ASP-Core-2
Input Language Format

ASP Standardization Working Group
Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski,
Thomas Krennwallner, Nicola Leone, Francesco Ricca, Torsten Schaub

November 3, 2015

ASP-Core-2
Input Language Format

Change Log

Current version: 2.03b.

- **2.00** Nov. 16th, 2012. First public release of the document.
- **2.01** Nov. 21th, 2012. Explicit support of negative integers in grammar table.
- **2.02** Dec. 4th, 2012. Addition of optimize statements as syntactic shortcuts.
- **2.03** Dec. 6th, 2012. Simplifications and rewritings of grammar and restrictions.
- **2.03b** Dec. 13th, 2012. Some fixes to Section 2.3 and 2.4.
- **2.03c** Nov. 03rd, 2014. Minor.

1 Language Syntax

For the sake of readability, the language specification is herein given in the traditional mathematical notation. A lexical matching table from the following notation to the actual raw input format is provided in Section 5.

1.1 Terms.

Terms are either *constants*, *variables*, *arithmetic terms* or *functional terms*. Constants can be either *symbolic constants* (strings starting with some lowercase letter), *string constants* (quoted strings) or *integers*. Variables are denoted by strings starting with some uppercase letter. An *arithmetic term* has form $-(t)$ or $(t \diamond u)$ for terms t and u with $\diamond \in \{$"+", "−", "∗", "/"$\}$; parentheses can optionally be omitted in which case standard operator precedences apply. Given a *functor* f (the *function name*) and terms $t_1, \ldots, t_n$, the expression $f(t_1, \ldots, t_n)$ is a *functional term* if $n > 0$, whereas $f()$ is a synonym for the symbolic constant f.

1.2 Atoms and Naf-Literals.

A *predicate atom* has form $p(t_1, \ldots, t_n)$, where p is a *predicate name*, $t_1, \ldots, t_n$ are terms and $n \geq 0$ is the arity of the predicate atom; a predicate atom $p()$ of arity 0 is likewise represented by its predicate name p without parentheses. Given a predicate atom q, q and $\neg q$ are *classical atoms*. A *built-in atom* has form $t \prec u$ for terms t and u with $\prec \in \{$"<", "≤", "=", "≠", ">", "≥"$\}$. Built-in atoms a as well as the expressions a and **not** a for a classical atom a are *naf-literals*.

1.3 Aggregate Literals.

An *aggregate element* has form

$$t_1, \ldots, t_m : l_1, \ldots, l_n$$

where $t_1, \ldots, t_m$ are terms and $l_1, \ldots, l_n$ are naf-literals for $m \geq 0$ and $n \geq 0$.

An *aggregate atom* has form

$$\#\mathrm{aggr}\{e_1; \ldots; e_n\} \prec u$$

where $e_1, \ldots, e_n$ are aggregate elements for $n \geq 0$, #aggr ∈ {"#count", "#sum", "#max", "#min"} is an *aggregate function name*, $\prec \in \{$"<", "≤", "=", "≠", ">", "≥"$\}$ is an *aggregate relation* and u is a term. Given an aggregate atom a, the expressions a and **not** a are *aggregate literals*.

In the following, we write *atom* (resp., *literal*) without further qualification to refer to some classical, built-in or aggregate atom (resp., naf- or aggregate literal).

1.4 Rules.

A *rule* has form

$$h_1 \mid \ldots \mid h_m :- b_1, \ldots, b_n.$$

where $h_1, \ldots, h_m$ are classical atoms and $b_1, \ldots, b_n$ are literals for $m \geq 0$ and $n \geq 0$.

1.5 Weak Constraints.

A *weak constraint* has form

$$:\sim b_1, \ldots, b_n. [w@l, t_1, \ldots, t_m]$$

where $t_1, \ldots, t_m$ are terms and $b_1, \ldots, b_n$ are literals for $m \geq 0$ and $n \geq 0$; w and l are terms standing for a *weight* and a *level*. Writing the part "@l" can optionally be omitted if $l = 0$; that is, a weak constraint has level 0 unless specified otherwise.

1.6 Queries.

A *query* Q has form a?, where a is a classical atom.

1.7 Programs.

An *ASP-Core-2 program* is a set of rules and weak constraints, possibly accompanied by a (single) query.[1] A program (rule, weak constraint, query, literal, aggregate element, etc.) is *ground* if it contains no variables.

2 Semantics

We herein give the full model-theoretic semantics of ASP-Core-2. As for non-ground programs, the semantics extends the traditional notion of Herbrand interpretation, taking care of the fact that *all* integers are part of the Herbrand universe. The semantics of propositional programs is based on [8], extended to aggregates according to [4, 5]. Choice atoms [15] are treated in terms of the reduction given in Section 3.2.

We restrict the given semantics to programs containing non-recursive aggregates (see Section 6 for this and further restrictions to the family of admissible programs), for which the general semantics presented herein is in substantial agreement with a variety of proposals for adding aggregates to ASP [2, 3, 6, 7, 9–16].

2.1 Herbrand Interpretation.

Given a program P, the *Herbrand universe* of P, denoted by U_P, consists of all integers and (ground) terms constructible from constants and functors appearing in P. The *Herbrand base* of P, denoted by B_P, is the set of all (ground) classical atoms that can be built by combining predicate names appearing in P with terms from U_P as arguments. A (Herbrand) *interpretation* I for P is a *consistent* subset of B_P; that is, $\{q, \neg q\} \nsubseteq I$ must hold for each predicate atom $q \in B_P$.

2.2 Ground Instantiation.

A *substitution* σ is a mapping from a set V of variables to the Herbrand universe U_P of a given program P. For some object O (term, classical atom, rule, weak constraint, query, literal, aggregate element, etc.), we denote by $O\sigma$ the object obtained by replacing each occurrence of a variable $v \in V$ by $\sigma(v)$ in O.

A variable is *global* in a rule, weak constraint or query r if it appears outside of aggregate elements in r. A substitution from the set of global variables in r is a *global substitution for* r; a substitution from the set of variables in an aggregate element e is a (local) *substitution for* e. A

[1] Unions of conjunctive queries (and more) can be expressed by including appropriate rules in a program.

global substitution σ for r (or substitution σ for e) is *well-formed* if the arithmetic evaluation, performed in the standard way, of any arithmetic subterm ($-(t)$ or $(t \diamond u)$ with $\diamond \in \{$"+", "−", "∗", "/"$\}$) appearing outside of aggregate elements in $r\sigma$ (or appearing in $e\sigma$) is well-defined.

Given a collection $\{e_1; \ldots ; e_n\}$ of aggregate elements, the *instantiation* of $\{e_1; \ldots ; e_n\}$ is the following set of aggregate elements:

$$\text{inst}(\{e_1; \ldots ; e_n\}) = \bigcup_{1 \le i \le n} \{e_i\sigma \mid \sigma \text{ is a well-formed substitution for } e_i\}$$

A *ground instance* of a term, classical atom, naf-literal, rule, weak constraint, or query r is obtained in two steps: (1) a well-formed global substitution σ for r is applied to r; (2) for every aggregate atom $\#\text{aggr}\{e_1; \ldots ; e_n\} \prec u$ appearing in $r\sigma$, $\{e_1; \ldots ; e_n\}$ is replaced by $\text{inst}(\{e_1; \ldots ; e_n\})$ (where aggregate elements are syntactically separated by ";").

The *arithmetic evaluation* of a ground instance r of some term, classical atom, naf-literal, rule, weak constraint or query is obtained by replacing any maximal arithmetic subterm appearing in r by its integer value, which is calculated in the standard way.[2] The *ground instantiation* of a program P, denoted by $grnd(P)$, is the set of arithmetically evaluated ground instances of rules and weak constraints in P.

2.3 Term Ordering and Satisfaction of Naf-Literals.

A ground classical atom $a \in B_P$ is *true* w.r.t. a (consistent) interpretation $I \subseteq B_P$ if $a \in I$. Let t and u be arithmetically evaluated ground terms. To determine whether a built-in atom $t \prec u$ (with $\prec \in \{$"<", "≤", "=", "≠", ">", "≥"$\}$) holds, we rely on a total order $\preceq$ on terms in U_P defined as follows:

- $t \preceq u$ for integers t and u if $t \le u$;
- $t \preceq u$ for any integer t and any symbolic constant u;
- $t \preceq u$ for symbolic constants t and u if t is lexicographically smaller than or equal to u;
- $t \preceq u$ for any symbolic constant t and any string constant u;
- $t \preceq u$ for string constants t and u if t is lexicographically smaller than or equal to u;
- $t \preceq u$ for any string constant t and any functional term u;
- $t \preceq u$ for functional terms $t = f(t_1, \ldots , t_m)$ and $u = g(u_1, \ldots , u_n)$ if
  - $m < n$ (the arity of t is smaller than the arity of u),
  - $m \le n$ and $g \not\preceq f$ (the functor of t is smaller than the one of u, while arities coincide) or
  - $m \le n$, $f \preceq g$ and, for any $1 \le j \le m$ such that $t_j \not\preceq u_j$, there is some $1 \le i < j$ such that $u_i \not\preceq t_i$ (the tuple of arguments of t is smaller than or equal to the arguments of u).

Then, $t \prec u$ is *true* w.r.t. I if $t \preceq u$ for $\prec$ = "≤"; $u \preceq t$ for $\prec$ = "≥"; $t \preceq u$ and $u \not\preceq t$ for $\prec$ = "<"; $u \preceq t$ and $t \not\preceq u$ for $\prec$ = ">"; $t \preceq u$ and $u \preceq t$ for $\prec$ = "="; $t \not\preceq u$ or $u \not\preceq t$ for $\prec$ = "≠". A positive ground naf-literal a is *true* w.r.t. I if a is a classical or built-in atom that is *true* w.r.t. I; otherwise, a is *false* w.r.t. I. A negative ground naf-literal **not** a is *true* (or *false*) w.r.t. I if a is *false* (or *true*) w.r.t. I.

2.4 Satisfaction of Aggregate Literals.

An *aggregate function* is a mapping from sets of tuples of ground terms to ground terms. The aggregate functions associated with aggregate function names introduced in Section 1.3 map a set T of tuples of ground terms to a ground term as follows:

[2] Note that the outcomes of arithmetic evaluation are well-defined relative to well-formed substitutions.

- $\#\text{count}(T) = |T|$;
- $\#\text{sum}(T) = \sum_{(t_1,\ldots,t_m) \in T,\, t_1 \text{ is an integer}} t_1$;
- $\#\text{max}(T) = \max\{t_1 \mid (t_1,\ldots,t_m) \in T\}$;
- $\#\text{min}(T) = \min\{t_1 \mid (t_1,\ldots,t_m) \in T\}$.

The terms selected by $\#\text{max}(T)$ and $\#\text{min}(T)$ are determined relative to the total order $\leq$ on terms in U_P (see Section 2.3); in the special case of an empty set, i.e. $T = \emptyset$, we adopt the convention that $\#\text{max}(\emptyset) \leq u$ and $u \leq \#\text{min}(\emptyset)$ for every term $u \in U_P$. An expression $\#\text{aggr}(T) \prec u$ is *true* (or *false*) for $\#\text{aggr} \in \{$"#count", "#sum", "#max", "#min"$\}$, an aggregate relation $\prec \in \{$"<", "≤", "=", "≠", ">", "≥"$\}$ and a ground term u if $\#\text{aggr}(T) \prec u$ is *true* (or *false*) according to the corresponding definition for built-in atoms given in Section 2.3.

An interpretation $I \subseteq B_P$ maps a collection E of aggregate elements to the following set of tuples of ground terms:

$$\text{eval}(E, I) = \{(t_1,\ldots,t_m) \mid t_1,\ldots,t_m : l_1,\ldots,l_n \text{ occurs in } E \text{ and } l_1,\ldots,l_n \text{ are } \textit{true} \text{ w.r.t. } I\}$$

A positive aggregate literal $a = \#\text{aggr}\{e_1;\ldots;e_n\} \prec u$ is *true* (or *false*) w.r.t. I if $\#\text{aggr}(\text{eval}(\{e_1;\ldots;e_n\}, I)) \prec u$ is *true* (or *false*) w.r.t. I; **not** a is *true* (or *false*) w.r.t. I if a is *false* (or *true*) w.r.t. I.[3]

2.5 Answer Sets.

Given a program P and a (consistent) interpretation $I \subseteq B_P$, a rule $h_1 \mid \ldots \mid h_m {:-}\, b_1,\ldots,b_n.$ in $grnd(P)$ is *satisfied* w.r.t. I if some $h \in \{h_1,\ldots,h_m\}$ is *true* w.r.t. I when $b_1,\ldots,b_n$ are *true* w.r.t. I; I is a *model* of P if every rule in $grnd(P)$ is satisfied w.r.t. I. The *reduct* of P w.r.t. I, denoted by P^I, consists of the rules $h_1 \mid \ldots \mid h_m {:-}\, b_1,\ldots,b_n.$ in $grnd(P)$ such that $b_1,\ldots,b_n$ are *true* w.r.t. I; I is an *answer set* of P if I is a $\subseteq$-minimal model of P^I. In other words, an answer set I of P is a model of P such that no proper subset of I is a model of P^I.

The semantics of P is given by the collection of its answer sets, denoted by $AS(P)$.

2.6 Optimal Answer Sets.

To select optimal members of $AS(P)$, we map an interpretation I for P to a set of tuples as follows:

$$\text{weak}(P, I) = \{(w@l, t_1,\ldots,t_m) \mid \\ :\sim b_1,\ldots,b_n.\,[w@l, t_1,\ldots,t_m] \text{ occurs in } grnd(P) \text{ and } b_1,\ldots,b_n \text{ are } \textit{true} \text{ w.r.t. } I\}$$

For any integer l, let

$$P^I_l = \sum_{(w@l, t_1,\ldots,t_m) \in \text{weak}(P,I),\, w \text{ is an integer}} w$$

denote the sum of integers w over tuples with $w@l$ in $\text{weak}(P, I)$. Then, an answer set $I \in AS(P)$ is *dominated* by $I' \in AS(P)$ if there is some integer l such that $P^{I'}_l < P^I_l$ and $P^{I'}_{l'} = P^I_{l'}$ for all integers $l' > l$. An answer set $I \in AS(P)$ is *optimal* if there is no $I' \in AS(P)$ such that I is dominated by I'. Note that P has some (and possibly more than one) optimal answer set if $AS(P) \neq \emptyset$.

[3] In view of the aforementioned extension of $\leq$ to $\#\text{max}(\emptyset)$ and $\#\text{min}(\emptyset)$, the truth values of $\#\text{min}(\emptyset) \prec u$ and $\#\text{max}(\emptyset) \prec u$ are well-defined (solely relying on $\prec \in \{$"<", "≤", "=", "≠", ">", "≥"$\}$). For instance, $\#\text{min}(\text{eval}(\{0 : p, \textbf{not } p\}, I)) > 0$ and $\#\text{min}(\text{eval}(\{0 : p, \textbf{not } p\}, I)) \neq 0$ evaluate to *true* for any interpretation I; this still applies when arbitrary other terms are used in place of 0. On the other hand, $\#\text{max}(\text{eval}(\{0 : p, \textbf{not } p\}, I)) > 0$ and $\#\text{max}(\text{eval}(\{0 : p, \textbf{not } p\}, I)) = 0$ are *false* w.r.t. any interpretation I.

2.7 Queries.

Given a program P along with a (single) query $a?$, let $Ans(a, P)$ denote the set of arithmetically evaluated ground instances a' of a such that $a' \in I$ for all $I \in AS(P)$. The set $Ans(a, P)$, which includes all arithmetically evaluated ground instances of a if $AS(P) = \emptyset$, constitutes the *answers* to $a?$. That is, query answering corresponds to cautious (or skeptical) reasoning as defined in [1].

3 Syntactic Shortcuts

This section specifies additional constructs by reduction to the language introduced in Section 1.

3.1 Anonymous Variables.

An *anonymous variable* in a rule, weak constraint or query is denoted by "_" (character underscore). Each occurrence of "_" stands for a fresh variable in the respective context (i.e., different occurrences of anonymous variables represent distinct variables).

3.2 Choice Rules.

A *choice element* has form

$$a : l_1, \ldots, l_k$$

where a is a classical atom and $l_1, \ldots, l_k$ are naf-literals for $k \geq 0$.

A *choice atom* has form

$$\{e_1; \ldots; e_m\} \prec u$$

where $e_1, \ldots, e_m$ are choice elements for $m \geq 0$, $\prec$ is an aggregate relation (see Section 1.3) and u is a term. The part "$\prec u$" can optionally be omitted if $\prec$ stands for "$\geq$" and $u = 0$.

A *choice rule* has form

$$\{e_1; \ldots; e_m\} \prec u \colon - \, b_1, \ldots, b_n.$$

where $\{e_1; \ldots; e_m\} \prec u$ is a choice atom and $b_1, \ldots, b_n$ are literals for $n \geq 0$.

Intuitively, a choice rule means that, if the body of the rule is *true*, an arbitrary subset of $\{e_1, \ldots, e_m\}$ can be chosen as *true* in order to comply with the provided aggregate relation to u. In the following, this intuition is captured by means of a proper mapping of choice rules to rules without choice atoms (in the head).

For any predicate atom $q = p(t_1, \ldots, t_n)$, let $\widehat{q} = \hat{p}(1, t_1, \ldots, t_n)$ and $\widehat{\neg q} = \hat{p}(0, t_1, \ldots, t_n)$, where $\hat{p} \neq p$ is an (arbitrary) predicate and function name that is uniquely associated with p, and the first argument (which can be 1 or 0) indicates the "polarity" q or $\neg q$, respectively.[4]

Then, a choice rule stands for the rules

$$a_i \mid \widehat{a_i} \colon - \, b_1, \ldots, b_n, l_{1_i}, \ldots, l_{k_i}.$$

for each $1 \leq i \leq m$ along with the single constraint

$$: - \, b_1, \ldots, b_n, \textbf{not} \; \#\text{count}\{\widehat{a_1} : a_1, l_{1_1}, \ldots, l_{k_1}; \ldots; \widehat{a_m} : a_m, l_{1_m}, \ldots, l_{k_m}\} \prec u.$$

The first group of rules expresses that the classical atom a_i in a choice element $a_i : l_{1_i}, \ldots, l_{k_i}$ for $1 \leq i \leq m$ can be chosen as *true* (or *false*) if $b_1, \ldots, b_n$ and $l_{1_i}, \ldots, l_{k_i}$ are *true*. This "generates" all

[4] It is assumed that fresh predicate and function names are outside of possible program signatures and cannot be referred to within user input.

subsets of the atoms in choice elements. On the other hand, the second rule, which is an integrity constraint, requires the condition $\{e_1; \dots; e_m\} \prec u$ to hold if $b_1, \dots, b_n$ are *true*.[5]

For illustration, consider the choice rule

$$\{p(a) : q(2); \neg p(a) : q(3)\} \leq 1 {:-}\, q(1).$$

Using the fresh predicate and function name $\hat{p}$, the choice rule is mapped to three rules as follows:

$$\begin{aligned} p(a) \mid \hat{p}(1, a) &{:-}\, q(1), q(2). \\ \neg p(a) \mid \hat{p}(0, a) &{:-}\, q(1), q(3). \\ &{:-}\, q(1), \textbf{not}\ \#\text{count}\{\hat{p}(1, a) : p(a), q(2); \hat{p}(0, a) : \neg p(a), q(3)\} \leq 1. \end{aligned}$$

Note that the three rules are satisfied w.r.t. an interpretation I such that $\{q(1), q(2), q(3), \hat{p}(1, a), \hat{p}(0, a)\} \subseteq I$ and $\{p(a), \neg p(a)\} \cap I = \emptyset$. In fact, when $q(1)$, $q(2)$, and $q(3)$ are *true*, the choice of none or one of the atoms $p(a)$ and $\neg p(a)$ complies with the aggregate relation "$\leq$" to 1.

3.3 Aggregate Relations.

An aggregate or choice atom

$$\#\text{aggr}\{e_1; \dots; e_m\} \prec u \quad \text{or} \quad \{e_1; \dots; e_m\} \prec u$$

may be written as

$$u \prec^{-1} \#\text{aggr}\{e_1; \dots; e_m\} \quad \text{or} \quad u \prec^{-1} \{e_1; \dots; e_m\}$$

where "$<$"$^{-1}$ = "$>$"; "$\leq$"$^{-1}$ = "$\geq$"; "$=$"$^{-1}$ = "$=$"; "$\neq$"$^{-1}$ = "$\neq$"; "$>$"$^{-1}$ = "$<$"; "$\geq$"$^{-1}$ = "$\leq$".

The left and right notation of aggregate relations may be combined in expressions as follows:

$$u_1 \prec_1 \#\text{aggr}\{e_1; \dots; e_m\} \prec_2 u_2 \quad \text{or} \quad u_1 \prec_1 \{e_1; \dots; e_m\} \prec_2 u_2$$

Such expressions are mapped to available constructs according to the following transformations:

◇ $u_1 \prec_1 \{e_1; \dots; e_m\} \prec_2 u_2 {:-}\, b_1, \dots, b_n.$ stands for

$$\begin{aligned} &u_1 \prec_1 \{e_1; \dots; e_m\} {:-}\, b_1, \dots, b_n. \\ &\{e_1; \dots; e_m\} \prec_2 u_2 {:-}\, b_1, \dots, b_n. \end{aligned}$$

◇ $h_1 \mid \dots \mid h_k {:-}\, b_1, \dots, b_{i-1}, u_1 \prec_1 \#\text{aggr}\{e_1; \dots; e_m\} \prec_2 u_2, b_{i+1}, \dots, b_n.$ stands for

$$h_1 \mid \dots \mid h_k {:-}\, b_1, \dots, b_{i-1}, u_1 \prec_1 \#\text{aggr}\{e_1; \dots; e_m\}, \#\text{aggr}\{e_1; \dots; e_m\} \prec_2 u_2, b_{i+1}, \dots, b_n.$$

◇ $h_1 \mid \dots \mid h_k {:-}\, b_1, \dots, b_{i-1}, \textbf{not}\ u_1 \prec_1 \#\text{aggr}\{e_1; \dots; e_m\} \prec_2 u_2, b_{i+1}, \dots, b_n.$ stands for

$$\begin{aligned} &h_1 \mid \dots \mid h_k {:-}\, b_1, \dots, b_{i-1}, \textbf{not}\ u_1 \prec_1 \#\text{aggr}\{e_1; \dots; e_m\}, b_{i+1}, \dots, b_n. \\ &h_1 \mid \dots \mid h_k {:-}\, b_1, \dots, b_{i-1}, \textbf{not}\ \#\text{aggr}\{e_1; \dots; e_m\} \prec_2 u_2, b_{i+1}, \dots, b_n. \end{aligned}$$

◇ $:\sim b_1, \dots, b_{i-1}, u_1 \prec_1 \#\text{aggr}\{e_1; \dots; e_m\} \prec_2 u_2, b_{i+1}, \dots, b_n. [w@l, t_1, \dots, t_k]$ stands for

$$:\sim b_1, \dots, b_{i-1}, u_1 \prec_1 \#\text{aggr}\{e_1; \dots; e_m\}, \#\text{aggr}\{e_1; \dots; e_m\} \prec_2 u_2, b_{i+1}, \dots, b_n. [w@l, t_1, \dots, t_k]$$

◇ $:\sim b_1, \dots, b_{i-1}, \textbf{not}\ u_1 \prec_1 \#\text{aggr}\{e_1; \dots; e_m\} \prec_2 u_2, b_{i+1}, \dots, b_n. [w@l, t_1, \dots, t_k]$ stands for

$$\begin{aligned} &:\sim b_1, \dots, b_{i-1}, \textbf{not}\ u_1 \prec_1 \#\text{aggr}\{e_1; \dots; e_m\}, b_{i+1}, \dots, b_n. [w@l, t_1, \dots, t_k] \\ &:\sim b_1, \dots, b_{i-1}, \textbf{not}\ \#\text{aggr}\{e_1; \dots; e_m\} \prec_2 u_2, b_{i+1}, \dots, b_n. [w@l, t_1, \dots, t_k] \end{aligned}$$

---

[5] In disjunctive heads of rules of the first form, an occurrence of $\widehat{a_i}$ denotes an (auxiliary) *atom* that is linked to the original atom a_i. Given the relationship between a_i and $\widehat{a_i}$, the latter is reused as a *term* in the body of a rule of the second form. That is, we overload the notation $\widehat{a_i}$ by letting it stand both for an atom (in disjunctive heads) and a term (in #count aggregates).

3.4 Optimize Statements.

An *optimize statement* has form

$$\#\text{opt}\{w_1@l_1, t_{1_1}, \ldots, t_{m_1} : b_{1_1}, \ldots, b_{n_1}; \ldots; w_k@l_k, t_{1_k}, \ldots, t_{m_k} : b_{1_k}, \ldots, b_{n_k}\}.$$

where #opt $\in$ {"#minimize", "#maximize"}, $w_i, l_i, t_{1_i}, \ldots, t_{m_i}$ are terms and $b_{1_i}, \ldots, b_{n_i}$ are naf-literals for $k \geq 0$, $1 \leq i \leq k$, $m_i \geq 0$ and $n_i \geq 0$. Similar to weak constraints (cf. Section 1.5), w_i and l_i stand for a *weight* and a *level*, and writing "$@l_i$" can optionally be omitted if $l_i = 0$.

An optimize statement stands for the weak constraints

$$:\sim b_{1_1}, \ldots, b_{n_1}. [w'_1@l_1, t_{1_1}, \ldots, t_{m_1}] \qquad \ldots \qquad :\sim b_{1_k}, \ldots, b_{n_k}. [w'_k@l_k, t_{1_k}, \ldots, t_{m_k}]$$

where $w'_i = w_i$ (or $w'_i = -w_i$) for $1 \leq i \leq k$ if #opt = "#minimize" (or #opt = "#maximize").

4 EBNF Grammar

```
<program>            ::= [<statements>] [<query>]

<statements>         ::= [<statements>] <statement>
<query>              ::= <classical_literal> QUERY_MARK

<statement>          ::= CONS [<body>] DOT
                       | <head> [CONS [<body>]] DOT
                       | WCONS [<body>] DOT
                         SQUARE_OPEN <weight_at_level> SQUARE_CLOSE
                       | <optimize> DOT

<head>               ::= <disjunction> | <choice>
<body>               ::= [<body> COMMA]
                         (<naf_literal> | [NAF] <aggregate>)

<disjunction>        ::= [<disjunction> OR] <classical_literal>

<choice>             ::= [<term> <binop>]
                         CURLY_OPEN [<choice_elements>]
                         CURLY_CLOSE [<binop> <term>]
<choice_elements>    ::= [<choice_elements> SEMICOLON]
                         <choice_element>
<choice_element>     ::= <classical_literal> [COLON [<naf_literals>]]

<aggregate>          ::= [<term> <binop>] <aggregate function>
                         CURLY_OPEN  [<aggregate_elements>]
                         CURLY_CLOSE [<binop> <term>]
<aggregate_elements> ::= [<aggregate_elements> SEMICOLON]
                         <aggregate_element>
<aggregate_element>  ::= [<terms>] [COLON [<naf_literals>]]
<aggregate_function> ::= AGGREGATE_COUNT
```

```
                     | AGGREGATE_MAX
                     | AGGREGATE_MIN
                     | AGGREGATE_SUM

<optimize>           ::= <optimize_function>
                       CURLY_OPEN  [<optimize_elements>]
                       CURLY_CLOSE
<optimize_elements>  ::= [<optimize_elements> SEMICOLON]
                       <optimize_element>
<optimize_element>   ::= <weight_at_level> [COLON [<naf_literals>]]
<optimize_function>  ::= MAXIMIZE | MINIMIZE

<weight_at_level>    ::= <term> [AT <term>] [COMMA <terms>]

<naf_literals>       ::= [<naf_literals> COMMA] <naf_literal>
<naf_literal>        ::= [NAF] <classical_literal> | <builtin_atom>

<classical_literal>  ::= [MINUS] ID [PAREN_OPEN [<terms>] PAREN_CLOSE]
<builtin_atom>       ::= <term> <binop> <term>

<binop>              ::= EQUAL
                     | UNEQUAL
                     | LESS
                     | GREATER
                     | LESS_OR_EQ
                     | GREATER_OR_EQ

<terms>              ::= [<terms> COMMA] <term>
<term>               ::= ID [PAREN_OPEN [<terms>] PAREN_CLOSE]
                     | NUMBER
                     | STRING
                     | VARIABLE
                     | ANONYMOUS_VARIABLE
                     | PAREN_OPEN <term> PAREN_CLOSE
                     | MINUS <term>
                     | <term> <arithop> term>
<arithop>            ::= PLUS
                     | MINUS
                     | TIMES
                     | DIV
```

5 Lexical Matching Table

<table>
<tr><th>Token Name</th><th>Mathematical Notation
used within this document (exemplified)</th><th>Lexical Format
(Flex Notation)</th></tr>
<tr><td>ID</td><td>$a, b, anna, \ldots$</td><td>[a-z][A-Za-z0-9_]*</td></tr>
<tr><td>VARIABLE</td><td>$X, Y, Name, \ldots$</td><td>[A-Z][A-Za-z0-9_]*</td></tr>
<tr><td>STRING</td><td>“http://bit.ly/cw6lDS”, “Peter”, . . .</td><td>\"([^\"]|\\\")*\"</td></tr>
<tr><td>NUMBER</td><td>1, 0, 100000, . . .</td><td>"0"|[1-9][0-9]*</td></tr>
<tr><td>ANONYMOUS_VARIABLE</td><td>_</td><td>"_"</td></tr>
<tr><td>DOT</td><td>.</td><td>"."</td></tr>
<tr><td>COMMA</td><td>,</td><td>","</td></tr>
<tr><td>QUERY_MARK</td><td>?</td><td>"?"</td></tr>
<tr><td>COLON</td><td>:</td><td>":"</td></tr>
<tr><td>SEMICOLON</td><td>;</td><td>";"</td></tr>
<tr><td>OR</td><td>|</td><td>"|"</td></tr>
<tr><td>NAF</td><td>not</td><td>"not"</td></tr>
<tr><td>CONS</td><td>: −</td><td>":-"</td></tr>
<tr><td>WCONS</td><td>:~</td><td>":~"</td></tr>
<tr><td>PLUS</td><td>+</td><td>"+"</td></tr>
<tr><td>MINUS</td><td>− or ¬</td><td>"-"</td></tr>
<tr><td>TIMES</td><td>*</td><td>"*"</td></tr>
<tr><td>DIV</td><td>/</td><td>"/"</td></tr>
<tr><td>AT</td><td>@</td><td>"@"</td></tr>
<tr><td>PAREN_OPEN</td><td>(</td><td>"("</td></tr>
<tr><td>PAREN_CLOSE</td><td>)</td><td>")"</td></tr>
<tr><td>SQUARE_OPEN</td><td>[</td><td>"["</td></tr>
<tr><td>SQUARE_CLOSE</td><td>]</td><td>"]"</td></tr>
<tr><td>CURLY_OPEN</td><td>{</td><td>"{"</td></tr>
<tr><td>CURLY_CLOSE</td><td>}</td><td>"}"</td></tr>
<tr><td>EQUAL</td><td>=</td><td>"="</td></tr>
<tr><td>UNEQUAL</td><td>≠</td><td>"<>"|"!="</td></tr>
<tr><td>LESS</td><td><</td><td>"<"</td></tr>
<tr><td>GREATER</td><td>></td><td>">"</td></tr>
<tr><td>LESS_OR_EQ</td><td>≤</td><td>"<="</td></tr>
<tr><td>GREATER_OR_EQ</td><td>≥</td><td>">="</td></tr>
<tr><td>AGGREGATE_COUNT</td><td>#count</td><td>"#count"</td></tr>
<tr><td>AGGREGATE_MAX</td><td>#max</td><td>"#max"</td></tr>
<tr><td>AGGREGATE_MIN</td><td>#min</td><td>"#min"</td></tr>
<tr><td>AGGREGATE_SUM</td><td>#sum</td><td>"#sum"</td></tr>
<tr><td>MINIMIZE</td><td>#minimize</td><td>"#minimi"[zs]"e"</td></tr>
<tr><td>MAXIMIZE</td><td>#maximize</td><td>"#maximi"[zs]"e"</td></tr>
<tr><td>COMMENT</td><td>% this is a comment</td><td>"%"([^*\n][^\n]*)?\n</td></tr>
<tr><td>MULTI_LINE_COMMENT</td><td>%* this is a comment *%</td><td>"%*"([^*]|*[^%])*"*%"</td></tr>
<tr><td>BLANK</td><td></td><td>[\t\n]+</td></tr>
</table>

Lexical values are given in Flex[6] syntax. The COMMENT, MULTI_LINE_COMMENT and BLANK tokens can be freely interspersed amidst other tokens and have no syntactic or semantic meaning.

[6] http://flex.sourceforge.net/

6 Using ASP-Core-2 in Practice – Restrictions

To promote declarative programming as well as practical system implementation, ASP-Core-2 programs are supposed to comply with the restrictions listed in the following. This particularly applies to input programs in the System Track of the *4th Answer Set Programming Competition*.

6.1 Safety.

For enabling a retrieval of values for variables from atoms within the variables' scope, any rule, weak constraint or query is required to be safe. To this end, for a set V of variables and literals $b_1, \ldots, b_n$, we inductively (starting from an empty set of bound variables) define $v \in V$ as *bound* by $b_1, \ldots, b_n$ if v occurs outside of arithmetic terms in some b_i for $1 \leq i \leq n$ such that b_i is

- a classical atom,
- a built-in atom $t = u$ or $u = t$ and any member of V occurring in t is bound by $b_1, \ldots, b_n$ or
- an aggregate atom $\#\text{aggr}\, E = u$ and any member of V occurring in E is bound by $b_1, \ldots, b_n$.

The entire set V of variables is *bound* by $b_1, \ldots, b_n$ if each $v \in V$ is bound by $b_1, \ldots, b_n$.

A rule, weak constraint or query r is *safe* if the set V of global variables in r is bound by $b_1, \ldots, b_n$ (taking a query r to be of form b_1?) and, for each aggregate element $t_1, \ldots, t_k : l_1, \ldots, l_m$ in r with occurring variable set W, the set $W \setminus V$ of local variables is bound by $l_1, \ldots, l_m$. For instance, the rule $p(X, Y) :- q(X), \#\text{sum}\{S, X : r(T, X), S = (2 * T) - X\} = Y.$ is safe, while $p(X, Y) :- q(X), \#\text{sum}\{S, X : r(T, X), S + X = 2 * T\} = Y.$ is not safe.

6.2 Finiteness.

Input programs in the System Track of the *4th Answer Set Programming Competition* must not have infinite or infinitely many answer sets. For example, a program including $p(X + 1) :- p(X).$ or $p(f(X)) :- p(X).$ along with a fact like $p(0).$ is not an admissible input in the System Track. Finiteness must be witnessed via a known maximum integer and maximum function nesting level per problem instance, which correctly limit the absolute values of integers as well as the depths of functional terms occurring as arguments in the atoms of answer sets.

6.3 Aggregates.

For the sake of an uncontroversial semantics, we require aggregates to be non-recursive. To make this precise, for any predicate atom $q = p(t_1, \ldots, t_n)$, let $q^v = p/n$ and $\neg q^v = \neg p/n$. We further define the directed *predicate dependency graph* $D_P = (V, E)$ for a program P by

- the set V of vertices a^v for all classical atoms a appearing in P and
- the set E of edges $(h_i^v, h_1^v), \ldots, (h_i^v, h_m^v)$ and $(h_1^v, a^v), \ldots, (h_m^v, a^v)$ for all rules $h_1 \mid \ldots \mid h_m :- b_1, \ldots, b_n.$ in P, $1 \leq i \leq m$ and classical atoms a appearing in $b_1, \ldots, b_n$.

The aggregates in P are *non-recursive* if, for any classical atom a appearing within aggregate elements in a rule $h_1 \mid \ldots \mid h_m :- b_1, \ldots, b_n.$ in P, there is no path from a^v to h_i^v in D_P for $1 \leq i \leq m$.

6.4 Predicate Arities.

The arity of atoms sharing some predicate name is not assumed to be fixed. However, system implementers are encouraged to issue proper warning messages if an input program includes classical atoms with the same predicate name but different arities.

6.5 Undefined Arithmetics.

While substitutions that lead to undefined arithmetic subterms (and are thus not well-formed) are "automatically" excluded by ground instantiation as specified in Section 2.2, rooting the semantics of a program on such clearance would make grounding cumbersome in practice. For instance, the (single) answer set of the one-rule program $p{:}-$ **not** $q(0/0)$. must be empty, and any a priori simplification relying on the absence of a definition for predicate q is probably mistaken.

In order to avoid grounding complications, however, a program P shall be invariant under undefined arithmetics; that is, $grnd(P)$ is supposed to be equivalent to any ground program P' obtainable from P by freely replacing arithmetic subterms with undefined outcomes by arbitrary terms from U_P instead of dropping an underlying (non-well-formed) substitution. As a matter of fact, the one-rule program considered above does not satisfy this condition (e.g., it is not equivalent to $p{:}-$ **not** $q(0)$.), while the semantics of the alternative program $p{:}-r,$ **not** $q(0/0)$. is invariant under undefined arithmetics.

7 Acknowledgements

The authors of this document thank Vladimir Lifschitz and all the Texas Action Group mailing list members for their useful comments and suggestions for improvements.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate functions in DLV. In Marina De Vos and Alessandro Provetti, editors, *Proceedings of the 2nd International Workshop on Answer Set Programming (ASP 2003)*, pages 274–288, 2003. Online at `http://CEUR-WS.org/Vol-78/`.
3. Marc Denecker, Nikolay Pelov, and Maurice Bruynooghe. Ultimate well-founded and stable model semantics for logic programs with aggregates. In Philippe Codognet, editor, *Proceedings of the 17th International Conference on Logic Programming (ICLP 2001)*, pages 212–226. Springer Verlag, 2001.
4. Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In José Júlio Alferes and João Leite, editors, *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA 2004)*, pages 200–212. Springer Verlag, 2004.
5. Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011. Special Issue: John McCarthy's Legacy.
6. Paolo Ferraris. Answer sets for propositional theories. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2005)*, pages 119–131. Springer Verlag, 2005.
7. Michael Gelfond. Representing knowledge in A-prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, pages 413–451. Springer Verlag, 2002.
8. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
9. David B. Kemp and Peter J. Stuckey. Semantics of logic programs with aggregates. In Vijay A. Saraswat and Kazunori Ueda, editors, *Proceedings of the 8th International Symposium on Logic Programming (ISLP 1991)*, pages 387–401. MIT Press, 1991.
10. Mauricio Osorio and Bharat Jayaraman. Aggregation and negation-as-failure. *New Generation Computing*, 17(3):255–284, 1999.

11. Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Partial stable models for logic programs with aggregates. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004)*, pages 207–219. Springer Verlag, 2004.
12. Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*, 7(3):301–353, 2007.
13. Nikolay Pelov and Mirosław Truszczyński. Semantics of disjunctive programs with monotone aggregates – an operator-based approach. In James Delgrande and Torsten Schaub, editors, *Proceedings of the 10th International Workshop on Non-Monotonic Reasoning (NMR 2004)*, pages 327–334, 2004.
14. Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997.
15. Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.
16. Allen Van Gelder. The well-founded semantics of aggregation. In Moshe Y. Vardi and Paris C. Kanellakis, editors, *Proceedings of the 11th Symposium on Principles of Database Systems (PODS 1992)*, pages 127–138. ACM Press, 1992.