LOAF.

SPECIFICATION

LOAF

A language of manifests: syntax, semantics, vocabulary and admitted rules.

Version 0.1 · 2026-10-30 · DRAFT

OCamlPro SAS — forge.ocamlpro.com/OSS/loaf — CC-BY-4.0

Contents

1 Purpose . . . 2
1.1 Principles . . . 2
1.2 Example . . . 2
2 Lexicon and grammar . . . 5
2.1 Lexicon . . . 5
2.2 Grammar . . . 5
3 Semantics . . . 7
3.1 Terms and atoms . . . 7
3.2 Elaboration . . . 7
3.3 Model . . . 11
3.4 Conformance . . . 12
4 Vocabulary and typing . . . 13
4.1 Sorts . . . 13
4.2 Declarations . . . 14
4.3 Open and closed predicates . . . 14
4.4 Cardinality . . . 15
4.5 Checks . . . 17
5 Admitted fragment . . . 18
5.1 Termination . . . 18
6 Templates . . . 20
7 Validations and constraints . . . 21
7.1 Validations . . . 21
7.2 Constraints . . . 21
8 Canonical form . . . 22
9 Interoperability . . . 23
9.1 Writing . . . 23
9.2 Interchange format . . . 23
9.3 Document view . . . 24
9.4 ASP form . . . 24
References . . . 26

SECTION 1

Purpose

This document specifies LOAF (*Logic Objects As Facts*), a language of manifests: declarative descriptions made of blocks, facts and rules. LOAF files have the extension `.loaf`. The language is a block syntax whose rules are written in the notation of ASP (*answer set programming*). A set of files denotes a logic program, whose meaning is its model (§3). An application that uses the language provides its vocabulary (§4) and its own rules, and uses the models of the program. Hamac is its first application; its manifests are described in the document “Hamac manifests”.

The words **MUST**, **MUST NOT**, **SHOULD** and **MAY** have the meaning of RFC 2119 (1) and 8174 (2).

The notation of rules is that of the ASP-Core-2 standard (3), with the aggregates of the clingo dialect (4), restricted to the fragment of §5. Their meaning is defined in §3.

1.1 Principles

1. **Open language.** Its grammar and semantics are public and versioned. Any tool can produce manifests by writing text in this language.
2. **A single semantics.** A manifest denotes facts. Validation, deductions and constraints are rules over these facts, in a single language (§3).
3. **Guaranteed termination.** The rules admitted in a manifest belong to a fragment whose evaluation terminates (§5).
4. **Declared vocabulary.** Every predicate and every enumerated value is declared; what is not declared is an error (§4).
5. **A canonical form.** Every manifest has a single writing, which the formatter produces (§8).

1.2 Example

The example describes the members and the teams of an association, and who may read which documents. It comes in three parts: the vocabulary, the rules of the association, and the records of the persons and of the teams. Each file starts with the version of the language it follows, `loaf 0.1.` (§3.2.3); the excerpts of this document omit that line.

1.2.1 The vocabulary

```
% What a person block contains: each field with the sort of its value.
block person {
  first string.
  last string.
  team* team.  % a person may belong to several teams
  role* rbac::role.
  email? text.
}

block team {
  lead person.             % exactly one lead
  derived member* person.  % computed, never written in a block
  size? int.
  meeting* day {           % one meeting block per meeting day
    room name.
    start int.
  }
}

% The members of a team are the persons who name it.
T.member(P) :- P.team(T).

% The values allowed for the sort day.
enum day { monday; tuesday; wednesday; thursday; friday }.

% A predicate outside any block, computed by the rules of this file.
derived predicate reads(person, name).
```

A `block` declaration has the shape of the blocks it describes: each value is replaced by its sort. The sorts `person` and `team` denote the subjects of the blocks of that name, `string`, `text`, `int` and `name` are primitive, `day` takes its values from the `enum` declaration, and `rbac::role` from the roles declared to the `rbac` library (§3.2.4). A `derived` field is computed: the rule that follows its declaration defines it, and no block can write it. A field has exactly one value per block, unless it is marked: `email?` is optional, `team*` admits any number of teams (§4.4). `T.member(P)` reads "team `T` has member `P`". A predicate that belongs to no block is declared by `predicate`, or by `derived predicate` if it is computed.

1.2.2 The rules of the association

```
% Loads another file: here a library that defines the RBAC security
% policy (role-based access control), in a version compatible with 0.1.
use rbac 0.1.

% The subjects of the policy are the persons, its objects and its
% operations are names.
sort rbac::subject = person.
sort rbac::object = name.
sort rbac::operation = name.

% The roles and what they allow. The treasurer also has the rights of a
% volunteer.
rbac::role treasurer; secretary; volunteer.
rbac::inherits treasurer, volunteer.
rbac::perm volunteer, agenda, read.
rbac::perm treasurer, accounts, read.

% The roles declared in the records feed the library.
rbac::assigned(P, R) :- P.role(R).
rbac::active(P, R) :- rbac::assigned(P, R).

% Validations: the treasurer sits on the board, every lead is a member of
% their team, and nobody leads more than two teams.
error(fmt("{P} holds the treasury without being a member of the board"),
      P.role(treasurer)) :-
    P.role(treasurer), not board.member(P).
error(fmt("{P} leads {T} without being a member of it"), T.lead(P)) :-
    T.lead(P), not T.member(P).
error(fmt("{P} leads more than two teams"), person(P)) :-
    person(P), #count { T : T.lead(P) } > 2.

% Who may read what.
reads(P, O) :- rbac::allow(P, O, read).
```

The names defined by the library are written `rbac::…`: the prefix tells where each term comes from. The library does not fix what its subjects, its objects and its operations are: the `sort` statements fix them for the association (§4.1). The rules of the association feed the library from the records. Three validations, one of them with an aggregate, report inconsistent descriptions: each produces an error that carries a message and the fact at fault (§7). `T.lead(P)` reads "team `T` has lead `P`": it is the field `lead` of the `team` blocks. As an argument of `error`, it denotes that fact itself.

1.2.3 The records

```
person alice {
  first "Alice".
  last "Martin".
  team board; workshop.  % two teams
  role treasurer.
  email fmt("{~.first}.{~.last}@asso.example").
}

person bruno {
  first "Bruno".
  last "Petit".
  team workshop.
  role volunteer.
}
```

```
team board { lead alice. }

team workshop {
  lead bruno.
  size N :- N = #count { P : ~.member(P) }.

  meeting thursday {  % a meeting of the team
    room room_b.
    start 18.
  }
}
```

The block `person alice` describes a person without repeating its name: each line applies to `alice`. `team board; workshop.` gives two values at once. The address is built by a template from the attributes of the record. In a `team` block, `~` denotes the team: the size of `workshop` is deduced by a rule, from its members. The `meeting` block is nested in the team.

1.2.4 What the program deduces

- `alice` has the address `Alice.Martin@asso.example`.
- The team `workshop` has two members, `alice` and `bruno`, and its `size` is 2.
- `alice` reads `accounts` and, by inheritance from the role `volunteer`, `agenda`; `bruno` reads `agenda`.
- If `board` had `bruno` as its lead, who is not a member of it, the second validation would produce the error “bruno leads board without being a member of it”, attached to the line `lead bruno.` of the record.
- `meeting saturday` would be refused before any evaluation: `saturday` does not belong to the enumeration `day` (**L002**).
- `member alice.` in a `team` block would be refused likewise: `member` is a computed field (**L001**).
- A `team` record without `lead` would produce the error **L007**: the field is not marked, it requires a value.
- A second file that wrote `person alice { first "Alicia". }` would give two first names to `alice`: the field `first` admits only one, and each of the two facts would produce the error **L006**.

SECTION 2

Lexicon and grammar

2.1 Lexicon

Element	Form
comment	`%` up to the end of the line
constant	`[a-z][a-zA-Z0-9_]*`: `alice`, `room_b`
variable	`[A-Z_][a-zA-Z0-9_]*`: `P`, `T1`, `_`
number	decimal integer
version	integers separated by dots, `0.1`, `2.0.0`, after `loaf`, `module` or `use` (§3.2.3); a single integer, `use rbac 2.`, is also a version
string	between double quotes; `\"`, `\\`, `\n`, `\t` and `\r` denote a double quote, a backslash, a new-line, a tab and a carriage return
attribute	`.` with no space between an identifier or `)` and a lowercase letter: `P.first`, `~.member`
end of statement	`.` followed by a blank, a `%`, a `}` or the end of the file

The dot thus has two uses, told apart by what surrounds it: `P.first` is an access to an attribute, `lead alice.` ends a statement. Decimal numbers do not exist: a quantity is written as a compound term, whose meaning the application fixes.

The words `loaf`, `module`, `use`, `namespace`, `block`, `derived`, `predicate`, `enum`, `sort` and `render` are reserved.

2.2 Grammar

```
file        ::= header element*
header      ::= "loaf" version "."
element     ::= use | module | namespace | block | declaration | statement
use         ::= "use" constant version "."
module      ::= "module" constant version "."
namespace   ::= "namespace" constant "{" element* "}"
block       ::= word term? "{" element* "}"
declaration ::= "block" word "{" field* "}"
              | ["derived"] "predicate" constant "(" sort { "," sort } ")" "."
              | "enum" constant [ "(" arguments ")" ]
                "{" term { ";" term } "}" "."
              | "sort" name [ "=" sort ] "."
              | "render" constant "(" sort { "," sort } ")" "."
field       ::= ["derived"] word [mark] [ values ] "."
              | word [mark] sort "{" field* "}"
mark        ::= "?" | "*" | "+"
values      ::= sorts [ "=>" sorts ] | "[" sort "]"
sorts       ::= sort { "," sort }
sort        ::= name
statement   ::= fact | rule | constraint | directive
fact        ::= name arguments? "."
rule        ::= name arguments? ":-" body "."
constraint  ::= ":-" body "." | ":~" body "." "[" term "@" term "]"
directive   ::= "#minimize" "{" aggregate-elements "}" "."
              | "#maximize" "{" aggregate-elements "}" "."
arguments   ::= term { "," term }
term        ::= constant | variable | number | string | template
              | constant "(" arguments ")"
              | term ";" term                  % pool
              | term ".." term                 % interval
              | "[" [ arguments ] "]"          % list, in a statement
              | term operator term
              | access
access      ::= subject "." word { "." word }
subject     ::= variable | constant | "~" { "~" }
body        ::= literal { "," literal }
literal     ::= ["not"] atom | term comparison term | aggregate
```

```
atom        ::= name [ "(" arguments ")" ] | field-of
field-of    ::= subject "." name "(" arguments? ")" { "." name "(" arguments? ")" }
name        ::= word | constant "::" word | "::" word
template    ::= "fmt" "(" "\"" ( text | "{{" | "}}" | "{" hole "}" )* "\"" ")"
hole        ::= variable | access
```

The body and the aggregates follow the notation of ASP (`#count`, `#sum`, `#min`, `#max`, comparisons, `not`), limited to the fragment of §5.

A brace opens a block when it follows the first words of an element, before any `:-`, `:~` or `#`; otherwise it belongs to an aggregate. The rule is decided in a single pass, without backtracking.

SECTION 3

Semantics

The meaning of a set of files is defined in two steps. Elaboration (§3.2) associates a program with the files: facts and rules over atoms. The model (§3.3) is the set of true atoms, which the rules determine from the facts. An implementation is conforming when it computes this model (§3.4).

3.1 Terms and atoms

Terms are constants, integers, strings and compound terms `f(t1, …, tn)`. Two terms are equal if they are identical; only arithmetic expressions over integers (`N + 1`) are evaluated. Two more terms, `#inf` and `#sup`, the smallest and the greatest of all terms, are only the values of some aggregates (§3.3): no file can write them.

A predicate is a path of names, global or qualified by a namespace. A block `team` defines the predicate `team`, its field `lead` the predicate `team.lead`, and the field `room` of its sub-block `meeting` the predicate `team.meeting.room`. Two different paths are two distinct predicates, as are `p` and `ns::p`.

An atom is a predicate applied to terms; an atom without variables is ground. The arity of a predicate is its number of arguments, and `p/n` denotes the predicate `p` of arity n: `team.lead/2` is the predicate of the atoms `team.lead(T, P)`. This document writes `team.meeting.room(workshop, thursday, room_b)` for the atom of the predicate `team.meeting.room`; in the language, it is written by a block or by an access to the field (§3.2.1). An interpretation is a set of ground atoms: those that are true.

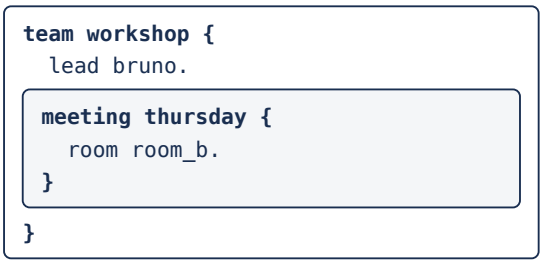
3.2 Elaboration

An element is elaborated in a context made of a path π (a sequence of names, empty at the level of a file) and of a sequence of subjects σ. We write $\pi.p$ for the path extended with `p`.

Element, in the context (π, σ)	Denotation
block `k s { E }`	the fact $(\pi.k)(\sigma, s)$, and the elements `E` elaborated in $(\pi.k, \sigma.s)$
statement `p t1, …, tn.`	the fact $(\pi.p)(\sigma, t_1, ..., t_n)$
rule `p t1, …, tn :- B.`	the rule $(\pi.p)(\sigma, t_1, ..., t_n)$ `:-` `B`
fact, rule or constraint written as an atom (`p(t) :- B.`)	themselves, at the level of a file
`enum d(a) { v1; …; vn }.`	the facts $d(a, v_i)$
`block`, `predicate`, `sort`, `render`	no fact: declarations (§4)

Example	Denotation

`team workshop {`
`  lead bruno.`
`  meeting thursday {`
`  room room_b.`
	`team(workshop)`
	`team.lead(workshop, bruno)`
	`team.meeting(workshop, thursday)`
	`team.meeting.room(workshop, thursday, room_b)`



in `team workshop`: π = `team`, σ = (`workshop`)
~ = `workshop`
`lead bruno.` denotes `team.lead(workshop, bruno)`

in `meeting thursday`: π = `team.meeting`, σ = (`workshop`, `thursday`)
~ = `thursday`, ~~ = `workshop`
`room room_b.` denotes
`team.meeting.room(workshop, thursday, room_b)`

Figure 1: Each block extends the path π, which names the predicate, and the sequence of subjects σ, which gives its first arguments. A statement written deep inside denotes a fact whose arguments start with all of σ; ~ denotes the last subject, ~~ the one before (§3.2.1).

Pools. An argument `a; b` denotes several facts, one per value: `team board; workshop.` in the block `person alice` denotes `person.team(alice, board)` and `person.team(alice, workshop)`. Several pools in one statement denote one fact per combination of their values. If the i-th argument has the set of values V_i (a single value forms a one-element set), the statement `p t1, …, tn.` denotes the facts $(\pi.p)(\sigma, v_1, ..., v_n)$ for all $(v_1, ..., v_n) \in V_1 \times ... \times V_n$:

Statement	Facts denoted

`rbac::perm treasurer; secretary, accounts; agenda, read.`	`rbac::perm(treasurer, accounts, read)`
	`rbac::perm(treasurer, agenda, read)`
	`rbac::perm(secretary, accounts, read)`
	`rbac::perm(secretary, agenda, read)`

Attributes. `has` is the relation of attributes: `has(s1, …, sn, f, v)` says that the subject of sequence $\sigma = (s_1, ..., s_n)$ has the value `v` for the attribute `f`, of sorts `term` for the subjects, `name` and `text`. A top-level subject has `has/3` attributes, that of a sub-block `has/4`. The language puts nothing in it: the attributes declared by fields are read through the fields themselves (§3.2.1), and an application **MAY** define others by rules with head `has(…)`.

Sets of files. A set of files denotes the union of the denotations of the files it contains and of the modules that `use` reaches (§3.2.2), each one once. A block opened several times denotes the union of its contents. The order of files and of elements has no effect.

3.2.1 Field access

In a rule, `X.f(a)` denotes the atom `k.f(X, a)`, where `k` is the sort of `X`. An access to a sub-block extends: `X.f(a).g(b)` denotes `k.f.g(X, a, b)`. As an argument, of `error` in particular, it denotes the term of the same shape.

Expression	Atom denoted
`T.lead(P)`	`team.lead(T, P)`
`T.meeting(D).room(R)`	`team.meeting.room(T, D, R)`

The sort of `X` (§4.1) is deduced from the `block` declarations: it is the block that declares all the fields through which the rule accesses `X`, each with the arity of its access: `X.level(A, L)` designates a field of arity 3. In `:- T.lead(P), not T.member(P).`, only `team` declares `lead` and `member`. If several blocks fit, a literal `k(X)` of the rule chooses; if the ambiguity remains, or if no block fits, the rule is refused (**L001**).

A field that a namespace declares (§3.2.4) is read `X.ns::f(a)`; inside the namespace `ns` itself, `X.f(a)` suffices. Unqualified, `X.f(a)` denotes the field of the namespace where the rule is written if it declares one with that path, the global field otherwise.

Subjects of the block. In a block, `~` denotes the last subject of σ, and `~.f(a)` the atom $(\pi.f)(\sigma, a)$. Each added `~` goes up one block: `~~` denotes the subject before last, and `~~.f(a)` the field `f` of the enclosing block. In general, if $\sigma = (s_1, ..., s_n)$, the sign repeated k times denotes s_{n-k+1}, and followed by `.f(a)` the atom $(\pi_{n-k+1}.f)(s_1, ..., s_{n-k+1}, a)$, where π_i is the path of the first i blocks. The arguments `a` are written in the current block. A block has only one enclosing block: the number of signs suffices to denote each of its subjects. More signs than subjects is refused (**L001**).

Rule, in `team workshop`	Rule denoted
`size N :- N = #count { P : ~.member(P) }.`	`team.size(workshop, N) :-` `  N = #count { P : team.member(workshop, P) }.`
in `meeting thursday`: `start H :- ~~.size(N), H = 17 + N.`	`team.meeting.start(workshop, thursday, H) :-` `  team.size(workshop, N), H = 17 + N.`

`~~` serves in particular to read, from a sub-block, a table indexed by the subject that surrounds it (§4.4).

Without parentheses, in term position, `X.attr` denotes the value of an attribute of the block of `X`: a field of that block with at most one value and a single sort, declared outside any namespace. A rule that contains `X.attr` is equivalent to the rule where the expression is replaced by a variable `V` that appears nowhere else, and where the atom of the field, `k.attr(X, V)`, is added to the body. The block `k` is deduced as for a field access; failing that, it is named by a literal `k(X)` of the rule, or it is the only

block that declares the attribute. A subject of several blocks thus has the attributes of each, kept apart: if `pg` is both a `service` and a `database`, `P.user` reads the field `user` of `database`. In a block, `~.attr` denotes the attribute of the current block, indexed by all of σ: in `meeting thursday` of `team workshop`, `~.room` adds `team.meeting.room(workshop, thursday, V)`, and `~~.size` adds `team.size(workshop, V)`.

References. When the field of an attribute has the sort of a block, the access continues into that block: `X.ref.attr` reads the attribute `attr` of the block that `X.ref` designates, and so on. In a consumption that designates its database by `db? database.`, `~.db.user` adds `service.consumes.db(S, N, D)` and `database.user(D, V)`. Each attribute of the chain is checked against the declaration of its block.

Attributes defined by an application. When no block declares the attribute, `X.attr` reads the relation of attributes, `has(X, attr, V)` (§3.2), which an application **MAY** complete by rules of head `has(…)`. An attribute read in a body, by `X.attr` or by a literal `has(…, attr, …)`, that no field declares (§4.4) and no rule of head `has(…)` defines, with the same number of subjects, is refused. A rule whose head attribute is a variable bound by `has` literals of its body (`has(S, N, K, V) :- …, has(P, K, V).`) defines the attributes that these literals read, for its own number of subjects; bound otherwise, the variable may name any attribute. A chain that goes through a field that does not designate a block, or that names an attribute that its block does not declare, is refused likewise (**L001**): a misspelt attribute would otherwise make the rule inapplicable without notice.

3.2.2 Modules

`use m 1.2.` denotes the module `m`, in a version compatible with 1.2 (§3.2.3): a file that declares its name and its version (`module m 1.4.0.`), among the files given or searched for by its name in the library directories. A module generally defines the namespace of the same name (§3.2.4), whose names are then written `m::…`. It **MAY** complete another one: a module `m2` that loads `m` by `use` and opens `namespace m { … }` declares its own fields and predicates there, and feeds the open predicates of `m`; it **MUST NOT** define the closed predicates of `m` (§4.3).

3.2.3 Versions

A file carries two kinds of versions: that of the language it follows, and those of the modules it loads.

Language version. The first element of a file is `loaf x.y.`, where `x.y` is the version of this specification that it follows; only comments precede it. An implementation refuses a file without this declaration, or of a version it does not know (**L008**). This document specifies version 0.1.

Module version. A module declares its version by `module m x.y.z.`, where `m` is its name. A version has three components, major, minor and patch, compared in this order. Two versions are compatible if they have the same major, or, when it is 0, the same minor: 1.2.0 and 1.4.1 are compatible, 0.1.3 and 0.2.0 are not.

Constraints. `use m c.` requires of `m` a version compatible with `c` and greater than or equal to `c`, where `c` has one, two or three components; missing components are 0. `use m 1.2.` admits 1.2.0 and 1.9.3, not 1.1.0 nor 2.0.0. A version is written without quotes; the last dot ends the declaration: in `use rbac 2.0.`, the constraint is 2.0.

Selection. For each module, the version selected is the smallest available version that satisfies all the constraints: those of the files and those of every module version that their `use` reach. A set of files loads a single version of each module. The selection depends only on the constraints and on the available versions: it is reproducible without a lock file. A `use` without a constraint, a module without a `module` declaration, constraints that are incompatible with each other (`0.1` and `0.2`) or that no available version satisfies are refused (**L009**).

```
% A file.
loaf 0.1.
use m 0.1.
use n 1.  % n 1.4.0 requires use m 0.1.2: m 0.1.3 is selected
```

Checksums. The selection picks a version; the checksum guarantees its contents. A checksum file, `loaf.sum`, gives for each selected module its version and the SHA-256 of its file, one line per module:

```
rbac 0.1.3 sha256:248864c85be9c2ee397192910e76c4071de2371a7a5c797112ca11e9ee5af8f0
```

When a checksum file comes with a set of files, an implementation refuses a selected module that is not listed in it, or whose contents do not match its checksum (**L011**). Two parties that exchange manifests and their checksum file thus load the same modules, byte for byte.

Compatibility. The vocabulary of a module is what it declares: blocks, fields, predicates, enumerations, renderings and sorts. The difference between the vocabularies of two versions fixes the minimal version bump:

Bump	Changes of the vocabulary
major	removing a block, a field, a predicate, an enumeration value, a rendering or an abstract sort; changing the sorts of a field or of a predicate, its keys, or the mark of a field; adding a required field (unmarked or `+`) to an existing block; making an open predicate computed; changing the instantiation of a sort
minor	adding a block, a predicate, an optional field (`?`, `*`) or a flag, an enumeration value, a rendering or an abstract sort; opening a computed predicate
patch	no change of the vocabulary

Changing a mark is always major: admitting more values makes false the rules that assume one value, admitting fewer invalidates the manifests that write several or none. A major bump changes the major, or the minor of a version 0.y.z; a minor bump changes the minor, or the patch of a version 0.y.z. A tool that compares two versions of a module refuses a declared version whose bump is lower than the minimal bump (**L010**). The difference of vocabularies does not see the rules: a change of rules alone is at least a patch, and its author judges whether it is more.

3.2.4 Namespaces

A block `namespace ns { … }` makes local the predicates that belong to it: those that a rule or a fact defines in it, and those it declares by `block` or `predicate` (§4). A local predicate `p` is the predicate `ns::p`. Like a block, a namespace **MAY** be opened several times, in one or several files: its local predicates are those of all its blocks.

- From outside, a local predicate is named `ns::p(…)`, in a head as in a body: this is how the inputs of a namespace are fed and its outputs read.
- From inside, an unqualified name denotes the local predicate if it exists, the global predicate otherwise; `::p(…)` always denotes the global predicate.
- In a block, a qualified statement keeps the path of the block: `k s { ns::p v. }` denotes the fact `ns::k.p(s, v)`, that of the field `p` that the namespace `ns` declares in the `k` blocks (example below).
- Terms belong to no namespace: `c(a, nil)` is the same term everywhere.
- `has`, the relation of attributes, is global.

A module thus adds its own fields to the blocks of a vocabulary. The module `gdpr` below adds to the `person` blocks the field `consent`, which lists the purposes a person consents to, and deduces from it who is listed in the directory:

```
namespace gdpr {
  enum purpose { directory; newsletter; photos }.

  block person { consent* purpose. }

  listed(P) :- P.consent(directory).
}
```

A record fills this field by a qualified statement:

Record	Facts denoted

`person alice {`
`  gdpr::consent directory; photos.`
	`person(alice)`
	`gdpr::person.consent(alice, directory)`
	`gdpr::person.consent(alice, photos)`

In the module, `P.consent(directory)` denotes `gdpr::person.consent(P, directory)`, and the rule gives `gdpr::listed(alice)`. The field belongs to the namespace `gdpr`: another module can declare a field `consent` in the same blocks without conflict. Outside the module, it is read by `P.gdpr::consent(F)`.

Two namespaces can thus define the same names without mixing:

```
use rbac 0.1.  % defines rbac::allow
use acl 2.     % another library, which also defines acl::allow

% An access requires the agreement of both.
allow(P, O, Op) :- rbac::allow(P, O, Op), acl::allow(P, O, Op).
```

3.3 Model

Elaboration gives a program P: a finite set of facts F and of rules R, all in the admitted fragment (§5). A rule has the form `h :- l1, …, ln.`, where each l_i is an atom, the negation `not a` of an atom, a comparison or an aggregate.

Strata. The dependency graph of P links the head predicate of each rule of R to the predicates of its body; an edge is negative if it goes through `not` or through an aggregate. The fragment forbids any cycle that contains a negative edge; the predicates are therefore arranged in strata $S_1, ..., S_m$, so that a predicate depends on the lower strata only through a negative edge.

The following figure arranges in this way the rules of the introductory example (§1.2) that concern the teams:

```
T.member(P) :- P.team(T).

team workshop { size N :- N = #count { P : ~.member(P) }. }

error(fmt("{P} holds the treasury without being a member of the board"),
      P.role(treasurer)) :-
    P.role(treasurer), not board.member(P).
error(fmt("{P} leads {T} without being a member of it"), T.lead(P)) :-
    T.lead(P), not T.member(P).
error(fmt("{P} leads more than two teams"), person(P)) :-
    person(P), #count { T : T.lead(P) } > 2.
```

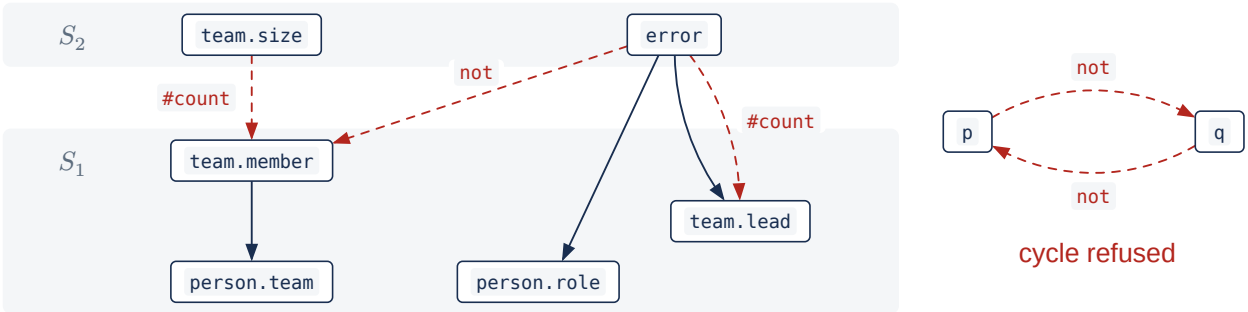


Figure 2: On the left, the dependency graph of part of the example: a solid edge is positive, a dashed edge negative. `team.member` depends on `person.team` through `T.member(P) :- P.team(T).`; `team.size` and `error` depend on `team.member` through an aggregate or through `not`, and are therefore arranged in the upper stratum. On the right, `p :- not q.` and `q :- not p.` form a cycle of negative edges, outside the fragment.

Computation. The model is computed stratum by stratum, from the facts. We write R_i for the rules of R whose head predicate belongs to S_i, and M_i for the set of atoms established once the strata $S_1, ..., S_i$ are computed. At the start, only the facts are established: $M_0 = F$. For i from 1 to m, M_i is the smallest set of atoms that contains M_{i-1} and is closed under R_i: whenever a ground instance of a rule of R_i has a true body, its head belongs to M_i. It is obtained by applying the rules of R_i until no new atom appears. In the body of a rule of R_i:

- an atom is true if it belongs to M_i;

- `not a` is true if $a \notin M_{i-1}$: `a` belongs to a lower stratum, whose computation is complete;
- an aggregate `#count { t : L }` is the number of distinct tuples t for which the literals `L` are true in M_{i-1}; `#sum`, `#min` and `#max` are defined likewise;
- a comparison is true if it holds between the values of its terms.

Values. Arithmetic (`+`, `-`, `*`, `/`, `\`, `**`) applies to integers; `/` and `\` give the quotient and the remainder of the division truncated toward zero: `-7 / 2` is `-3`, and `-7 \ 2` is `-1`. An undefined expression (an operand that is not an integer, a division by zero, a negative exponent) makes false the rule instance that contains it. Terms are totally ordered: integers, in the order of numbers, precede compound terms, which precede strings. Two compound terms are compared by arity, then by name, then argument by argument; a constant is a compound term of arity 0, and a tuple has the empty name. Two strings are compared byte by byte. `#min` of an empty set is `#sup`, greater than any term, and `#max` of an empty set is `#inf`, smaller than any term.

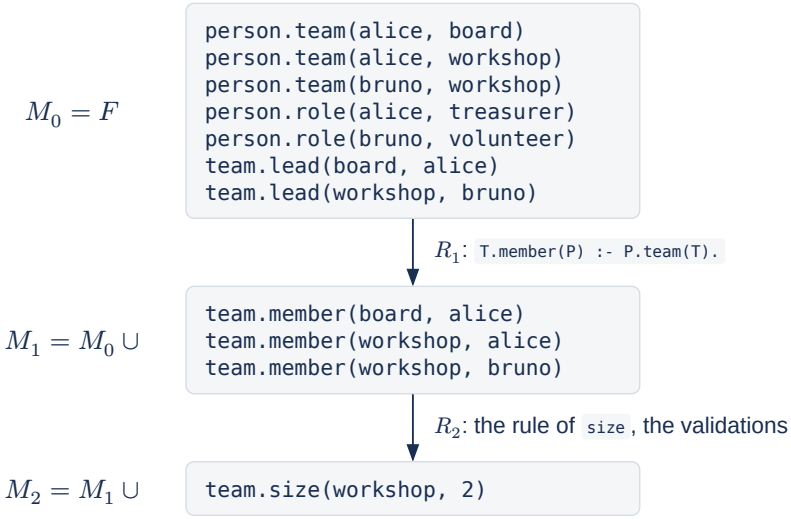


Figure 3: The computation of the model of the part of the example shown above. R_1 adds the members of the teams to the facts, R_2 the size of `workshop`, counted in M_1. The validations produce no error: in M_1, whose computation is complete before that of S_2, every lead is a member of their team and `alice`, who holds the treasury, is a member of `board`.

The model of P is $M = M_m$. It does not depend on the chosen arrangement in strata: it is the perfect model of P (5), which is also its unique stable model (6).

Validations. The atoms `error(Reason, Fact)` of M are the errors of the description (§7). A set of files is accepted if it passes the checks (§4.5) and if M contains no error.

Constraints and choices of an application. A constraint `:- B.` adds no atom: it excludes the models where `B` is true. An application adds its own rules A to P, among them choice rules; the meaning of $P \cup A$ is then the set of its stable models in the sense of the ASP-Core-2 standard (3), and weak constraints, `#minimize` and `#maximize`, designate the optimal ones. Without choice rules, there is only one: M, if it satisfies the constraints. The ASP form (§9.4) gives P in the syntax of the standard, to which an application adds A.

3.4 Conformance

An implementation is conforming if, for every set of files:

- it refuses, with the same code, the files that the checks and the admitted fragment refuse;
- it computes the same model M, restricted to the declared predicates, and therefore the same errors;
- it exports this model in the interchange format (§9.2), byte for byte, and, if it provides one, in the document view (§9.3).

Its internal representations (names of predicates, encoding of namespaces and attributes, shape of rules) are not observable.

SECTION 4

Vocabulary and typing

A rule that names a predicate or a value that does not exist never applies (§3.3). So that a misspelt name does not disable a rule, every predicate and every enumerated value **MUST** be declared.

4.1 Sorts

A sort is a set of values. Each argument of a declared predicate has a sort, which its declaration names (§4.2): in `block team, lead person.` gives the field `team.lead` the shape `team.lead(team, person)`, whose first argument is a team and the second a person.

Sort	Values	Example
`int`	the integers	`start 18.`
`string`	the strings	`first "Alice".`
`name`	the constants	`room room_b.`
`text`	the terms that have a rendering (§6)	`cat("Alice", ".", "Martin")`
`term`	all ground terms	`route(payroll, "/salaries")`
name of a block: `person`, `team`	the subjects of the blocks of that name	`lead bruno.` requires a block `person bruno`
name of an enumeration: `day`	the values of the `enum` declaration	`meeting thursday` requires `thursday` in `day`
name of a predicate of a namespace: `rbac::role`	the first arguments of the facts of that predicate	`role treasurer.` requires the fact `rbac::role(treasurer)`

In its namespace, the sort of a predicate is named without qualification. The module `rbac` thus declares `predicate role(name).` and `predicate inherits(role, role).`; from outside, the sort is named `rbac::role`, as in the declaration `role* rbac::role.` of the block `person`.

Abstract sorts. `sort s.` declares an abstract sort: a generic model thus names the sorts of its arguments without fixing them. `sort ns::s = k.` instantiates it with the sort `k`, whose values it takes. An abstract sort is instantiated once and only once in the set of files, by any one of them:

```
namespace rbac {
  sort subject.
  sort object.
  sort operation.
  predicate role(name).
  predicate assigned(subject, role).
  predicate perm(role, object, operation).
}

% In a file of the association:
sort rbac::subject = person.
sort rbac::object = name.
sort rbac::operation = name.
```

With these instantiations, a fact `rbac::assigned(carol, volunteer)` without a block `person carol` is refused.

The extension of a sort is fixed by the facts that the files denote: the subjects of the blocks, the values of the enumerations, the first arguments of the facts of a predicate. A value outside the sort of its argument is refused (**L002**): in a fact, before any evaluation (§4.5); in an atom that a rule deduces, once the model is computed, at the location of the rule. A rule that deduces `team.member(board, carol)` without a block `person carol`, or a `size` equal to `#sup`, is thus refused. An unknown sort, or an abstract sort whose instantiation is missing, repeated or forms a cycle, is refused before any evaluation.

4.2 Declarations

The vocabulary is written in the language itself, by `block`, `predicate`, `enum` and `sort` declarations, as in the first part of the example. An application ships its own; a file **MAY** declare its own predicates, with the same syntax.

A declaration `block k { … }` declares the predicate `k/1` and, for each field, the predicate that the statements of this field denote (§3.2). The second column gives the shape of the facts of this predicate, the sort of each argument in its place:

Declaration, in `block team`	Shape of the facts	Example of a fact
`lead person.`	`team.lead(team, person)`	`team.lead(workshop, bruno)`
`meeting day {`	`team.meeting(team, day)`	`team.meeting(workshop, thursday)`
`  room name.`	`team.meeting.room(team, day, name)`	`team.meeting.room(workshop, thursday, room_b)`

A field marked `derived` is computed. It is defined only by rules written in the file that declares it: those of the application for its vocabulary, those of a manifest for its own fields. A fact that writes it, in a block or elsewhere, is refused (**L001**), as is a rule of another file that defines it (§4.3). It is read like any other field.

In the introductory example (§1.2), the vocabulary computes the members of a team from the records of the persons:

```
block team {
  lead person.
  derived member* person.  % computed, never written in a block
}

T.member(P) :- P.team(T).

person alice { team board; workshop. }
person bruno { team workshop. }
```

The records give three members:

- `team.member(board, alice)`;
- `team.member(workshop, alice)`;
- `team.member(workshop, bruno)`.

A record `team board { member carol. }` is refused (**L001**).

A manifest declares its own computed fields in the same way:

```
block person { derived colleague* person. }  % shares at least one team

P.colleague(Q) :- P.team(T), Q.team(T), P != Q.
```

This file adds the field `colleague` to the `person` blocks. With the previous records, `person.colleague(alice, bruno)` and `person.colleague(bruno, alice)` are true: both are members of `workshop`.

Only this file defines `colleague`. Another file cannot add cases, for instance make the lead of a team the colleague of its members:

```
P.colleague(Q) :- T.lead(P), Q.team(T).  % refused (L001)
```

Like a block, a `block` declaration **MAY** be opened several times: its fields add up. A library thus completes the blocks of an application: in a namespace, the declared fields are local, and the predicates of the blocks themselves remain global (`k`, and `k.q` for a sub-block `q`).

4.3 Open and closed predicates

Each predicate is open or closed:

- a predicate declared by `predicate`, by a field of `block` or by `enum` is **open**: any file **MAY** define it, by facts or rules. This is how a record fills the fields of a block and how a file feeds the inputs of a module;
- a predicate declared `derived` is **closed**: only the file that declares it defines it, by rules for a field, by facts and rules for a `derived predicate`;
- a predicate that is not declared is **closed**: a single file defines it. These are the intermediate predicates and the results of a module, such as `rbac::senior` and `rbac::allow`. They are written as atoms (`senior(R, R) :- role(R).`): a block, and an instruction written in a block, always use a declared block and field, so that a misspelt name (`servcie web { … }`, `prot 8080.`) is refused (**L001**).

`error/2` and `has`, declared by the language, are open. A fact or a rule that defines a closed predicate outside its file, or a fact that writes a computed field, is refused (**L001**). Reopening a namespace changes nothing: the rule applies to the file, not to the namespace.

No construct removes a fact or a rule: a file only adds. The rules that define a closed predicate are therefore all written in a single file, and another file acts on it only through the open predicates it reads. The module `rbac` thus declares its inputs (`role`, `assigned`, `perm` …); its results `allow` and `error` are defined only in it:

```
use rbac 0.1.

rbac::perm treasurer, accounts, write.  % an input: admitted

% A result: refused (L001), written here or in "namespace rbac { … }".
rbac::allow(P, 0, read) :- rbac::assigned(P, _), rbac::perm(_, 0, _).
```

4.4 Cardinality

The values of a field form a set: for a given block, the field `f` has as many values as there are facts $(\pi.f)(\sigma, ...)$, without order or duplicates. A field has exactly one value per block; a mark attached to its name, in its declaration, admits other numbers:

Declaration	Values per block	Example
`f s.`	exactly one	`first string.`
`f? s.`	at most one	`email? text.`
`f* s.`	any number	`team* team.`
`f+ s.`	at least one	`phone+ string.`

For a field with several sorts, the tuples of values are counted (`slot* day, int.`); for a sub-block, its subjects (`meeting* day { … }`). A field without a sort (`public.`) is a flag: present or absent, it carries no mark. All the declarations of the same field carry the same mark: a `block` declaration reopened to add fields to a sub-block repeats it (`route* string { role* role. }`).

Cardinality does not change the facts that the blocks denote: it adds validations (§7), which the language writes for each field `f`:

Mark	Validations added
none, `?`	if a block has several values of `f`, the fact `error(l006, F)` for each of its facts `F` of the field: `error(l006, person.first(alice, "Alice"))` (**L006**)
none, `+`	if a block has no value of `f`, the fact `error(l007(f), B)`, where `B` is the fact of the block: `error(l007(lead), team(board))` (**L007**)

They apply to the model: a field that rules fill, computed or not, is counted with the values they produce. A block reopened in several files unites their contents; two files that each give a value to a field that admits only one therefore produce **L006**, with the locations of both facts:

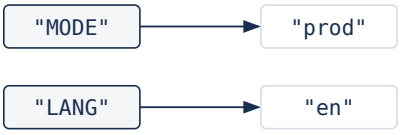
```
person alice { first "Alice". }   % one file
person alice { first "Alicia". }  % another: L006 for each fact
```

Tables. An arrow splits the sorts of a field into keys and values: `f* k => v.` declares a table, which associates a single value with each key. Statements and facts do not change: `env "MODE", "prod".` denotes `service.env(api, "MODE", "prod")`. The mark then counts the keys, and each key present has exactly one value:

```
block service { env* string => text. }

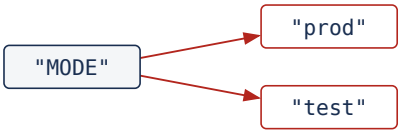
service api {
  env "MODE", "prod".
  env "LANG", "en".                  % two keys: admitted
}
service api { env "MODE", "test". }  % L006 for MODE
```

A table adds the validation `error(l006, F)` for each fact `F` of a key that has several values, besides those of its mark. In the document view (§9.3), a table is a JSON object indexed by its keys: `{"MODE": "prod", "LANG": "en"}`. Keys and values may have several sorts (`route* string, name => int.`).



one arrow per key: admitted

```
"env": { "MODE": "prod", "LANG": "en" }
```



two arrows from one key: **L006** for each fact

```
"env": { "MODE": ? }
```

Figure 4: The table `env` of the block `service api`. On the left, each key has one value, and the table is written as a JSON object. On the right, two files each give a value to `"MODE"`: no object represents this case. A field without an arrow (`env* string, text.`) would count the pairs and admit both facts.

A table is often read from a sub-block, through the subject that surrounds it (`~~`, §3.2.1). Here, a table of the block `site` associates its room with each team, and the meeting of the team reads its own from it:

```
block site { room_of* team => name. }  % the room assigned to each team

site hall {
  room_of workshop, room_b.
  room_of board, room_a.
}

team workshop {
  meeting thursday {
    venue hall.  % a field venue site. of meeting
    room R :- ~.venue(S), S.room_of(~~, R).
  }
}
```

1 `~.venue(S)`

→ `team.meeting.venue(workshop, thursday, S)`

team.meeting.venue

team	day	site
workshop	thursday	hall
board	monday	hall

`S = hall`

2 `S.room_of(~~, R)`

→ `site.room_of(hall, workshop, R)`

site.room_of (table)

site	key	value
hall	workshop	room_b
hall	board	room_a

`R = room_b`

→ `team.meeting.room(workshop, thursday, room_b)`

Figure 5: The rule of `room` in `meeting thursday`, where σ = (`workshop` , `thursday`). The first literal reads the site of the meeting; the second reads the row of the table whose key is the enclosing subject, `~~` = `workshop`. Since each key has only one value, the meeting receives at most one room.

Lists. `f* [s].` declares a list: a table whose key is an integer position, `f* int => s.`, and whose document view (§9.3) is a JSON array, ordered by increasing positions. A list `[t1, …, tn]`, as an argument of a statement, denotes one fact per element, from position 1 to n; a statement contains at most one list.

```
block service { dns* [text]. }

service api { dns ["10.0.0.53", "10.0.0.54"]. }  % positions 1 and 2
service api { dns 15, "10.0.0.1". }              % position 15, at the end
```

The facts are `service.dns(api, 1, "10.0.0.53")`, `service.dns(api, 2, "10.0.0.54")` and `service.dns(api, 15, "10.0.0.1")`; the document view is `["10.0.0.53", "10.0.0.54", "10.0.0.1"]`. Positions need not be contiguous: a file, or a rule (`dns I, V :- …`), places an element between two others by its position. The mark counts the elements, and two equal elements remain distinct by their position. As for any table, two files that each write a list at the same place give two values to position 1 (**L006**): lists are never concatenated. A field declared as a list and as a table is refused (**L001**).

In the example (§1.2), `lead person.` thus requires one lead per team: a `team` record without `lead` produces `error(l007(lead), team(T))`. `size? int.` admits a team without a size: only `workshop` computes one.

Attributes. A field with at most one value (unmarked, or `?`), with a single sort, declared outside any namespace, is an attribute of its block, read by `X.attr` (§3.2.1). `P.first` thus denotes the first name of `P`, whether the block writes it or a rule computes it (`size N :- …` gives the `size` of `workshop`). An attribute of a sub-block keeps all its subjects: in `meeting thursday`, `~.room` reads `team.meeting.room(workshop, thursday, R)` distinct from the room of another team meeting on the same day. A field marked `*` or `+` is not an attribute, nor is a table: reading `P.team` is refused (**L001**, §3.2.1).

4.5 Checks

The refusals of this specification carry a code, `L` followed by three digits; an application numbers its own diagnostics in another space. Before any evaluation, the tool checks the following conditions; only **L002** is also checked on the deduced atoms, and **L014** on the instructions, once the model is computed (§4.1, §6):

Code	Condition
L001	predicate not declared, or used with another arity; block, or instruction in a block, that no declaration describes (§4.3); access to a field whose block cannot be deduced, or to an attribute that nothing defines (§3.2.1); closed predicate defined outside its file, or computed field written by a fact (§4.3); field declared with two different cardinality marks, or as a table and as a field, or as a list and as a table (§4.4); statement that contains more than one list; `~` repeated more times than the block has subjects (§3.2.1)
L002	value outside its sort: outside its enumeration (`meeting saturday`), subject that denotes no block of the right sort (`lead zoe.` without a block `person zoe`), value not declared in a namespace (`role auditor.` without a declaration `rbac::role auditor`), in a fact or in a deduced atom (§4.1)
L003	rule outside the admitted fragment (§5)
L004	unknown sort; abstract sort without instantiation, instantiated twice or through a cycle; instantiation of a sort that is not abstract (§4.1)
L005	hole of a template whose sort is not included in `text` (§6)
L008	file without a `loaf` declaration, or of an unknown language version (§3.2.3)
L009	`use` without a constraint, module without a `module` declaration, constraints that are incompatible or that no available version satisfies (§3.2.3)
L011	selected module missing from the checksum file, or whose contents do not match its checksum (§3.2.3)
L014	instruction whose template has no value in the model (§6)

These checks apply to the facts and rules that the files denote (§3.2), sort by sort.

SECTION 5

Admitted fragment

The rules that a manifest **MAY** contain form a fragment of ASP:

- facts, normal rules, integrity constraints, weak constraints, aggregates, comparisons, `#minimize` and `#maximize`;
- **safe rules**: every variable of a rule appears in a positive atom of its body, outside any aggregate, or is defined by an assignment `X = t` or `X = #count { … }` whose other variables are themselves safe; a variable local to an aggregate appears in a positive atom of its condition;
- **stratified negation**: a predicate **MUST NOT** depend negatively on itself, directly or through other rules;
- **stratified aggregates**: an aggregate **MUST NOT** range over a predicate that depends on the head predicate;
- **bounded term construction**: in a rule that belongs to a cycle of the dependency graph of predicates, each variable of the head **MUST** be bound by a positive atom of the body, and each compound term of the head **MUST** be, whatever the values of its variables, at most as large as an argument of a positive atom of the body: it does not count more symbols, an anonymous variable `_` of the body counting for one, and no variable occurs in it more often. An arithmetic term (`N + 1`) has no bounded size. Such a rule can decompose and recompose existing terms, never build larger ones;
- **no choice rule**: choices belong to the application, which adds its own generation rules;
- no `#script`, `#external`, `#include`, `#program`: a manifest reads nothing but the facts, and has no side effect.

The dependency graph has one vertex per predicate and one edge from the head of each rule to each predicate of its body. The relation of attributes counts in it as one vertex per attribute name: `has(S, first, V)` and `has(S, email, V)` are two vertices, and a variable attribute (`has(S, A, V)`) denotes all of them.

A rule outside this fragment is refused (**L003**), with the rule and, where relevant, the cycle at fault.

5.1 Termination

The fragment guarantees that evaluation terminates, in two steps.

1. **Grounding.** The terms of a program are those of its facts, finitely many, and those that the rules build. A rule that is in no cycle applies a finite number of times: these rules are arranged in levels, each of which produces a finite number of terms from a finite set. A rule of a cycle produces only terms at most as large as terms already present, made of the symbols of the program; the terms of bounded size over a finite set of symbols are finitely many. The universe of terms is therefore finite, and so is the ground program. Intervals `a..b` are finite, their bounds coming from facts.
2. **Solving.** A finite ground program has a finite number of candidate models; the search for a stable model, and for an optimal model, terminates.

Stratified negation and aggregates give the deductive part of the program a unique model, computed level by level (5). The only non-determinism comes from the generation rules of the application, over finite domains.

Termination does not mean speed: a choice problem is combinatorial, and solving it may take exponential time in the worst case.

The rule `p(f(X)) :- p(X).` is refused: it belongs to a cycle and its head builds a term larger than those of its body. Likewise, `count(M) :- count(N), M = N + 1.` is refused: `M` is bound only by an assignment. The suffixes of a list `l(a, l(b, nil))`, on the other hand, are computed by a recursive rule, `suffix(T) :- suffix(l(_, T)).`, which produces only subterms. With `cat(D) :- parent(_, D).`, the rule `parent(c(T, s(X, R)), c(T, X)) :- cat(c(T, s(X, R))).` belongs to a cycle, and it is admitted: it recomposes a term smaller than that of its body.

Stratification excludes in particular default values written on the predicate they complete: `T.meeting(M).start(18) :- T.meeting(M), not T.meeting(M).start(_).` has no stable model and is refused (**L003**). A default value is written on another predicate:

```
start(T, M, H) :- T.meeting(M).start(H).
start(T, M, 18) :- T.meeting(M), not declared_start(T, M).
declared_start(T, M) :- T.meeting(M).start(_).
```

SECTION 6

Templates

A template `fmt("…{X.attr}…")` is a string in which each hole `{…}` will be replaced by the value of its expression; `{{` and `}}` give braces. A hole contains a variable or an access to an attribute (`P.first`, `~.last`). A template denotes the term `cat(…)` that alternates its texts and the terms of its holes; an access to an attribute in a hole follows the rule of §3.2.1:

Template	Term denoted
`fmt("{P.first}@{P.domain}")`	`cat(V1, "@", V2)`, with `person.first(P, V1)` and `person.domain(P, V2)` added to the body, for `P` of sort `person`

Rendering. The rendering of a term is a text. It is defined for the terms of the sort `text`:

- an integer is rendered in decimal, a string by its contents, a constant by its name;
- `cat(t1, …, tn)` is rendered by the concatenation of the renderings of its arguments;
- a term `f(t1, …, tn)` whose constructor the application declares by `render f(s1, …, sn).`, and each argument t_i of which is of sort s_i, is rendered as the application defines.

```
render qty(int, name).  % a quantity and its unit: qty(512, mi)
```

No other term has a rendering: `route(payroll, "/salaries")` is an ordinary compound term.

Sort of a hole. Each hole has a sort, which **MUST** be included in `text`. That of an access to an attribute is the sort of the field it reads, and `text` for an attribute that an application defines (§3.2.1). That of a variable is the sort of a position where it occurs in a positive literal of the body, or `int` if an aggregate defines it (`N = #count { … }`); a single one of these sorts included in `text` suffices. A sort is included in `text` if it is `int`, `string`, `name` or `text`, or if all its values have a rendering. A hole none of whose sorts is included in `text`, `term` included, is refused (**L005**):

```
block person { badge term. }

greeting(P, fmt("badge {B}")) :- P.badge(B).  % refused: B is of sort term
```

Only a string passed to `fmt` is interpolated: other strings are literal, braces included. A variable of a hole that is bound nowhere makes the rule unsafe: it is refused (**L003**).

Instructions with a template. An instruction written without `:-` asserts its fact; its templates make it a rule, whose body reads the attributes of the holes. If that body is never true in the model, the fact would be missing without notice: the set of files is refused (**L014**), at the location of the instruction. A field whose value may be absent is marked `?`, and the instruction is left out, or written as a rule (`url U :- U = fmt(…).`).

Other compound terms are ordinary terms. An application **MAY** interpret some of them when it uses a model; in the rules, they remain terms, which are compared and passed on without being interpreted.

SECTION 7

Validations and constraints

The language distinguishes two ways of ruling out what does not fit.

7.1 Validations

A validation is a rule with head `error(Reason, Fact)`. `Fact` is the fact at fault, written as an atom: `team(T)`, or `T.lead(P)` for the fact `team.lead(T, P)`. `Reason` is a message, a string or an `fmt` template, or a code that an application documents in its catalogue of errors. `error/2` is declared by the language, which itself writes the validations of the cardinality of fields, with codes **L006** and **L007** (§4.4).

Each fact keeps track of the location, file and line, of the element that denotes it. The diagnostic gives the location of the fact at fault and the reason:

```
association.loaf:84: bruno leads board without being a member of it
```

A set of files that produces an error is refused; all the errors are reported together.

7.2 Constraints

A constraint `:- body.` forbids the body from being true: it eliminates the models where it is. It serves to restrict the choices of an application (§5). If an application chooses the room of each meeting, this constraint rules out the choices that put two teams in the same room on the same day:

```
:- T1.meeting(D).room(R), T2.meeting(D).room(R), T1 != T2.
```

On a description without choices, a violated constraint leaves the program without any model, without saying why. In the introductory example (§1.2), the records fix the rooms: a record `team board { meeting thursday { room room_b. } }` would violate the constraint, and the program would have no model left. A property that the description must satisfy is therefore written as a validation, which designates the fact at fault:

```
error(fmt("{T1} and {T2} meet on {D} in room {R}"),
      T2.meeting(D).room(R)) :-
    T1.meeting(D).room(R), T2.meeting(D).room(R), T1 < T2.
```

With the same record, it produces the error “board and workshop meet on thursday in room room_b”, attached to the line `room room_b.` of the team `workshop`. The comparison `T1 < T2` reports each pair of teams only once.

SECTION 8

Canonical form

The formatter rewrites a manifest into a single form:

- one element per line, two spaces per block level; a block that contains a single element is written on one line if it fits in 80 characters: `person alice { first "Alice". }`;
- the facts of a block before its nested blocks, in the original order;
- a space after each comma, none before; a space on each side of operators and comparisons, none around `..`;
- rules longer than 80 characters broken after `:-`, the body indented by four spaces, one literal per line if it does not fit on one; a head that is too long broken between its arguments, aligned under the parenthesis;
- facts longer than 80 characters broken between their arguments, aligned under the first one; an `enum` declaration longer than 80 characters written one value per line;
- comments kept: those that precede an element stay before it, the one that ends its line still ends it, two spaces after the longest element of a run of consecutive lines that have one; an empty line between two elements is kept, a single one.

The canonical form is a fixed point: formatting twice changes nothing. A generator that writes it directly produces manifests that reviews and diffs handle without noise.

SECTION 9

Interoperability

9.1 Writing

Producing a manifest requires writing text that follows the grammar of §2. A generator only needs to escape strings; the canonical form (§8) gives it an unambiguous writing.

9.2 Interchange format

The model M of a set of files is exported in an interchange format, which a tool reads without parsing the language. Every conforming implementation produces the same export, byte for byte (§3.4).

The export is a UTF-8 text in the JSON Lines format: each line is a JSON object (7), ended by a newline. The first line is the header: the version of the language, and the selected version of each module (§3.2.3).

Member	Contents
`loaf`	the version of the language: `"0.1"`
`modules`	an object that associates each loaded module with its selected version: `{"rbac": "0.1.0"}`

Each following line is an atom of M whose predicate is declared, by a vocabulary or by the language (`error/2`, `has`). The object has three members:

Member	Contents
`pred`	the path of the predicate, qualified by its namespace if it has one: `"team.lead"`, `"rbac::role"`
`args`	the arguments, encoded according to the following table
`at`	the locations of the elements that denote the atom (§3.2), sorted by file, line and column; the list is empty for an atom that is only deduced

Term	Encoding	Example
integer	JSON number	`18`
constant	JSON string	`"bruno"`
string	object `str`	`{"str": "Alice"}`
tuple `(t1, …, tn)`	object `tuple`	`{"tuple": ["shell", 2]}`
compound term `f(t1, …, tn)`	object `f`, `args`	`{"f": "qty", "args": [512, "mi"]}`
`#inf`, `#sup`	object `bound`	`{"bound": "sup"}`

A fact designated as an argument by an access to a field (`T.lead(P)`, §3.2.1) is the compound term that bears the name of its predicate: `{"f": "team.lead", "args": ["board", "alice"]}`.

A location is an object `{"file": …, "line": …, "column": …}`. `file` is the path of the file as the implementation received it, or as `use` found it for a module (§3.2.2). `line` and `column` give the position of the first character of the element, from 1; the column is counted in Unicode characters. The element of a fact `k(s)` is the block `k s { … }`; that of another fact, the statement or the `enum` declaration that denotes it.

Each line is written in the canonical form of RFC 8785 (8): members sorted, no whitespace. The atom lines are sorted in the order of their bytes. In the introductory example, the header, a fact of the records and a deduced atom are exported as follows:

```
{"loaf":"0.1","modules":{"rbac":"0.1.0"}}
{"args":["workshop","bruno"],"at":
[{"column":3,"file":"association.loaf","line":87}],"pred":"team.lead"}
{"args":["workshop",2],"at":[],"pred":"team.size"}
```

The JSON schema (9) of a line, header or atom, is published with this specification: `schema/loaf-model-0.1.schema.json`. In case of divergence, the specification prevails.

9.3 Document view

The document view presents the blocks of the model M of an accepted set of files as a single JSON object, in the shape of a configuration document. It is written in the canonical form of RFC 8785 (8). It contains only the atoms of the blocks and of their fields: the other predicates, `has` and `error/2` do not appear in it.

Keys. The key of a term is its rendering (§6): a constant gives its name, a string its contents, an integer its decimal writing. The view has one member per block name that has subjects; its value is an object with one member per subject, keyed by the subject, whose value is the object of the fields of that block.

Fields. The object of a block has one member per field that has at least one value for this block, keyed by the name of the field, prefixed by `ns::` if a namespace declares it (`"gdpr::consent"`). Its value depends on the declaration of the field:

Declaration	Value	Example
flag `f.`	`true`	`"public": true`
`f s.`, `f? s.`	the value	`"first": "Alice"`
`f* s.`, `f+ s.`	an array of the values, in the order of terms (§3.3)	`"team": ["board", "workshop"]`
table `f* k => v.`	an object keyed by the keys; one level per key sort	`"env": {"LANG": "en", "MODE": "prod"}`
list `f* [s].`	an array of the values, by increasing position	`"dns": ["10.0.0.53", "10.0.0.54"]`
sub-block `f* s { … }`	an object keyed by the subjects of the sub-block, whose values are the objects of their fields	`"meeting": {"thursday": {"room": "room_b", "start": 18}}`

A value with several sorts (`slot* day, int.`) is an array of its components: `"slot": [["monday", 9], ["thursday", 18]]`.

Values. Each value is written according to the sort of its position: a number for `int`; the interchange encoding (§9.2) for `term`; for any other sort, the rendering as a JSON string when the language defines it (an integer, a string, a constant, or `cat(…)` of such terms), the interchange encoding otherwise, for a term whose rendering an application defines (`render`, §6): `"limit": {"f": "qty", "args": [500, "milli"]}`. The introductory example (§1.2) has the following view, here indented:

```
{"person": {"alice": {"email": "Alice.Martin@asso.example",
                      "first": "Alice", "last": "Martin",
                      "role": ["treasurer"], "team": ["board", "workshop"]},
            "bruno": {"first": "Bruno", "last": "Petit",
                      "role": ["volunteer"], "team": ["workshop"]}},
 "team": {"board": {"lead": "alice", "member": ["alice"]},
          "workshop": {"lead": "bruno", "member": ["alice", "bruno"],
                       "meeting": {"thursday": {"room": "room_b",
                                                "start": 18}},
                       "size": 2}}}
```

Undefined view. The view is undefined, and an implementation refuses to produce it (**`L012`**), when a subject or a key has no rendering defined by the language, when a value of a sort other than `int` and `term` has no rendering, or when two distinct terms take the same place: two subjects of a block or two keys of a table with the same key (`alice` and `"alice"`), two values of a field with the same writing.

9.4 ASP form

The ASP form writes the program P of a set of files (§3.3) in the syntax of ASP-Core-2 (3), so that an application can add its own rules A and hand $P \cup A$ to an ASP solver. It is produced after the checks that precede evaluation (§4.5), without evaluating P: the rules of A may change its models.

The form contains every fact and every rule of P, those that the language adds for cardinality included (§4.4), and its weak constraints; `#maximize { w@p, t : L }` is written as the weak constraint

`:~ L. [-(w)@p, t]`. Its first line is a comment that gives the version of the language and the selected version of each module; a directive `#defined p/n.` of clingo (4) follows for each declared predicate, so that a predicate without facts is not taken for a misspelt one.

Names. A predicate path, or a function symbol named after one (§9.2), becomes an identifier: `::` is replaced by `__` and `.` by `_`. `team.lead` gives `team_lead`, `rbac::person.consent` gives `rbac__person_consent`, and the fact designated by `T.lead(P)` is the term `team_lead(T, P)`. The other names are unchanged; the variables introduced by elaboration receive fresh names. If two distinct predicates, or two distinct function symbols, of the same arity give the same identifier (`a.b_c` and `a_b.c` both give `a_b_c`), the form is undefined (**L013**).

```
% loaf 0.1, modules: rbac 0.1.0
team_member(T, P) :- person_team(P, T).
team_lead(board, alice).
error(cat(P, " leads ", T, " without being a member of it"), team_lead(T, P)) :-
    team_lead(T, P), not team_member(T, P).
```

For a set of files without choice rules, the unique stable model of the ASP form is the model M, with the predicate names of this section.

References

1. BRADNER, Scott. Key words for use in RFCs to Indicate Requirement Levels. *Request for Comments*. March 1997. No. 2119. DOI 10.17487/RFC2119.
2. LEIBA, Barry. Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. *Request for Comments*. May 2017. No. 8174. DOI 10.17487/RFC8174.
3. CALIMERI, Francesco, FABER, Wolfgang, GEBSER, Martin, IANNI, Giovambattista, KAMINSKI, Roland, KRENNWALLNER, Thomas, LEONE, Nicola, MARATEA, Marco, RICCA, Francesco and SCHAUB, Torsten. ASP-Core-2 Input Language Format. *Theory and Practice of Logic Programming*. 2020. Vol. 20, no. 2, p. 294–309. DOI 10.1017/S1471068419000450.
4. GEBSER, Martin, KAMINSKI, Roland, KAUFMANN, Benjamin and SCHAUB, Torsten. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*. 2019. Vol. 19, no. 1, p. 27–82. DOI 10.1017/S1471068418000054.
5. APT, Krzysztof R., BLAIR, Howard A. and WALKER, Adrian. Towards a Theory of Declarative Knowledge. In : *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988. p. 89–148.
6. GELFOND, Michael and LIFSCHITZ, Vladimir. The Stable Model Semantics for Logic Programming. In : *Proceedings of the Fifth International Conference and Symposium on Logic Programming*. MIT Press, 1988. p. 1070–1080.
7. BRAY, Tim. The JavaScript Object Notation (JSON) Data Interchange Format. *Request for Comments*. December 2017. No. 8259. DOI 10.17487/RFC8259.
8. RUNDGREN, Anders, JORDAN, Bret and ERDTMAN, Samuel. JSON Canonicalization Scheme (JCS). *Request for Comments*. June 2020. No. 8785. DOI 10.17487/RFC8785.
9. WRIGHT, Austin, ANDREWS, Henry, HUTTON, Ben and DENNIS, Greg. JSON Schema: A Media Type for Describing JSON Documents, draft 2020-12. Online. June 2022. Available from: https://json-schema.org/draft/2020-12/json-schema-core

Funding

« Ce projet a été financé par le Gouvernement dans le cadre du plan France 2030 opéré par Cap Digital et financé par l'Union européenne – NextGeneration EU »
« Ce projet a été financé par le gouvernement dans le cadre de France 2030 »
« Financé par l'Union européenne - Next Generation EU dans le cadre du plan France Relance »







