

An Unofficial FastLAS Tutorial: A Programmer's Guide

FABIO AURELIO D'ASARO

University of Verona, Italy

University College London, United Kingdom

(e-mail: fabioaurelio.dasaro@univr.it; fabio.d'asaro.14@ucl.ac.uk)

Abstract

FastLAS is a scalable system for Inductive Logic Programming (ILP): you give it some background knowledge, a *language bias*, and a set of *examples*, and it searches for a set of logic-program rules (a *hypothesis*) that explains the examples. These notes are a hands-on introduction to *writing* FastLAS programs. They are organised as a programmer's guide: syntax first, then a ladder of worked, numbered examples of increasing difficulty. Every FastLAS example here has been run against FastLAS 2.2.0 and shows the tool's actual output; the few Clingo and ILASP snippets are marked as such. We keep theory to the minimum needed to write correct programs; throughout, set-off notes flag where FastLAS differs from its sibling system ILASP, and where the two learning algorithms (`--op1` and `--nop1`) behave differently. The document is intended as an unofficial tutorial to FastLAS 2.2.0, not as an official language specification.

Disclosure. This is an unofficial guide written from a user, for the users. The author is not affiliated with ILASP LTD and has never participated in the development of ILASP or FastLAS.

KEYWORDS: FastLAS; Inductive Logic Programming; Learning from Answer Sets; Answer Set Programming; mode declarations; tutorial

Contents

1	Introduction	3
1.1	What FastLAS does	3
1.2	FastLAS, ILASP, and this guide	3
1.3	What you need	3
1.4	Notation used in this manual	4
2	FastLAS at a glance	4
2.1	The shape of a learning task	4
2.2	The four ingredients	5
2.3	What FastLAS computes	7
2.4	Your first task	8
2.5	Running FastLAS from the command line	8

3	The anatomy of a .las file	8
3.1	Background knowledge: ASP in one page	9
3.2	The language bias: mode declarations	10
3.3	Examples: context-dependent partial interpretations	13
3.4	Scoring: the #bias mini-language	17
4	Choosing the algorithm: --opl vs. --nopl	21
5	Writing effective programs	23
5.1	Writing an efficient mode bias	23
5.2	Working within FastLAS's limits	24
5.3	Prediction queries: #predict	27
5.4	Incremental learning from a stream (the cache)	29
6	Classic problems, from Clingo to FastLAS	30
6.1	Guess and test in Clingo	30
6.2	The same problems as learning tasks	31
6.3	Preferences: where FastLAS stops and ILASP starts	33
7	Case studies	35
7.1	Event recognition in CAVIAR	35
7.2	Access-control policy learning	37
8	Running FastLAS and reading its output	38
8.1	Command-line flags	38
8.2	Interpreting the output	39
8.3	Seeing the solve program: --output-solve-program	39
9	FastLAS vs. ILASP: a porting cheat-sheet	41
9.1	Deciding whether the task ports at all	41
9.2	A task ported line by line	41
9.3	How the directives correspond	42
9.4	What to do when a port fails	42
10	Quick reference	43
11	Exercises	43
Appendix A	Solutions to the exercises	47
References		51

How this guide is organised. Section 2 gives a first task at a glance. Sections 3 to 5 are the tutorial proper: the anatomy of a .las file, the choice of learning algorithm, and how to write programs that stay fast and stay within the system's limits. Section 6 revisits the classic ASP exercises as learning tasks, and Section 7 works two published applications, CAVIAR event recognition and access-control policy learning, as case studies. The remaining sections are compact reference material to consult rather than read straight through, followed by exercises whose solutions are in Appendix A.

1 Introduction

1.1 What FastLAS does

FastLAS (Law et al. 2020) learns *Answer Set Programs* (ASP) (Gebser et al. 2012) from examples: ordinary logic-programming rules of the form `head :- body`. You describe:

- **Background knowledge** B : facts and rules that are always true;
- **A language bias** M (*mode declarations*): the vocabulary the learned rules may use;
- **Examples** E : snapshots of what should (and should not) be entailed;
- **A scoring function** (*bias*): what makes one hypothesis “better” than another.

FastLAS returns the lowest-scoring hypothesis H such that $B \cup H$ explains every example. Two things characterise it: it is built to *scale* to tasks with tens of thousands of examples, and its notion of a *best* hypothesis is yours to define rather than fixed in advance.

1.2 FastLAS, ILASP, and this guide

FastLAS is built by Mark Law and collaborators at Imperial College London, on the same *Learning from Answer Sets* (LAS) foundations (Law et al. 2014; Law 2018) as **ILASP** (Law et al. 2020). The two systems share almost all of their *input* syntax (mode declarations, the example format, typed variables), so if you know one, most of the other is familiar. FastLAS trades generality for speed and adds custom scoring; ILASP is more general (recursion, choice rules, preferences). We point out the concrete differences in DIFFERENCE FROM ILASP notes as we go. At the time of writing there is no other dedicated FastLAS syntax tutorial; the closest general reference is the ILASP manual. This document is an unofficial, example-driven guide to FastLAS 2.2.0, not a formal language specification. For the official materials alongside it, see the FastLAS project page <https://spike-imperial.github.io/FastLAS/>, the FastLAS installation and running notes <https://spike-imperial.github.io/FastLAS/installation.html>, and the ILASP manual <https://doc.ilasp.com/>.

1.3 What you need

Two executables, both of which must be on your `PATH`. FastLAS is distributed from its project page, <https://spike-imperial.github.io/FastLAS/>, which carries the releases and build instructions; this guide uses version 2.2.0. FastLAS runs the **Clingo** ASP solver as an external process, so Clingo must be installed as well. Check both before starting, since a task cannot run without either:

```
$ FastLAS --version
$ clingo --version
```

A FastLAS task is usually a single text file, conventionally with the extension `.las`, though several files given on one command line are simply concatenated, which lets you factor out a shared background or bias. FastLAS ships *two* learning algorithms, chosen by a mandatory command-line flag:

- `--opl`: the original algorithm (*Observational Predicate Learning*), which assumes the predicate you are learning is one the examples actually talk about;
- `--nopl`: the later algorithm (*Non-Observational Predicate Learning*), which drops that assumption and so handles more tasks, at some cost in speed.

Section 4 is devoted to the difference. For now, just know that *every* run needs one of these flags.

Keep the official references close by while reading: the project page above, the FastLAS installation/running notes <https://spike-imperial.github.io/FastLAS/installation.html>, and the ILASP manual <https://doc.ilasp.com/>. When a command-line detail or corner-case matters, those are the authoritative sources.

PITFALL. Running FastLAS `task.las` with **no** algorithm flag prints `ERROR: usage: FastLAS [ --opl | --nopl ] file_name` and does nothing. Always pass `--opl` or `--nopl`.

1.4 Notation used in this manual

Three kinds of monospaced block appear throughout. A *task file*, or a fragment of one, is printed on its own:

```
#modeh(cycle).
#modeb(rain).
```

A *command* is introduced by a dollar prompt, which you do not type. Everything on the line after the prompt is what you enter at a shell:

```
$ FastLAS --opl examples/ex01_cycle.las
```

The *output* FastLAS writes in response is printed in the same way but without a prompt, so a command and its result can be read as a transcript:

```
cycle :- not rain.
```

In running text we set directives, atoms, rules and flags in typewriter font, as in `#modeh`, `cycle :- not rain.` and `--opl`. Command-line options keep the two hyphens FastLAS expects. Every task file shown here is named where it is first used, and all of them are in the `examples/` directory of the repository accompanying this guide:

https://github.com/dasaro/fastlas_manual

Paths such as `examples/ex01_cycle.las` are relative to a clone of that repository.

2 FastLAS at a glance

2.1 The shape of a learning task

A `.las` file is a plain-text file mixing up to four kinds of content, in any order and not all of them required (the first task below has no background knowledge at all):

Background knowledge	ordinary ASP facts/rules (<code>p :- q.</code>)
Mode declarations	<code>#modeh(...)</code> , <code>#modeb(...)</code> , <code>#maxv(N)</code>
Examples	<code>#pos(...)</code> , <code>#neg(...)</code>
Scoring bias	<code>#bias("...")</code>

Comments start with `%` and run to the end of the line (as in ASP/Prolog, *not* `//` or `#`).

The rest of this section says what each of the four means, and what exactly FastLAS computes from them. It is the only place in the guide with any formal machinery, and there is little of it: four definitions and one optimisation problem.

2.2 The four ingredients

Background knowledge B . What you already know, written as an ordinary ASP program: facts such as `bird(tweety).` and rules such as `flies(X) :- bird(X), not penguin(X).` It is fixed, it is never learned, and it is the same for every example. Formally it is a set of normal rules, read under the stable-model (answer set) semantics of Gebser et al. (2012): a program does not have a meaning but a set of *answer sets*, each one a consistent set of atoms that the program justifies. A program with no choices, like a set of facts and non-recursive rules, has exactly one.

Language bias M : the mode declarations. What the learned rules are *allowed* to say. `#modeh(A)` declares that the atom A may appear as a rule head; `#modeb(L)` declares that the literal L may appear in a rule body. Inside them, `var(t)` stands for a variable of type t and `const(t)` for a constant of that type, so one declaration stands for many concrete literals. `#maxv(N)` caps the number of distinct variables in any one rule.

Together these generate the *search space* (or hypothesis space) S_M : the set of every rule

$$h \leftarrow b_1, \dots, b_n$$

such that h is an instance of some `#modeh`, each b_i is an instance of some `#modeb`, and the rule uses at most `#maxv` distinct variables. This set is what FastLAS searches. A *hypothesis* H is any subset of it, $H \subseteq S_M$; nothing outside S_M can ever be learned, which is why an over-tight mode bias produces **UNSATISFIABLE** and an over-loose one produces a long wait.

It helps to see S_M written out once. ILASP, FastLAS’s sibling system, will do exactly that: its `-s` flag prints the whole hypothesis space generated by a mode bias. This is an **ILASP** facility, not a FastLAS one, and we borrow it here purely to make the definition concrete. Take this mode bias, which is all that `examples/ex29_space_ilasp.las` contains:

```
bird(a). bird(b).
#modeh(flies(var(bird))).
#modeb(winged(var(bird))).
#modeb(penguin(var(bird))).
#maxv(1).
```

One head mode, two body modes, one variable. ILASP enumerates what that generates:

```
$ ILASP --version=4 -s examples/ex29_space_ilasp.las
1 ~ :- winged(V1).
1 ~ :- penguin(V1).
2 ~ flies(V1) :- penguin(V1).
2 ~ :- penguin(V1); not winged(V1).
2 ~ flies(V1) :- winged(V1).
2 ~ :- winged(V1); penguin(V1).
2 ~ :- winged(V1); not penguin(V1).
3 ~ flies(V1) :- penguin(V1); not winged(V1).
3 ~ flies(V1) :- winged(V1); not penguin(V1).
3 ~ flies(V1) :- winged(V1); penguin(V1).
```

Ten candidate rules, each tagged with its length. Every combination of the declared literals appears, positive and negated, and so do the headless constraints that ILASP is

also willing to learn. That set is what “search space” means, and you can see how quickly it would grow with one more body mode or one more permitted variable.

FastLAS goes about it differently, and this is the heart of why it scales. It never builds S_M . Instead it works example by example: for each one it computes a small set of candidate rules that is *sufficient* in the sense that some optimal hypothesis can still be assembled from it, and searches only that (Law et al. 2020). The space is defined by your modes exactly as above, but it is approached from the data rather than enumerated in advance. What `--space-size` reports is the size of the candidate set FastLAS actually kept, so on the same modes with a single example it prints

```
$ FastLAS --opl --space-size examples/ex29_space_fastlas.las
% SPACE SIZE: 1
flies(V0) :- bird(V0).
```

one, not ten. The number therefore answers “how much did FastLAS have to consider here”, not “how large is S_M ”. It is still the right number to watch when a task is slow, because it is the one that grows when your bias is too loose (Section 5.1).

Examples E. What should and should not follow. An example is a four-slot *context-dependent partial interpretation*:

```
#pos( id , { inclusions } , { exclusions } , { context } ).
```

The *context* is a small ASP program describing one scenario, and it is added to the background for that example only. The *inclusions* and *exclusions* are sets of atoms saying what must, and must not, hold in that scenario. Write $B \cup C \cup H$ for the background, this example’s context and the hypothesis taken together. Then:

- a **positive** example is *covered* when **some** answer set A of $B \cup C \cup H$ satisfies $inc \subseteq A$ and $exc \cap A = \emptyset$;
- a **negative** example is *covered* when **no** answer set does.

“Some”, not “every”: coverage is *brave*. If the background admits several answer sets, a positive example is satisfied as soon as one of them fits, even if the others do not.

Example 1. *Brave coverage*

The background below has two answer sets, $\{x, p\}$ and $\{y\}$. Only the first contains q , and that is enough for $e1$.

```
% Positive examples are covered BRAVELY: an inclusion need hold in only
% SOME answer set, not all of them. The background has two answer sets,
% {x,p} and {y}; only the first contains q, and that is enough for e1
% to be covered.
1 { x ; y } 1.
p :- x.
#modeh(q).
#modeb(p).
#pos(e1, {q}, {}, {} ).
% context forces y, so p fails and q must not hold
#pos(e2, {}, {q}, { :- x. } ).
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex25_brave.las
q :- p.
```

An example with no weight is *hard*: it must be covered. Attaching a penalty with `@` makes

it *noisy*, and FastLAS may then leave it uncovered and pay that penalty instead, which is what lets it tolerate mislabelled data.

Scoring bias. Which hypothesis you want, when several cover the examples. A `#bias("...")` string is a small ASP program that assigns a cost to a candidate rule by inspecting it through the reserved predicates `in_head/1` and `in_body/1`. The familiar pair

```
#bias("penalty(1, head)    :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").
```

charges one point per literal, which makes “best” mean “shortest”. Section 3.4 shows costs that no fixed objective could express.

PITFALL. Without a #bias there is no objective at all. Every rule then costs zero, every covering hypothesis is equally optimal, and `--score-only` duly reports 0. FastLAS still returns *something*, but nothing is minimising length on your behalf, and which of several covering hypotheses you get is not something you should rely on. Every worked example in this guide supplies a scoring bias explicitly.

2.3 What FastLAS computes

Putting those together: given B , M , E and a scoring bias, FastLAS searches for a hypothesis $H \subseteq S_M$ that

1. covers every *hard* example, and
2. minimises

$$\mathcal{S}(H) = \sum_{r \in H} \text{cost}(r) + \sum_{e \in \text{uncovered}} \text{penalty}(e)$$

where the first sum is the cost your `#bias` assigns to the rules you keep, and the second is the price of the noisy examples you chose not to cover.

and prints it, one rule per line. If no $H \subseteq S_M$ covers the hard examples, it prints UNSATISFIABLE. Both halves of that sum are observable: `--score-only` prints $\mathcal{S}(H)$ for the returned hypothesis.

Example 2. *The two halves of the score*

Here `e1` and `e2` contradict each other. `e1` is hard, so it must be covered; `e2` carries penalty 1, so FastLAS covers `e1`, leaves `e2` uncovered, and pays.

```
% e1 and e2 contradict each other. e1 is hard, so it must be covered.
% e2 carries penalty 1, so FastLAS may leave it uncovered and pay 1.
#modeh(p).
#modeb(a).
#pos(e1, {p}, {}, { a. }).
#pos(e2@1, {}, {p}, { a. }).
#bias("penalty(1, head)    :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex26_noisy.las
p.
$ FastLAS --opl --score-only examples/ex26_noisy.las
2
```

The 2 is one point for the learned rule `p.` and one for the abandoned example.

2.4 Your first task

Let us learn the rule “I cycle to work unless it rains”. We will observe two days.

Example 3. *Hello, FastLAS (propositional)*

```
#modeh(cycle).
#modeb(rain).
#modeb(not rain).

#pos(d1, {cycle}, {}, {}).
#pos(d2, {}, {cycle}, {rain.}).

#bias("penalty(1, head) :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").
```

Read it top to bottom:

- `#modeh(cycle).` says the head of a learned rule may be the atom `cycle`.
- `#modeb(rain).` and `#modeb(not rain).` let rule bodies use `rain` or its negation `not rain`.
- `#pos(d1, {cycle}, {}, {}).` is example `d1`: in the scenario with empty context `{}` (a clear day), `cycle` must hold. The four slots are *id*, *inclusions*, *exclusions*, *context*.
- `#pos(d2, {}, {cycle}, {rain.}).` is example `d2`: when the context contains the fact `rain`, `cycle` must *not* hold (it is in the exclusions). Note the full stop in `{rain.}`: the fourth slot is an ASP *program*, so every fact in it ends with a dot, whereas the two set slots hold bare atoms and do not. Omitting that dot is a syntax error, and it is the commonest slip when copying this line as a template.
- The two `#bias` lines say “charge 1 point per head atom and 1 point per body atom”, i.e. prefer the shortest rule.

Run it:

```
$ FastLAS --opl examples/ex01_cycle.las

cycle :- not rain.
```

FastLAS found exactly the intended rule. Because this task is fully observational, both algorithms agree: `FastLAS --nopl examples/ex01_cycle.las` prints the same thing.

2.5 Running FastLAS from the command line

The basic invocation is

```
$ FastLAS --opl task.las      # original FastLAS algorithm
$ FastLAS --nopl task.las     # FastNonOPL (non-observational learning)
```

The remaining flags (`--version`, `--help`, `--debug`, `--force-safety`, `--threads N`, `--timeout T`) are summarised in Section 8. The learned hypothesis is printed to standard output, one rule per line. If no hypothesis explains the examples, FastLAS prints UNSATISFIABLE.

3 The anatomy of a .las file

Every FastLAS task is built from the same pieces: *background knowledge*, a *mode-declaration* language bias, *examples*, and a *scoring* function. This section covers each in turn: the whole input language in one place.

3.1 Background knowledge: ASP in one page

Background knowledge is an ASP program in the subset accepted by FastLAS, close to Clingo for ordinary facts and rules but not identical to full Clingo syntax. If you know Prolog/Datalog it will look familiar. The pieces you need:

Construct	Example
Fact	<code>bird(tweety).</code>
Rule	<code>flies(X) :- bird(X), not penguin(X).</code>
Integrity constraint	<code>:- smaller, greater.</code> (“never both”)
Choice rule	<code>1 { a ; b } 1 :- h.</code>
Interval / range	<code>num(1..100).</code>
Arithmetic	<code>+ - * / \ **</code> and <code> X </code> (absolute value)
Comparisons	<code>= != < <= > >=</code>
Comment	<code>% this is a comment</code>

One word on **not**, since it is the source of most early confusion. It is *negation as failure*, not classical negation: `not penguin(X)` does not assert that *X* is provably not a penguin, only that nothing in the program manages to derive that it is. Absence of evidence is treated as evidence of absence, which is what makes these programs *non-monotonic*: adding the fact `penguin(tweety)` later can retract a conclusion that held before.

Variables start with an **uppercase** letter (*X*, *Time*); predicate and constant names start **lowercase** (*bird*, *tweety*); `_` is an anonymous variable; strings are double- or single-quoted. Ranges like `num(1..100)` and arithmetic such as `X+1` are expanded by Clingo. You will typically put universal rules and type definitions in the background, and the specifics of each example in that example’s *context* (Section 3.3).

A program, built up. It is worth writing one small program before going further, because the background knowledge of a learning task is exactly this and nothing more. Facts state what is true, and a range states many facts at once, so `num(1..5).` is five facts. A rule derives new atoms from old, and its variables are read as “for every”:

```
composite(N) :- N = 1..5, I = 2..N-1, N\I = 0.
```

This says that *N* is composite when some *I* strictly between 1 and *N* divides it, `\` being the remainder operator. Negation as failure then gives the primes, with `not composite(N)` holding for every *N* the first rule failed to derive:

```
prime(N) :- N = 2..5, not composite(N).

$ clingo examples/asp/asp01_primes.lp
prime(2) prime(3) prime(5)
```

That is the whole of what a FastLAS background program does: derive further atoms from the facts of the task. The remaining ASP constructs, choice rules and constraints, do something different. They describe a *space* of possible answers rather than deriving one, which is how Clingo is used to solve puzzles and how ILASP describes hypotheses. FastLAS does accept them in the background, with one restriction: a choice rule must carry *both* bounds, so `1 { x ; y } 1.` parses but `{ x }.` is a syntax error. What FastLAS will not do is *learn* them; they can only be background. Section 6 shows what that means in practice: the classic guess-and-test programs stay in Clingo, and what FastLAS contributes is learning the rules inside them.

PITFALL. FastLAS's ASP is not full Clingo. Its parser rejects *conditional literals*, the : “for every” construct. A background rule like

```
defended(X) :- arg(X), out(Y) : att(Y,X).      % rejected
```

fails with `syntax error, unexpected T_COLON`. Rewrite it with an auxiliary predicate and negation. Name the *opposite* property and derive yours from its absence:

```
spoiled(X)  :- att(Y,X),
               not out(Y).                % some attacker of X is not out
defended(X) :- arg(X),
               not spoiled(X).             % no such attacker exists
```

This parses, and `defended` is then usable as an ordinary `#modeb` feature.

Dialect limits: what FastLAS's ASP won't accept. Conditional literals (just above) are one instance of a general rule: FastLAS's own lexer knows only a fixed set of `#`-keywords and constructs, so most Clingo “extras” are rejected outright. There is no fall-through to Clingo for them. (To see what FastLAS *does* produce, `--output-solve-program` is described in Section 8.)

You write	FastLAS says
<code>#show sel/0.</code>	Unknown token: '#'
<code>q :- #count{...} > 0. (aggregates)</code>	Unknown token: '#'
<code>#minimize{...}. / #maximize</code>	Unknown token: '#'
<code>:~ sel. [1@1] (weak constraint)</code>	syntax error, unexpected T_COLON
<code>p(X) :- q(Y) : r(X,Y). (cond. lit.)</code>	syntax error, unexpected T_COLON

Work within it: compute what you need with plain normal rules and negation (Section 3.1), and supply any optimisation (`#minimize`) or weak constraints at *deploy* time around the learned theory rather than inside the task. This is the generate-and-constrain pattern of Section 5.2.

3.2 The language bias: mode declarations

Mode declarations define the *hypothesis space*: the set of rules FastLAS is allowed to build. A learned rule takes its head from a `#modeh` declaration and each body literal from a `#modeb` declaration.

#modeh and #modeb. In the propositional case (Example 3) the declarations are just atoms. The power comes from *typed variables*.

Typed variables: `var(t)`. Inside a mode, `var(t)` is a placeholder for a variable of “type” `t`. The type `t` is an ordinary unary background predicate `t/1` that enumerates the type's values. For instance `#modeh(flies(var(animal)))` says “a learned rule may have head `flies(X)` where `X` ranges over things satisfying `animal(X)`”.

PITFALL. Safety. As in ASP, every variable in a rule must be *bound by a positive body literal*. A rule like `flies(X) :- not penguin(X).` is unsafe on its own, because `X` occurs only under negation. FastLAS does not reject such a rule: it appends the type atom (e.g. `animal(X)`) to the body, which makes it safe, and that is why you see the type atom in every learned rule. Example 7 learns exactly such a rule, `flies(V0) :-`

not flightless(V0), animal(V0)., whose only declared body literal is negated. If you would rather FastLAS satisfied safety from your *declared* modes instead of leaning on the type atom, pass `--force-safety`; on that same example it returns the longer `flies(V0) :- winged(V0), not flightless(V0), animal(V0)`. The flag changes the answer, so do not reach for it casually while debugging.

Example 4. *Typed variables and generalisation*

Two birds, both winged, both fly. What is the simplest rule?

```
animal(eagle). animal(sparrow).

#modeh(flies(var(animal))).
#modeb(winged(var(animal))).
#maxv(1).
#bias("penalty(1, X) :- in_head(X).").
#bias("penalty(1, X) :- in_body(X).").

#pos(p1, {flies(eagle)}, {}, { winged(eagle). }).
#pos(p2, {flies(sparrow)}, {}, { winged(sparrow). }).

$ FastLAS --opl examples/ex02_flies_general.las

flies(V0) :- animal(V0).
```

Note two things. First, FastLAS names the learned variable `V0`. Second, with no counter-examples the *most general* rule wins: “all animals fly”. The body mode `winged` was available but not needed, and the shortest hypothesis is preferred. To learn something less trivial we need examples that push back, and the per-example data must live in the *context* (note `winged(eagle)` sits in `p1`’s context). This is the single most important idiom in FastLAS; we return to it in Section 3.3.

Constants: `const(t)`. `const(t)` is a placeholder for a specific *constant* of type `t` to be baked into the rule (rather than a variable). In FastLAS the candidate constants are drawn from `t(C)` facts, whether those facts sit in the background or in a positive example’s context.

Example 5. *A learned constant with `const(t)`*

“`sel` holds exactly when the chosen digit is 2.”

```
#modeh(sel).
#modeb(chosen(const(digit))).
#bias("penalty(1, X) :- in_head(X).").
#bias("penalty(1, X) :- in_body(X).").

#pos(p1, {sel}, {}, { chosen(2). digit(2). }).
#pos(p2, {}, {sel}, { chosen(1). digit(1). }).
#pos(p3, {}, {sel}, { chosen(3). digit(3). }).

$ FastLAS --opl examples/ex05_const_select.las

sel :- chosen(2).
```

The constant 2 was selected because `digit(2)` appears in a positive example’s context. The other examples rule out `chosen(1)` and `chosen(3)`.

DIFFERENCE FROM ILASP. In ILASP, `const(t)` values come from `#constant(t, v)`. FastLAS has no `#constant` directive. Instead, supply the constants as ordinary `t(v)` facts in your positive example contexts (or background).

Negation as failure in modes. To allow a learned rule to use `not p`, declare it explicitly: `#modeb(not penguin(var(animal)))`. A body mode with no `not` counterpart can never appear negated.

DIFFERENCE FROM ILASP. ILASP *auto-generates* the negated version of each `#modeb` literal, and you write the option `(positive)` to switch that off. FastLAS is the other way round: negation is *opt-in*, declared with a separate `#modeb(not ...)`. FastLAS also does not support the ILASP mode-option tuples `(positive)`, `(anti_reflexive)`, `(symmetric)`.

Recall and #maxv. A leading integer in a body mode is a *recall* bound, written `#modeb(1, actual(var(number), var(number)))`, which in other systems caps how many times that literal may appear in one rule. In FastLAS the control that matters is `#maxv(N)`, which caps the number of distinct variables in any single learned rule: it is the main lever you have on the size of the search, and Section 5.1 measures its effect.

Comparisons and arithmetic as body modes. Relational operators can themselves be body modes, letting learned rules contain inequalities:

```
#modeb(var(number) < var(number)).
#modeb(var(number) >= var(number)).
```

Arithmetic in the *contexts* of examples is evaluated by Clingo before learning starts, and Example 10 relies on that. Declaring arithmetic as a body *mode* is a different matter: FastLAS is not built to learn arithmetic relations. Write the numeric argument as `const(t)` rather than as a literal, `#modeb(sz(var(thing), const(t)))` rather than `#modeb(sz(var(thing), 3))`, and reach for `num_var` (below) when what you want is a numeric *bound*.

Learning numeric thresholds: num_var. `var(t)` and `const(t)` range over *symbolic* values. For *numeric* data FastLAS has a third placeholder, `num_var(t)`, that does something the others cannot: it makes FastLAS *synthesise the comparison bounds itself*. You never write the threshold. FastLAS searches for the `>=`/`<=` constants that fit the examples. As in the examples below, the task should define the relevant numeric domain and provide the observed values in the example contexts.

Example 6. *Synthesising a numeric interval*

Four cars: two accepted (speeds 60 and 80) and two rejected (30 and 110), so the accepted band is bounded on *both* sides.

```
car(c1). car(c2). car(c3). car(c4).
speed_reading(0..120).
#modeh(ok(var(car))).
#modeb(observed(var(car), num_var(speed_reading))).
#maxv(1).
#pos(p1, {ok(c1)}, {}, { observed(c1, 60). }).
#pos(p2, {ok(c2)}, {}, { observed(c2, 80). }).
#pos(n1, {}, {ok(c3)}, { observed(c3, 30). }).
```

```
#pos(n2, {}, {ok(c4)}, { observed(c4, 110). }).
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --nopl examples/ex17_numvar.las

ok(V0) :- observed(V0,V_0_speed_reading), V_0_speed_reading >= 60,
          V_0_speed_reading <= 80, car(V0).
```

FastLAS invented *both* bounds (≥ 60 and ≤ 80) from the data; no comparison was declared as a mode. The synthesised numeric variable is named $V_{<i><type>}$. With negatives on only one side you would get a single bound (e.g. $V \geq 70$); a plain `var` in place of `num_var` synthesises nothing and the task is UNSATISFIABLE.

Several numeric conditions at once. Two flags govern how many numeric comparisons a rule may carry:

- `--max-conditions N` (default 1): the number of distinct bounded numeric variables allowed *per rule*. One variable's $\geq$ and $\leq$ together count as one.
- `--num-var-count N` (default 1): the number of numeric slots available *per numeric type*.

To bound two *different* quantities you must lift the default cap:

```
car(c1). car(c2). car(c3).
speed_reading(0..120). weight_reading(0..3000).
#modeh(ok(var(car))).
#modeb(fast(var(car), num_var(speed_reading))).
#modeb(heavy(var(car), num_var(weight_reading))).
#maxv(1).
#pos(p1, {ok(c1)}, {}, { fast(c1,80). heavy(c1,1500). }).
#pos(n1, {}, {ok(c2)}, { fast(c2,40). heavy(c2,1500). }).
#pos(n2, {}, {ok(c3)}, { fast(c3,80). heavy(c3,500). }).
#bias("penalty(1, body(X)) :- in_body(X).").

% the default cap of 1 makes this UNSATISFIABLE
$ FastLAS --nopl examples/ex17_numvar_multi.las
$ FastLAS --nopl --max-conditions 2 examples/ex17_numvar_multi.las

ok(V0) :- fast(V0,V_0_speed_reading), heavy(V0,V_0_weight_reading),
          V_0_weight_reading >= 1500, V_0_speed_reading >= 80, car(V0).
```

At the default cap of one numeric condition the task is UNSATISFIABLE; `--max-conditions 2` lets the rule bound both speed and weight.

3.3 Examples: context-dependent partial interpretations

An example is a *Context-Dependent Partial Interpretation* (CDPI). Its four slots are:

```
#pos( id , { inclusions } , { exclusions } , { context } ).
%          bare atoms      bare atoms      an ASP program:
%          (no dots)       (no dots)       every fact ends in a dot
```

Meaning: *there is an answer set of $B \cup H$ together with the **context**, that contains every **inclusion** atom and none of the **exclusion** atoms.* The *context* is a little ASP program local to that example: the scenario the example is about. The *id* may carry a penalty (*Weighted (noisy) examples*, below); it can even be omitted.

The context slot is the one that does most of the work in real tasks, so it is worth

dwelling on. Everything you put in it is added to the background *for that example only*, which means one file can hold thousands of unrelated situations without them interfering. A medical task might give each patient their own context; an access-control task gives each request one (Example 13); the CAVIAR study of Section 7.1 gives each video frame one. Contexts may contain rules, not just facts, so a scenario can carry its own local definitions:

```
% two patients, two scenarios, one file
#pos(p1, {treat(ann)}, {},
    { fever(ann). cough(ann). ill(X) :- fever(X), cough(X). }).
#pos(p2, {}, {treat(bob)},
    { cough(bob). ill(X) :- fever(X), cough(X). }).
```

Two things follow. Anything shared by *every* example belongs in the background instead, where it is written once; and anything that varies between scenarios must be in the context, because an example is otherwise indistinguishable from any other with the same inclusions and exclusions.

PITFALL. Put the scenario in the context. The most common beginner mistake is to leave contexts empty (`{}`) and pile all the data into global background. Examples grounded only through global facts typically come back **UNSATISFIABLE**. Rule of thumb: universal rules and type definitions go in the background; the facts describing *this* example go in *its* context.

Exclusions: expressing the negative case. Putting an atom in the *exclusions* of a positive example says “in this scenario, that atom must *not* be derivable”. This is the usual way to express a counter-example.

Example 7. Rules with exceptions (inclusions vs. exclusions)

“Winged animals fly, unless they are flightless.”

```
animal(eagle). animal(ostrich). animal(sparrow).

#modeh(flies(var(animal))).
#modeb(winged(var(animal))).
#modeb(not flightless(var(animal))).
#maxv(1).
#bias("penalty(1, X) :- in_head(X).").
#bias("penalty(1, X) :- in_body(X).").

#pos(p1, {flies(eagle)}, {}, { winged(eagle). }).
#pos(p2, {flies(sparrow)}, {}, { winged(sparrow). }).
#pos(p3, {}, {flies(ostrich)},
    { winged(ostrich). flightless(ostrich). }).

$ FastLAS --opl examples/ex03_flies_exception.las

flies(V0) :- not flightless(V0), animal(V0).
```

Example p3 is a positive example whose *exclusion* `{flies(ostrich)}` forbids the ostrich from flying; its context supplies `flightless(ostrich)`. That single counter-example forces the `not flightless(V0)` literal into the rule.

Negative examples #neg. A negative example states that *no* answer set of $B \cup H$ with the given context satisfies the inclusion/exclusion pattern. Syntactically it looks like **#pos** but with **#neg**. Where a positive example says *this must be possible*, a negative one says *this must be impossible*, and that is a much stronger demand: one stray answer set is enough to violate it. Negative examples are how you stop a learner settling for a rule that is too permissive.

Example 8. *A negative example rules out the over-general rule*

Polly flies and Tweety, a penguin, must not. The positive example alone is satisfied by `flies(V0) :- bird(V0).`, since nothing yet forbids Tweety from flying; the negative example is what forces the exception into the rule.

```
% A negative example says: no answer set may look like this. Here it is
% what stops the learner settling for "all birds fly".
bird(tweety). bird(polly). penguin(tweety).

#modeh(flies(var(bird))).
#modeb(bird(var(bird))).
#modeb(not penguin(var(bird))).
#maxv(1).

#pos(p1, {flies(polly)}, {}, {}). % polly does fly
#neg(n1, {flies(tweety)}, {}, {}). % tweety must NOT fly

#bias("penalty(1, head) :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --nopl examples/ex27_neg_penguin.las
flies(V0) :- not penguin(V0), bird(V0).
```

Delete the **#neg** line and re-run: the answer degrades to `flies(V0) :- bird(V0).`, which covers the one positive example just as well and happens to be shorter. The negative example is carrying the whole distinction.

PITFALL. Run tasks containing #neg with --nopl. Under `--opl` a task with a negative example in it returns **UNSATISFIABLE**; the same file solves under `--nopl`:

```
$ FastLAS --opl examples/ex23_neg.las
UNSATISFIABLE
$ FastLAS --nopl examples/ex23_neg.las
cycle :- not rain.
```

Since most of this guide runs under `--opl`, the safe habit is to state counter-examples as *exclusions* of a **#pos** wherever you can, as Example 7 does. Reach for **#neg** only when you genuinely need “no answer set may look like this”, and then pass `--nopl`.

Two smaller points about the syntax. **#neg** **must** use the four-slot form: FastLAS accepts **#pos** with either three slots (*id*, inclusions, exclusions) or four, but **#neg(e, {a}, {b})** is a syntax error, and you want **#neg(e, {a}, {b}, {})** with an explicit, possibly empty, context. And the *id* may carry a penalty just as a positive example may, so **#neg(e@5, ...)** is a noisy negative example.

Computed inclusions and exclusions. The two set slots can also be filled *by the background program* rather than written out. `inclusion/1` and `exclusion/1` are reserved: whatever the background derives for them, in the presence of an example’s context, is

added to that example's inclusion and exclusion sets. So this task learns `flies` even though both slots of both examples are empty:

```
% Computed inclusions and exclusions: the two set slots of every example
% are empty, and the background derives them instead (Section 3.3).
animal(a). animal(b).
inclusion(flies(X)) :- animal(X), goal(X).
exclusion(flies(X)) :- animal(X), not goal(X).
#modeh(flies(var(animal))).
#modeb(winged(var(animal))).
#pos(e1, {}, {}, { goal(a). winged(a). }).
#pos(e2, {}, {}, { }).
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex24_computed_slots.las
flies(V0) :- winged(V0), animal(V0).
```

Rename the two predicates to anything else and the same task returns the empty hypothesis, which is the quickest way to convince yourself the names are special. This is how tasks with thousands of examples are written in practice: each example carries only its scenario, and two background rules say what should follow. The CAVIAR study of Section 7.1 is built entirely on this mechanism.

Weighted (noisy) examples. Attach a penalty to an example id with @: `#pos(a1@1, {smaller}, {greater}, {...})`, and likewise `#neg(n1@1, ...)`. An unweighted example is *hard* and must be covered; a weighted one may be left uncovered at a cost equal to its penalty. Real data disagrees with itself, and this is what lets FastLAS shrug off the disagreement instead of returning UNSATISFIABLE: an outlier that would force a contorted rule can simply be bought off, if buying it off is cheaper than the extra literals.

Because coverage and hypothesis cost are added together (Section 2.3), the penalties are not merely labels. They decide the answer.

Example 9. The weights decide

Two examples flatly contradict each other, and both are noisy. FastLAS keeps whichever is more expensive to abandon.

```
% Two examples that contradict each other, both noisy. FastLAS keeps the
% one that is more expensive to abandon: here e1, at the price of
% leaving e2 uncovered.
% Swap the two weights and the answer flips to the empty hypothesis.
#modeh(p).
#modeb(a).

#pos(e1@3, {p}, {}, { a. }). % costs 3 to ignore
#pos(e2@1, {}, {p}, { a. }). % costs 1 to ignore

#bias("penalty(1, head) :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex28_weights.las
p.
$ FastLAS --opl --score-only examples/ex28_weights.las
2
```

It covered `e1` and paid 1 for abandoning `e2`: one point for the rule `p.` and one for the

abandoned example. Now swap the two weights, so that **e1** costs 1 to ignore and **e2** costs 3, and run it again. FastLAS returns *nothing at all* with a score of 1: the empty hypothesis costs nothing, covers **e2**, and abandons **e1** for a single point. Same data, same modes, opposite conclusions.

DIFFERENCE FROM ILASP. The example format (the 4-tuple CDPI and the **id@penalty** weighting) is byte-for-byte the same as ILASP, and so is the acceptance condition for a positive example: *brave*, meaning some answer set extends it. What differs is what each system asks of the *background*. `--opl` assumes $B \cup \text{context}$ is categorical, one answer set per example, which is what lets it be fast; ILASP makes no such assumption, and neither does `--nopl`. See Section 4.

A harder example: learning arithmetic. This example brings together typed numeric variables, arithmetic in contexts, and a length bias.

Example 10. *Learning $\text{result}(X)$ from expressions*

```
num(1..100).
#modeh(result(var(num))).
#modeb(expr(var(num))).
#maxv(1).
#bias("penalty(1, X) :- in_head(X).").
#bias("penalty(1, X) :- in_body(X).").

#pos(eg1, {result(5)}, { result(1) }, { expr(4+1). }).
#pos(eg2, {result(5)}, { }, { expr((4+1)). }).
#pos(eg3, {result(10)}, { }, { expr((4+1)*2). }).
#pos(eg4, {result(2)}, { }, { expr((4+1)/2). }).
#pos(eg5, {result(25)}, { }, { expr((4+1)**2). }).
#pos(eg6, {result(6)}, { }, { expr(|(4-7)*2|). }).

#pos(eg7, {result(6)}, { }, { expr(4+1*2). }).
#pos(eg8, {result(8)}, { }, { expr(4/1*2). }).
#pos(eg9, {result(1)}, { }, { expr(4-2-1). }).
#pos(eg10, {result(3)}, { }, { expr(4-(2-1)). }).

$ FastLAS --opl examples/ex04_arithmetic.las

result(V0) :- expr(V0), num(V0).
```

Each example puts an arithmetic expression in its context (`expr(4+1)`, `expr((4+1)*2)`, `expr(|(4-7)*2|)`, ...); Clingo evaluates the arithmetic, and FastLAS learns the single rule that maps every expression to its value. Note again the auto-appended type atom `num(V0)` that keeps the rule safe.

3.4 Scoring: the *#bias* mini-language

This is FastLAS's signature feature. Each `#bias("...")` contains a fragment of an ASP program that FastLAS evaluates *per candidate rule* to compute that rule's *cost*. The total cost of a hypothesis is the sum of its rule costs (plus penalties for any uncovered weighted examples); FastLAS returns a *minimum-cost* hypothesis. The reserved vocabulary:

- `penalty(W, Id)`: charge weight `W` (an integer, possibly negative, or an arithmetic expression) under identifier `Id`. Equal-`Id` penalties are counted once.
- `in_head(X)`: true iff atom `X` is in the candidate rule's head.
- `in_body(X)`: true iff literal `X` is in the candidate rule's body.
- plus arbitrary ASP: your own predicates, `#count`, arithmetic, comparisons... This is not a contradiction of the dialect limits of Section 3.1: the string inside `#bias` is handed to Clingo, so Clingo's aggregates are available there even though the same constructs are rejected in the background program.

The idiomatic “prefer shortest rule” scoring is

```
#bias("penalty(1, head)      :- in_head(X).").
#bias("penalty(1, body(X))   :- in_body(X).").
```

Here `head` and `body(X)` are just penalty *identifiers*; the special predicates are `in_head/in_body`. This pair is the one you will write most often, and it is what makes “best” mean “shortest”; as Section 2.2 noted, if you write no `#bias` at all there is no objective to minimise, so it is worth putting in even when it looks like boilerplate.

The scoring function chooses the hypothesis. When several hypotheses fit the data, the bias decides which one is returned. The next two examples share identical background, modes, and examples; only the `#bias` differs.

Example 11. *Scoring, take 1: plain length*

```
#modeh(sel).
#modeb(a).
#modeb(b).

#pos(p1, {sel}, {}, { a. b. }).
#pos(p2, {}, {sel}, { c. }).

#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex06_bias_length.las

sel :- a.
```

Both `sel :- a.` and `sel :- b.` explain the data and both have length 1, so FastLAS breaks the tie (here in favour of `a`).

Example 12. *Scoring, take 2: make one literal expensive*

Now we charge 10 for using `a` but only 1 for `b`:

```
#modeh(sel).
#modeb(a).
#modeb(b).

#pos(p1, {sel}, {}, { a. b. }).
#pos(p2, {}, {sel}, { c. }).

#bias("penalty(10, body(a)) :- in_body(a).").
#bias("penalty(1, body(b)) :- in_body(b).").

$ FastLAS --opl examples/ex07_bias_custom.las

sel :- b.
```

The same data gives a different answer, because the scoring function is a first-class part of the specification. This is how you encode domain-specific criteria: cost, risk, coverage, generality. Weights may be any integers, negative ones included, which is the basis of the generality objective in the policy experiments of Law et al. (2020). A second-stage `#final_bias("...")` lets you score aggregate features of a whole rule after the ordinary `#bias` has run.

DIFFERENCE FROM ILASP. `#bias` means *opposite things* in the two systems. In FastLAS it assigns *costs* (a scoring function). In ILASP it holds *pruning constraints* (`:- body(...)`) that add/remove rules from the search space but never assign cost; ILASP's objective is fixed to hypothesis length plus example penalties. Consequently `#bias` lines **do not port** between the systems.

Example 13. A policy in miniature

Custom scoring is why FastLAS was built: Law et al. (2020) introduced it to learn access-control policies from logs of granted and denied requests, where the policy that merely fits the log is rarely the one you want. Here is the idea at its smallest. Each request is one example, its attributes in the context; a granted request includes `accept`, a denied one excludes it, and each attribute gets a `const` body mode:

```
% Access-control policy learning (AAAI 2020 style): learn an 'accept'
% rule from granted and denied requests. Each request's attributes live
% in that example's context.
role(manager). role(staff).  clearance(high). clearance(low).
#modeh(accept).
#modeb(subject_role(const(role))).
#modeb(subject_clearance(const(clearance))).

% granted
#pos(g1, {accept}, {},
      { subject_role(manager). subject_clearance(high). }).
% denied
#pos(d1, {}, {accept},
      { subject_role(staff). subject_clearance(low). }).

% the usual per-literal bias: "best" means "shortest"
#bias("penalty(1, head)    :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex18_policy.las

accept :- subject_role(manager).
```

Both `subject_role(manager)` and `subject_clearance(high)` are single-literal policies consistent with the log, so under the per-literal bias they tie at cost 2 and FastLAS returns the role-based one. Now suppose your organisation prefers *clearance*-based policies; add one line that charges for role conditions:

```
#bias("penalty(1, prefer_clearance) :- in_body(subject_role(V)).").

$ FastLAS --opl examples/ex18_policy_clearance.las

accept :- subject_clearance(high).
```

Same log, a different *policy* chosen by the scoring criterion, not the data. The experiments of Law et al. (2020) push this to real access logs under length, coverage and generality objectives; Section 7.2 reproduces one of them.

Advanced scoring: #final_bias. The #bias scoring of Section 3.4 is *decomposable*: a rule's cost is a sum of *local*, per-literal contributions (*in_head*, *in_body*). That decomposability is what lets FastLAS score candidate rules independently and stay fast. But some criteria are not decomposable: they depend on an *aggregate* property of the whole rule. The classic case: “charge once if a rule uses negation-as-failure *at all*, however many negated literals it has.” You cannot write that as a sum of per-literal penalties (that would count each one).

#final_bias handles these *semi-decomposable* functions with a second scoring stage:

1. In #bias, compute *intermediate*/1 *feature* atoms describing the whole rule. These may be non-local. A negated body literal appears as `neg(X)`, so `intermediate(naf) :- in_body(neg(X)).` makes the single atom `naf` a feature that is present *iff* the rule uses any negation.
2. #final_bias assigns the cost over those features, in a final stage after the main search: `#final_bias("penalty(1, naf) :- intermediate(naf).")`.

Example 14. A flat cost for using negation

The data below admits exactly one rule, `p :- not a, not b`, with two negated literals.

```
% Semi-decomposable scoring: charge ONCE for using negation-as-failure,
% no matter how many negated literals a rule has. The only rule that
% fits here is
%   p :- not a, not b.
% It has two negations, yet its 'naf' cost is only 1.
#modeh(p).
#modeb(not a).
#modeb(not b).

#pos(e1, {p}, {}, {}).      % nothing present -> p holds
#pos(e2, {}, {p}, { a. }).  % a present      -> p false
#pos(e3, {}, {p}, { b. }).  % b present      -> p false

% stage 1 (#bias): compute an aggregate feature of the whole rule
#bias("intermediate(naf) :- in_body(neg(X)).").
% stage 2 (#final_bias): assign the cost over those features
#final_bias("penalty(1, naf) :- intermediate(naf).").

$ FastLAS --opl examples/ex16_final_bias.las
p :- not a, not b.
$ FastLAS --opl --score-only examples/ex16_final_bias.las
1
```

Its negation cost is **1**, not **2**: the `naf` feature is charged once regardless of the two `not` literals. Score the very same rule with a plain per-literal `#bias("penalty(1, X) :- in_body(neg(X)).")` instead and it costs **2**, one per negated literal. That per-literal counting is exactly the decomposable behaviour #final_bias lets you escape.

Reach for #final_bias when your objective genuinely depends on a whole-rule property (`naf` usage, the number of *distinct* predicates, the presence of a particular pattern)

that a per-literal `#bias` cannot express. The two stages compose: `#bias` still does the ordinary decomposable scoring, and `#final_bias` adds the aggregate part on top.

4 Choosing the algorithm: `--opl` vs. `--nopl`

Almost everything so far ran under `--opl`. The exceptions were the `num_var` runs of Section 3.2, which give the same answer either way, and the `#neg` task of Section 3.3, which does not: that one is **UNSATISFIABLE** under `--opl` and solvable under `--nopl`. The flag is not a cosmetic switch: it selects a different learning algorithm, and *the same .las file can succeed under one and fail under the other*. There is no marker in the file itself; the choice is entirely on the command line.

What `--opl` assumes. `--opl` runs the original FastLAS algorithm and does *Observational Predicate Learning*. It is valid only when, for every example:

1. the predicate(s) you are learning (the `#modeh` heads) are **directly observed**, appearing in the examples' inclusions/exclusions; and
2. $B \cup \text{context}$ has **exactly one** answer set (a *categorical* background: no unresolved choice rules and no even negation loops; FastLAS's parser has no disjunction at all).

When these hold, `--opl` is the right choice and the faster one: it skips the extra work described next.

What `--nopl` adds. `--nopl` runs FastNonOPL (Law et al. 2021), which inserts a *possibility-generation* phase (abduction) before solving. This lets it learn a predicate that is **never observed** in the examples but only influences observed predicates through the background, and it copes with backgrounds that have *multiple* answer sets. In principle anything `--opl` can solve, `--nopl` can too, and the converse is false. In practice it can also be substantially slower. Reach for `--nopl` when you need it, not by default.

Example 15. *A non-observational target: `--opl` fails, `--nopl` succeeds*

We observe `fault`; we want to learn `permitted`, which appears only inside the background rule `fault :- did(A), not permitted(A), act(A)`, never in an example.

```

fault :- did(A), not permitted(A), act(A).

#modeh(permitted(var(act))).
#modeb(authorised(var(act))).
#maxv(1).
#bias("penalty(1, X) :- in_head(X).").
#bias("penalty(1, X) :- in_body(X).").

#pos(e1, {}, {fault}, { act(a1). did(a1). authorised(a1). }).
#pos(e2, {fault}, {}, { act(a2). did(a2). }).

$ FastLAS --opl examples/ex08_nopl_permitted.las
UNSATISFIABLE

$ FastLAS --nopl examples/ex08_nopl_permitted.las
permitted(V0) :- authorised(V0), act(V0).
```

Under `--opl` the only learnable head is `permitted`, but the *observed* atom is `fault`; FastLAS has no way to connect them, so no hypothesis covers the examples. Under

--nopl, possibility generation reasons backwards through the background rule `fault :- ... , not permitted(A), ...` to infer what `permitted` must look like, and recovers the target rule.

Example 16. *Non-observational learning on a real task*

This task learns `valid_move`, which appears only in the background rule `violation :- agent_at(C,T), not valid_move(C,T-1), time(T-1)`, from examples that mention only `violation`:

```
#pos(eg3, {}, {violation}, {
    agent_at(cell(10,1),1).
    agent_at(cell(10,2),2).
    agent_at(cell(10,1),3).
    agent_at(cell(9,1),4).
    agent_at(cell(8,1),5).
}).

#pos(eg0, {violation}, {}, {
    agent_at(cell(10,1),1).
    agent_at(cell(10,2),2).
    agent_at(cell(8,1),3).
}).

cell(cell(1..10, 1..10)).
adjacent(cell(X,Y),cell(X+1,Y)) :- cell(cell(X,Y)), cell(cell(X+1,Y)).
adjacent(cell(X+1,Y),cell(X,Y)) :- cell(cell(X,Y)), cell(cell(X+1,Y)).
adjacent(cell(X,Y+1),cell(X,Y)) :- cell(cell(X,Y)), cell(cell(X,Y+1)).
adjacent(cell(X,Y),cell(X,Y+1)) :- cell(cell(X,Y)), cell(cell(X,Y+1)).

violation :- agent_at(C, T), not valid_move(C, T-1), time(T-1).

leq(T, T2) :- time(T), time(T2), T <= T2.
time(1..5).

#modeh(valid_move(var(cell), var(time))).
#modeb(agent_at(var(cell), var(time))).
#modeb(adjacent(var(cell), var(cell))).
#bias("penalty(1, head(X)) :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex09_agent_abduction.las
UNSATISFIABLE

$ FastLAS --nopl examples/ex09_agent_abduction.las
valid_move(V0,V1) :- agent_at(V2,V1), adjacent(V2,V0), cell(V0),
    time(V1), cell(V2).
```

OPL vs. NOPL. **Decision rule.** Use --opl when your target predicates are directly observed and the background is deterministic (one answer set per context); it is faster. Switch to --nopl when either (a) the predicate you are learning never appears in the examples (non-observational), or (b) the background/context can have more than one answer set, which in FastLAS comes from a choice rule or from an even loop through

negation such as `a :- not b.` with `b :- not a.` A symptom of needing `--nopl` is an `--opl` run that returns `UNSATISFIABLE` even though you believe a rule exists.

	<code>--opl</code>	<code>--nopl</code>
Algorithm	original FastLAS	FastNonOPL
Target must be observed?	yes	no
Background may have > 1 answer set?	no	yes
Extra abduction phase	no	yes
Speed	faster	slower
Generality	subset	superset (in principle)

5 Writing effective programs

Beyond getting a task to run, four topics recur in practice: keeping the search fast, working within FastLAS's expressiveness limits, querying a learned theory, and learning from a stream.

5.1 Writing an efficient mode bias

The mode declarations decide both *what* FastLAS can learn and *how long it takes*. FastLAS builds candidate rules by grounding the modes over variable assignments, and that work grows combinatorially with the number of variables you allow (`#maxv`) and the number and kind of body modes. The catch is that this cost is paid *even when the final hypothesis is tiny*: a bias that is looser than the target needs, but still contains it, gives the same answer more slowly. Two rules of thumb dominate.

Keep `#maxv` as small as the target needs. The bound on distinct variables per rule is the strongest lever you have on the size of the search, as the following experiment shows.

Example 17. *A performance experiment (tightening the variable bound)*

This task learns `rel(V0,V1) :- edge(V1,V0)`, a target that needs only two variables.

```

node(a). node(b). node(c). node(d). node(e). node(f). node(g). node(h).
edge(a,b). edge(b,a). edge(b,c). edge(c,b). edge(c,d). edge(d,c).
edge(e,f). edge(f,e). edge(f,g). edge(g,f). edge(g,h). edge(h,g).
near(a,c). near(c,a). near(e,g). near(g,e).
far(a,h). far(h,a). far(a,g). far(d,e).

#modeh(rel(var(node), var(node))).
#modeb(edge(var(node), var(node))).
#modeb(near(var(node), var(node))).
#modeb(far(var(node), var(node))).
#modeb(var(node) != var(node)).

#pos(p1, {rel(a,b)}, {}, { edge(a,b). }).
#pos(p2, {rel(c,d)}, {}, { edge(c,d). }).
#pos(p3, {rel(f,g)}, {}, { edge(f,g). }).
#pos(p4, {}, {rel(a,c)}, { near(a,c). }).
#pos(p5, {}, {rel(a,h)}, { far(a,h). }).

```

```

#bias("penalty(1, head) :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").
#maxv(2).

$ FastLAS --opl --space-size examples/ex12_modebias.las
% SPACE SIZE: 1
rel(V0,V1) :- edge(V1,V0), node(V0), node(V1).

```

It learns the intended rule already at `#maxv(2)`; `#maxv` is a directive rather than a flag, so to reproduce the rows below, copy the file and edit its last line. Watch what happens to the running time as we raise `#maxv`. The times below were measured on a MacBook Pro (13-inch, 2020, model `MacBookPro17,1`) with an Apple M1 system-on-chip (8 cores: 4 performance and 4 efficiency) and 16 GB of unified memory, using FastLAS 2.2.0 and Clingo 5.8.0; absolute times will differ on other machines, and what matters is how sharply they grow:

#maxv	learned rule	final space size	wall-clock time
2	<code>rel(V0,V1):-edge(V1,V0)</code>	1	0.06 s
3	(same)	2	0.09 s
4	(same)	2	1.7 s
5	—	—	killed after 2 min

Neither the learned hypothesis nor the final hypothesis-space size FastLAS reports (`SPACE SIZE`, the number of candidate rules FastLAS keeps after pruning) grows past two rules, yet raising `#maxv` from 2 to 5 turns a 0.06-second run into one that does not finish in two minutes. Every extra permitted variable multiplies the assignments FastLAS must ground. **Set `#maxv` to the smallest value in which your intended rule can be written.**

Declare only the body modes you need. Extra body modes enlarge the same grounding. *Comparison* modes such as `var(t) != var(t)` or `var(t) < var(t)` are the worst offenders, because they range over *every pair* of the allowed variables. On the same task, at a fixed `#maxv(4)`, growing from one body mode to three takes 0.09 s to 0.44 s, and adding the single `!=` comparison as a fourth mode takes it to 1.74 s: a further factor of four for one declaration, with no change to the learned rule.

PITFALL. Symptoms of an over-loose bias: a run that takes far longer than the size of the answer would suggest, or that appears to hang. Tighten it by (1) lowering `#maxv`; (2) removing modes the target does not use, especially comparison and high-arity modes; and (3) using `const(t)` instead of `var(t)` wherever a fixed value suffices. `--space-size` reports the final space size and `--debug` shows which phase is slow.

5.2 Working within FastLAS's limits

FastLAS learns *non-recursive* normal rules; constraint-style checks are learned through a fixed head such as `violated`, as below. Under `--opl` the background must also be *categorical*, one answer set per example; `--nopl` lifts that particular restriction (Section 4), but neither algorithm learns recursion. Three practical consequences follow, each with an idiom that works around it.

Recursion belongs in the background. A *learned* rule may not be recursive, but the background may. Precompute any recursive feature (reachability, transitive closure, ...) in the background and expose it to the learner as an ordinary body mode.

Example 18. *A recursively-defined feature in the background*

```
% Recursion is allowed in the BACKGROUND: reach/2 is a transitive
% closure, while the LEARNED rule stays non-recursive and just uses the
% derived feature connected/0.
reach(X,Y) :- edge(X,Y).
reach(X,Z) :- edge(X,Y), reach(Y,Z).
connected  :- reach(a,d).

#modeh(target).
#modeb(connected).

% path a..d -> connected
#pos(p1, {target}, {}, { edge(a,b). edge(b,c). edge(c,d). }).
% gap -> not connected
#pos(n1, {}, {target}, { edge(a,b). edge(c,d). }).

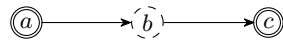
#bias("penalty(1, head)    :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex13_recursion_bg.las

target :- connected.
```

`reach/2` is defined recursively in the background; the learned rule `target :- connected.` is non-recursive and merely *uses* the reachability feature.

Reframe a coupled target as a verifier. The running example here is *abstract argumentation* (Dung 1995), and one paragraph of it is all we need. An argumentation framework is a directed graph: the nodes are arguments, and an edge `att(X,Y)` says that `X` attacks `Y`. A *labelling* marks each argument *in* (accepted, drawn with a double border) or *out* (rejected, dashed), and the basic sanity condition, *conflict-freeness*, says that no *in* argument may attack another *in* argument.



In the framework drawn above, the labelling shown (*a* and *c* in, *b* out) is conflict-free; labelling *a* and *b* both in is not, because *a* attacks *b*.

Here is the trap. Suppose you try to learn such a labelling *in/out* directly, and the feature that would discriminate, “*X* is attacked by an *in* argument”, is itself defined from *in*. That asks FastLAS for a *recursive* rule (the target appears in the definition of its own body feature), which it will not produce: the direct task either returns **UNSATISFIABLE** or collapses to a trivial over-general rule.

The fix is the *verifier reframing*: give the labelling as **context** (so the feature becomes deterministic *input*, not something derived from the target), and learn a *constraint*, a single violated head (or `#modeh(false)`), that recognises the bad cases. Valid examples *exclude* violated; invalid ones *include* it.

Example 19. *Learning a verifier over a given labelling*

```

% Verifier reframing: the labelling in/1 and out/1 is GIVEN as context,
% so attacked_by_in/1 is deterministic input rather than something
% learned through the target predicate.
% We learn a constraint-style verifier:
%   violated :- <features>.
arg(a). arg(b).
attacked_by_in(X) :- att(Y,X), in(Y).

#modeh(violated).
#modeb(in(var(arg))).
#modeb(out(var(arg))).
#modeb(attacked_by_in(var(arg))).
#modeb(not attacked_by_in(var(arg))).
#maxv(1).

% valid -> not violated
#pos(cf1, {}, {violated}, { att(a,b). in(a). out(b). }).
% valid -> not violated
#pos(cf2, {}, {violated}, { att(a,b). out(a). in(b). }).
% invalid -> violated
#pos(bad1, {violated}, {}, { att(a,b). in(a). in(b). }).

#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex14_verifier.las

violated :- in(V0), attacked_by_in(V0), arg(V0).

```

FastLAS learns the textbook conflict-freeness check: a labelling is bad when an `in` argument is attacked by an `in` argument. Because the labelling is context (deterministic, no choice rules), this runs under the faster `--opl`.

Rebuild the full semantics at deploy time. A verifier only *checks* labellings; to *enumerate* them, pair the learned rule with a fixed *generate* step in a plain Clingo program. Everything here is hand-written except the one learned line:

```

% Deploying a learned verifier (clingo, not FastLAS): a fixed generate
% step plus the rule FastLAS learned in the verifier example, turned
% into a constraint. clingo then enumerates the conflict-free
% labellings of the framework.
arg(a). arg(b). att(a,b).
0 { in(X) } 1 :- arg(X).
out(X) :- arg(X), not in(X).
attacked_by_in(X) :- att(Y,X), in(Y).
violated :- in(V0), attacked_by_in(V0), arg(V0).    % learned
:- violated.
#show in/1.

$ clingo examples/asp/asp10_deploy.lp --models=0
Answer: 1

Answer: 2
in(b)
Answer: 3
in(a)
SATISFIABLE

```

Exactly the three conflict-free labellings of the one-attack framework, no more. Any *global* objective, preferring larger labellings with a fixed `#maximize`, say, also lives in this deploy program rather than in the learning task: FastLAS learns the local *check*, and the fixed part provides the rest.

OPL vs. NOPL. This trap is not the non-observational case of Section 4: there the target never appears in the examples, whereas here it is exactly what the examples record. The obstacle is recursion, the target appearing in the definition of its own body feature, and neither `--opl` nor `--nopl` learns recursion, so the direct task stays UNSATISFIABLE under both. The verifier reframing is the remedy, and it keeps you in the faster `--opl`.

5.3 Prediction queries: `#predict`

Sometimes you do not want the hypothesis itself but an answer to a question: *given what FastLAS would learn, does the theory predict X in situation Y?* The `#predict` directive asks exactly that. Syntactically it mirrors an example, a partial interpretation plus a context,

```
#predict( id , { inclusions } , { exclusions } , { context } ).
```

but instead of constraining learning, it *queries* the learned theory. At most one `#predict` may appear in a task.

What FastLAS prints. Adding a `#predict` changes the output. Instead of one hypothesis, FastLAS prints **two**, each followed by a small JSON cost summary:

- the cheapest hypothesis (consistent with all the examples) that *satisfies* the query, and
- the cheapest one that does *not* satisfy it.

Reading off the verdict is easy:

- if the *not-satisfying* side is UNSATISFIABLE, the query is **necessarily entailed**: no admissible theory can avoid it (a confident yes);
- if the *satisfying* side is UNSATISFIABLE, the query is **impossible** (a confident no);
- if both sides return a hypothesis, compare their **Length**: the optimal theory behaves like the cheaper side (equal cost means the data leaves the query undetermined).

Example 20. *A prediction that is necessarily entailed*

We reuse the cycling task and ask: on a clear day (empty context), does the theory predict `cycle`?

```
#modeh(cycle).
#modeb(rain).
#modeb(not rain).

#pos(d1, {cycle}, {}, {}).
#pos(d2, {}, {cycle}, {rain.}).

#bias("penalty(1, head) :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").

#predict(clear_day, {cycle}, {}, {} ).
```

```

$ FastLAS --opl examples/ex10_predict_entailed.las

% Optimal hypothesis satisfying the prediction:

cycle :- not rain.

{
  "Length": 2,
  "Noisy Example Penalty": 0,
  "Uncovered Examples": [ ],
  "Final Semi-decomposable Representation": [ ]
}

% Optimal hypothesis not satisfying the prediction:

UNSATISFIABLE

{
  "Length": 0, ...
}

```

The not-satisfying side is UNSATISFIABLE: *no* hypothesis consistent with the training data can make `cycle` false on a clear day, so the prediction is a confident yes.

Example 21. *A prediction that is impossible*

The mirror-image question: on a rainy day, does the theory predict `cycle`? Only the context changes (`{rain.}`).

```

#modeh(cycle).
#modeb(rain).
#modeb(not rain).

#pos(d1, {cycle}, {}, {}).
#pos(d2, {}, {cycle}, {rain.}).

#bias("penalty(1, head) :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").

#predict(rainy_day, {cycle}, {}, { rain. }).

$ FastLAS --opl examples/ex11_predict_impossible.las

% Optimal hypothesis satisfying the prediction:

UNSATISFIABLE

{ "Length": 0, ... }

% Optimal hypothesis not satisfying the prediction:

cycle :- not rain.

{ "Length": 2, ... }

```

Now the *satisfying* side is UNSATISFIABLE: no admissible theory makes `cycle` hold when it rains, a confident no. (When both sides come back with a hypothesis, you instead

compare the two `Length` values; the FastLAS source distribution ships such a task as `FastLAS2/testing/prediction_task.las`.)

5.4 Incremental learning from a stream (the cache)

When examples arrive over time, a *stream* of *windows* rather than one fixed training set, you do not want to redo all the work each time a new window shows up. FastLAS can cache the hypothesis space and the per-example analysis computed so far and reload it when the next window arrives (the IncrementalLAS algorithm of Law et al. 2022). The guarantee: after each window the result is the *same* optimal hypothesis you would get by learning from every window at once. Two flags drive it:

<code>--write-cache FILE</code>	after solving, write the computed state (space + per-example data) to <code>FILE</code>
<code>--read-cache FILE</code>	resume from a previously-written cache instead of re-computing it

Example 22. *A two-window stream*

Each window file carries the fixed parts (mode bias, `#bias`, background) plus only that window's *new* examples. Window 1 has two examples; window 2 adds a single new one:

```
% fixed parts (repeated every window): mode bias + scoring
#modeh(p).
#modeb(a).
#modeb(b).
#bias("penalty(1, body(X)) :- in_body(X).").

% window 1 examples
#pos(e1, {p}, {}, { a. b. }).
#pos(e2, {}, {p}, { }).

% fixed parts (repeated every window): mode bias + scoring
#modeh(p).
#modeb(a).
#modeb(b).
#bias("penalty(1, body(X)) :- in_body(X).").

% window 2 brings ONE new example (the cache carries e1, e2)
#pos(e3, {}, {p}, { a. }).
```

Learn window 1 and save its cache; then learn window 2 *reading* that cache and saving a new one:

```
$ FastLAS --opl --write-cache cache1 examples/ex15_window1.las

p :- a.

$ FastLAS --opl --read-cache cache1 --write-cache cache2 \
  examples/ex15_window2.las

p :- b.
```

The hypothesis *refined* from `p :- a.` to `p :- b.` once the new example ruled out the first rule. Window 2 was passed *only* the new example `e3`; `e1` and `e2` came from the cache. Chain further windows the same way (`--read-cache cache2 --write-cache cache3 ...`).

The guarantee. Learning the same three examples in one batch gives the identical result:

```
$ FastLAS --opl examples/ex15_batch.las          # e1, e2, e3 together
p :- b.
```

Incremental equals batch: the cache saves work, not accuracy.

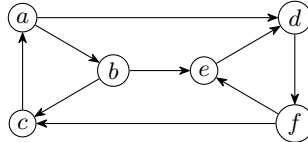
PITFALL. Each window file must still contain the `#modeh/#modeb/#bias` declarations and any global background. The cache carries the computed *per-example* data and the hypothesis-space schemas, but the mode bias must be present for FastLAS to fold the new window's examples in; drop it and the new examples are silently ignored (you get the previous window's answer). Treat the cache as opaque. Do not hand-edit it.

The payoff is what makes streaming tractable: each window reuses the earlier windows' analysis instead of recomputing it, while the guarantee above keeps the running hypothesis optimal over everything seen so far.

6 Classic problems, from Clingo to FastLAS

The problems in this section are the standard exercises of an ASP course: cliques, colouring, vertex cover, and a small preference puzzle. Each one can be read twice. Written for Clingo, the program *solves* an instance, and you supply the rules. Written for FastLAS, the rules are what you are missing, and the instances are what you supply. Several of the encodings below go back to the textbook treatments of Lifschitz (2019) and Gebser et al. (2012).

The Clingo programs below all use the same six-vertex graph (the learning tasks later in the section use smaller ones, so that the examples stay readable):



```
vertex(a;b;c;d;e;f).
edge(a,b; b,c; c,a; d,f; f,e; e,d; a,d; f,c; b,e).
```

6.1 Guess and test in Clingo

The Clingo idiom for these problems is *guess and test*: a choice rule guesses a candidate, and constraints reject the candidates that are not solutions. A clique is one line of each.

```
{ in(V) : vertex(V) } = 3.
:- in(V1), in(V2), V1!=V2, not edge(V1,V2), not edge(V2,V1).

$ clingo examples/asp/asp02_clique.lp
in(e) in(f) in(d)
```

Vertex cover is the same shape, and introduces the auxiliary predicate that the learning task below will have to discover:

```

{in(X) : vertex(X)}.
covered(X,Y) :- edge(X,Y), in(X).
covered(X,Y) :- edge(X,Y), in(Y).
:- edge(X,Y), not covered(X,Y).

$ clingo examples/asp/asp05_vertexcover.lp
in(b) in(c) in(d) in(e)

```

None of this is FastLAS input, starting with the very first line: the pooling semicolon of `vertex(a;b;c;d;e;f)` is a syntax error there, and so are a choice rule whose head begins with a left brace (FastLAS wants both bounds), conditional literals with `:`, and `#show`. All are outside the ASP dialect it accepts (Section 3.1); these programs are for Clingo, and they are in `examples/asp/`.

Two more classics ship in `examples/asp/` for reading rather than walking through: `asp04_queens.lp`, and `asp03_hamiltonian.lp`, which carries a caveat worth the detour. Its reachability rule follows `edge/2` rather than the guessed `in/2`, so the test is vacuous: the file admits 440 models, where the corrected version (one literal changed to `in(V1,V)`) admits 6, the same single cycle counted once per guessed start vertex. Hand-written encodings are delicate; learning the rule from examples, as the tasks below do, is one way of not hand-writing them.

6.2 The same problems as learning tasks

Turn each program around. Instead of writing `covered/2` and asking Clingo for a cover, give FastLAS some graphs in which you already know what is covered, and let it write the rule.

Example 23. Learning the vertex-cover rule

The graph and the guessed set `in` become example contexts, and the head is the predicate we want defined.

```

% Learn one half of the vertex-cover definition:
%   covered(X,Y) :- edge(X,Y), in(X).
% The graph and the guessed set "in" are supplied as example contexts.
vertex(a). vertex(b).
#modeh(covered(var(vertex), var(vertex))).
#modeb(edge(var(vertex), var(vertex))).
#modeb(in(var(vertex))).
#maxv(2).
#pos(p1, {covered(a,b)}, {}, { edge(a,b). in(a). }).
#pos(p2, {}, {covered(a,b)}, { edge(a,b). in(b). }).
#pos(p3, {}, {covered(a,b)}, { edge(a,b). }).
% in(a) but no edge
#pos(p4, {}, {covered(a,b)}, { in(a). }).
#bias("penalty(1, head) :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex19_vertexcover.las
covered(V0,V1) :- edge(V0,V1), in(V0), vertex(V0), vertex(V1).

```

That is the first of the two rules above. Example `p4` is the one that earns its keep. Drop it and the answer degrades to `covered(V0,V1) :- in(V0), vertex(V0), vertex(V1).`, which explains every remaining example and is one literal shorter, because nothing in

the task ever shows a vertex that is in without an edge leaving it. It is the concrete form of the rule in Section 3.3: an example is only informative if some candidate rule gets it wrong.

Example 24. *Learning the colouring conflict*

A constraint is learned as a rule with a fixed head, the verifier reframing of Section 5.2. Proper colourings exclude `violated`, improper ones include it.

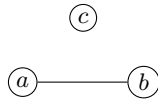
```
% Learn the graph-colouring conflict as a verifier:
%   violated :- edge(V0,V1), colour(V0,V2), colour(V1,V2).
vertex(a). vertex(b). shade(red). shade(blue).
#modeh(violated).
#modeb(edge(var(vertex), var(vertex))).
#modeb(colour(var(vertex), var(shade))).
#maxv(3).
#pos(ok1, {}, {violated},
    { edge(a,b). colour(a,red). colour(b,blue). }).
#pos(ok2, {}, {violated},
    { edge(a,b). colour(a,blue). colour(b,red). }).
#pos(bad1, {violated}, {},
    { edge(a,b). colour(a,red). colour(b,red). }).
#pos(bad2, {violated}, {},
    { edge(a,b). colour(a,blue). colour(b,blue). }).
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex20_colouring.las
violated :- edge(V1,V2), colour(V2,V0), colour(V1,V0), shade(V0),
    vertex(V1), vertex(V2).
```

Two vertices joined by an edge carrying the same shade `V0`: the Clingo constraint `:- edge(X,Y), colour(X,C1), colour(Y,C2), C1 = C2.` recovered from examples alone.

Example 25. *Learning the clique conflict*

The same reframing recovers the clique test of Section 6.1. The graph is the three-vertex one below, with a single edge, and each context pairs it with one chosen set: $\{a, b\}$ and the singleton $\{a\}$ are cliques, $\{a, c\}$ and $\{b, c\}$ are not.



The background makes the adjacency symmetric, so one negated literal is enough to say “not neighbours”:

```
% Learning the clique conflict: a chosen set is bad as soon as it holds
% two distinct vertices that are not adjacent. The background makes the
% adjacency symmetric so one literal suffices.
vertex(a). vertex(b). vertex(c).
adj(X,Y) :- edge(X,Y).
adj(X,Y) :- edge(Y,X).

#modeh(violated).
#modeb(in(var(vertex))).
#modeb(not adj(var(vertex), var(vertex))).
#modeb(var(vertex) != var(vertex)).
```

```
#maxv(2).

#pos(ok1, {}, {violated}, { edge(a,b). in(a). in(b). }).
#pos(ok2, {}, {violated}, { edge(a,b). in(a). }).
#pos(bad1, {violated}, {}, { edge(a,b). in(a). in(c). }).
#pos(bad2, {violated}, {}, { edge(a,b). in(b). in(c). }).

#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex33_clique.las
violated :- in(V0), in(V1), not adj(V1,V0), V1 != V0, vertex(V0),
           vertex(V1).
```

Two chosen, distinct, non-adjacent vertices: the clique constraint of Section 6.1, recovered with the comparison mode `var(t) != var(t)` of Section 3.2 doing the “distinct” part.

PITFALL. The learned rule is stable, but **the order of its body literals is not**. Across repeated runs of the same task, the colouring rule above comes back with `edge/2` first or with a `colour/2` literal first, and the pet-owner rule of Example 26 alternates between `eats(V0,V1)` and `eats(V1,V0)` with the two `own/1` literals swapped to match. These are the same rule up to the order of a conjunction and the naming of variables. Compare hypotheses as sets of literals, not as strings, and do not build a regression test that diffs the output character by character.

6.3 Preferences: where FastLAS stops and ILASP starts

The pet-owner puzzle has three animals, and cats eat fish while dogs eat cats. A farmer wants as many animals as possible, and would rather they did not eat one another. In Clingo those are two weak constraints at different priorities. The no-eating one carries the higher priority `@2`, so Clingo satisfies it first and sacrifices animals to it:

```
0 { own(A) } 1 :- animal(A).
:- own(A).[-1@1, A]
:- own(A), own(B), eats(A, B).[1@2, A, B]

$ clingo examples/asp/asp08_petowner.lp
own(fish) own(dog)
Optimization: 0 -2
```

Two animals, no conflicts. FastLAS cannot learn that pair of weak constraints, and cannot learn the choice rule either. What it can learn is the conflict itself, as a hard rule.

Example 26. *Learning the pet-owner conflict*

```
% The pet-owner problem. Clingo scores the conflict
% with a weak constraint; FastLAS learns the conflict itself as a hard
% rule:
% violated :- own(V0), own(V1), eats(V0,V1).
animal(dog). animal(cat). animal(fish).
#modeh(violated).
#modeb(own(var(animal))).
#modeb(eats(var(animal), var(animal))).
#maxv(2).
#pos(ok1, {}, {violated},
      { own(dog). own(fish). eats(cat,fish). eats(dog,cat). }).
```

```

#pos(ok2, {}, {violated}, { own(cat). eats(cat,fish). eats(dog,cat). }).
#pos(bad1, {violated}, {},
    { own(cat). own(fish). eats(cat,fish). eats(dog,cat). }).
#pos(bad2, {violated}, {},
    { own(dog). own(cat). eats(cat,fish). eats(dog,cat). }).
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex21_petowner.las
violated :- own(V0), own(V1), eats(V0,V1), animal(V0), animal(V1).

```

DIFFERENCE FROM ILASP. Preferences are the sharpest line between the two systems. ILASP learns weak constraints directly: you label examples and order them with `#brave_ordering(id, eg1, eg2).` and `#cautious_ordering(...)`, and it returns the `:~` rules that reproduce the ordering. This is what makes it usable for preference learning on data such as the SUSHI rankings of Kamishima (2003), where each observation is one item preferred to another. FastLAS has no ordering examples and no `#modeo`, so an ordering task has to be reformulated as a classification task before FastLAS can take it, as in Example 27.

Example 27. *A SUSHI taste, as a classification task*

The SUSHI data of Kamishima (2003) describes each item by style, major and minor group, oiliness, price and frequency. Rather than an ordering over items, ask which items a user likes, and let the numeric features find their own thresholds with `num_var` (Section 3.2).

```

% A SUSHI-style preference task using the feature encoding (style, major
    group, oiliness, price; cf. Kamishima's SUSHI
% preference data).
% The taste to be recovered: seafood that is oily enough.
% likes(S) :- oiliness(S,0), 0 >= 3, seafood(S).
sushi(s1). sushi(s2). sushi(s3). sushi(s4). sushi(s5). sushi(s6).
oiliness_val(0..4).

#modeh(likes(var(sushi))).
#modeb(oiliness(var(sushi), num_var(oiliness_val))).
#modeb(seafood(var(sushi))).
#modeb(maki(var(sushi))).
#maxv(1).

#pos(e1, {likes(s1)}, {}, { oiliness(s1,4). seafood(s1). }).
#pos(e2, {likes(s2)}, {}, { oiliness(s2,3). seafood(s2). }).
% too lean
#pos(e3, {}, {likes(s3)}, { oiliness(s3,1). seafood(s3). }).
% too lean
#pos(e4, {}, {likes(s4)}, { oiliness(s4,0). seafood(s4). }).
% oily, not seafood
#pos(e5, {}, {likes(s5)}, { oiliness(s5,4). maki(s5). }).
% oily, not seafood
#pos(e6, {}, {likes(s6)}, { oiliness(s6,3). maki(s6). }).
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex22_sushi.las
likes(V0) :- oiliness(V0,V_0_oiliness_val), seafood(V0),
    V_0_oiliness_val >= 3, sushi(V0).

```

Seafood, oily enough. The threshold 3 was never declared: it is the smallest oiliness among the items the user liked, which is as much as the data supports.

7 Case studies

This section works two applications from the FastLAS literature end to end: the CAVIAR event-recognition benchmark, and access-control policy learning, the application FastLAS was originally built for.

7.1 Event recognition in CAVIAR

CAVIAR is a benchmark of CCTV video in which people are tracked frame by frame. The low-level activity of each person (*walking, running, active, inactive, appear, ...*) and their pairwise distances are given; the task is to learn definitions of high-level *composite* events: here, when two people are *meeting*. It is the flagship scalability benchmark in the FastLAS papers, and it exercises almost everything in this guide at once: typed variables, `const` thresholds, weighted examples, and computed inclusions and exclusions.

The task. Events are modelled in the *Event Calculus* (Kowalski and Sergot 1986). A *fluent* such as `meeting(P1,P2)` holds over time; the two predicates we learn, `initiatedAt(F,T)` and `terminatedAt(F,T)`, say when a fluent *starts* and *stops* holding, while the background `happensAt(E,T)` records the low-level events. So FastLAS is learning the *start* and *stop* conditions of a meeting.

The encoding. Two excerpts show the structure. First, the background defines the fluent, derives the distance events `close/further` from raw distances against a threshold, and (using the computed inclusions and exclusions of Section 3.3) *computes* each example's target atoms from a `goal(holdsAt(...))` annotation:

```
fluent(meeting(X,Y)) :- person(X), person(Y).
happensAt(close(Id1,Id2,Th),T) :- dist(Id1,Id2,T,D), dist(Th),
    D <= Th.
happensAt(further(Id1,Id2,Th),T) :- dist(Id1,Id2,T,D), dist(Th),
    D > Th.

inclusion(initiatedAt(A,1)) :- not holdsAt(A,1), goal(holdsAt(A,2)).
exclusion(initiatedAt(A,1)) :- fluent(A), not goal(holdsAt(A,2)).
% (symmetric rules give inclusion/exclusion for terminatedAt)

% the const(dist) candidates
dist(24). dist(25). dist(27). dist(34). dist(40).
```

Second, the mode declarations. Note `const(dist)`, which lets a rule pin a specific distance threshold drawn from those `dist/1` facts, and the trailing `%2/%10` comments the authors use to record each mode's intended recall:

```
#modeh(initiatedAt(meeting(var(person), var(person)), var(time))).
#modeh(terminatedAt(meeting(var(person), var(person)), var(time))).

#modeb(happensAt(active(var(person)), var(time))).           %2
#modeb(happensAt(walking(var(person)), var(time))).           %2
#modeb(happensAt(running(var(person)), var(time))).           %2
```

```
#modeb(not happensAt(walking(var(person)), var(time))). %2
#modeb(happensAt(close(var(person), var(person), const(dist)),
    var(time))). %10
#modeb(happensAt(further(var(person), var(person), const(dist)),
    var(time))). %10
% ... and likewise for inactive, abrupt, appear and disappear
```

Each example is one frame transition. Its context lists who is present and what they are doing, the goal(`holdsAt(meeting(...),2)`) says which meetings should hold next, and the `@100` weight marks it as a (soft) noisy example:

```
#pos(p_22732@100, {}, {}, {
    person(id0). person(id1). person(id2). person(id3).
    goal(holdsAt(meeting(id2,id3),2)).
    happensAt(inactive(id0),1). happensAt(running(id2),1).
    happensAt(walking(id3),1).
    % ... plus the raw distances, e.g. dist(id0,id1,1,175). ...
}).
```

Running it. Read this part as a large-scale worked sketch rather than as a self-contained repository example.

PITFALL. The data fold is not in the repository accompanying this guide: it ships with the FastLAS source distribution, as `FastLAS2/data/non_branching_caviar/fastlas_tasks/train_fold_0.las`. Its example ids are also not unique, so running it as-is stops with `Duplicated example ID`. Copy it into the current directory and make the ids unique first:

```
$ awk '{ if ($0 ~ /^#(pos|neg)\(\/) sub(\/\(/, "(u" NR "_"); print }' \
    train_fold_0.las > caviar.las
```

```
$ FastLAS --opl caviar.las
```

This returns a 13-rule theory. A representative subset (each rule wraps to fit the page):

```
initiatedAt(meeting(V0,V1),V2) :- happensAt(active(V0),V2),
    happensAt(active(V1),V2), happensAt(close(V0,V1,25),V2),
    person(V1), time(V2).
initiatedAt(meeting(V0,V1),V2) :- happensAt(active(V0),V2),
    happensAt(inactive(V1),V2), happensAt(close(V1,V0,24),V2),
    person(V0), person(V1), time(V2).
terminatedAt(meeting(V0,V1),V2) :- happensAt(running(V1),V2),
    person(V1), time(V2).
terminatedAt(meeting(V0,V1),V2) :- happensAt(walking(V0),V2),
    happensAt(inactive(V1),V2), person(V1), time(V2).
```

In English: a meeting *starts* when both people are **active** and **close** within distance 25, or when one is **active** beside an **inactive** companion within 24; it *ends* when one of them starts **running**, or walks off while the other is **inactive**. The 25 and the 24 are `const(dist)` values FastLAS selected from the `dist/1` pool: the payoff of the `const` mechanism from Section 3.2. The thirteen-rule size is stable from run to run; *which* of several equal-cost rules survives is not (the `const(dist)` thresholds in particular move between runs), so treat the listing above as one run's output rather than as the answer. This is the scale, tens of thousands of examples, that motivates FastLAS in the first place.

7.2 Access-control policy learning

Example 13 showed a policy learned from a toy log. The experiments of Law et al. (2020) do the same at full scale, on access logs generated from the *Project Management* case study of Xu and Stoller (2014): requests by users (*subjects*) to act on *resources*, each granted or denied by a ground-truth policy. The task files ship with the FastLAS source distribution, under `FastLAS1/data/policy_learning/files/`; this subsection reproduces the length-scored (`sf_len`) experiment that learns `accept`.

The task. Each logged request is one example. Its context lists the request’s attributes: who is asking (`subject_role`, `subject_projects`, ...), for what (`resource_type`, `resource_project_id`, ...) and to do what (`action_action_type`); thirteen attributes in all. A granted request includes `accept` and a denied one excludes it, exactly as in Example 13; the first of the ten cross-validation folds has 409 grants and 40 denials. The learned rules say when to accept, and a request matched by no rule is denied.

The encoding. The mode declarations give each attribute one `const` body mode, and the `#bias` lines implement the paper’s S_{len} objective, one unit per literal, exactly as in Section 3.4:

```
#modeh(accept).
#modeb(1, subject_role(const(subject_role))).
#modeb(1, resource_type(const(resource_type))).
#modeb(1, action_action_type(const(action_action_type))).
% ... ten more attribute modes ...
#maxv(1).
#bias("penalty(1, head(X)) :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).")
```

One example of each kind, contexts abbreviated:

```
#pos(eg(id0), {accept}, {}, {
  subject_role(employee).      resource_type(task).
  action_action_type(read).    % ... eight more attributes ...
}).
#pos(eg(id409), {}, {accept}, {
  subject_role(projectleader). resource_type(task).
  action_action_type(approve). % ...
}).
```

PITFALL. The files are in FastLAS 1 syntax, and declare their constant pools with a directive that FastLAS 2 no longer accepts: `#constant(subject_role, employee).` and so on. In FastLAS 2 a pool is ordinary background facts (Section 3.2), but do *not* port `#constant(t, v).` to `t(v).` when `t` is also a context predicate, as it is here. The pool facts would then hold in *every* example, so every candidate body would be true everywhere, no rule could avoid the denials, and the task would come back **UNSATISFIABLE**. Give each pool a fresh predicate instead:

```
$ sed -E \
-e 's/^#constant\((([a-z_]+), *([A-Za-z0-9_]+\))\)\.\/\1_t(\2).\/' \
-e 's/const\((([a-z_]+\))/const(\1_t)/g' \
PM_full_fold0_accept_sf_len_train.las > pm_fold0_len.las
```

The converted fold is `examples/pm_fold0_len.las` in the repository accompanying this guide.

```
$ FastLAS --opl examples/pm_fold0_len.las
```

This takes about twenty seconds on the machine of Section 5.1 and returns a theory of around 55 rules (53 to 59 across folds and runs). Some are genuinely general, others memorise a single user or resource:

```
accept :- action_action_type(setschedule), subject_role(planner).
accept :- resource_is_properietary(true), subject_role(employee).
accept :- subject_user_id(pjl3__0).
accept :- resource_dept_id(dept__4).
```

The distribution also ships each held-out fold as a clingo program that tries a theory on the unseen requests and classifies each as `tp`, `fp`, `tn` or `fn`. Save the learned rules to `hyp.lp` and count:

```
$ clingo --enum-mode=brave --quiet=1 -n 0 hyp.lp \
    PM_full_fold_0_accept_test.las
```

Repeating train-and-count over the ten folds gives precision 0.954, recall 0.905 and F_1 0.929; the published results for this experiment (Table 4 of Law et al. (2020)) are 0.951, 0.905 and 0.928. The port reproduces the paper.

The memorised rules are no accident. Every example above is hard, so each training grant that no general rule explains must be bought a rule of its own. The paper's objective, however, is S_{len} *plus* the penalty of uncovered examples, and weighted examples (Example 9) express exactly that: `examples/pm_fold0_len_pen.las` differs only in an `@1` on every example id, making an outlier cost one unit to leave uncovered but at least two to memorise. The same command then returns around nine rules:

```
accept :- action_action_type(read).
accept :- subject_role(employee).
accept :- resource_type(budget), subject_role(manager).
accept :- action_action_type(setcost), subject_role(accountant).
```

a policy readable at a glance (anyone may read, employees may act, managers handle budgets, accountants set costs, ...), at precision 0.950, recall 1.000 and F_1 0.974 over the ten folds. As with CAVIAR, *which* equal-cost rules survive changes from run to run; the sizes and the scores are what stay stable.

8 Running FastLAS and reading its output

8.1 Command-line flags

One of `--opl` or `--nopl` must always appear; the rest are occasional. These are the ones used in this manual.

<code>--opl / --nopl</code>	choose the algorithm (one is required)
<code>--version</code>	print the version
<code>--help</code>	list all options
<code>--debug</code>	verbose trace of the algorithm's phases
<code>--force-safety</code>	enforce the safety constraint on learned rules
<code>--threads N</code>	parallelism; the default is 8
<code>--timeout T</code>	time limit for the final solving stage
<code>--score-only</code>	print only the score of the solution
<code>--space-size</code>	prepend the size of the kept candidate space
<code>--output-solve-program</code>	print the final ASP program and stop
<code>--max-conditions N</code>	bounded numeric variables per rule (Section 3.2)
<code>--write-cache F / --read-cache F</code>	save/reuse the computed state (Section 5.4)

8.2 Interpreting the output

FastLAS is terse: it prints the hypothesis and nothing else. Five things can come back, and it is worth knowing what each one means before you start changing your task.

- **One or more rules:** the learned hypothesis. Learned variables are named `V0`, `V1`, ..., and type atoms may be appended for safety. The rule is stable but *the order of its body literals is not*: repeated runs of the same task may print the same conjunction in a different order, and may name the variables differently. Compare hypotheses as sets of literals, not as strings.
- **(blank):** the empty hypothesis is optimal, e.g. all examples are weighted and cheaper to leave uncovered. Note that when the background already entails an example, `--nopl` prints nothing but `--opl` still emits a redundant rule.
- **UNSATISFIABLE:** no hypothesis in the space explains the (hard) examples. Common causes: a needed rule is outside the modes; `#maxv` too low; the task is non-observational but you used `--opl`; the scenario was left out of the context. A task containing a `#neg` also needs `--nopl` (Section 3.3). If *every* task reports this, including a known-good one such as `examples/ex01_cycle.las`, check the installation first (Section 1.3).
- **Two hypotheses, each with a JSON summary:** your task contains a `#predict`; the output is a satisfying/not-satisfying pair (Section 5.3).
- **Unknown token: '#' or syntax error ...:** a parse error; check for unsupported directives such as `#constant`, or a missing context slot on `#neg`.

8.3 Seeing the solve program: `--output-solve-program`

`--output-solve-program` runs the FastLAS algorithm up to the final search, then prints the assembled ASP optimisation program to stdout and stops (no solving). It is the best window into a task that is unexpectedly UNSATISFIABLE or slow. Read it, or run it in Clingo yourself:

```
$ FastLAS --opl --output-solve-program examples/ex01_cycle.las

% d1
% d1 : d1
disj(0) :- in_h(0).
```

```

n_cov(d1) :- not disj(0).
n_cov(d1) :- disj(1).
n_cov(d1) :- #true, n_cov(d1).
:- n_cov(d1).
% d2
% d2 : d2
n_cov(d2) :- disj(1).
n_cov(d2) :- #true, n_cov(d2).
:- n_cov(d2).
0 {in_h(0)} 1.
rule_score(0, 2).
% ... plus the :-~ weak constraints, #show, and the lua #script block

```

Each `in_h(i)` is a candidate rule the search may switch on; `disj(i)` records that the *i*th of an example's coverage conditions is met; `n_cov(eg)` marks an example the current hypothesis fails to cover, and the `:- n_cov(eg)` constraints force coverage. The `% eg : eg` comments and the vacuous `#true` self-loops are bookkeeping, safe to skip, and the index *i* is assigned afresh on each run, so a rerun may print `in_h(1)` where this transcript shows `in_h(0)`. One caveat: on a `#predict` task only the prediction-*satisfying* program is emitted.

Diagnosing an UNSATISFIABLE task. The flag earns its keep when a task fails and you cannot see why. Consider this one, which asks for a relation between two nodes but permits only one variable per rule:

```

% Deliberately UNSATISFIABLE: the target needs two variables, but
% #maxv(1) allows only one, so no candidate rule can cover p1.
% Section 8.3 diagnoses it.
node(a). node(b). edge(a,b).
#modeh(rel(var(node), var(node))).
#modeb(edge(var(node), var(node))).
#maxv(1).
#pos(p1, {rel(a,b)}, {}, {}).
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex30_unsat_diagnosis.las
UNSATISFIABLE

```

Nothing there says what went wrong. The solve program does:

```

$ FastLAS --opl --output-solve-program examples/ex30_unsat_diagnosis.las
% p1
% p1 : p1
n_cov(p1) :- not disj(0).
n_cov(p1) :- disj(0).
n_cov(p1) :- #true, n_cov(p1).
:- n_cov(p1).

```

Skip the comment and the vacuous self-loop and read the remaining three lines together. The first two say that `p1` is uncovered when `disj(0)` fails *and* when it holds, so `n_cov(p1)` is derivable no matter what the search chooses; the third then forbids exactly that. The program is contradictory on its own, which is the signature of an example no candidate rule can satisfy. `--space-size` confirms it from the other direction:

```

$ FastLAS --opl --space-size examples/ex30_unsat_diagnosis.las

```

```
% SPACE SIZE: 0
UNSATISFIABLE
```

There are no candidates at all, because `rel(var(node), var(node))` needs two variables and `#maxv(1)` allows one. Raise it to `#maxv(2)` and the task is solved. The habit worth forming: when a task is `UNSATISFIABLE`, look at `--space-size` first, since a space of 0 means the fault is in the mode declarations rather than in the examples.

9 FastLAS vs. ILASP: a porting cheat-sheet

If you are moving a task between the two systems, most of it carries over unchanged: the example blocks (`#pos/#neg` with `@penalty`), `#maxv`, and the `#modeh/#modeb` predicate skeletons are written the same way. This section covers what does not: whether a given task is worth porting at all, what the line-by-line changes are, how the directives correspond, and what to do when a port fails.

9.1 Deciding whether the task ports at all

Before touching the syntax, ask what shape the answer has. If the thing you want to learn is a set of *definite conclusions*, a definition or a constraint, FastLAS will do it and will scale further. If what you want is a *space* of possibilities or an *ordering* over them, the task is ILASP's, and no amount of rewriting will move it.

If the target is...	then
a definition, <code>p :- q, not r.</code>	port it; this is FastLAS's home ground
a constraint, "never both"	port it as a <code>violated</code> head (Section 5.2)
a numeric threshold	port it, and use <code>num_var</code> (Section 3.2)
a domain-specific notion of "best"	port it to FastLAS; ILASP cannot express it
a choice, "either heads or tails"	stay in ILASP (Exercise 22)
a preference or ranking	stay in ILASP (Section 6.3)
a recursive definition	stay in ILASP, or move the recursion into the background

9.2 A task ported line by line

Here are the opening lines of the same learning problem in both systems, recognising an animal that flies. Only the head of each file is shown, enough to see the four differences; a task that actually learns the flightless exception needs the counter-example of Example 7 as well. On the left, ILASP; on the right, FastLAS.

% ILASP	% FastLAS
#modeh(flies(var(animal))).	#modeh(flies(var(animal))).
#modeb(winged(var(animal))).	#modeb(winged(var(animal))).
#modeb(flightless(var(animal))).	#modeb(not flightless(var(animal))).
#maxv(1).	#maxv(1).
#constant(animal, eagle).	animal(eagle). % a fact
#pos(p1, {flies(eagle)}, {}).	#pos(p1, {flies(eagle)}, {},
	{ winged(eagle). }).

Three changed lines, and a fourth difference that does not show up in a listing this small:

- **Negation is explicit.** ILASP generates the negated version of each `#modeb` for you (unless you write `(positive)`); FastLAS does not, so declare `#modeb(not p(...))` for every negated literal you want available.
- **Constants are facts.** `#constant(t, v).` is not a FastLAS directive and will stop the parser with **Unknown token: '#'**. Write `t(v).` in the background or, better, in the context of the examples that need it.
- **Examples carry their scenario.** ILASP examples often lean on one global background; FastLAS expects the per-example data in the fourth slot. This is the single most common cause of a ported task returning **UNSATISFIABLE** while nothing looks wrong.
- **#bias changes meaning.** In ILASP it prunes the search space; in FastLAS it assigns cost. A ported `#bias` is almost never still correct, and a silently wrong one is worse than a parse error.

9.3 How the directives correspond

With the shape of a port settled, this is the reference table: what each system calls a given feature, and where FastLAS simply has no counterpart.

Concept	ILASP	FastLAS 2.2.0
Head declarations	<code>#modeh</code> , <code>#modeha</code> (choice)	<code>#modeh</code> only
Body declarations	<code>#modeb</code> , auto-NAF	<code>#modeb</code> ; explicit <code>not</code> decls
Condition/aggregates	<code>#modec</code>	not supported
Weak-constraint bias	<code>#modeo</code> , <code>#weight</code> , <code>#maxp</code>	not supported
Mode options	<code>(positive)</code> , <code>(symmetric)</code> ...	not supported
Constants	<code>#constant(t,v)</code>	<code>t(v)</code> facts in contexts
Max variables	<code>#maxv</code>	<code>#maxv</code> (same)
Hypothesis-size cap	<code>#max_penalty</code> (default 15)	<code>#max_penalty</code> ; accepted, no default cap
<code>#bias("...")</code>	pruning constraints	scoring (<code>penalty/in_body</code>)
Objective	length + penalties (fixed)	user-defined ; no objective unless you write one
Ordering/preferences	<code>#brave_ordering</code> , <code>#cautious_ordering</code>	not supported
Recursion, choice	yes	no
CLI	ILASP <code>--version=N</code> task	FastLAS <code>--opl/--nopl</code> task

9.4 What to do when a port fails

Most failures announce themselves as a parse error or an unexpected **UNSATISFIABLE**. The table maps the message you see to the thing that usually caused it.

Symptom	Usual cause
Unknown token: '#'	#constant, #modeo, #modec, or a Clingo aggregate
unexpected T_BASIC_SYMBOL	#modeha, i.e. an ILASP choice head
syntax error, unexpected T_COLON	a conditional literal $p : q$, or a weak constraint $:~$
UNSATISFIABLE	the scenario is not in the context slot; or a negated body mode was never declared; or #maxv is too low
a rule that is too general	no example rules out the shorter candidate (Example 23)
the wrong rule of several ties	the objective, not the data: add a #bias (Section 3.4)

When none of these applies, `--output-solve-program` (Section 8) prints the exact ASP that FastLAS is about to solve, which is usually enough to see which example is doing the damage.

10 Quick reference

The user-facing directives of FastLAS 2.2.0:

#modeh(A).	head atom A allowed in learned rule heads
#modeh(false).	learn a constraint-style verifier with head false
#modeb(A). / #modeb(not A).	body literal allowed (optionally negated)
#modeb(N, A).	... with recall bound N
#maxv(N).	max distinct variables per rule
#max_penalty(N).	cap on hypothesis cost
#bias("...").	scoring program (costs); see Section 3.4
#final_bias("...").	second-stage scoring
#pos(id,{I},{E},{C}).	positive example (3-slot form also allowed)
#neg(id,{I},{E},{C}).	negative example (context slot mandatory)
#pos(id@W, ...).	weighted (noisy) example, penalty W
#predict(id,{I},{E},{C}).	prediction query (advanced)

Inside modes: `var(t)` typed variable, `const(t)` typed constant, `num_var(t)` numeric variable that learns $\geq/\leq$ bounds (Section 3.2). Comments: %.

The example files. Every numbered example in this guide, except where the text explicitly points to an external artifact, and every exercise solution of Appendix A, is a task file in the accompanying repository:

https://github.com/dasaro/fastlas_manual

The examples are under `examples/` and the solutions under `examples/solutions/`. Clone it and each one runs directly against FastLAS 2.2.0, e.g.

```
$ git clone https://github.com/dasaro/fastlas_manual.git
$ cd fastlas_manual
$ FastLAS --opl examples/ex03_flies_exception.las
```

11 Exercises

The exercises below run through the whole guide, from a first task to the case-study features. Every exercise has a solution: selected ones are worked in Appendix A, and the rest are tested task files in `examples/solutions/` of the repository (Section 10),

named where the appendix points to them. Every reference solution was checked against FastLAS 2.2.0. Try yours, then compare the learned hypothesis, not just the file. Unless stated otherwise, run with `--opl`.

Warm-up (the first task and the anatomy of a file)

Exercise 1 (a two-hop rule). A family is given by `parent/2` facts: `parent(ann,bob)`, `parent(bob,carl)`, `parent(carl,dee)`. Declare modes (`head grandparent`, `body parent`, `#maxv(3)`) and give one example whose *include* set holds the true grandparent pairs and whose *exclude* set holds the near-misses (parents and great-grandparents), so that FastLAS is forced to learn the two-parent join rather than a shorter over-general rule. Starter/solution: `examples/solutions/sol01_grandparent.las`.

Exercise 2 (fix a broken task). The task below returns UNSATISFIABLE: no rule over the given modes can make winged `ostrich` *not* fly while eagles and sparrows do. Add exactly one line to fix it.

```
animal(eagle). animal(ostrich). animal(sparrow).
#modeh(flies(var(animal))).
#modeb(winged(var(animal))).
#maxv(1).
#pos(p1, {flies(eagle)}, {}, { winged(eagle). }).
#pos(p2, {flies(sparrow)}, {}, { winged(sparrow). }).
#pos(p3, {}, {flies(ostrich)},
      { winged(ostrich). flightless(ostrich). }).
```

Solution: `examples/solutions/sol02_exception.las`.

Exercise 3 (pick the constant). With `#modeb(chosen(const(colour)))` and colours `red`, `blue`, `green`, learn which single colour makes `sel` true, given one accepted context `chosen(blue)` and two rejected ones (`chosen(red)`, `chosen(green)`). Solution: `examples/solutions/sol03_const.las`.

Exercise 4 (steer the scoring). Both `sel :- a.` and `sel :- b.` explain the data below equally well, and plain length returns `sel :- a..` Change *only* the `#bias` so FastLAS returns `sel :- b.` instead.

```
#modeh(sel). #modeb(a). #modeb(b).
#pos(p1, {sel}, {}, { a. b. }).
#pos(p2, {}, {sel}, { c. }).
#bias("penalty(1, body(a)) :- in_body(a).").
#bias("penalty(1, body(b)) :- in_body(b).").
```

Solution: `examples/solutions/sol04_bias.las`.

Intermediate (modes, algorithms and effective programs)

Exercise 5 (a learned threshold). Using `num_var`, learn `adult` from `age/2` observations: positives at ages 20 and 40, negatives at 10 and 17. Keep `#maxv(1)`. Which bound does FastLAS synthesise, and why that number rather than, say, `>= 18`? Solution: `examples/solutions/sol05_numvar.las`.

Exercise 6 (two thresholds in one rule). Cars are ok exactly when *both* a speed band and a weight band hold, and the data is arranged so neither attribute alone separates the classes. The default run prints UNSATISFIABLE. Which flag lets FastLAS bound two numeric variables in a single rule, and what rule results? Solution: `examples/solutions/sol06_numvar_multi.las`.

Exercise 7 (choose the algorithm). In the task below `permitted` never appears in any example; it only influences the observed `fault` through the background. Explain why `--opl` returns UNSATISFIABLE, and run it so it succeeds.

```

fault :- did(A), not permitted(A), act(A).
#modeh(permitted(var(act))).
#modeb(authorised(var(act))).
#maxv(1).
#pos(e1, {}, {fault}, { act(a1). did(a1). authorised(a1). }).
#pos(e2, {fault}, {}, { act(a2). did(a2). }).

```

Solution: `examples/solutions/sol07_nopl.las`.

Exercise 8 (translate a Clingo idiom). FastLAS rejects the aggregate below with `Unknown token: '#'`. Rewrite it as an auxiliary rule in FastLAS's dialect, then complete a task that learns `safe(X) :- not attacked(X)`.

```

attacked(X) :- #count{ Y : att(Y,X) } >= 1.    % illegal in FastLAS

```

Solution: `examples/solutions/sol08_dialect.las`.

Exercise 9 (ask a prediction). Take the cycle/rain task (`cycle :- not rain. is learnable`) and add a `#predict` asking whether “cycle holds on a dry day” (`cycle` included, empty context). From the two-hypothesis output, is that conclusion *entailed*? Solution: `examples/solutions/sol09_predict.las`.

Advanced (scoring, limits, streams and the case studies)

Exercise 10 (reframe a recursive target). Take the argumentation framework with arguments `a, b, c` and attacks `att(a,b)` and `att(b,c)` (Section 5.2 explains the setting). Build a verifier task for it: each context carries the framework and one labelling, conflict-free labellings (such as `a` and `c` in, `b` out) *exclude* violated, and labellings with an attack between two in arguments *include* it. Check that FastLAS learns the same conflict-freeness rule as Example 19, and say why learning `in/1` directly would not have worked. Solution: `examples/solutions/sol10_verifier.las`.

Exercise 11 (score a feature once). The only rule that fits the data is `p :- not a, not b` (two negations). Using `#final_bias` with an `intermediate/1` feature, charge a *single* unit for using negation-as-failure, regardless of how many negated literals appear. Confirm with `--score-only` that the winning rule scores 1, not 2. Solution: `examples/solutions/sol11_finalbias.las`.

Exercise 12 (learn from a stream). Window 1 holds one example forcing `p`; window 2 adds a contradicting example. With `--write-cache/--read-cache`, show that the incremental result after window 2 equals the batch result on both windows, and differs from window 1's answer. Solution files: `examples/solutions/sol12_window1.las`, `sol12_window2.las`, `sol12_batch.las`.

Exercise 13 (change the objective, change the policy). The access log of Example 13 yields `accept :- subject_role(manager). under plain length.` Add one domain `#bias` that charges for role conditions so FastLAS instead prefers the clearance-based policy `accept :- subject_clearance(high).` Solution: `examples/solutions/sol13_policy.las`.

From the classic problems (Section 6)

Exercise 14 (the other half of the cover). Example 23 learns `covered(X,Y) :- edge(X,Y), in(X).` The Clingo program also has the symmetric rule, covering an edge by its *second* endpoint. Change the examples so that FastLAS learns that one instead, and keep the task down to the four examples used before. Solution: `examples/solutions/sol14_cover_other.las`.

Exercise 15 (which streets are dangerous). The path problem of the course notes labels each street with a danger score, `dangerous(a,b,4)`, `dangerous(a,c,1)`, `dangerous(b,c,3)`, `dangerous(c,d,1)`, `dangerous(b,d,1)`. Suppose the two streets scoring 3 or more, `a,b` and `b,c`, are considered *risky* and the three scoring 1 are not. Use `num_var` to learn the threshold rather than writing it, and say why the bound comes out where it does. Solution: `examples/solutions/sol15_dangerous.las`.

With pen and paper

These three want no computer at all, though you can check each answer with one command afterwards.

Exercise 16 (enumerate a search space). Take this mode bias, which is propositional, so no variables are involved:

```
#modeh(p).
#modeb(a).
#modeb(b).
```

Write out, by hand, every rule that FastLAS may consider. Then answer a second question: ILASP's `-s` flag reports *nine* rules with head `p` for the same three lines. Why more, and what would you have to add to the FastLAS bias to obtain the same nine?

Exercise 17 (compute a score). The task below learns `q :- r.` Before running anything, work out what `--score-only` will print.

```
#modeh(q).
#modeb(r).
#modeb(s).
#pos(e1, {q}, {}, { r. s. }).
#pos(e2, {}, {q}, { s. }).
#bias("penalty(2, head) :- in_head(X).").
#bias("penalty(3, body(X)) :- in_body(X).").
```

Exercise 18 (why is nothing the best answer?). Take Example 9 and swap its two penalties, so that `e1` costs 1 to abandon and `e2` costs 3. FastLAS then returns the empty hypothesis. Explain, in terms of the objective of Section 2.3, why learning *nothing* is optimal here, and say what the score is.

At the keyboard

Exercise 19 (negative examples). A car is roadworthy unless it has failed its inspection. You are given `car(c1). car(c2). failed(c2).`, a head mode `roadworthy(var(car))`, body modes `car(var(car))` and not `failed(var(car))`, `#maxv(1)`, and the single positive example `#pos(p1, {roadworthy(c1)}, {}, {})`. As it stands, FastLAS answers `roadworthy(V0) :- car(V0).`, which would make the failed car roadworthy too. Add one negative example that rules this out, and say which algorithm flag you now need. Solution: `examples/solutions/sol17_neg_roadworthy.las`.

Exercise 20 (what a penalty buys). Take the two contradictory examples of Example 9, with `e2` fixed at penalty 1. Raise the penalty on `e1` one step at a time and find the smallest value at which FastLAS stops returning the empty hypothesis and commits to `p`. Explain the number you find by adding up the two costs it is comparing. Solution: `examples/solutions/sol18_weights.las`.

Exercise 21 (build a task only `--nopl` can solve). The exercises so far have handed you the task. Now write one. Construct, from scratch, a FastLAS task that returns a hypothesis under `--nopl` and UNSATISFIABLE under `--opl`, using no more than a handful of lines. State which property of your task is responsible, and check both runs. There is more than one way to do it. Solution: `examples/ex31_nopl_only.las`.

Exercise 22 (build a task only ILASP can solve). Harder, and worth the effort. Construct a learning task that ILASP solves but FastLAS cannot, where the obstacle is the *shape of the hypothesis* rather than a syntactic detail of the input. Write the task in both dialects, run ILASP `--version=4` on one and FastLAS on the other, and explain in a sentence what it is about the rules ILASP returns that FastLAS will never produce. Section 9.1 says where to look. Solution: `examples/ex32_coins_ilasp.las` with `examples/solutions/sol16_coins.las`.

Exercise 23 (a task FastLAS cannot do). Three coins are flipped and the outcome of each is observed, as in the introductory ILASP task of the course notes, one flip being `#pos(f1, {heads(c1), tails(c2), heads(c3)}, {tails(c1), heads(c2), tails(c3)}, {})`. ILASP learns the pair `heads(V) :- not tails(V), coin(V).` and `tails(V) :- not heads(V), coin(V).` Pose the same task to FastLAS, with both predicates as heads and their negations as body literals. What comes back, and what is it about this particular hypothesis that FastLAS will not produce? Solution: `examples/solutions/sol16_coins.las`.

Appendix A Solutions to the exercises

Every hypothesis below is the verbatim output of FastLAS 2.2.0 on the corresponding file in `examples/solutions/`. FastLAS appends the type atoms (`person(V0)`, `car(V0)`, ...) that its `var(t)` placeholders stand for; they are part of the learned rule.

This appendix is structured in three layers: selected worked solutions from the earlier tutorial sections, a complete worked set for the *With pen and paper* exercises, and a complete worked set for the *At the keyboard* exercises.

Selected earlier solutions in task-file form

The remaining early solutions are mechanical, and the tested task files carry them: each of those exercises names its file, and all of them live in `examples/solutions/`.

Solution 5. FastLAS `--opl examples/solutions/sol05_numvar.las` learns a one-sided lower bound:

```
adult(V0) :- age(V0,V_0_age_val), V_0_age_val >= 20, person(V0).
```

FastLAS synthesises `>= 20`, the smallest *observed* positive value, not the “textbook” 18: it can only propose bounds at values that appear in the data. Because every negative sits below the positives, the bound is one-sided.

Solution 6. The default caps a rule at one bounded numeric variable, so no single rule can bound both speed and weight and the task is UNSATISFIABLE. Raise the limit with `--max-conditions 2`:

```
$ FastLAS --opl --max-conditions 2 \
  examples/solutions/sol06_numvar_multi.las

ok(V0) :- speed(V0,V_0_speed_val), weight(V0,V_0_weight_val),
          V_0_weight_val >= 1400, V_0_speed_val >= 70,
          V_0_weight_val <= 1500, V_0_speed_val <= 80, car(V0).
```

(`--num-var-count` would *not* help here: that flag adds numeric slots per type, while the gate on two *different* bounded quantities in one rule is `--max-conditions`.)

Solution 7. `permitted` is *non-observational*: it appears in no example, so the OPL algorithm has nothing to generalise from and returns UNSATISFIABLE. FastNonOPL abduces it:

```
$ FastLAS --nopl examples/solutions/sol07_nopl.las

permitted(V0) :- authorised(V0), act(V0).
```

Solution 9. The prediction is **entailed**. FastLAS prints two hypotheses: the one *satisfying* the query is the normal optimum cycle `:- not rain.`, while the one *not satisfying* it comes back UNSATISFIABLE: there is no optimal theory that refutes the query, so it must hold.

```
% Optimal hypothesis satisfying the prediction:
cycle :- not rain.
% Optimal hypothesis not satisfying the prediction:
UNSATISFIABLE
```

Solution 11. The `#bias` builds one aggregate feature, `intermediate(naf)`, that is on as soon as the rule uses any negation; `#final_bias` then charges for the feature, not per literal:

```
#bias("intermediate(naf) :- in_body(neg(X)).").
#final_bias("penalty(1, naf) :- intermediate(naf).").
```

FastLAS `--opl examples/solutions/sol11_finalbias.las` learns `p :- not b, not a.`, and FastLAS `--opl --score-only examples/solutions/sol11_finalbias.las` prints 1: one charge for the whole rule, though it has two negations.

Solution 15. FastLAS `--opl examples/solutions/sol15_dangerous.las` learns a lower bound on the danger score:

```
risky(V0,V1) :- dangerous(V0,V1,V_0_danger_val), V_0_danger_val >= 3,
               place(V0), place(V1).
```

The bound is 3 because that is the smallest score among the streets labelled risky. Any threshold in $(1, 3]$ separates this data equally well, and FastLAS can only propose bounds at values that actually occur, so it takes the lowest positive one.

Worked solutions: With pen and paper

Solution 16. Four rules: `p.`, `p :- a.`, `p :- b.` and `p :- a, b.` Every subset of the two declared body literals, with `p` as the head. ILASP reports nine because it generates the negated form of each `#modeb` for you, so its body literals are `a`, `b`, `not a` and `not b`, which yield five further rules (`p :- not a.`, `p :- not b.`, `p :- a, not b.`, `p :- b, not a.` and `p :- not a, not b.`). In FastLAS negation is opt-in, so to get the same nine you would add `#modeb(not a).` and `#modeb(not b).`

Solution 17. 5. The bias charges 2 for the head and 3 for each body literal, and the learned rule `q :- r.` has one of each, so $2 + 3 = 5$. Both examples are hard and both are covered, so nothing is added for uncovered examples.

Solution 18. The score is 1. With the weights swapped, FastLAS compares two options. Learning `p.` covers `e1` but abandons `e2`, costing 1 for the rule plus 3 for the abandoned example, so 4. Learning nothing covers `e2`, since with no rule `p` never holds, and abandons `e1` for its penalty of 1; the empty hypothesis itself costs nothing, so the total is 1. The second is cheaper, and an empty answer is FastLAS telling you that on this data, at these prices, no rule is worth its cost.

Worked solutions: At the keyboard

Solution 19. Add `#neg(n1, {roadworthy(c2)}, {}, {})`. A negative example demands that *no* answer set makes `c2` roadworthy, which the over-general rule violates, so the exception is forced into the body:

```
roadworthy(V0) :- not failed(V0), car(V0).
```

Because the task now contains a `#neg`, it must be run with `--nopl`.

Solution 20. The answer flips at 3. FastLAS is comparing two options: learn `p.` and abandon `e2`, which costs $1 + 1 = 2$, or learn nothing and abandon `e1`, which costs whatever penalty `e1` carries. The rule wins only once that penalty is strictly greater than 2. At `e1@2` the two options tie at 2 and either answer may come back, on different runs of the same file; from `e1@3` the rule wins outright, with a score of 2.

Solution 21. Make the predicate you are learning one that no example mentions. In the task below only `alarm` is ever observed; `suppressed` is never in an inclusion or an exclusion, and reaches the observations only through the background rule.

```
% Solvable under --nopl, UNSATISFIABLE under --opl. The target predicate
% "suppressed" appears in no example: only "alarm" is ever observed, and
% suppressed reaches it through the background rule.
```

```

alarm :- fault, not suppressed.

#modeh(suppressed).
#modeb(safe).

#pos(e1, {}, {alarm}, { fault. safe. }).
#pos(e2, {alarm}, {}, { fault. }).

#bias("penalty(1, head) :- in_head(X).").
#bias("penalty(1, body(X)) :- in_body(X).").

$ FastLAS --opl examples/ex31_nopl_only.las
UNSATISFIABLE
$ FastLAS --nopl examples/ex31_nopl_only.las
suppressed :- safe.

```

This is exactly the non-observational case of Section 4: `--opl` has nothing to generalise from, because the target never appears in the data it is given, while `--nopl` abduces what `suppressed` would have to be. A second, quite different construction also works: put a `#neg` in the task, since those need `--nopl` too (Section 3.3).

Solution 22. Ask for a hypothesis that describes a *space* of possibilities rather than one conclusion. The three-coin task does it: each coin lands heads or tails, and no deterministic set of rules can produce both observed flips. In ILASP:

```

% An ILASP task, not a FastLAS one. Run it with
% ILASP --version=4 <file>
% Three coins are flipped twice. The only hypothesis covering both flips
% is a pair of rules forming an even loop through negation, which ILASP
% builds and FastLAS does not.
% The FastLAS rendering of the same task,
% examples/solutions/sol16_coins.las, returns UNSATISFIABLE.
coin(c1). coin(c2). coin(c3).

#modeh(heads(var(coin))).
#modeh(tails(var(coin))).
#modeb(heads(var(coin))).
#modeb(tails(var(coin))).
#modeb(coin(var(coin))).
#maxv(1).

#pos({heads(c1), tails(c2), heads(c3)},
    {tails(c1), heads(c2), tails(c3)}).
#pos({heads(c1), heads(c2), tails(c3)},
    {tails(c1), tails(c2), heads(c3)}).

$ ILASP --version=4 examples/ex32_coins_ilasp.las
tails(V1) :- coin(V1); not heads(V1).
heads(V1) :- coin(V1); not tails(V1).

```

Those two rules are an *even loop through negation*: each holds exactly when the other fails, so the program has two answer sets per coin, and the flips are covered by choosing between them. The same task posed to FastLAS is `examples/solutions/sol16_coins.las`, and it answers UNSATISFIABLE under both algorithms:

```

$ FastLAS --opl examples/solutions/sol16_coins.las

```

```

UNSATISFIABLE
$ FastLAS --nop1 examples/solutions/sol16_coins.las
UNSATISFIABLE

```

FastLAS does not construct such loops, and this is the boundary rather than an accident of the encoding: weak constraints and choice rules are out of reach for the same reason (Section 6.3). If the answer you want is a space of possibilities, the task is ILASP's.

Solution 23. FastLAS answers `UNSATISFIABLE`, under `--op1` and `--nop1` alike, and it does so even if you keep only the first flip. The task gives every coin the same description: the contexts are empty, the background says only `coin(c1). coin(c2). coin(c3).`, and so nothing distinguishes one coin from another. Any hypothesis FastLAS builds is therefore a set of rules that treats all three coins alike, and its answer set makes `heads` true for all of them or for none, which no example accepts. Covering the observation needs a hypothesis with *several* answer sets, one per way the coins could land, and the pair ILASP finds is exactly that: an even loop through negation, in which `heads(V)` holds when `tails(V)` does not and vice versa. FastLAS does not construct such loops. This is the same boundary as the choice rules and weak constraints of Section 6.3, and it is a good instinct to develop: if the answer you want is a *space* of possibilities rather than one definite conclusion, the task belongs to ILASP.

Acknowledgements and statements

This material grew out of a series of PhD-level lectures on ILASP and FastLAS given by the author, and reuses the slides, exercises and lecture notes written for them, in particular those from the intensive course *Logic Programming and Explainable AI* taught at the IECS Doctoral School, University of Trento, in 2023.

It is an unofficial guide written from a user, for the users. The author is not affiliated with ILASP LTD, has never participated in the development of ILASP or FastLAS, and speaks here only as an external user, teacher and document author.

Artificial intelligence was used to reorganise that material into these notes, and for editing, proofreading, and running and re-running the examples. All content was subsequently checked by hand by the author, who is responsible for any error that remains. Some will certainly have escaped both the machine and the author, and corrections are genuinely welcome: please write to `fabioaurelio.dasaro@univr.it` or `fabio.d'asaro.14@ucl.ac.uk`.

References

- DUNG, P. M. 1995. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n -person games. *Artificial Intelligence* 77, 2, 321–357.
- GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.
- KAMISHIMA, T. 2003. SUSHI preference data sets. <http://www.kamishima.net/sushi/>.
- KOWALSKI, R. AND SERGOT, M. 1986. A logic-based calculus of events. *New Generation Computing* 4, 1, 67–95.
- LAW, M. 2018. Inductive learning of answer set programs. Ph.D. thesis, Imperial College London.

- LAW, M., BRODA, K., AND RUSSO, A. 2022. Search space expansion for efficient incremental inductive logic programming from streamed data. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2697–2704.
- LAW, M., RUSSO, A., BERTINO, E., BRODA, K., AND LOBO, J. 2020. FastLAS: Scalable inductive logic programming incorporating domain-specific optimisation criteria. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. Vol. 34. 2877–2885.
- LAW, M., RUSSO, A., AND BRODA, K. 2014. Inductive learning of answer set programs. In *Logics in Artificial Intelligence (JELIA)*, LNCS 8761. 311–325.
- LAW, M., RUSSO, A., AND BRODA, K. 2020. The ILASP system for inductive learning of answer set programs. *The Association for Logic Programming Newsletter*.
- LAW, M., RUSSO, A., BRODA, K., AND BERTINO, E. 2021. Scalable non-observational predicate learning in ASP. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1936–1943.
- LIFSCHITZ, V. 2019. *Answer Set Programming*. Springer.
- XU, Z. AND STOLLER, S. D. 2014. Mining attribute-based access control policies from logs. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 276–291.