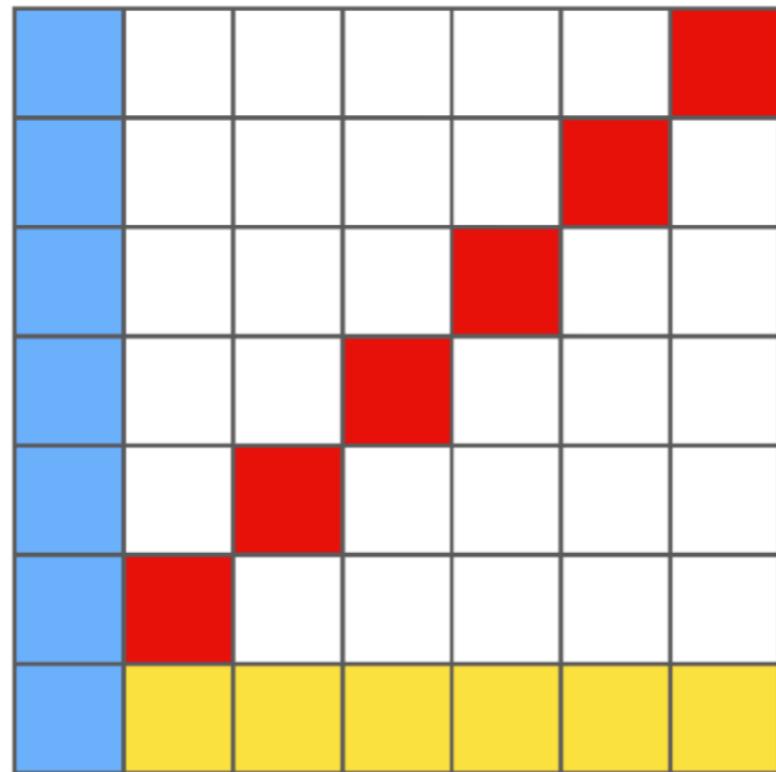


```
% bk  
in(1,1,blue).  
in(1,2,blue).  
in(1,3,blue).  
empty(2,1).  
empty(2,2).  
empty(2,3).
```

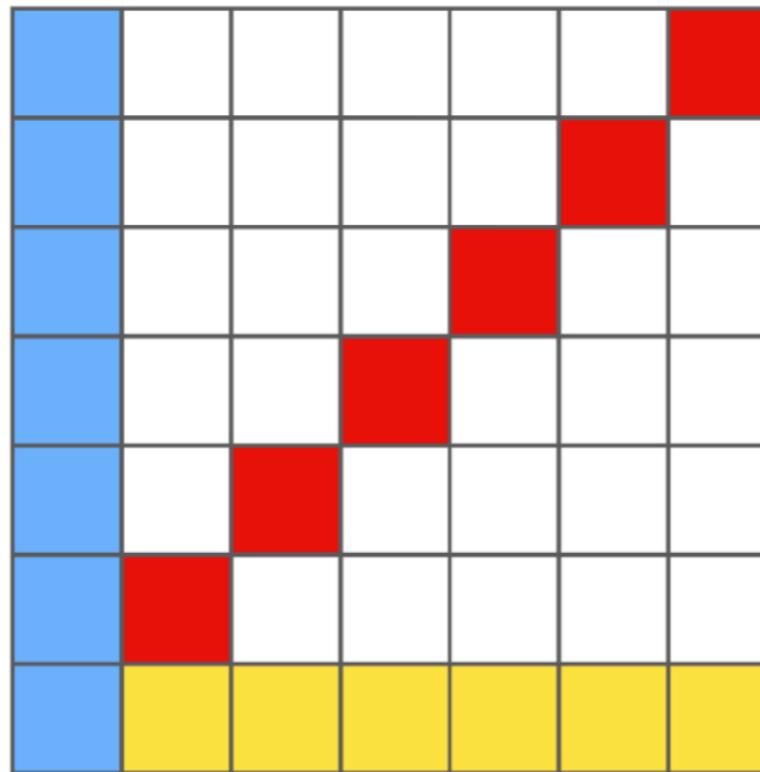
...

```
% bk  
succ(1,2).  
succ(2,3).  
add(1,1,2).  
add(1,2,3).  
height(7).  
width(7).
```

Output



Output



% examples

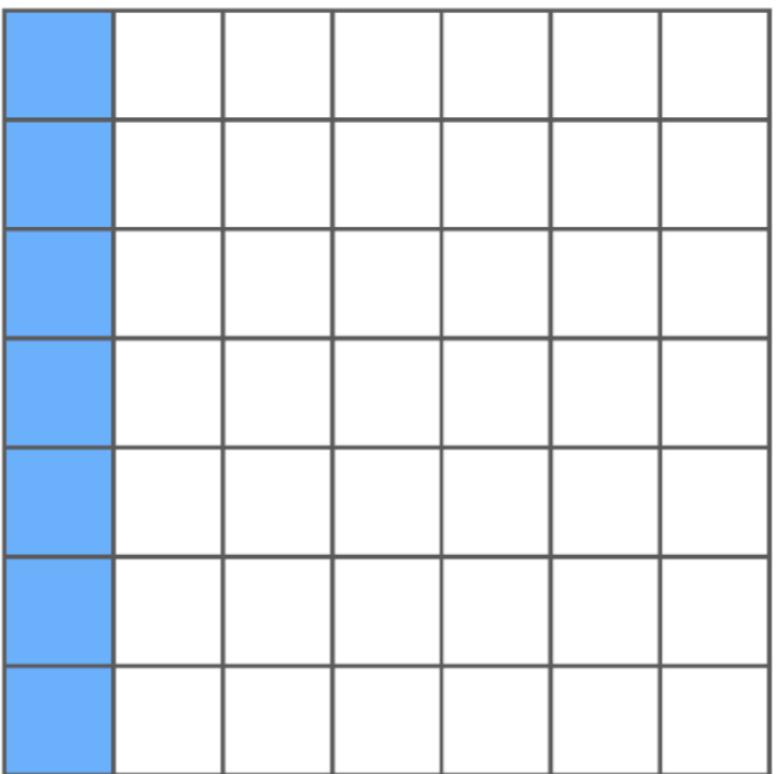
```
out(1,1,blue). out(1,7,yellow). out(2,6,red).
out(1,2,blue). out(2,7,yellow). out(3,5,red).
out(1,3,blue). out(3,7,yellow). out(4,4,red).
```

...

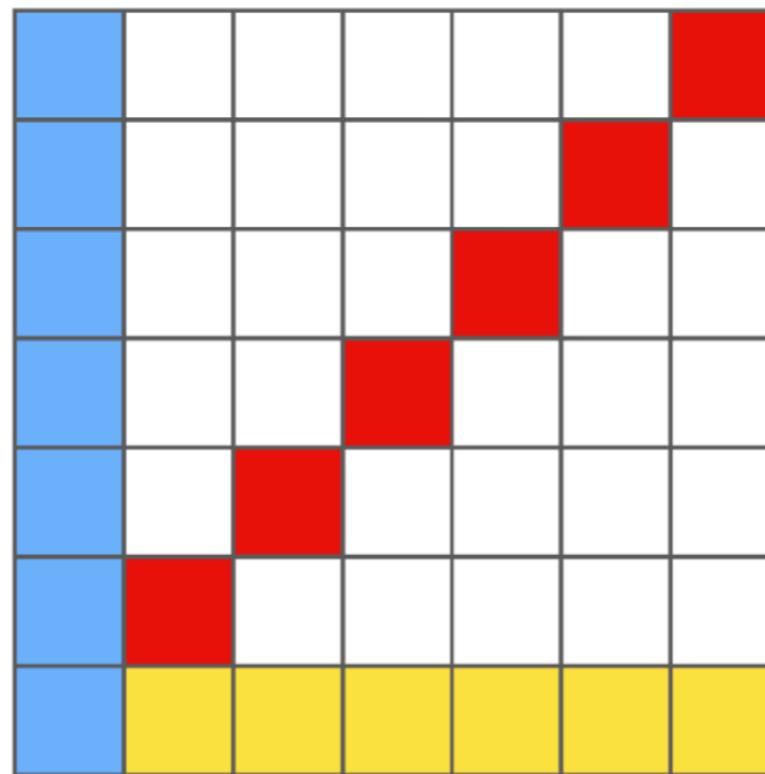
... .

... .

Input

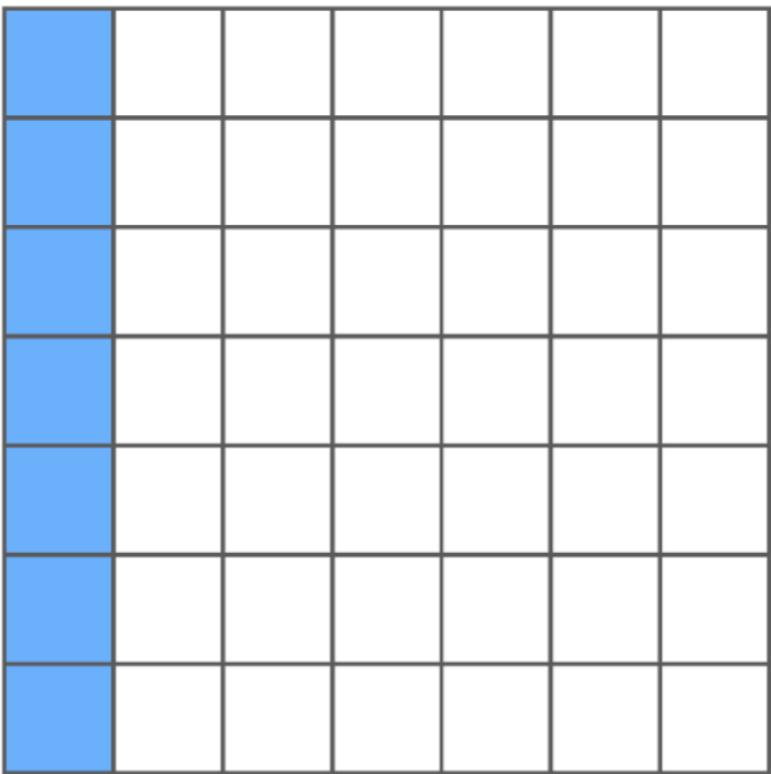


Output

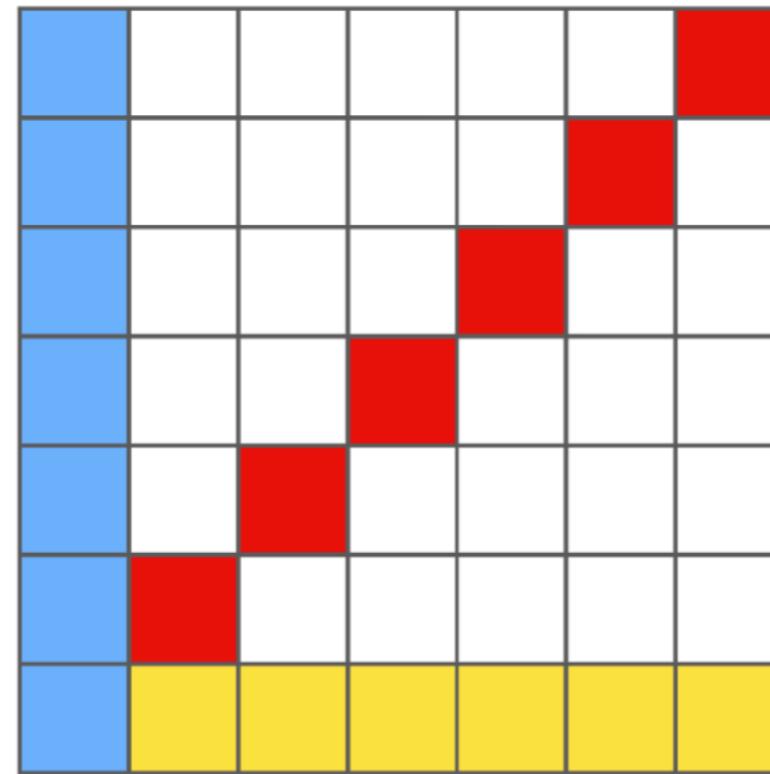


```
% hypothesis
out(X,Y,C) :- in(X,Y,C).
out(X,Y,yellow) :- empty(X,Y), height(X).
out(X,Y,red) :- empty(X,Y), height(X+Y-1).
```

Input



Output

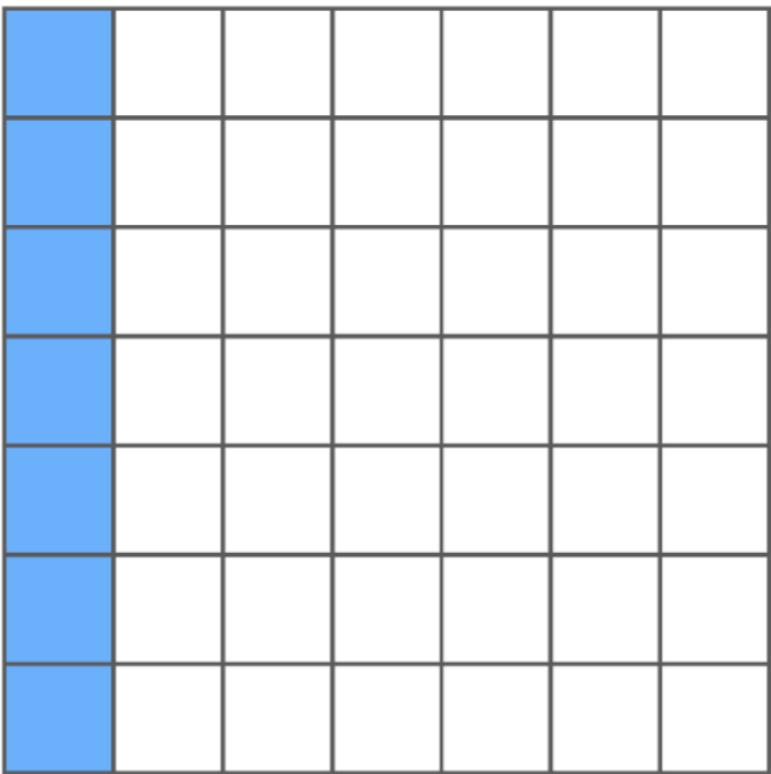


An output pixel is colour C if it is colour C in the input.

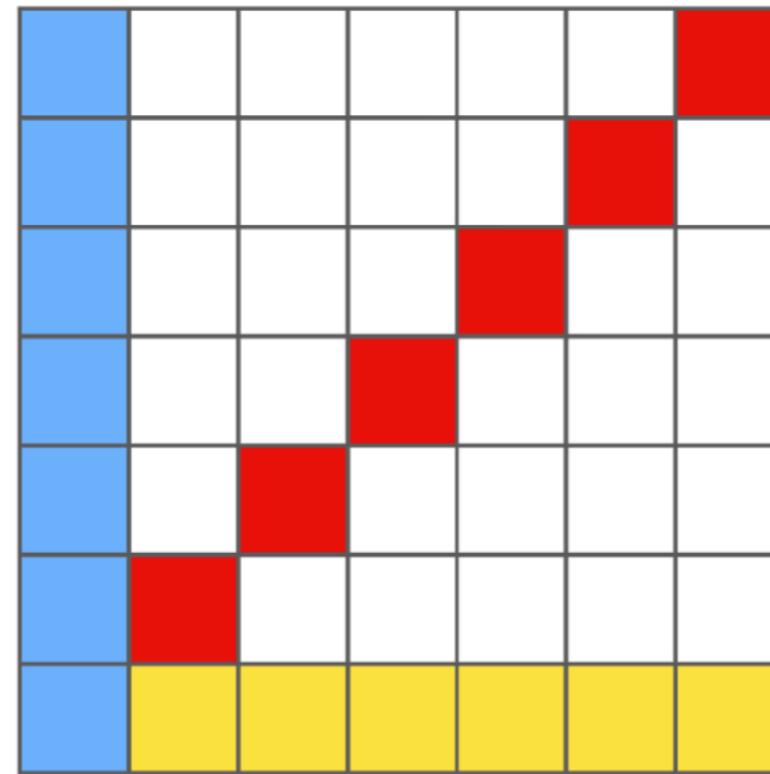
An output pixel is yellow if it is in the bottom row and empty in the input.

An output pixel is red if it is empty in the input and the sum of its coordinates X and Y equals $H + 1$, i.e. it is on the diagonal.

Input



Output

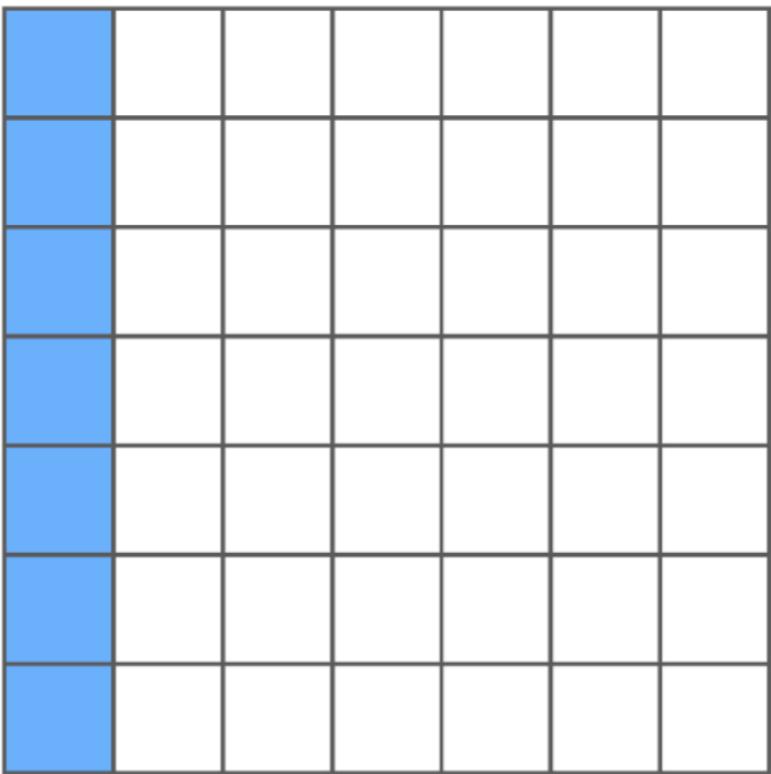


An output pixel is colour C if it is colour C in the input.

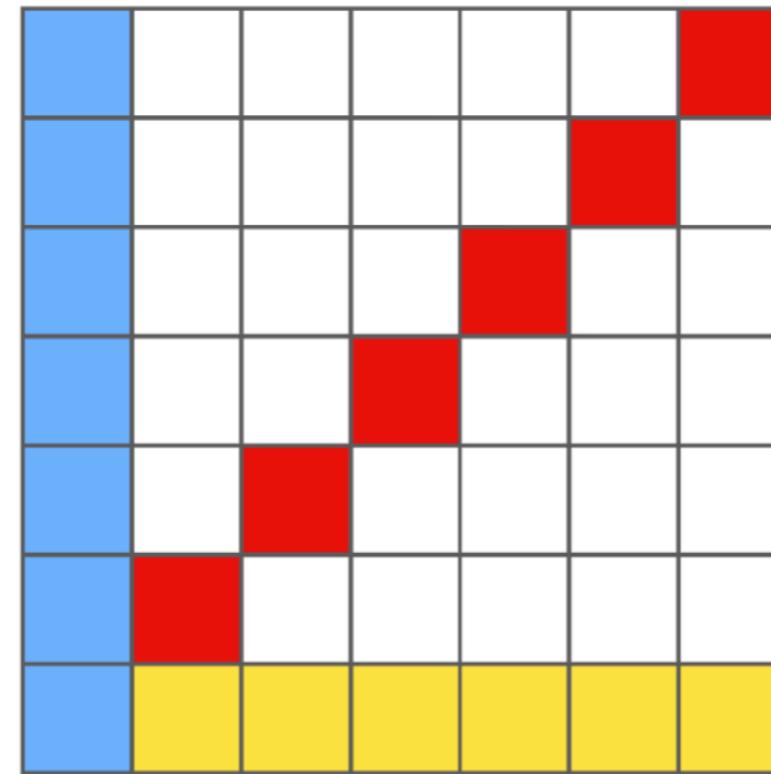
An output pixel is yellow if it is in the bottom row and empty in the input.

An output pixel is red if it is empty in the input and the sum of its coordinates X and Y equals $H + 1$, i.e. it is on the diagonal.

Input



Output



An output pixel is colour C if it is colour C in the input.

An output pixel is yellow if it is in the bottom row and empty in the input.

An output pixel is red if it is empty in the input and the sum of its coordinates X and Y equals $H + 1$, i.e. it is on the diagonal.

Program synthesis

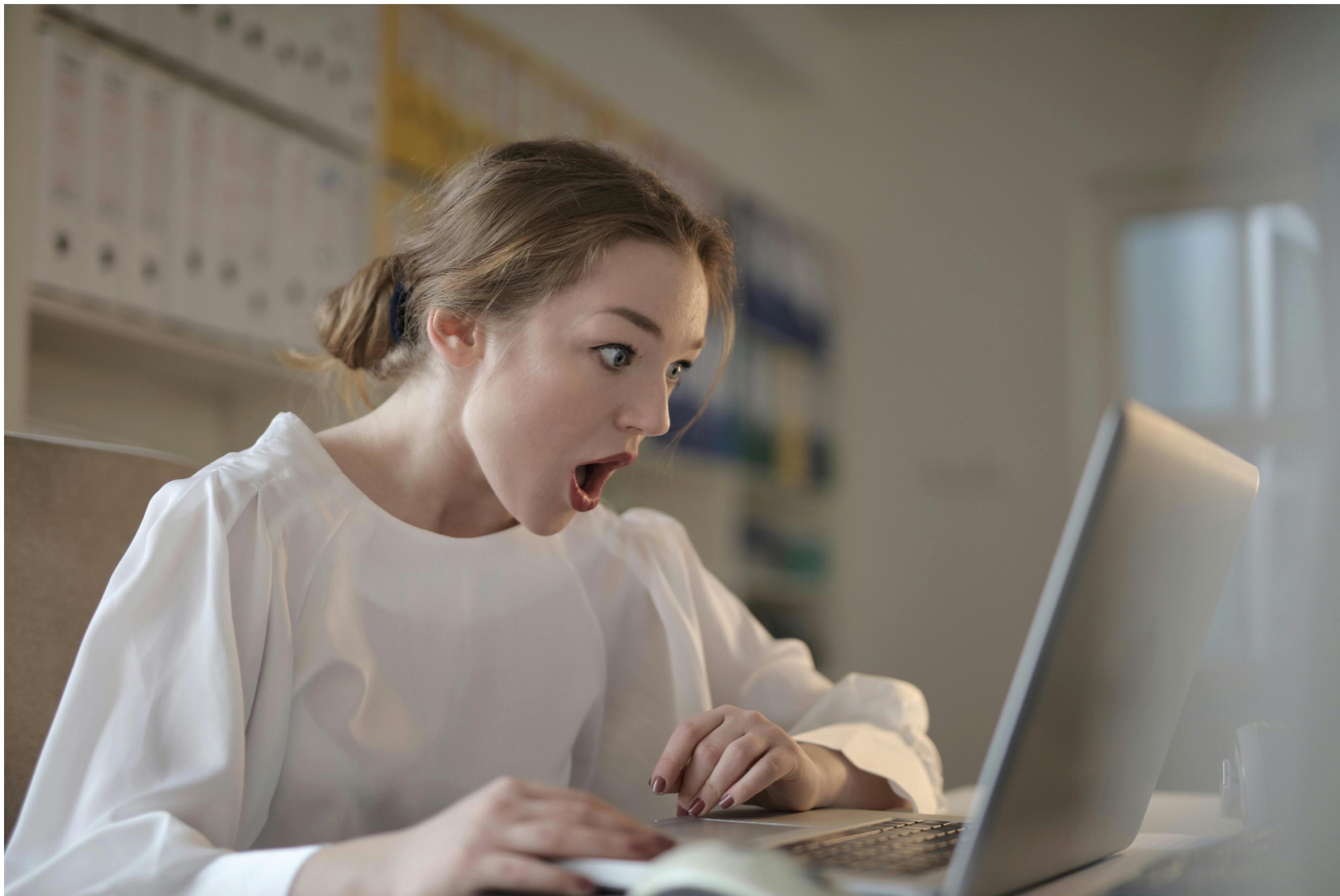
[3, 1, 9, 0, 7]	→ [1, 9, 0]
[2, 1, 3, 4, 6, 9]	→ [1, 3]
[4, 1, 2, 3, 5, 0, 7, 6, 9, 8]	→ [1, 2, 3, 5]
[1, 5, 4, 2, 8, 3, 0, 6]	→ [5]
[5, 2, 1, 0, 4, 3, 7, 6]	→ [2, 1, 0, 4, 3]

What is missing?

My (old) job



When scientists hear about ILP



When scientists try to use ILP

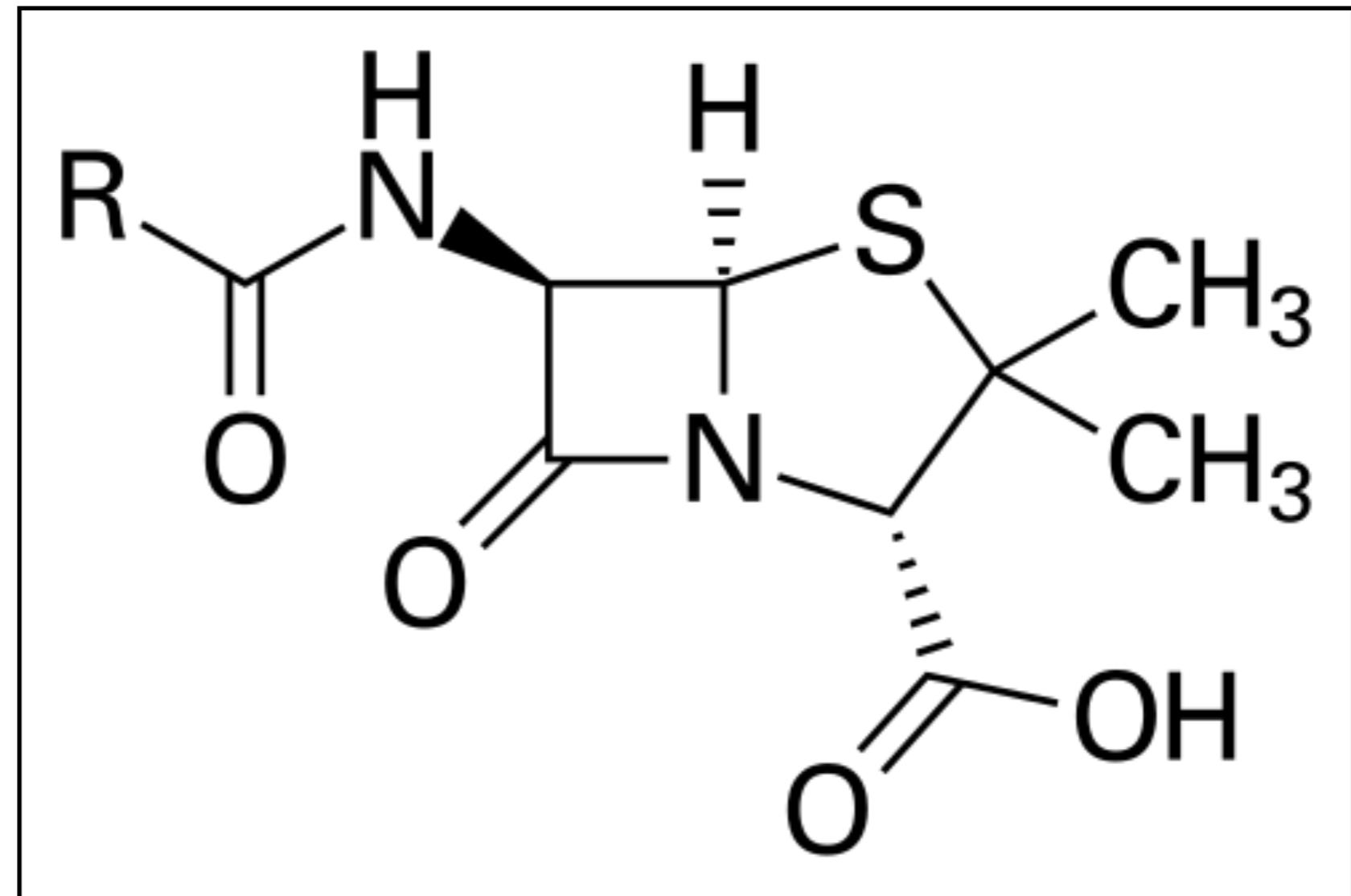


Usability

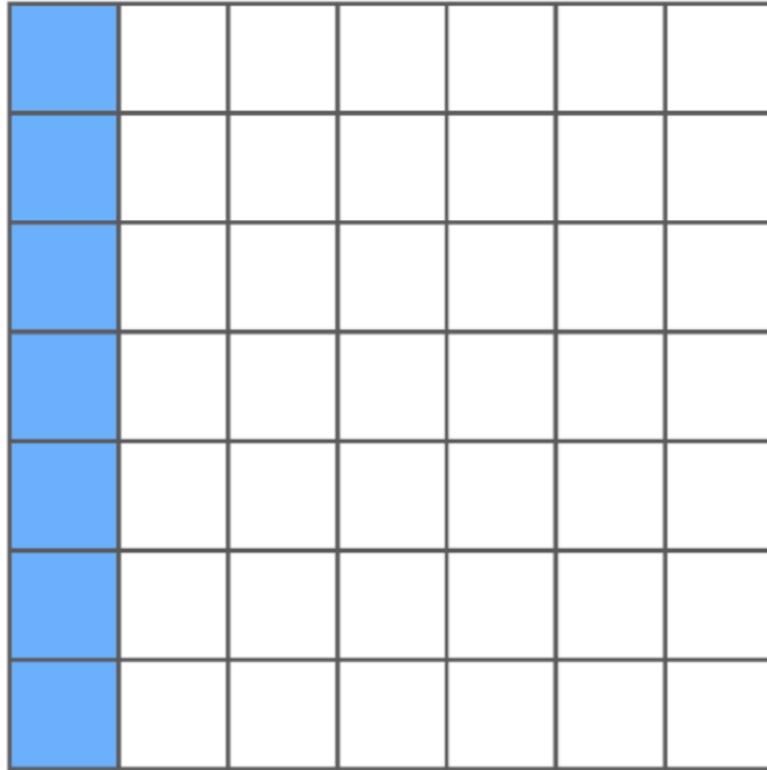
A user needs a PhD in ILP to use ILP

Usability

Prior
knowledge



Input



```
% bk  
in(1,1,blue).  
in(1,2,blue).  
in(1,3,blue).  
empty(2,1).  
empty(2,2).  
empty(2,3).  
  
...
```

% bk

~~succ(1,2)~~

~~succ(2,3)~~

~~add(1,2)~~

~~add(2,3)~~

~~height(1)~~

~~width(7)~~

Usability

Where does the prior knowledge come from?

Usability

Where does the prior knowledge come from?



ChatGPT

Much room for improvement

Better symbolic search

Better symbolic search

- >1 hours -> 17 seconds (symmetry breaking)

Better symbolic search

- >1 hours -> 17 seconds (symmetry breaking)
- >10 hours -> 2 seconds (preprocessing)

Better symbolic search

- >1 hours -> 17 seconds (symmetry breaking)
- >10 hours -> 2 seconds (preprocessing)
- >1 hours -> 12 seconds (constraint analysis)

Neural guidance

How to make bold conjectures?

Neural guidance

Can we learn which hypothesis to try next?

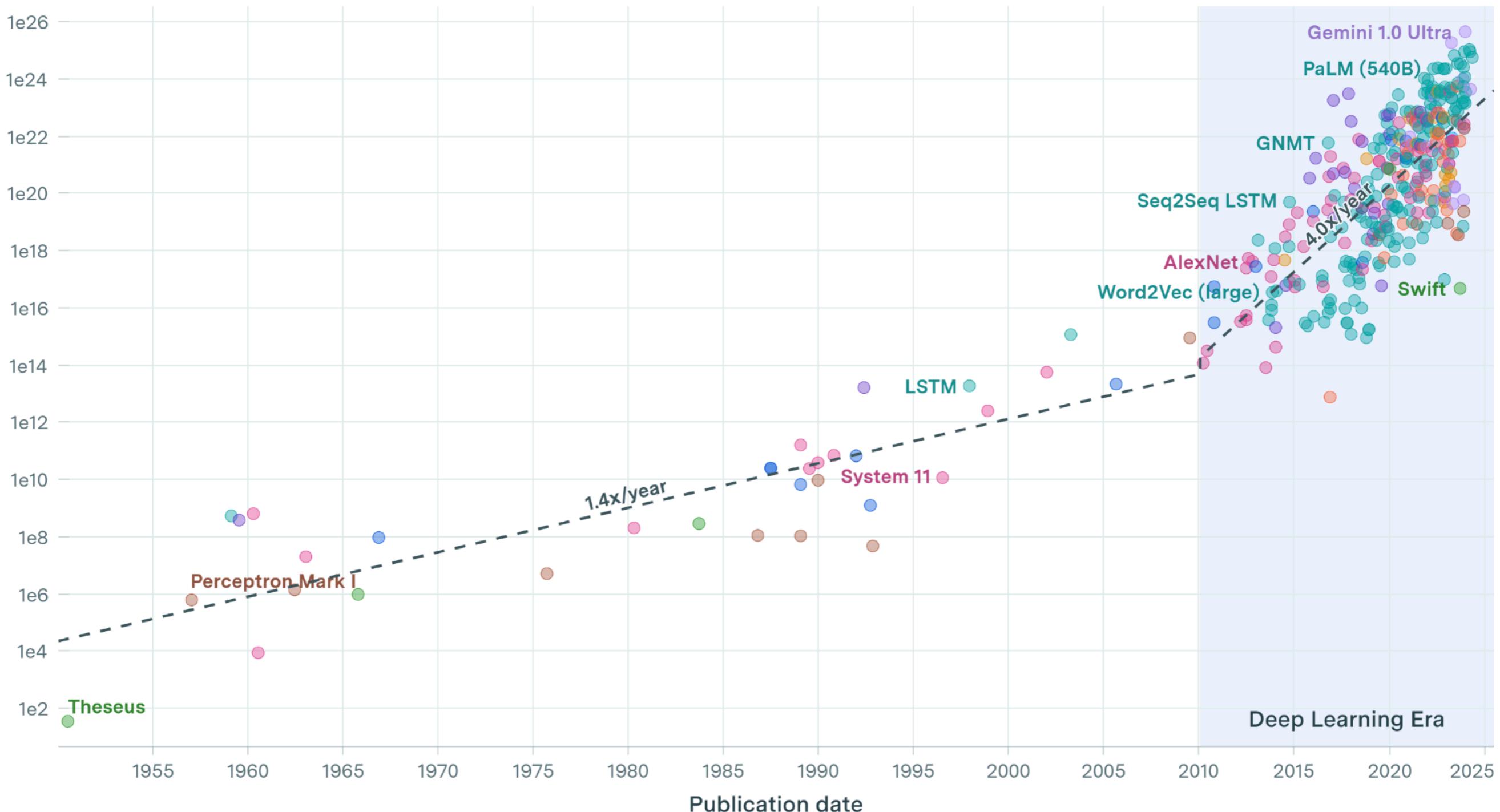
ILP now



Single CPU

Parallelisation

Training compute (FLOP)

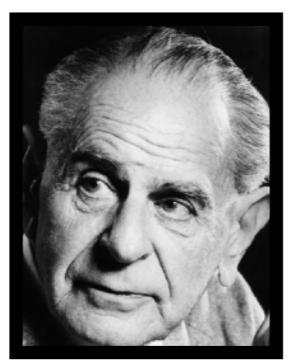


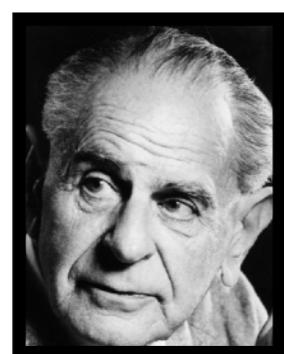
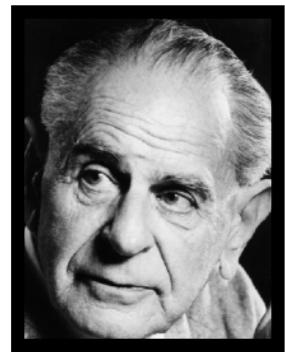
ILP future

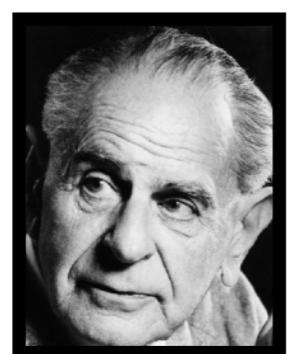
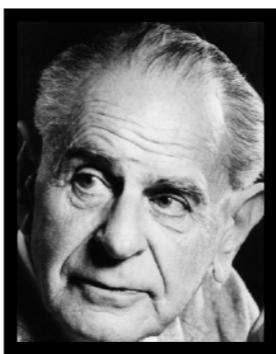
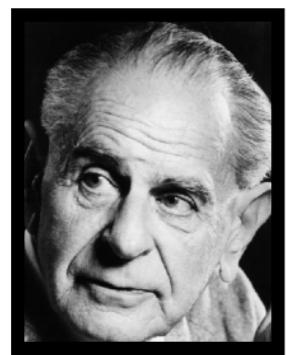


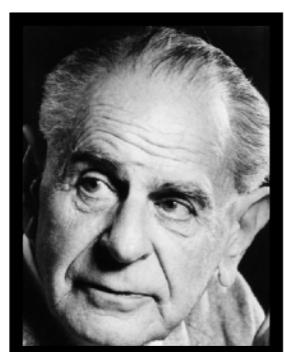
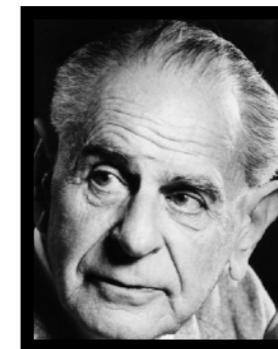
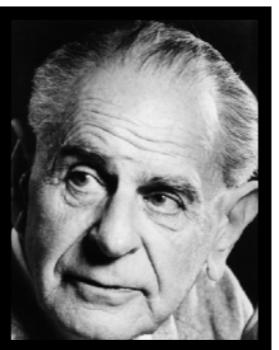
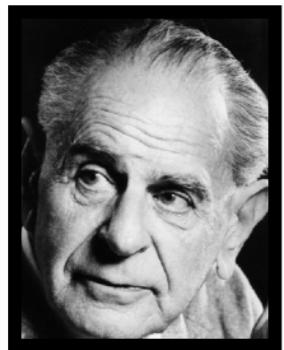
Parallelisation

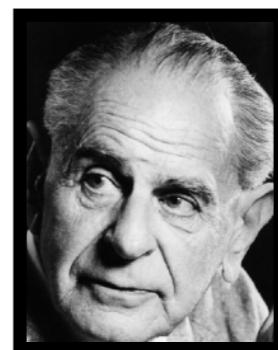
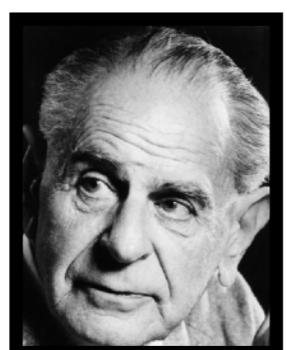
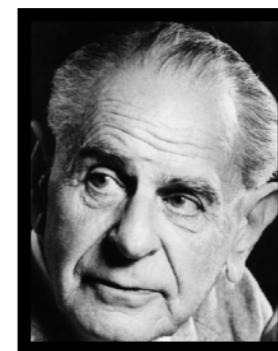
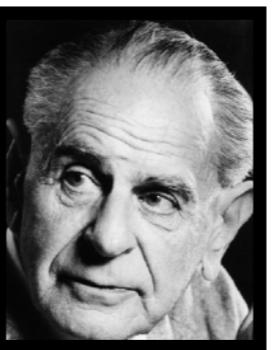
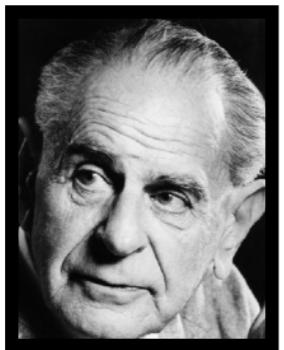
Can we achieve similar improvements as DL?

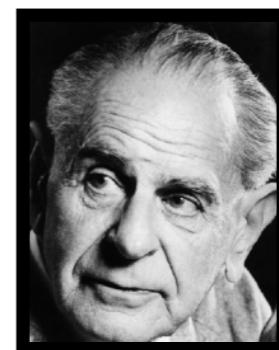
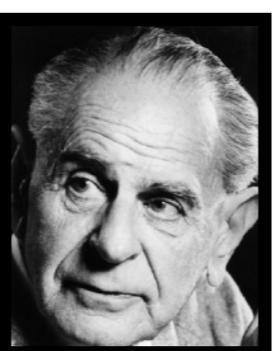
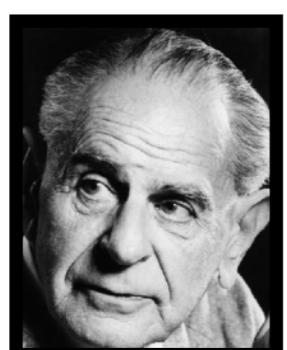
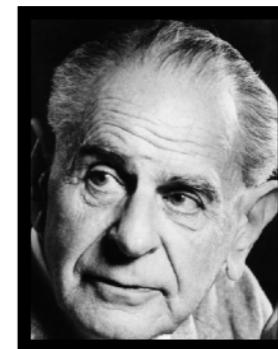
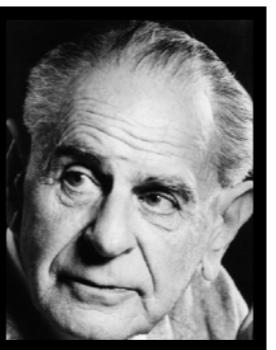
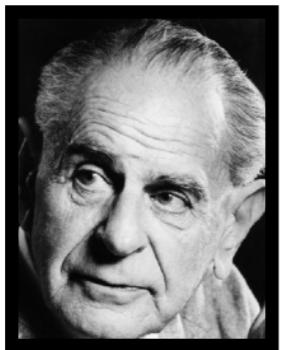


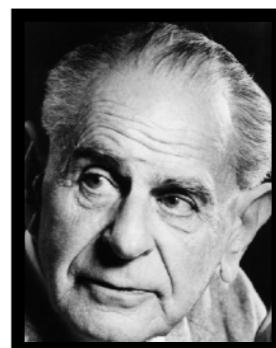
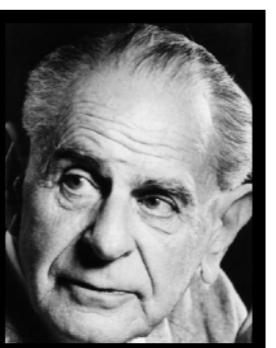
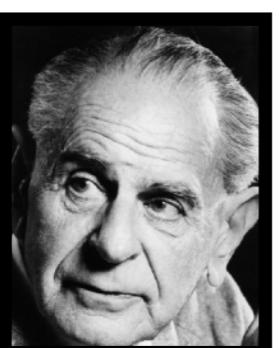
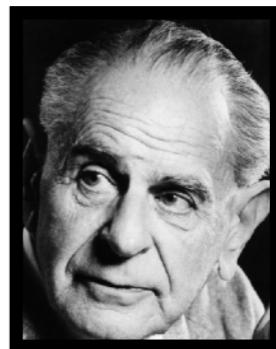
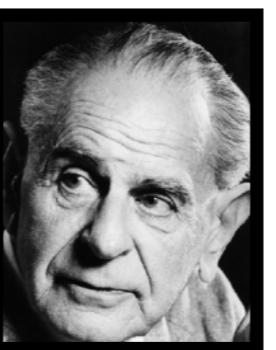
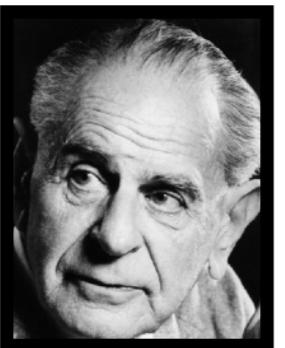


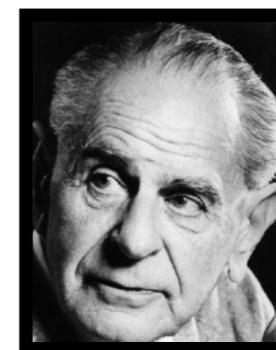
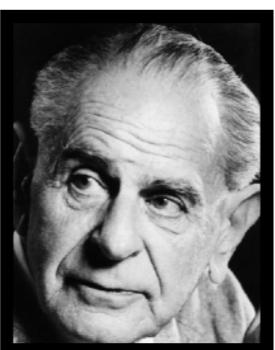
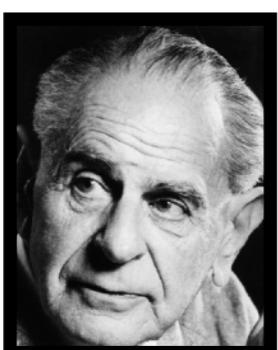
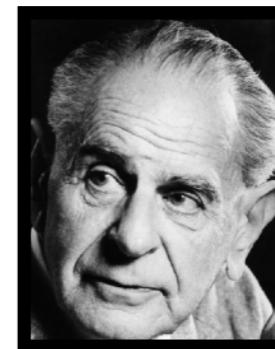
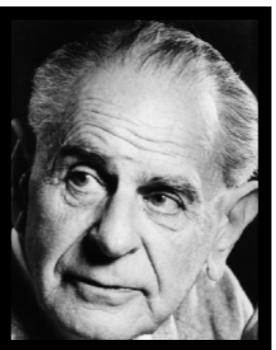
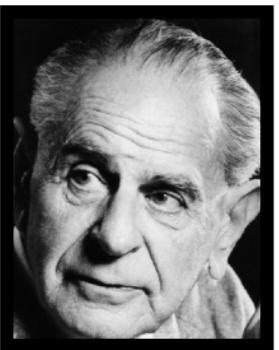


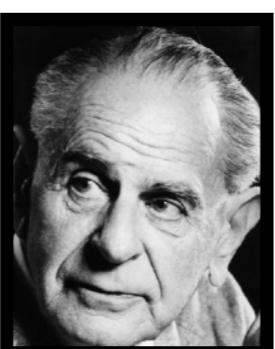
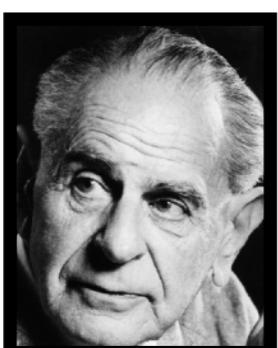
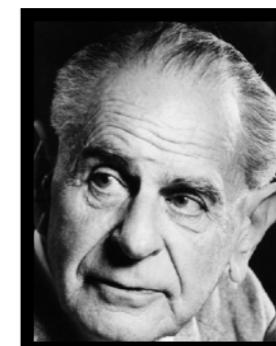
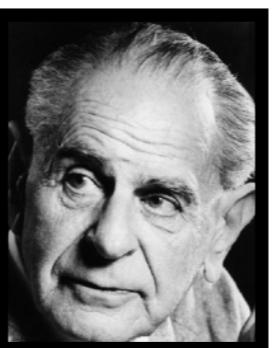
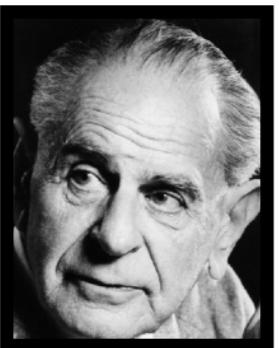


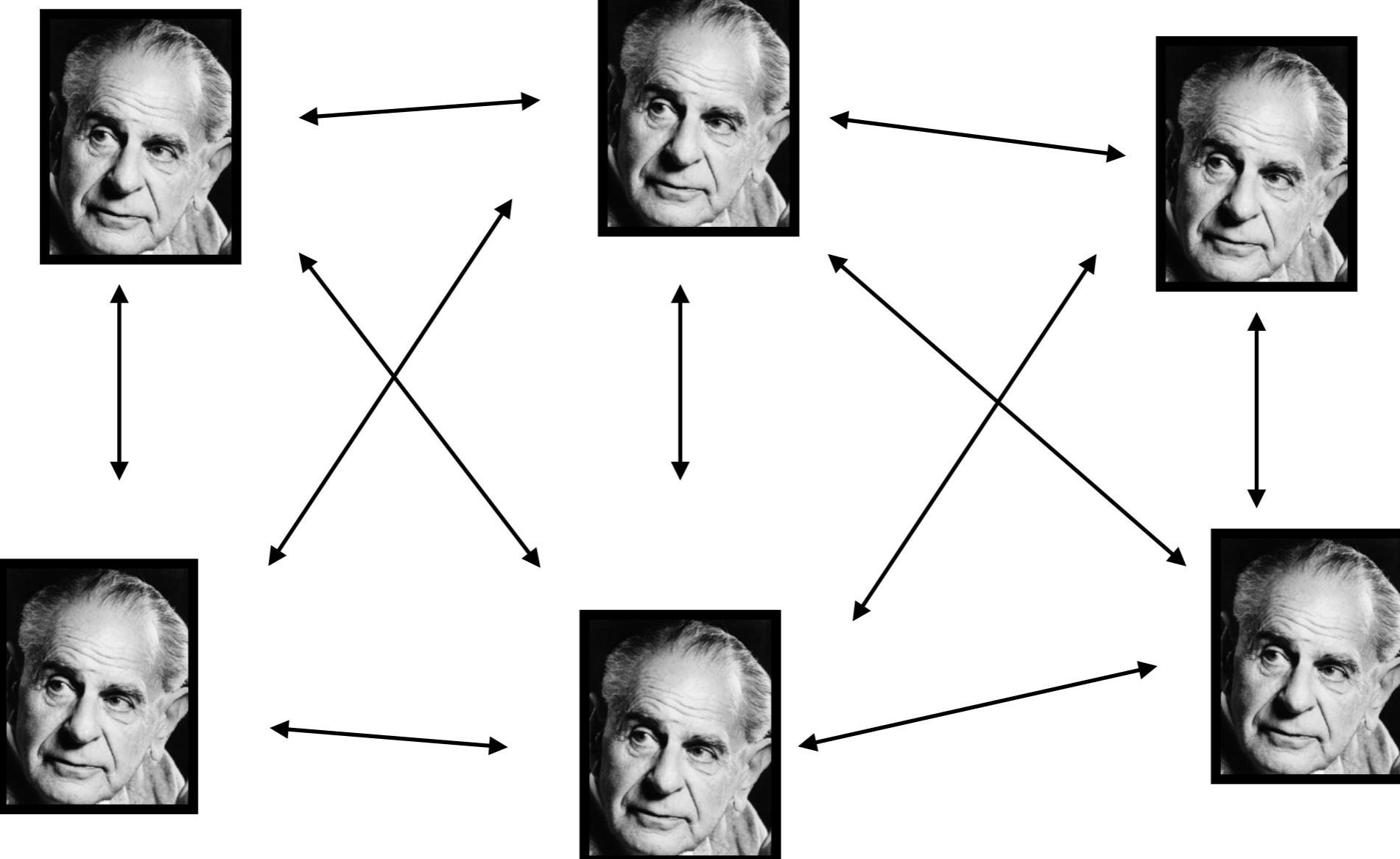












Predicate invention is really hard



```
goal(white, 100) :- line(w).
goal(white, 50) :- \+ line(r), \+ line(w), \+ open.
goal(white, 0) :- line(r).
goal(red, 100) :- line(r).
goal(red, 50) :- \+ line(r), \+ line(w), \+ open.
goal(red, 0) :- line(w).
```

```
% Line is row, column, or diagonal  
line(X) :- row(X).  
line(X) :- column(X).  
line(X) :- diag(X).
```

```
% Horizontal
row(Z) :-  
    true(cell(X1, Y, Z)),  
    Z \= b,  
    Z \= dirt,  
    succ(X1, X2),  
    true(cell(X2, Y, Z)),  
    succ(X2, X3),  
    true(cell(X3, Y, Z)),  
    succ(X3, X4),  
    true(cell(X4, Y, Z)).
```

```
% Vertical
column(Z) :-  
    true(cell(X, Y1, Z)),  
    Z \= b,  
    succ(Y1, Y2),  
    true(cell(X, Y2, Z)),  
    succ(Y2, Y3),  
    true(cell(X, Y3, Z)),  
    succ(Y3, Y4),  
    true(cell(X, Y4, Z)).
```

I do not have any ideas

Summary

- Popper can learn complex rules from noisy data

Summary

- Popper can learn complex rules from noisy data
- Much scope to make many fundamental advances

Summary

- Popper can learn complex rules from noisy data
- Much scope to make many fundamental advances

<https://github.com/logic-and-learning-lab/Popper>

(Google Popper + ILP)

Bias as constraints

$p(A)$

$p(B)$

$q(A)$

$q(B)$

$r(A)$

$r(B)$

$\text{odd}(A)$

$\text{odd}(A)$

$\text{even}(A)$

$\text{even}(B)$

$\text{succ}(A, B)$

$\text{succ}(B, A)$

Bias as constraints

p_a

p_b

q_a

q_b

r_a

r_b

odd_a

odd_a

even_a

even_b

sucab

sucba

Bias as constraints

p_a

p_b

q_a

q_b

r_a

r_b

odd_a

odd_a

even_a

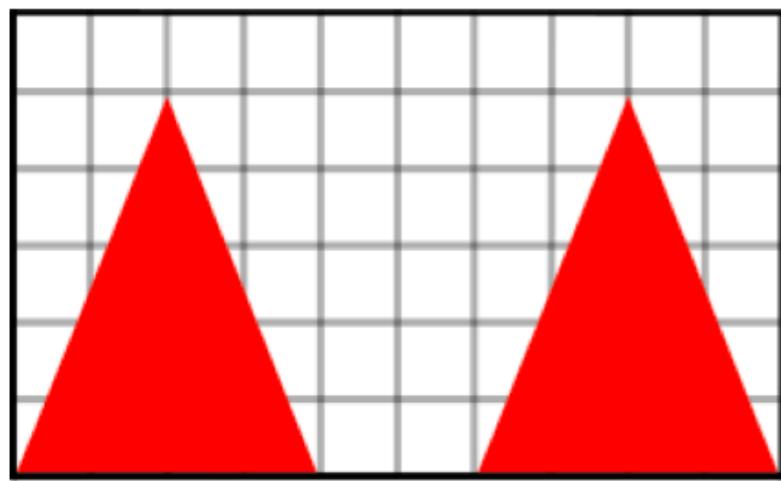
even_b

sucab

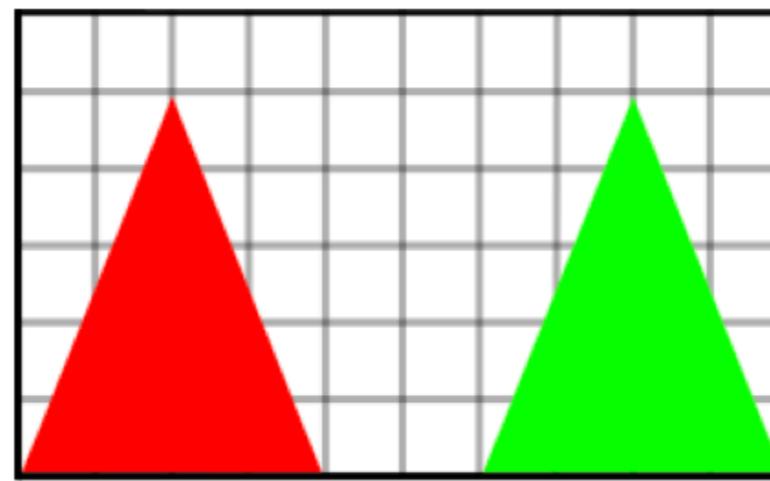
sucba

Predicate invention + negation

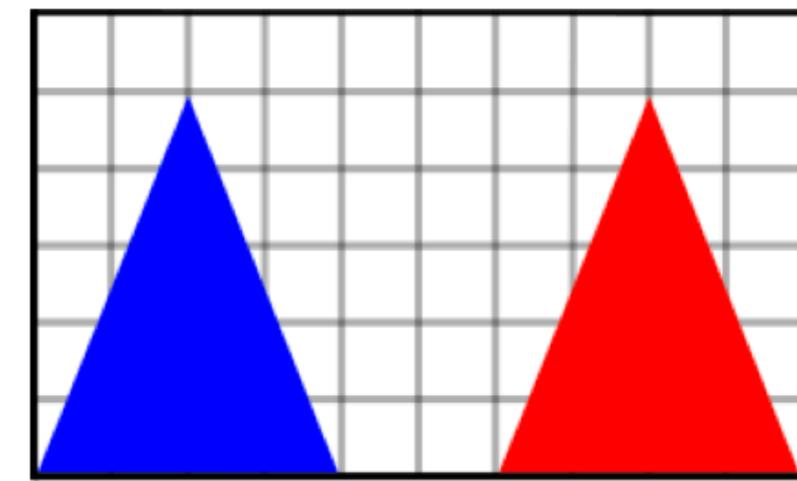
Predicate invention + negation



E^+

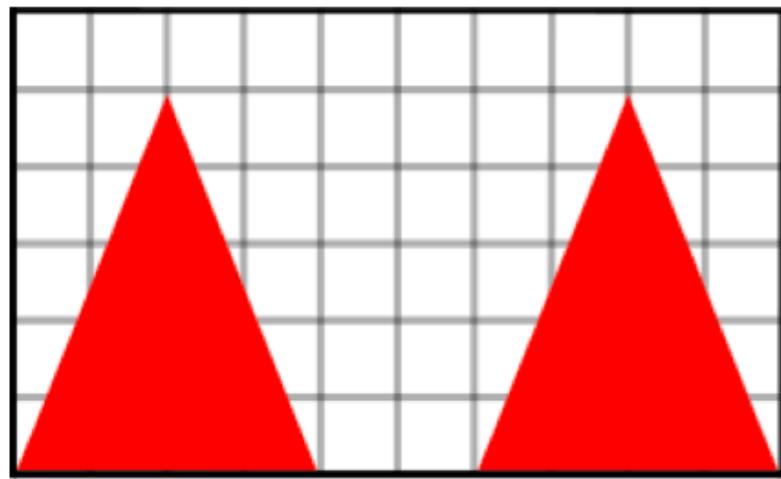


E^-

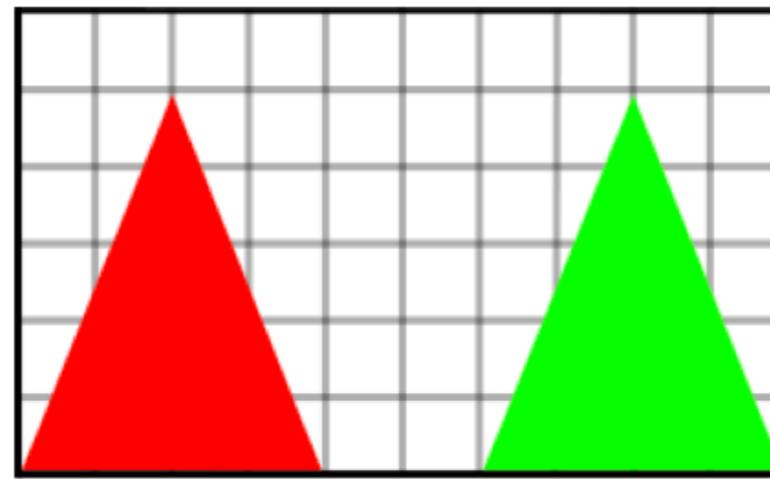


E^-

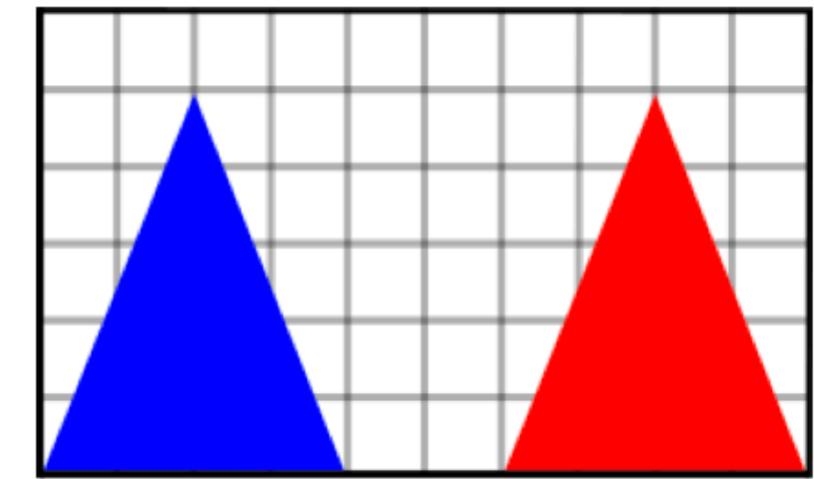
Predicate invention + negation



E^+



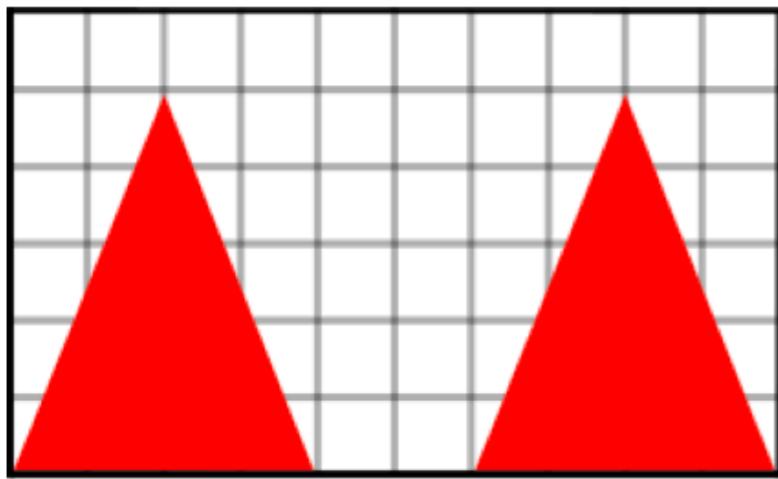
E^-



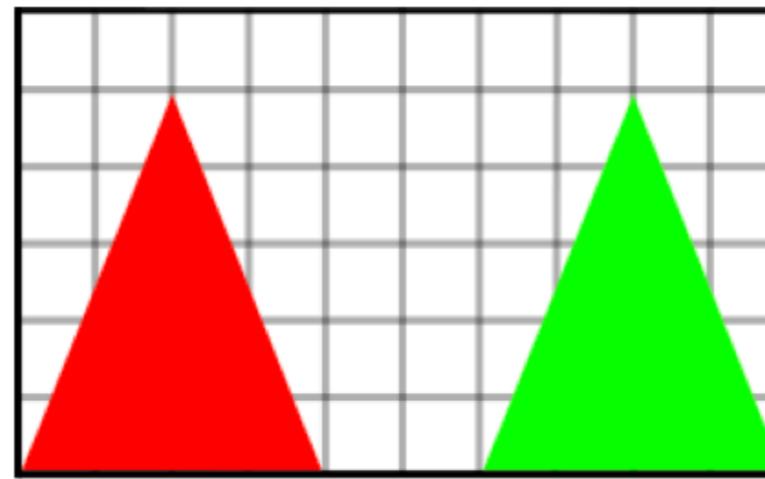
E^-

“there are two red cones”

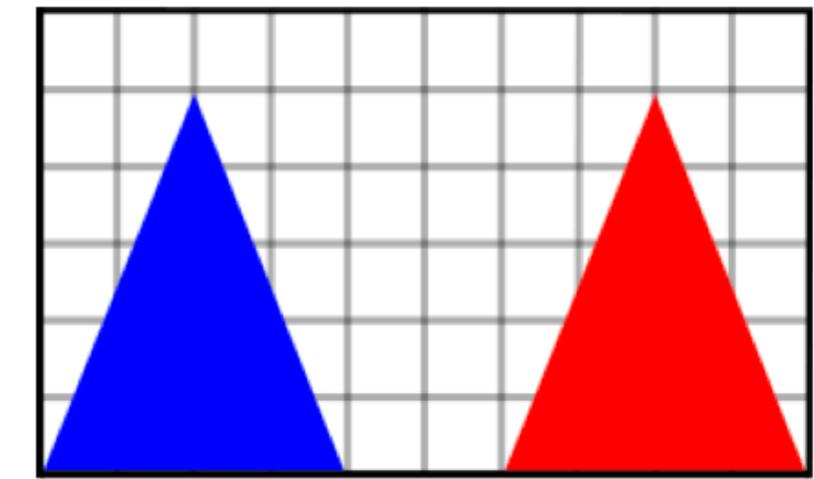
Predicate invention + negation



E^+



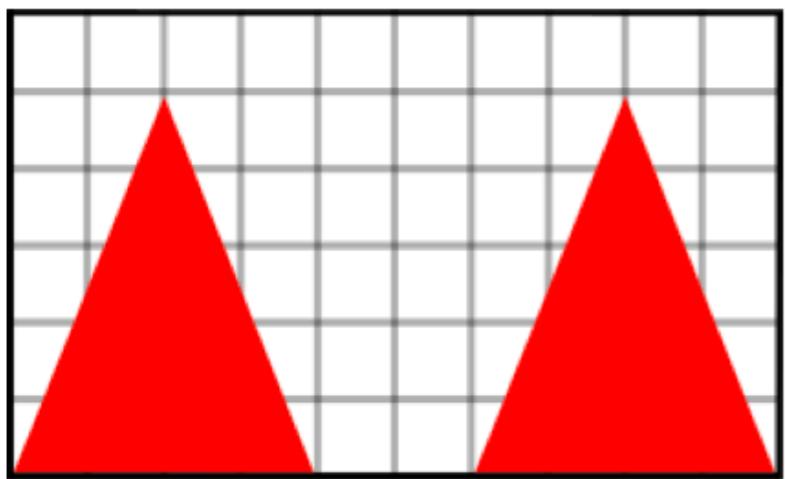
E^-



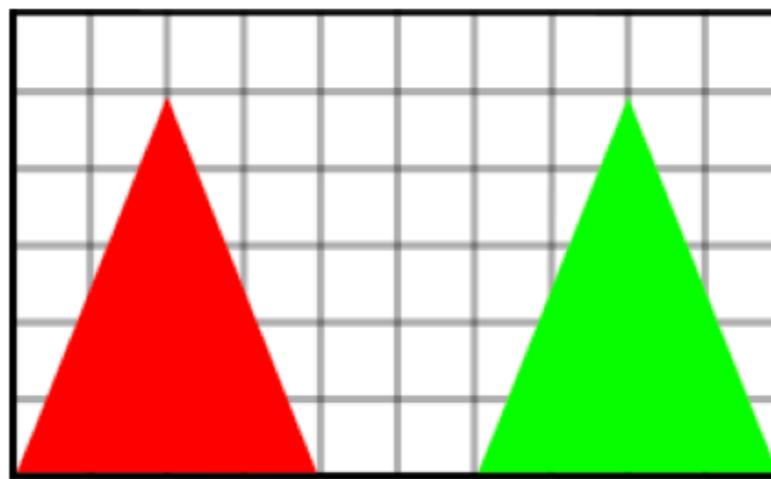
E^-

```
f(S):- cone(S,A), red(A), cone(S,B), red(B), all_diff(A,B)
```

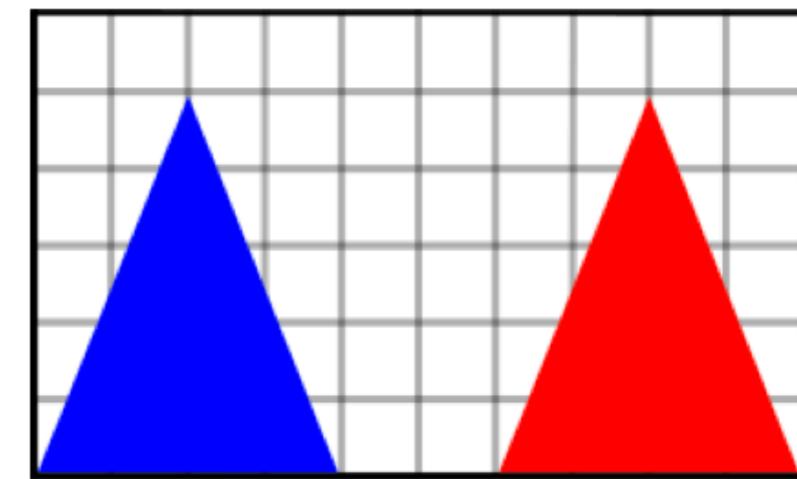
Predicate invention + negation



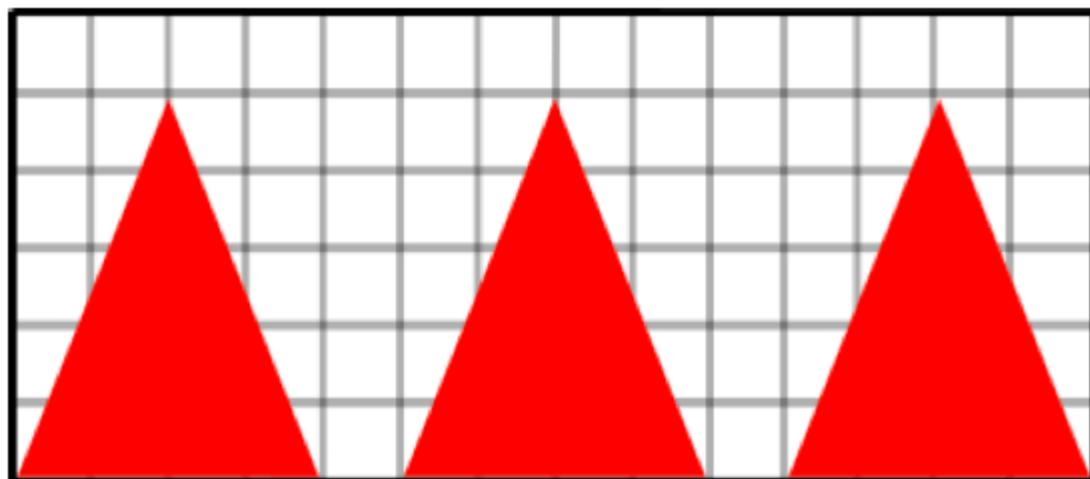
E^+



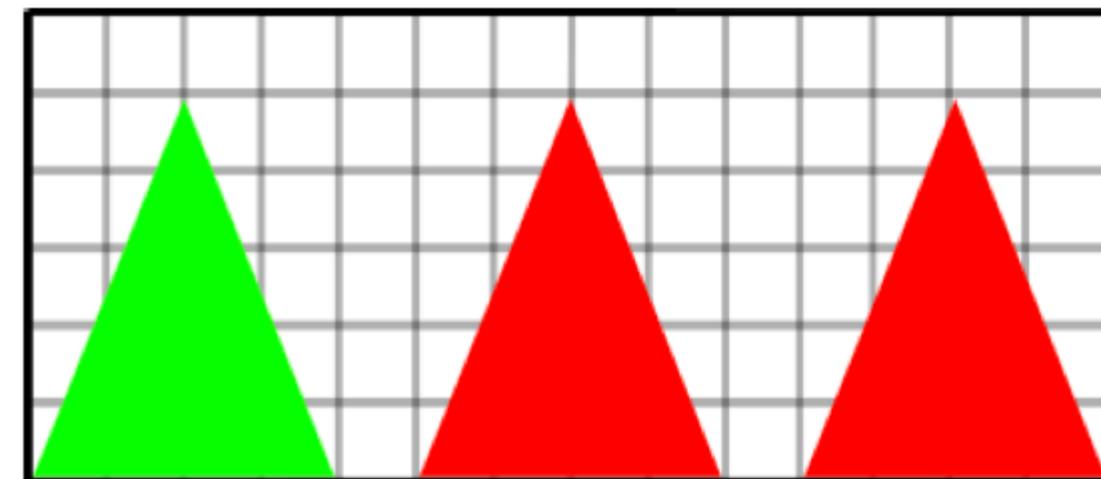
E^-



E^-



E^+



E^-

Predicate invention + negation

“there are exactly two cones and both are red”

or

“there are exactly three cones and all three are red”

Predicate invention + negation

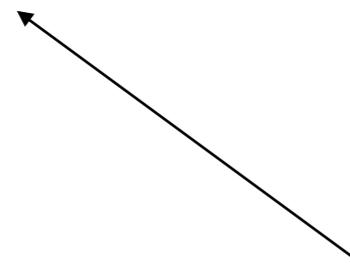
very messy program here

Predicate invention + negation

```
f(S):- not inv1(S)
inv1(S):- cone(S,P), not red(P)
```

Predicate invention + negation

```
f(S):- not inv1(S)  
inv1(S):- cone(S,P), not red(P)
```



there is a cone that is not red

Predicate invention + negation

it is not true that there is a cone that is not red

f(S):- not inv1(S)

inv1(S):- cone(S,P), not red(P)

there is a cone that is not red

Predicate invention + negation

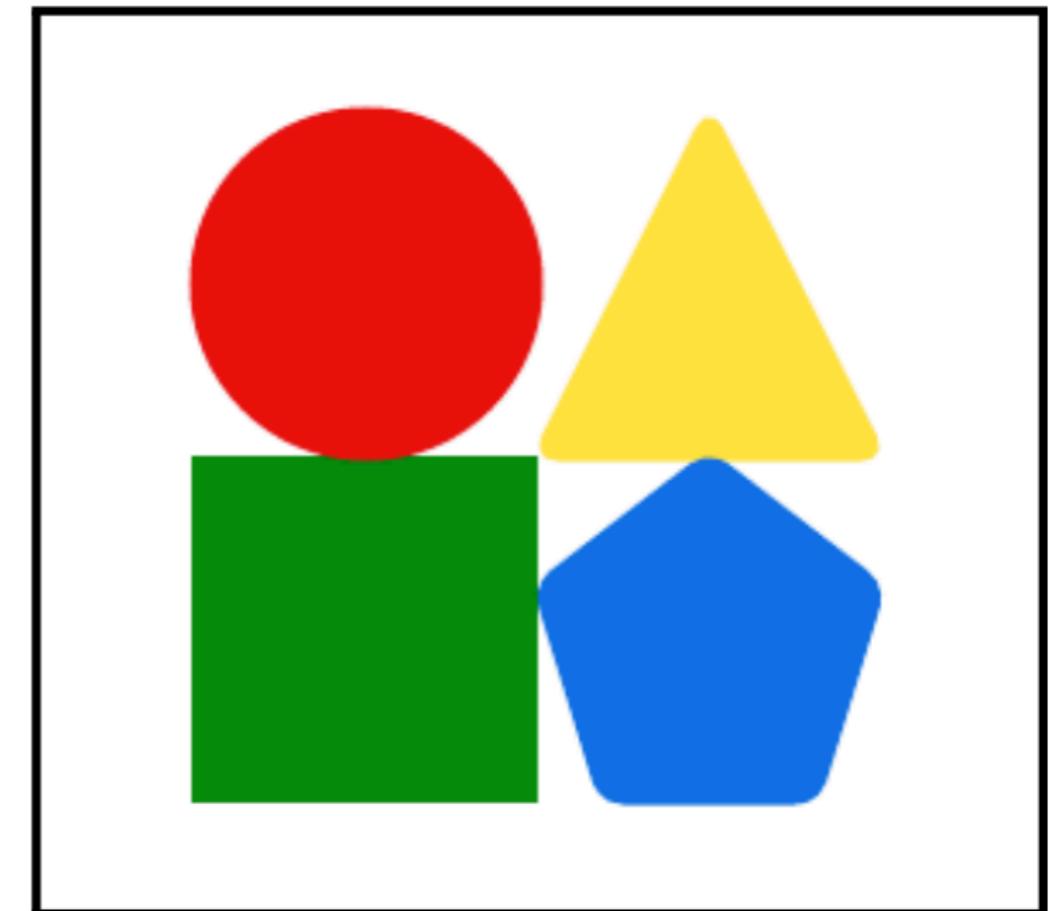
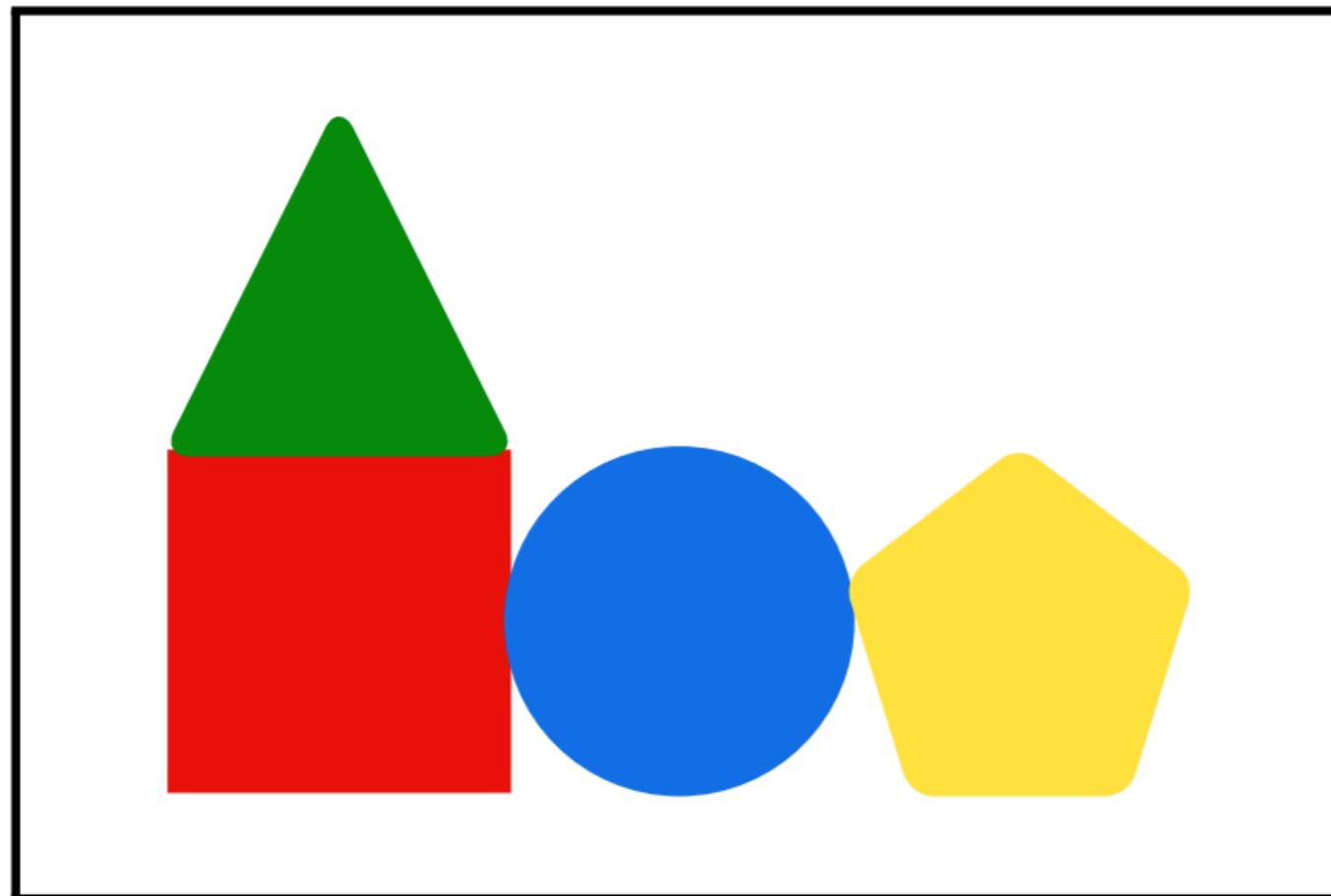
```
f(S):- not inv1(S)  
inv1(S):- cone(S,P), not red(P)
```

all the cones are red

Key idea

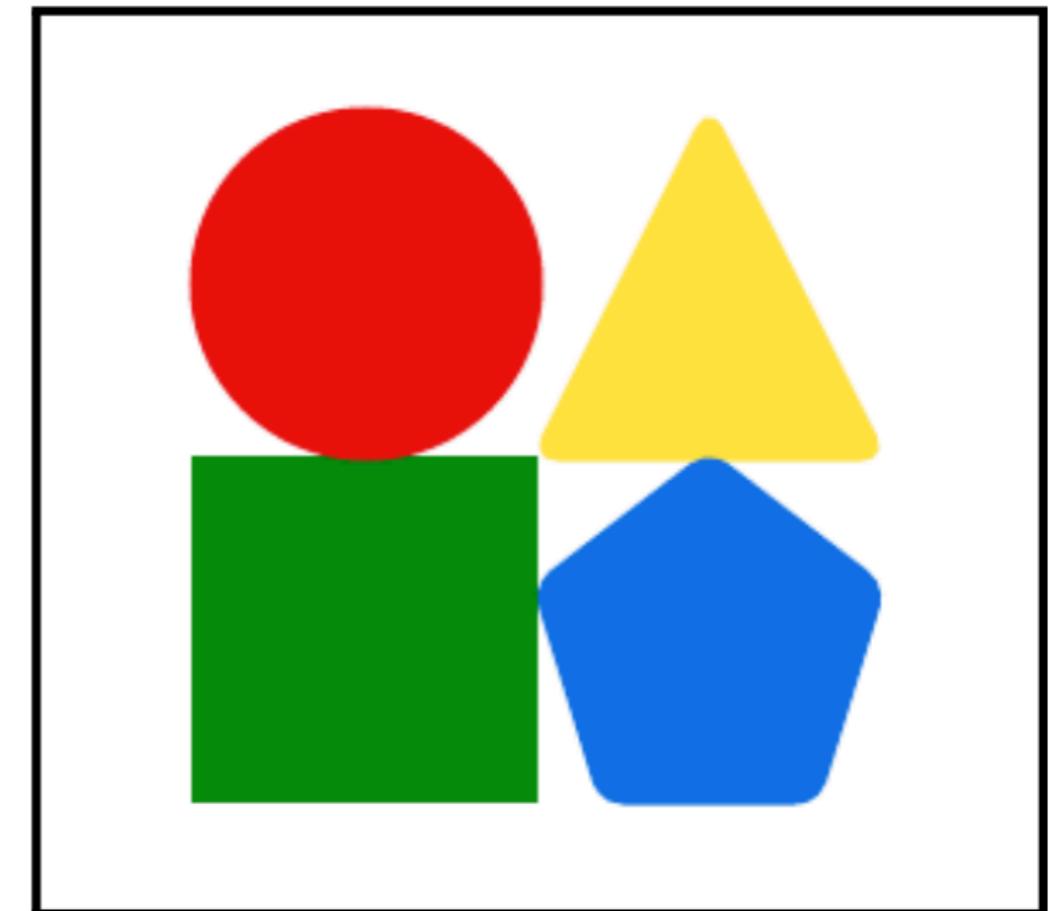
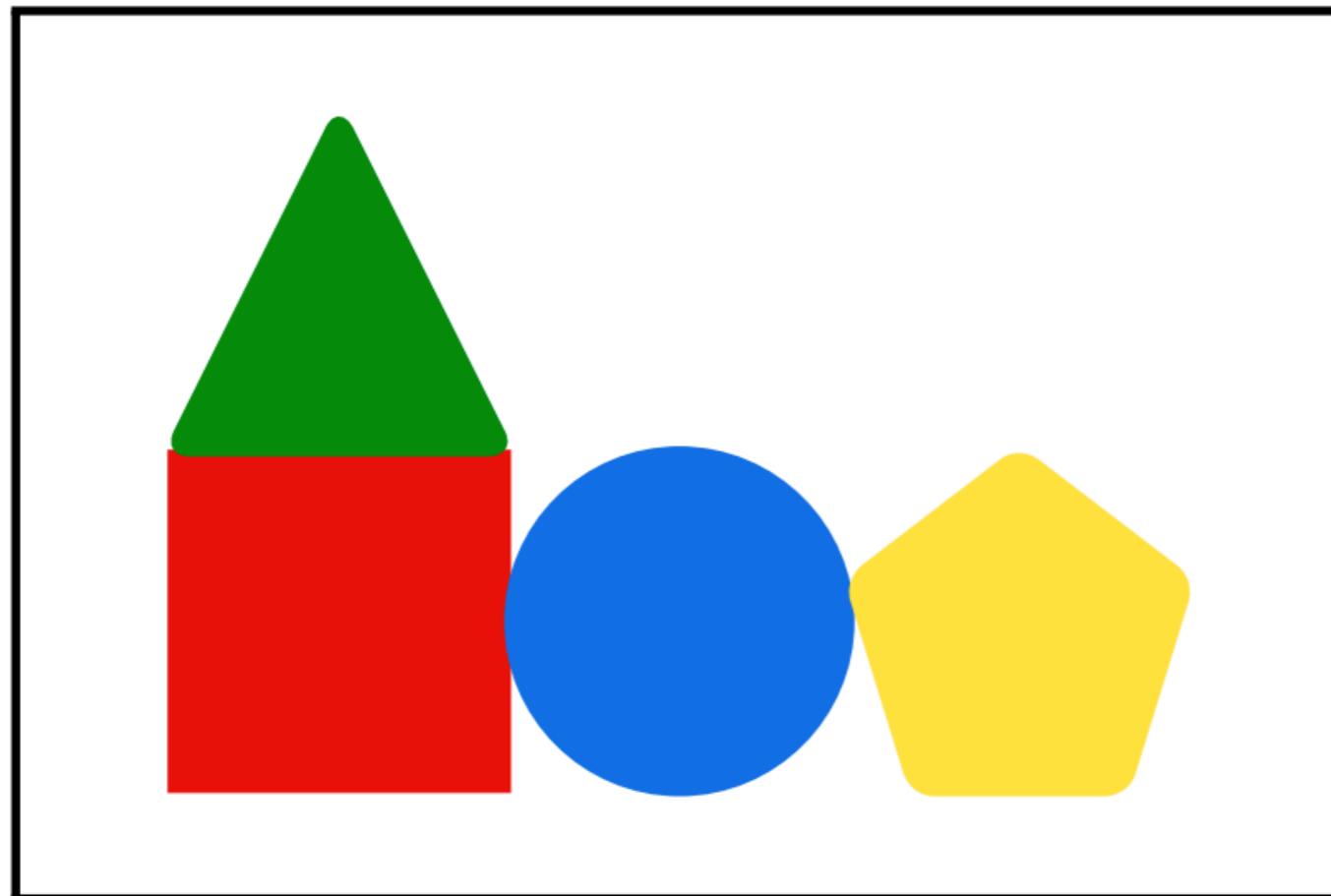
Join non-splittable rules

Joiner (Popper V4)



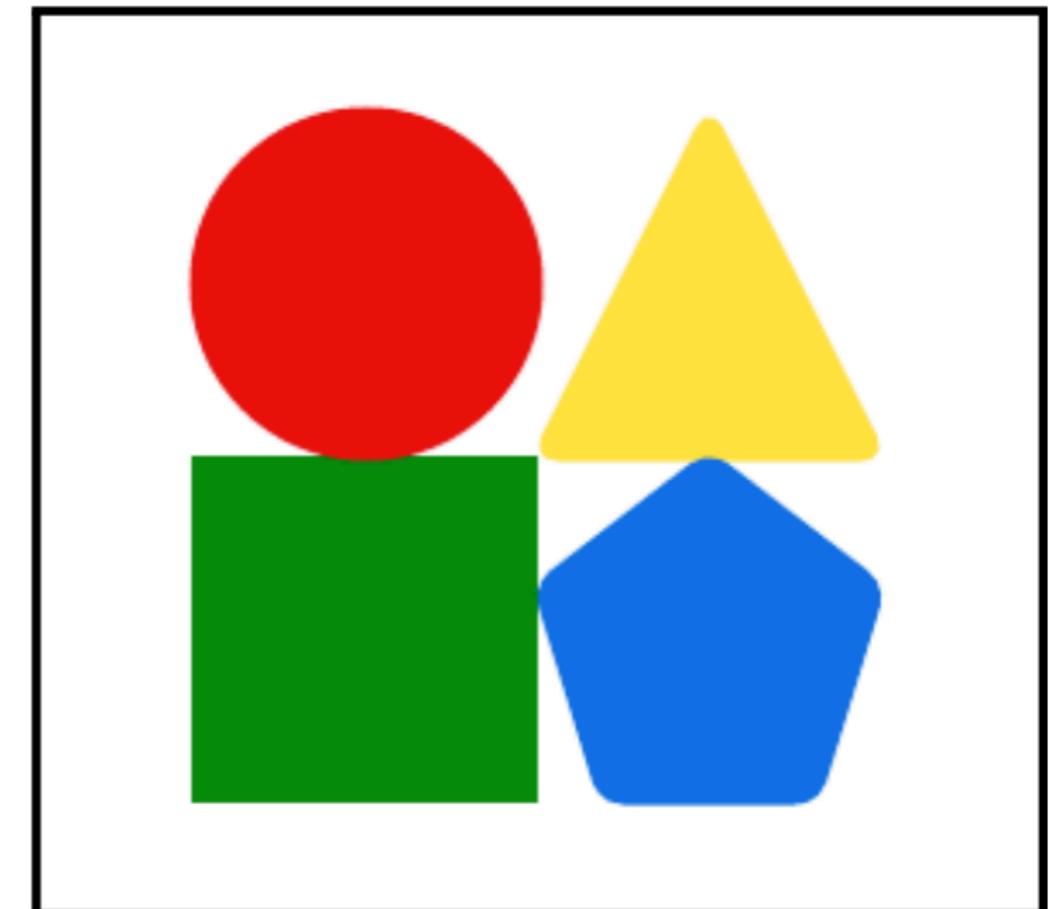
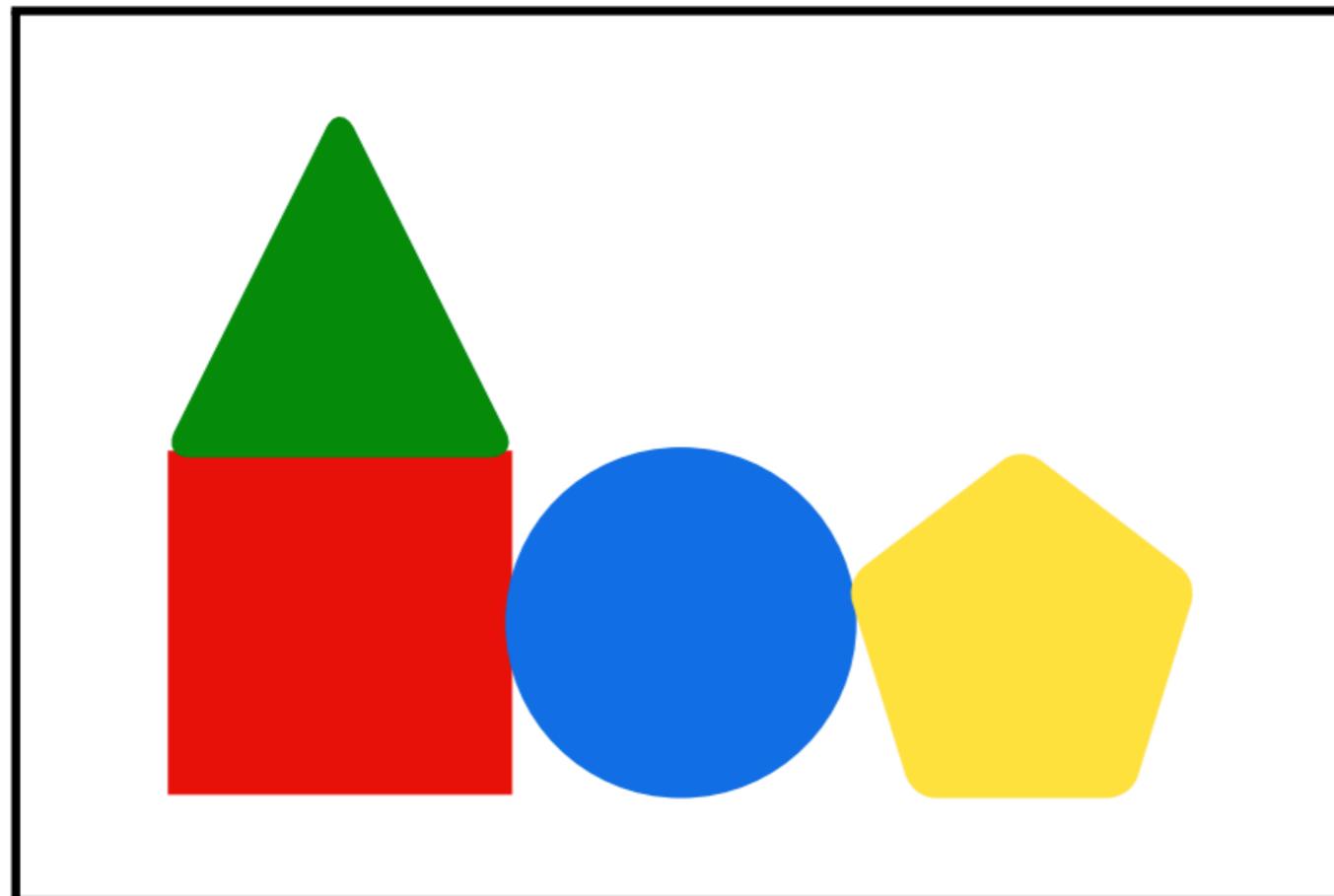
Positive examples

Joiner (Popper V4)



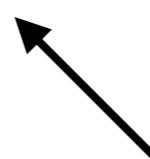
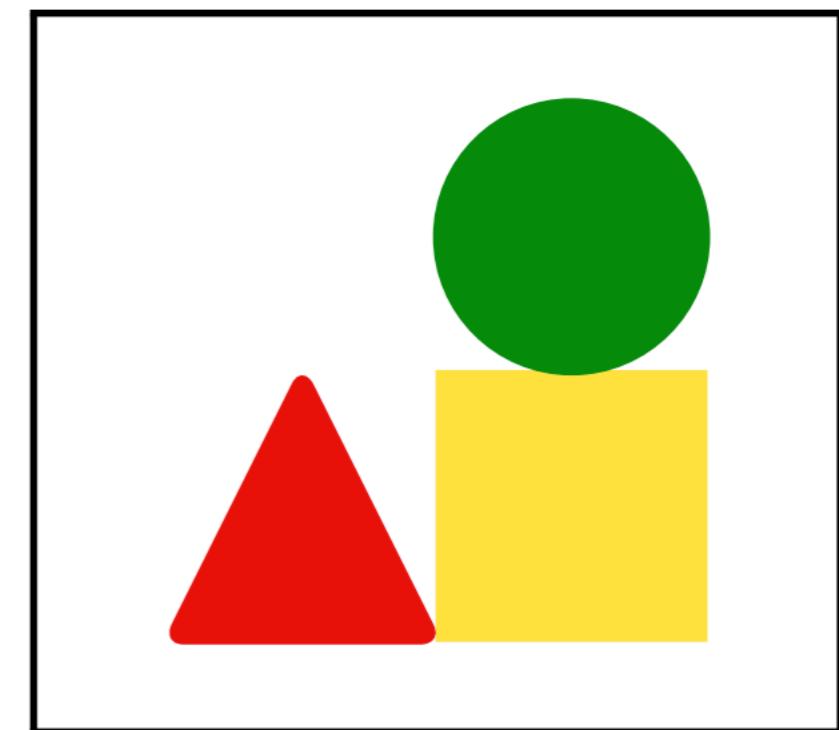
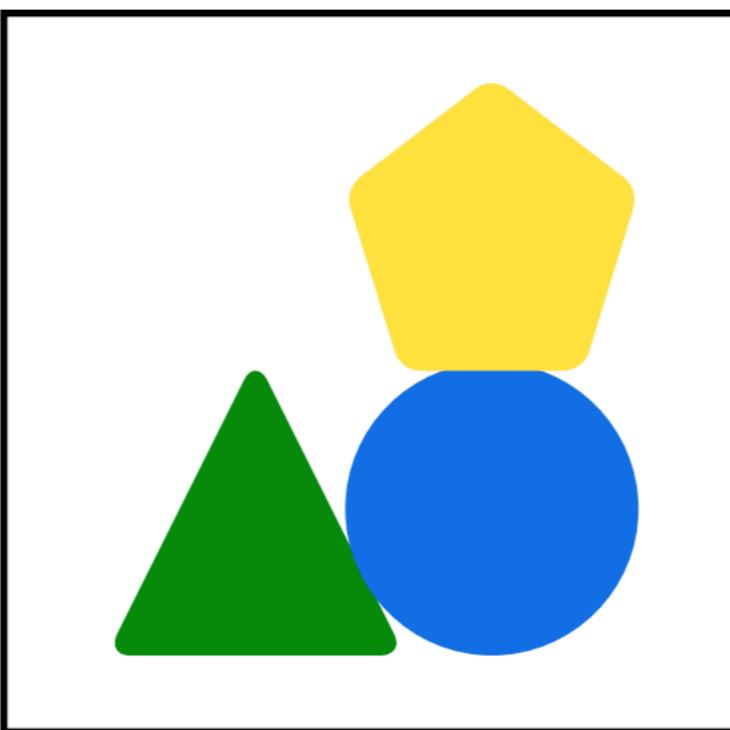
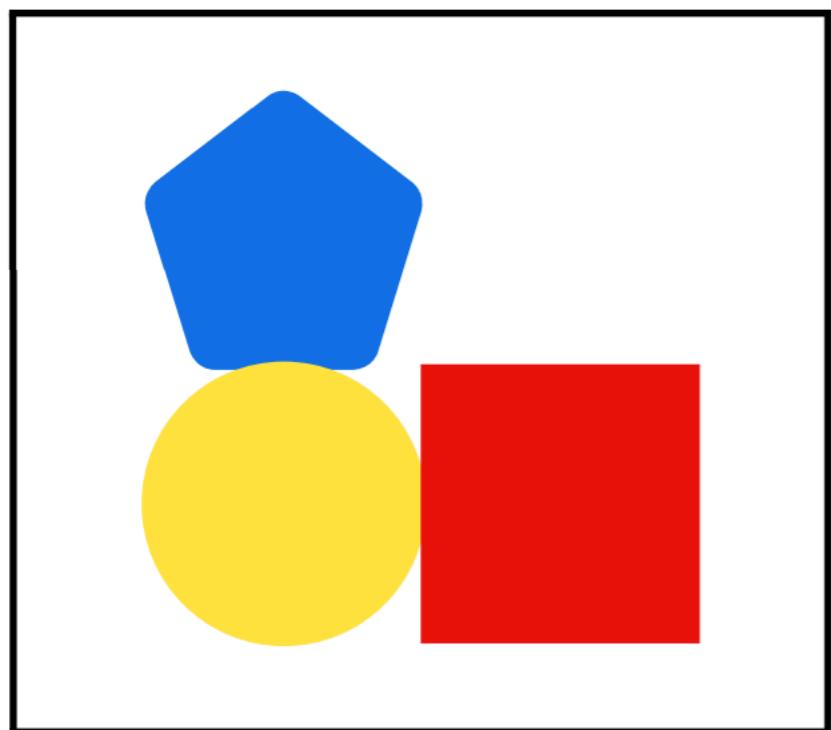
“There is a blue piece”

Joiner (Popper V4)



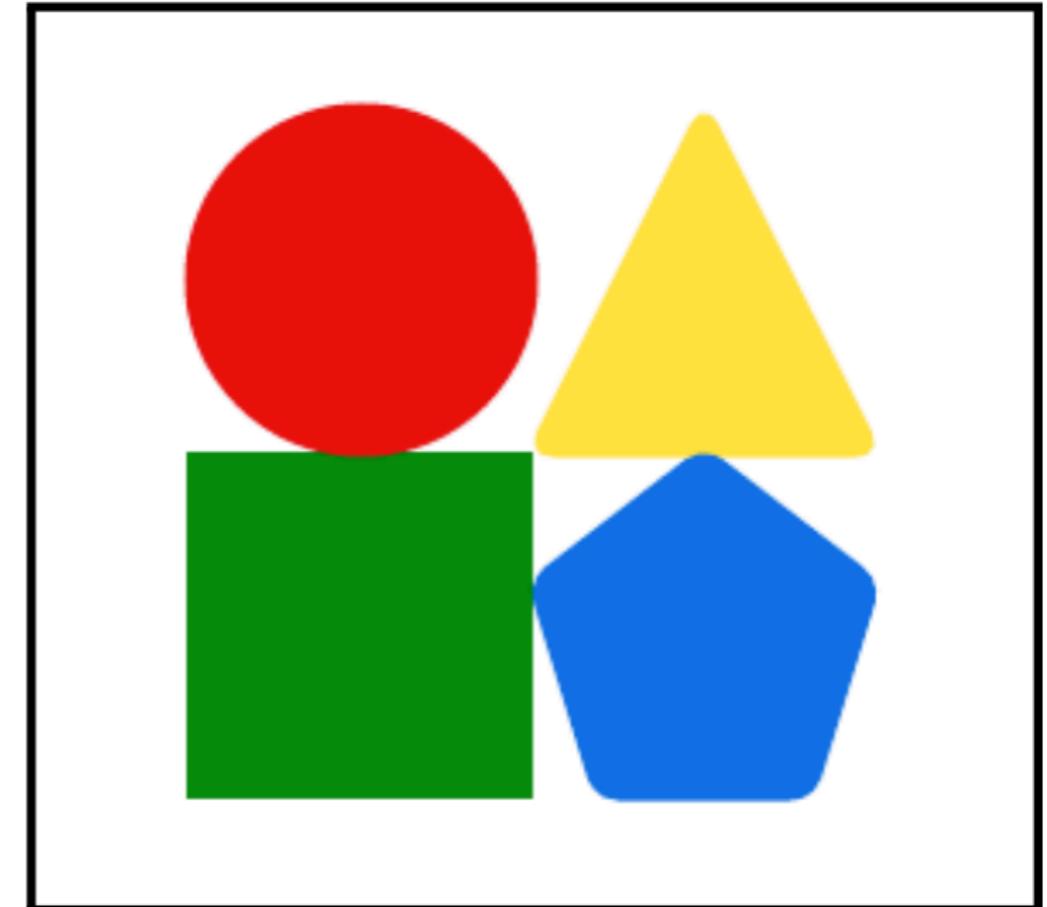
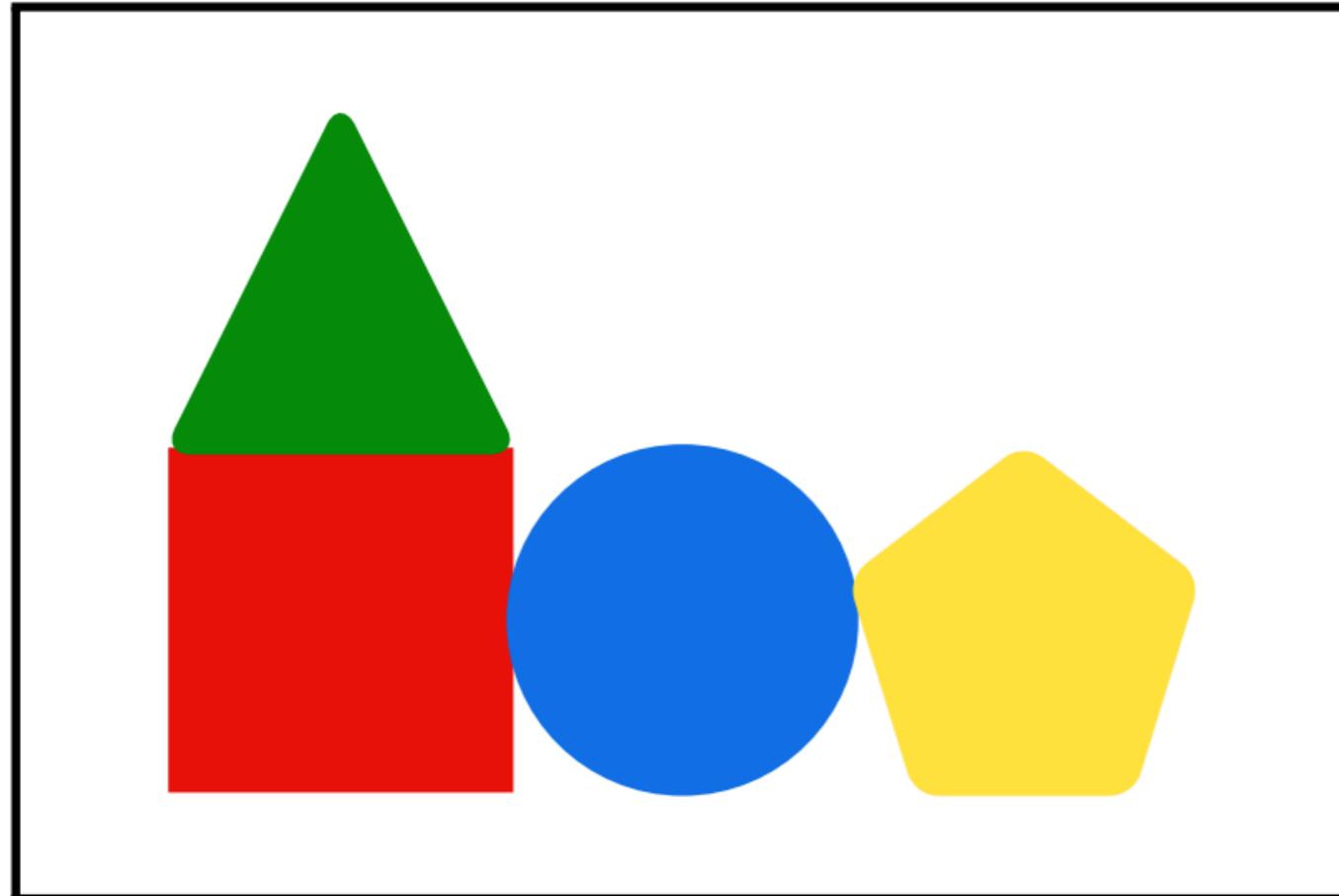
$h1 = \{ \text{zendo}(S) \leftarrow \text{piece}(S, B), \text{blue}(B) \}$

Joiner (Popper V4)



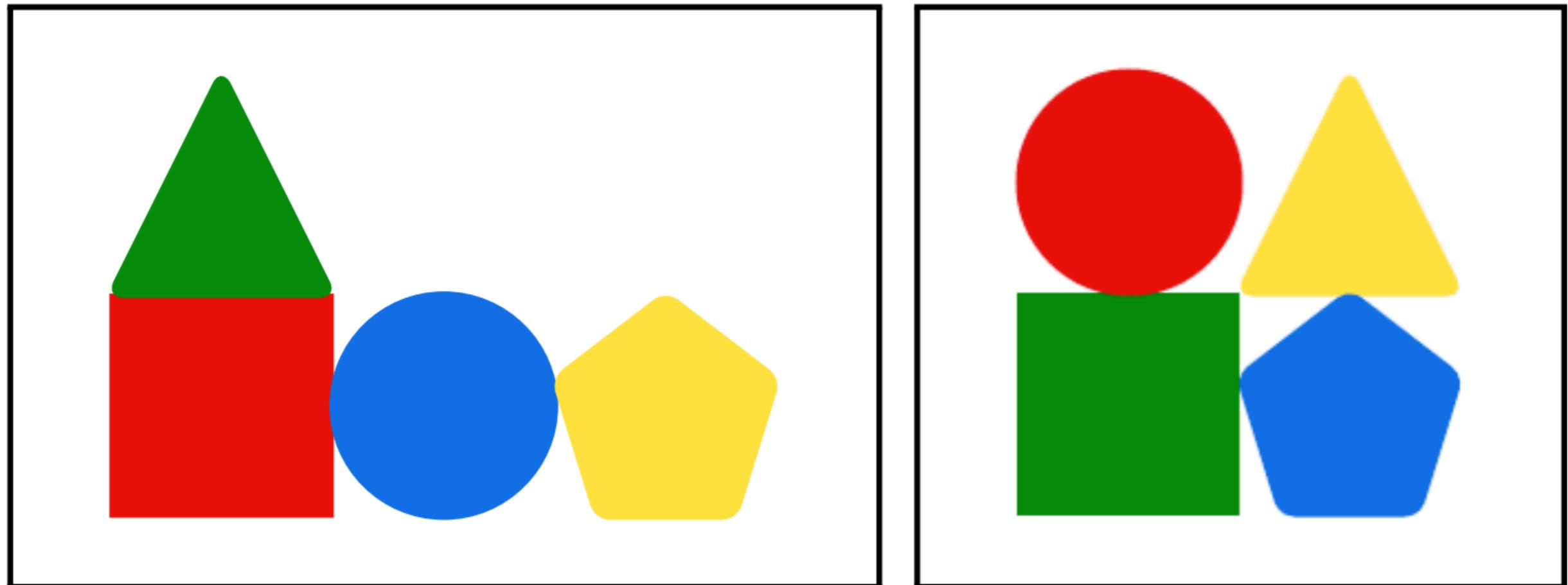
Negative examples

Joiner (Popper V4)



“There is a blue piece and a red piece and a green piece”

Joiner (Popper V4)



```
zendo(S) ← piece(S,B), blue(B), piece(S,R),  
red(R), piece(S,G), green(G)
```

Joiner (Popper V4)

```
zendo(S) ← piece(S,B), blue(B), piece(S,R),  
red(R), piece(S,G), green(G)
```

Joiner (Popper V4)

Splittable



zendo(S) \leftarrow piece(S, B), blue(B), piece(S, R),
red(R), piece(S, G), green(G)

Joiner (Popper V4)

Splittable



```
zendo(S) <
  piece(S,B), blue(B),
  piece(S,R), red(R),
  piece(S,G), green(G)
```

Joiner (Popper V4)

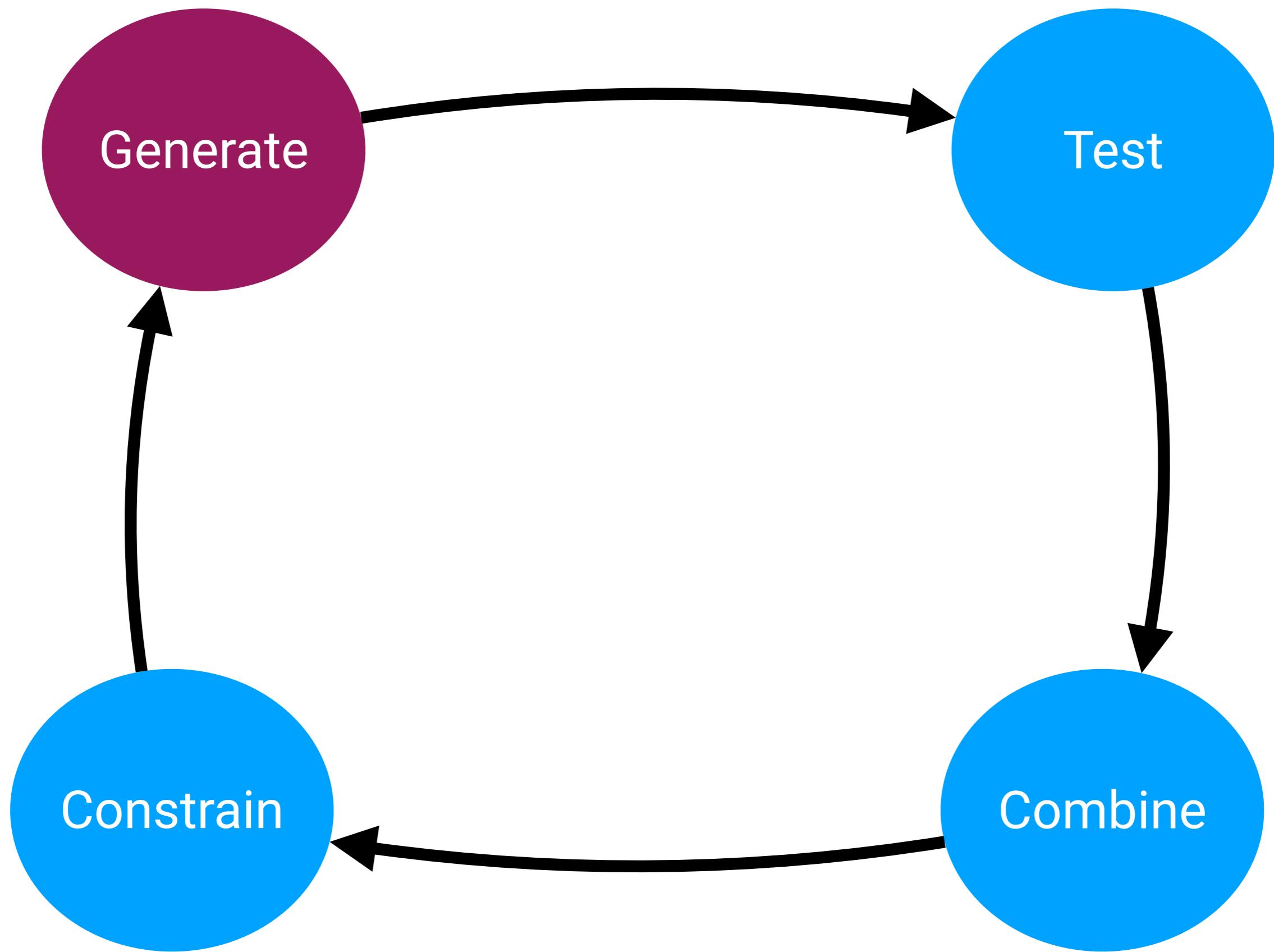
```
r1 = { zendo(S) ← piece(S,B), blue(B) }  
r2 = { zendo(S) ← piece(S,R), red(R) }  
r3 = { zendo(S) ← piece(S,G), green(G) }  
r4 = { zendo(S) ← piece(S,Y), yellow(Y) }
```

Joiner (Popper V4)

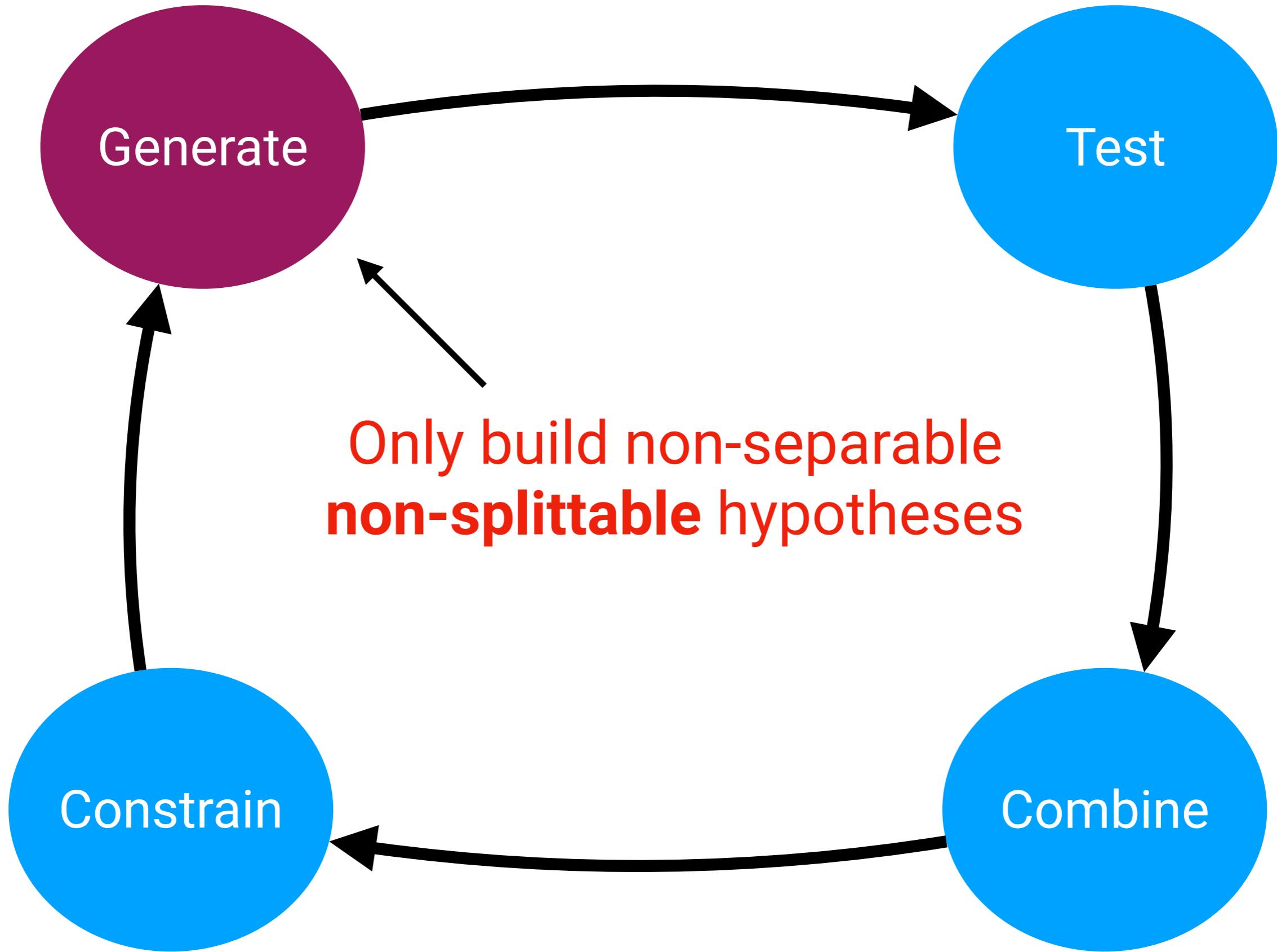
```
r1 = { zendo(S) ← piece(S,B), blue(B) }  
r2 = { zendo(S) ← piece(S,R), red(R) }  
r3 = { zendo(S) ← piece(S,G), green(G) }  
r4 = { zendo(S) ← piece(S,Y), yellow(Y) }
```

```
h = {r1 ∩ r2 ∩ r3}
```

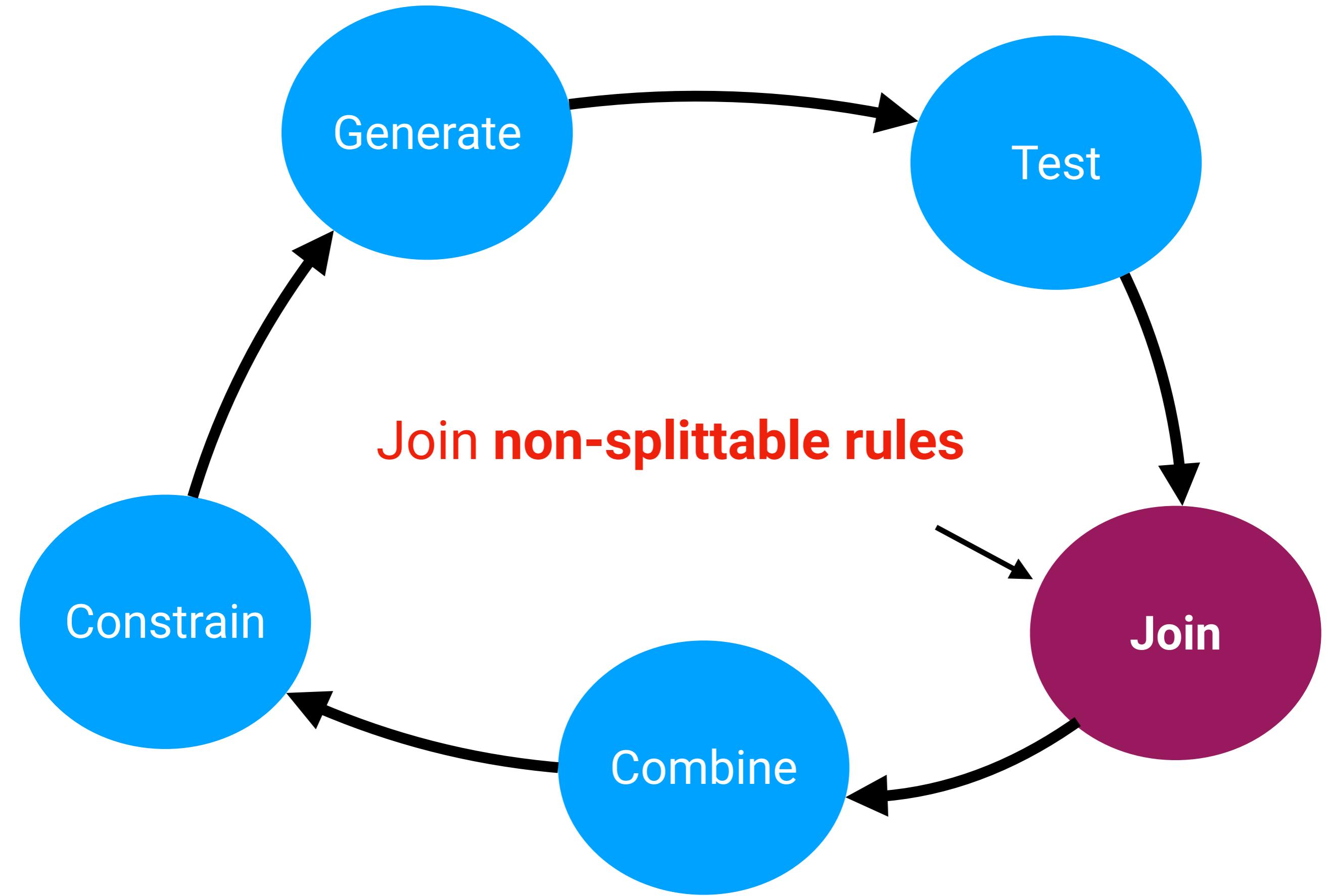
Combo (Popper V3)



Joiner (Popper V4)



Joiner (Popper V4)



Join

Given

- a set of non-splittable rules where each rule covers some positive and some negative examples

Join

Given

- a set of non-splittable rules where each rule covers some positive and some negative examples
- coverage of each rule

Join

Given

- a set of non-splittable rules where each rule covers some positive and some negative examples
- coverage of each rule
- size of each rule

Join

Given

- a set of non-splittable rules where each rule covers some positive and some negative examples
- coverage of each rule
- size of each rule

Find

- a subset of rules where the **intersection** of the coverage covers at least one positive example and no negative examples

Joiner (Popper V4)

SAT + MaxSAT

Joiner impact

Can learn rules with 100s of literals

Why?

Problem decomposition

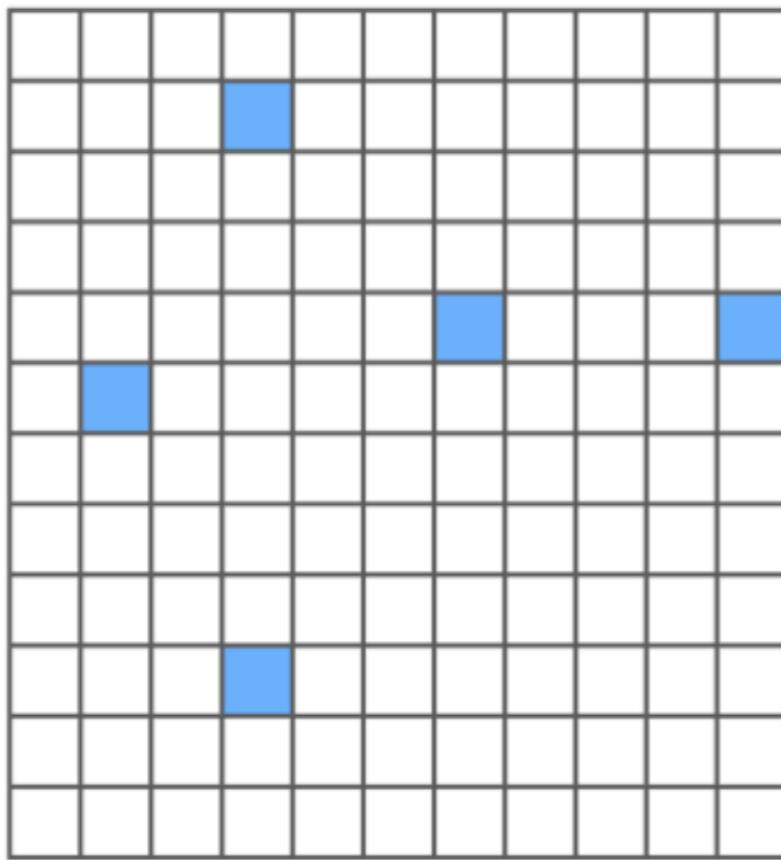
Why?

Generate stage is easier because the space
of non-splittable hypotheses is smaller

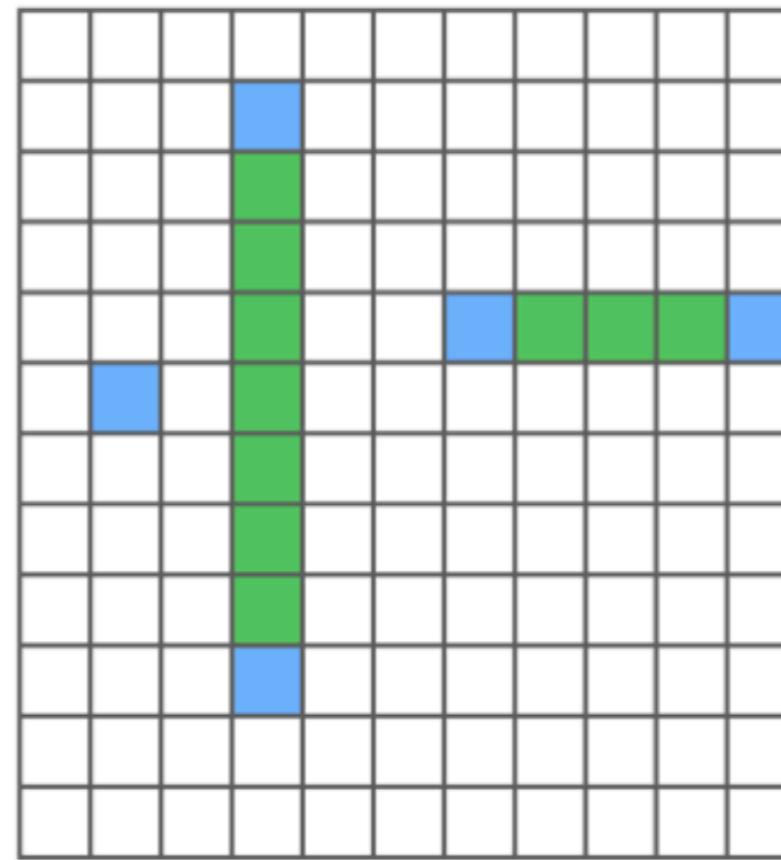
Why?

Join stage allows us to jump to learn big rules

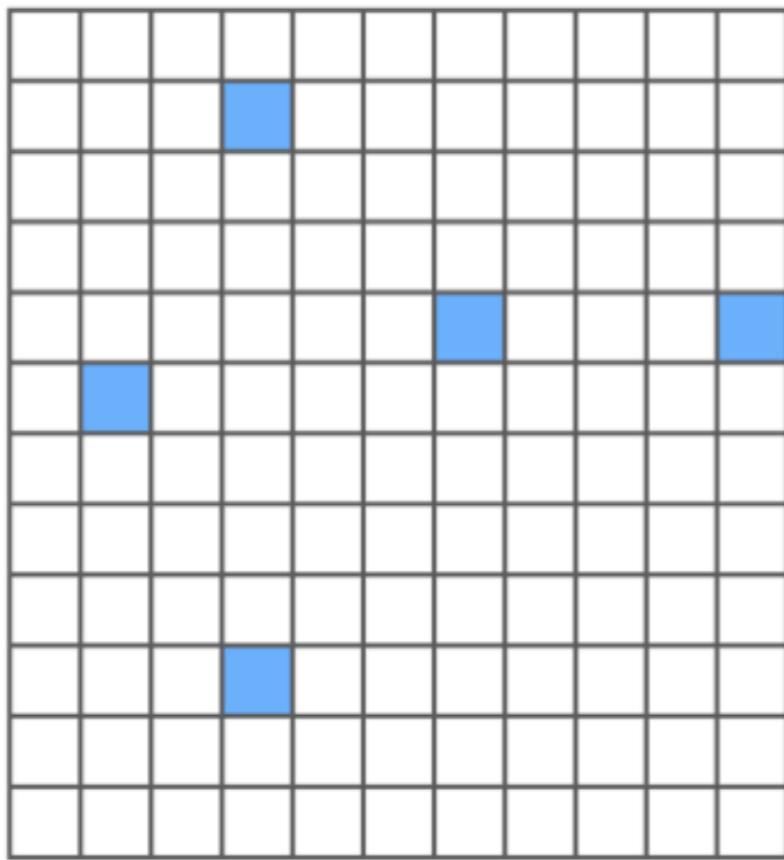
Input



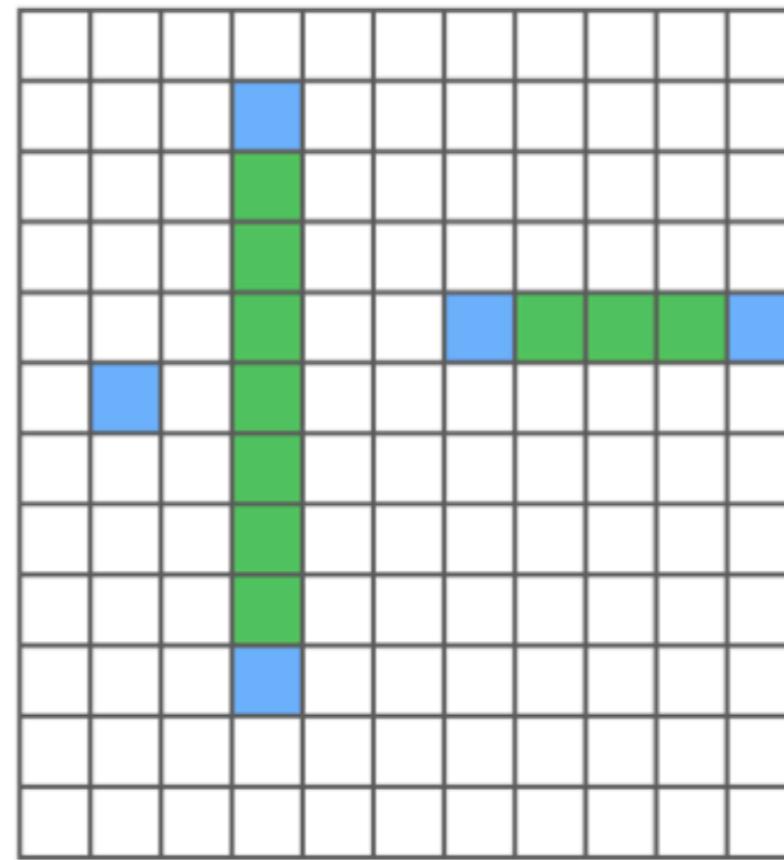
Output



Input



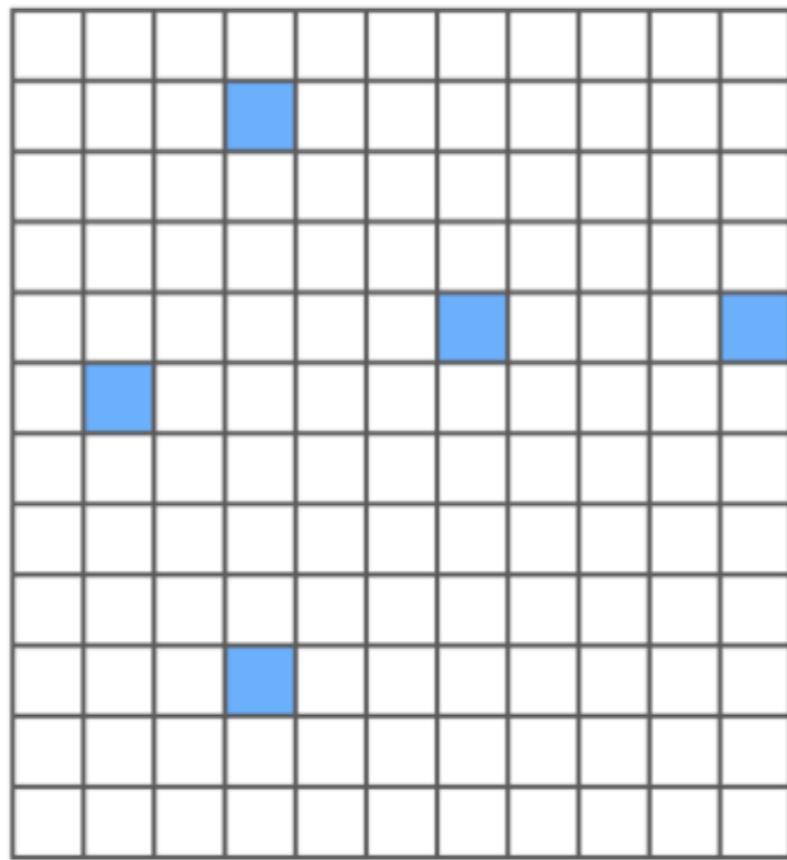
Output



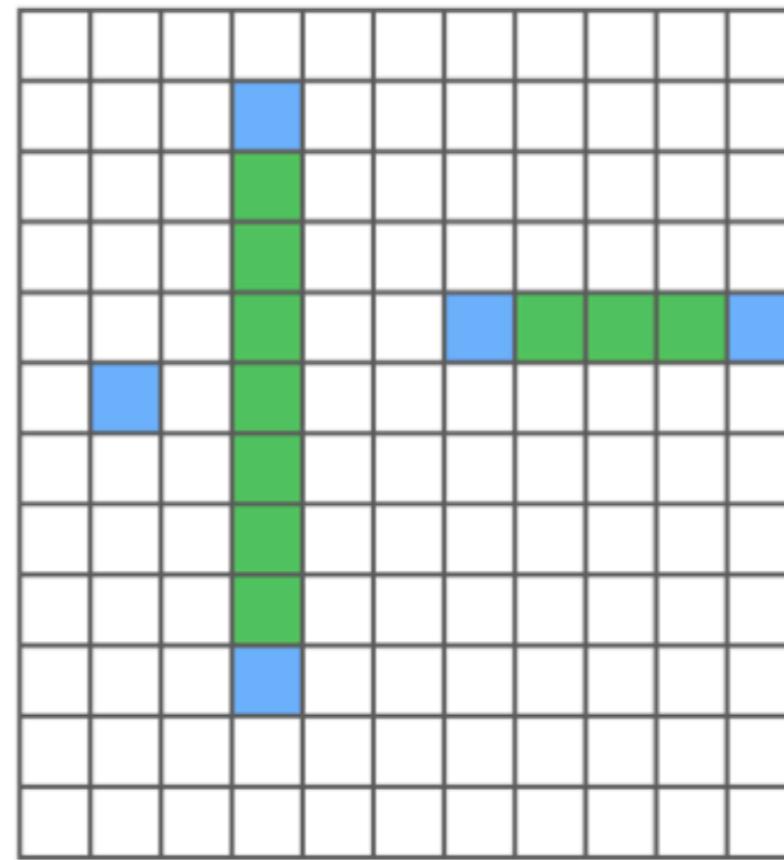
An out pixel has colour C if it is colour C in the input.

An out pixel is red if it is empty in the input and there is a pixel in the same row (X) and a pixel in the same column (Y) with the same colour (C) in the input.

Input



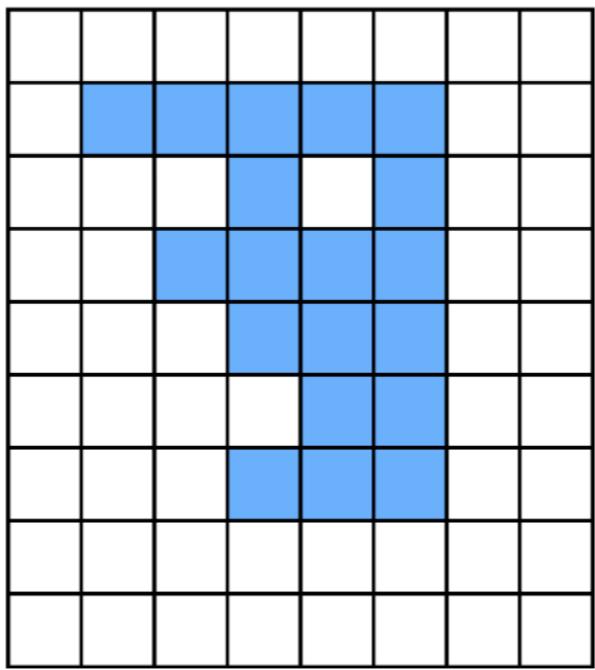
Output



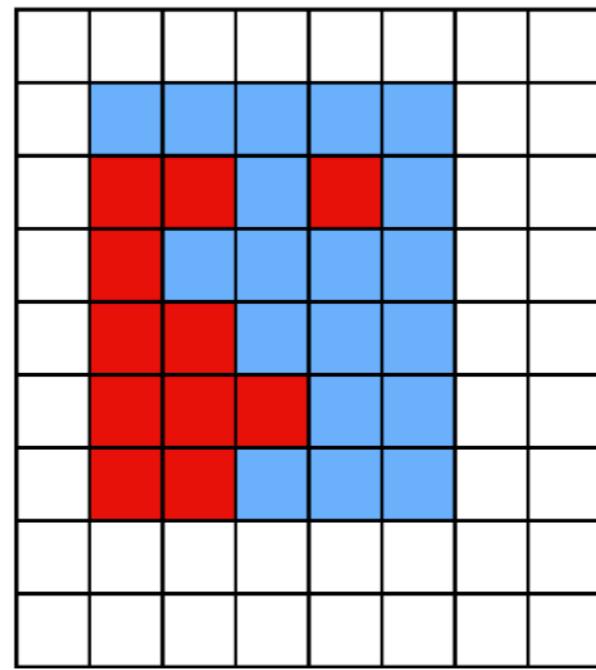
An out pixel has colour C if it is colour C in the input.

An out pixel is red if it is empty in the input and there is a pixel in the same row (X) and a pixel in the same column (Y) with the same colour (C) in the input.

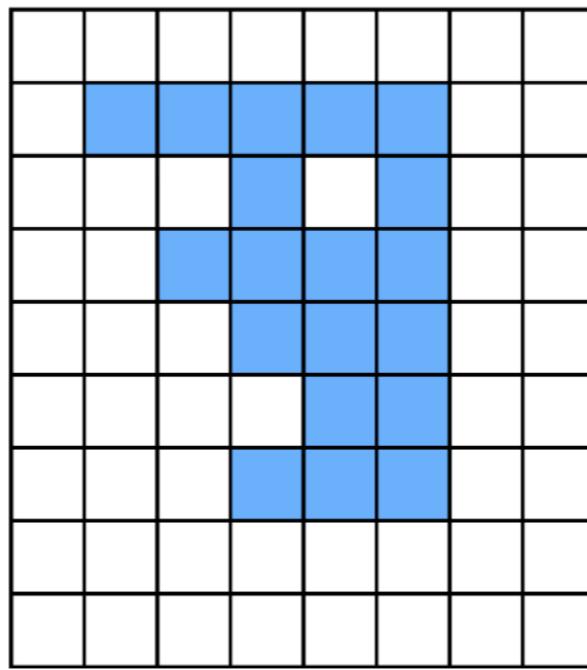
Input



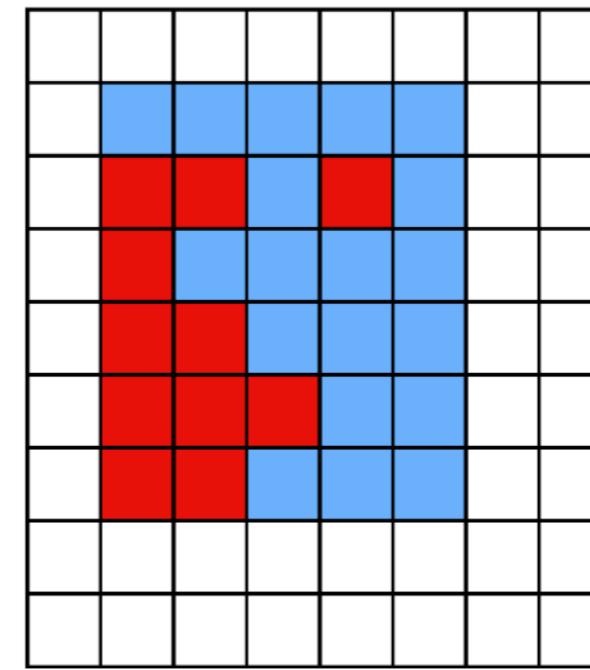
Output



Input



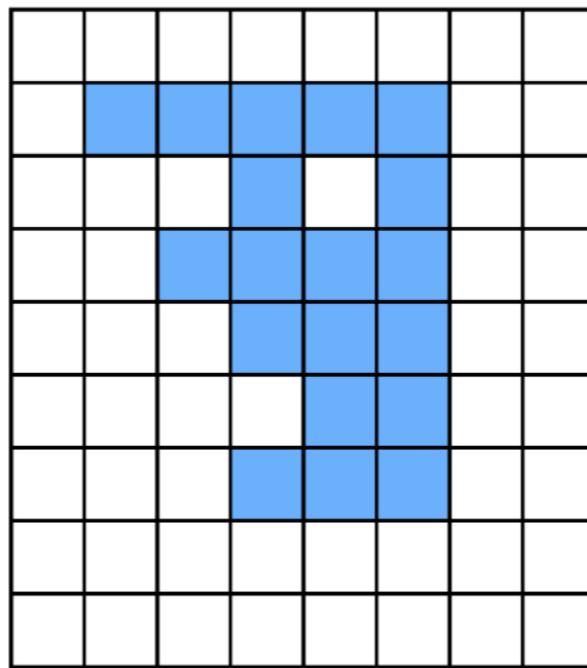
Output



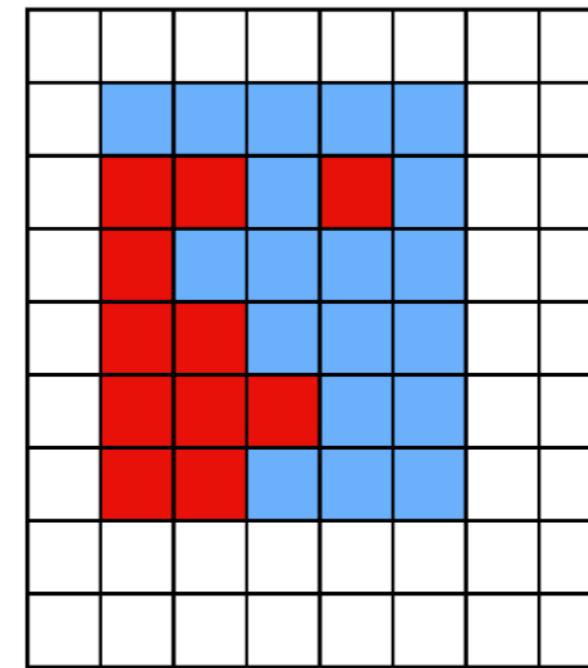
An output pixel is colour C if it is colour C in the input.

An output pixel is red if it is empty in the input and an input pixel in the row X has colour C, and an input pixel in the column Y has colour C.

Input



Output



An output pixel is colour C if it is colour C in the input.

An output pixel is red if it is empty in the input and an input pixel in the row X has colour C, and an input pixel in the column Y has colour C.