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

ASP-Core-2
Input Language Format

Change Log

Current version: 2.02.

2.0	Nov 16th, 2012. First public release of the document.
2.01	Nov 21th, 2012. Explicit support of negative integers in grammar table.
2.01b	Dec 3rd, 2012. Small grammar/lexical fixes. Addition of examples.

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 $n \geq 0$, $\#\mathrm{aggr} \in \{$"#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

$$h_1 \mid \ldots \mid h_m \leftarrow b_1, \ldots, b_n.$$

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

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

where $t_1, \dots, t_m$ are terms and $b_1, \dots, b_n$ 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 is of the form $q?$, where q 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. Note that unions of conjunctive queries (and more) can be easily expressed by means of the inclusion of appropriate rules in the program. A program (a rule, a literal, an atom, a term, a query) 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 [10], extended to aggregates according to [6, 7]. Choice atoms [17] 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 allowed programs), for which the general semantics presented herein is in substantial agreement with a variety of proposals for adding aggregates to ASP [4, 5, 8, 9, 11–18].

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 of 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 (aggregate element, literal, rule, weak constraint, etc.), we denote by $O\sigma$ the object obtained by replacing each variable $v \in V$ by $\sigma(v)$ in O.

A variable is *global* in a rule or weak constraint 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 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 rule or weak constraint 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 rule or weak constraint is obtained by replacing any maximal arithmetic subterm appearing in r by its integer value, which is calculated in the standard way.[1] 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 classical atom $a \in B_P$ is *true* w.r.t. a (consistent) interpretation $I \subseteq B_P$ if $a \in I$. To determine whether a built-in atom $t \prec u$ (with $\prec \in \{``<", ``\le", ``=", ``\ne", ``>", ``\ge"\}$) 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 $g \npreceq f$ (the functor of t is smaller than the one of u, while arities coincide) or
  - $m \le n$, $f \preceq g$ and, for any $j = 1, \ldots, m$ such that $t_j \npreceq u_j$, there is some $i = 1, \ldots, j{-}1$ such that $u_i \npreceq t_i$ (the tuple of arguments of t is smaller than or equal to arguments of u).

Then, $t \prec u$ is *true* w.r.t. I if $t \preceq u$ for $\prec = ``\le"$; $u \preceq t$ for $\prec = ``\ge"$; $t \preceq u$ and $u \npreceq t$ for $\prec = ``<"$; $u \preceq t$ and $t \npreceq u$ for $\prec = ``>"$; $t \preceq u$ and $u \preceq t$ for $\prec = ``="$; $t \npreceq u$ or $u \npreceq t$ for $\prec = ``\ne"$. A positive 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 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 terms to terms. The aggregate functions associated with aggregate function names introduced in Section 1.3 map a set T of tuples of terms to a term as follows:

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

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

The terms selected by #max(T) and #min(T) are determined relative to the total order $\preceq$ 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 #max($\emptyset$) $\preceq u$ and $u \preceq$ #min($\emptyset$) for every term $u \in U_P$. An expression #aggr(T) $\prec u$ is *true* (or *false*) for #aggr $\in$ {"#count", "#sum", "#max", "#min"}, an aggregate relation $\prec \in$ {"<", "≤", "=", "≠", ">", "≥"} and a term u if #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 terms:

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

A positive aggregate literal $a =$ #aggr$\{e_1; \ldots; e_n\} \prec u$ is *true* (or *false*) w.r.t. I if #aggr(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.[2]

2.5 Answer Sets.

Given a program P and a (consistent) interpretation $I \subseteq B_P$, a rule $h_1 \mid \ldots \mid h_m \leftarrow b_1, \ldots, b_n$. 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 $h_1 \mid \ldots \mid h_m \leftarrow b_1, \ldots, b_n$. 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 set of all answer sets for it, denoted by $AS(P)$.

2.6 Optimal Answer Sets.

To select optimal answer sets in $AS(P)$, we map an interpretation I for P to the following set of tuples:

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

For any integer l, let

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

denote the sum of integers w over tuples with $w@l$ in weak(P, I). Then, an answer set I of P is *dominated* by an answer set I' of P if there is some integer l such that $P_l^{I'} < P_l^I$ and $P_{l'}^{I'} = P_{l'}^I$ for all integers $l' > l$. An answer set I of P is *optimal* if there is no answer set I' of P such that I is dominated by I'. Note that a program P that has answer sets might have one or more optimal answer sets.

2.7 Ground Queries.

Given a ground query $Q = q?$ of a program P, Q is true if $\forall I \in AS(P)$ q is true w.r.t. I. Otherwise, Q is false. Note that, if $AS(P) = \emptyset$, all queries are true. Note that query answering, according to this definition, corresponds to cautious (skeptical) reasoning as defined in [1].

---

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

2.8 Non-Ground Queries.

Given the non-ground query $Q = q(t_1, \ldots, t_n)$? of a program P, let $Ans(Q, P)$ be the set of all substitutions σ for Q such that $Q\sigma$ is true. The set $Ans(Q, P)$ constitutes the set of answers to Q. Note that, if $AS(P) = \emptyset$, $Ans(Q, P)$ contains all possible substitutions for Q.

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 or weak constraint is denoted by "_" (character underscore). An occurrence of "_" stands for a fresh variable in the context of the rule or weak constraint at hand (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

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

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 (that can be 1 or 0) indicates the "polarity" q or $\neg q$, respectively.[3]

Then, a choice rule stands for the rules

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

for each $i, (1 \leq i \leq m)$, and for the single constraint

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

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

[3] It is assumed that fresh predicate/function names are outside of possible program signatures and cannot be used 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*.[4]

For illustration, consider the choice rule

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

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

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

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:

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

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

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

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

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

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

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

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

---

[4] 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).

4 EBNF Grammar

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

<rules>              ::= [<rule> <rules>]
<query>              ::= <classic literal> QUERY_MARK

<rule>               ::= CONS <body> DOT |
                         <head> [[CONS] <body>] DOT |
                         WCONS <body> DOT [<weights_at_levels>]

<head>               ::= <disjunction> |
                         <choice_atom>
<body>               ::= <conjunction>

<weights_at_levels>  ::= SQUARE_OPEN TERM [AT TERM]
                             [TERM_SEPARATOR <terms>] SQUARE_CLOSE

<disjunction>        ::= [<disjunction> HEAD_SEPARATOR]
                             <classic_literal>
<conjunction>        ::= [<conjunction> BODY_SEPARATOR]
                             (<naf_literal> | [NAF] <aggregate>)

<choice_atom>        ::= [<term> <binop>] CURLY_OPEN <choice_elements>
                             CURLY_CLOSE [<binop> <term>]
<choice_elements>    ::= [<choice_elements> SEMICOLON] <choice_element>
<choice_element>     ::= <atom> [COLON <naf_literals>]

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

<arithop>            ::= PLUS |
                         MINUS |
                         TIMES |
                         DIV

<aggregate_atom>     ::= [<term> <binop>] <aggregate function>
                             CURLY_OPEN  <aggregate_elements>
                             CURLY_CLOSE <binop> <term> |
                         <term> <binop> <aggregate function>
                             CURLY_OPEN <aggregate_elements>
                             CURLY_CLOSE [<binop> <term>]
<aggregate_elements> ::= [<aggregate_elements> SEMICOLON]
```

```
                             <aggregate_element>
<aggregate_element>  ::= <basic_terms> COLON <naf_literals>

<aggregate_function> ::= AGGR_COUNT |
                         AGGR_MAX |
                         AGGR_MIN |
                         AGGR_SUM

<atom>               ::= <predicate_name>
                             [PARAM_OPEN [<terms>] PARAM_CLOSE]
<builtin_atom>       ::= <term> <binop> <term>
<classic_literal>    ::= [NEG] <atom>

<naf_literals>       ::= [<naf_literals> BODY_SEPARATOR] <naf_literal>
<naf_literal>        ::= [NAF] <classic_literal> |
                         <builtin_atom>

<terms>              ::= [<terms> TERM_SEPARATOR] <term>
<basic_terms>        ::= [<basic_terms> TERM_SEPARATOR] <basic_term>
<term>               ::= <basic_term> |
                         <expression_term> |
                         <function_term>

<basic_term>         ::= <ground_term> |
                         <variable_term>
<ground_term>        ::= SYMBOLIC_CONSTANT |
                         STRING | [MINUS] NUMBER
<variable_term>      ::= VARIABLE |
                         ANON_VAR
<function_term>      ::= <predicate_name> PARAM_OPEN <terms>
                             PARAM_CLOSE
<expression_term>    ::= <expression_term> <arithop>
                             <expression_term> |
                         PARAM_OPEN <expression_term>
                             PARAM_CLOSE |
                          <ground_term> | [MINUS] VARIABLE

<predicate_name>     ::= ID |
                         STRING
```

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>$p, P, q_1, \ldots$</td><td>[A-Za-z]([A-Za-z] | [0-9] | "_")*</td></tr>
<tr><td>SYMBOLIC_CONSTANT</td><td>$a, b, anna, \ldots$</td><td>[a-z]([A-Za-z] | [0-9] | "_")*</td></tr>
<tr><td>VARIABLE</td><td>$X, Y, Name, \ldots$</td><td>[A-Z]([A-Za-z] | [0-9] | "_")*</td></tr>
<tr><td>STRING</td><td>“http : //bit.ly/cw6lDS”, “Peter”, ...</td><td>\"([^\"]|\\\")*\"</td></tr>
<tr><td>ANON_VAR</td><td>_</td><td>"_"</td></tr>
<tr><td>NUMBER</td><td>$1, 0, 100000, \ldots$</td><td>[0-9]+</td></tr>
<tr><td>DOT</td><td>.</td><td>"."</td></tr>
<tr><td>BODY_SEPARATOR</td><td>,</td><td>","</td></tr>
<tr><td>TERM_SEPARATOR</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>HEAD_SEPARATOR</td><td>$|$</td><td>"|"</td></tr>
<tr><td>NEG</td><td>$\neg$</td><td>"-"</td></tr>
<tr><td>NAF</td><td>not</td><td>"not"</td></tr>
<tr><td>CONS</td><td>$\leftarrow$</td><td>:-</td></tr>
<tr><td>PLUS</td><td>$+$</td><td>"+"</td></tr>
<tr><td>MINUS</td><td>$-$</td><td>"-"</td></tr>
<tr><td>TIMES</td><td>$*$</td><td>"*"</td></tr>
<tr><td>DIV</td><td>$/$</td><td>"/"</td></tr>
<tr><td>PARAM_OPEN</td><td>(</td><td>"("</td></tr>
<tr><td>PARAM_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>$\neq$</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>$\leq$</td><td>"<="</td></tr>
<tr><td>GREATER_OR_EQ</td><td>$\geq$</td><td>">="</td></tr>
<tr><td>AGGR_COUNT</td><td>$\#count$</td><td>"#count"</td></tr>
<tr><td>AGGR_MAX</td><td>$\#max$</td><td>"#max"</td></tr>
<tr><td>AGGR_MIN</td><td>$\#min$</td><td>"#min"</td></tr>
<tr><td>AGGR_SUM</td><td>$\#sum$</td><td>"#sum"</td></tr>
<tr><td>COMMENT</td><td>% this is a comment</td><td>"%".*$</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[5] syntax. The COMMENT and BLANK tokens can be freely interspersed amidst other tokens and have no syntactical or semantic meaning.

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

6 Using ASP-Core-2 in Practice – Restrictions

A number of restrictions and specific assumptions must be taken into account while writing ASP-Core-2 programs (in particular, all the following specifications are assumed within the System Track of *4th Answer Set Programming Competition*).

6.1 Safety.

Programs are assumed to be *safe*. A program P is safe if all its rules are safe; a rule r is *safe* if any variable X appearing in r is safe in the following sense:

1. if X is global, it is safe if either:
   - X appears in a positive predicate atom in the body of r, or
   - X appears in a built-in atom $X = Y \diamond Z$ in the body of r, having X as its left-hand side, and Y and Z are safe, or
   - X appears in a positive aggregate atom in the form $X = \#f\{Conj\}$ and all other variables in the atom are safe.
2. if X is local to an aggregate element $\{V : Conj\}$ appearing as a term in V, then it appears in an atom of *Conj* as well.

6.2 Programs with Function Symbols and Integers.

Programs with function symbols and integers are in principle subject to no restriction. However, for the sake of Competition, and in order to facilitate implementors, it is prescribed that

- each selected problem encoding P must provably have finitely many finite answer sets for any of its benchmark instance B_i, that is $AS(P \cup B_i)$ must be a finite set of finite elements. "Proofs" of finiteness can be given in terms of membership to a known decidable class of programs with functions and/or integers, or any other *formal* mean.
- a bound k_P on the maximum nesting level of terms, and a bound m_P on the maximum integer value appearing in answer sets originated from P must be known. That is, for any instance B_i and for any term t appearing in $AS(P \cup B_i)$, the nesting level of t must not be greater than k_P and, if t is an integer it must not exceed m_P.

The values m_P and k_P will be provided in input to participant systems, when invoked on P.

6.3 Aggregate Literals.

For *aggregate elements* in the form

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

$t_1, \ldots, t_m$ are assumed to be either constants or variables.

6.4 Non-Recursiveness of Aggregates and Conditional Literals.

Recursive aggregates shall not appear within an encoding selected for the Competition. Formally, given a ASP-Core-2 program P, we define the *(labeled) dependency graph* $DG(P)$ between predicates of P, for which

- a node is present for each predicate p appearing P;

- an arc $p \leftarrow q$ appears in $DG(P)$ if there is a rule $r \in P$ in which p appears in the head and q appears in a predicate atom in the body;
- an arc $p \leftarrow_a q$ appears in $DG(P)$ if there is a rule $r \in P$ in which p appears in the head and q appears in an aggregate body atom;
- two arcs $p \leftarrow q$ and $q \leftarrow p$ appear in $DG(P)$ if p and q both appear in the head of some rule $r \in P$.

We say that P has no recursive aggregates (or that P is stratified with respect to aggregation) if there is no cycle in $DG(P)$ containing an edge of the form $p \leftarrow_a q$.

6.5 Restrictions on Disjunction.

Arbitrary usage of disjunction in programs might cause a shift in complexity towards $F - \Sigma_2^P$ [3]. In order to encourage the participation of Systems not implementing full disjunction, encodings for problems belonging to the P and NP category shall be provided in terms of head-cycle free programs [2].

6.6 Invariance under 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 $p \leftarrow \textbf{not}\ q(0/0).$ 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, $grnd(P)$ shall 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 $p \leftarrow \textbf{not}\ q(0).$), while the semantics of the alternative program $p \leftarrow r, \textbf{not}\ q(0/0).$ is invariant under undefined arithmetics.

6.7 Predicate Arities.

The arity of predicate names is not assumed to be fixed. Implementors are suggested to issue proper warning messages, should an input encoding present predicate atoms with different arities and same predicate name.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.
3. Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
4. 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 ASP03 - Answer Set Programming: Advances in Theory and Implementation*, pages 274–288, Messina, Italy, September 2003. Online at `http://CEUR-WS.org/Vol-78/`.

5. 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*, pages 212–226. Springer Verlag, 2001.
6. 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 Artificial Intelligence (JELIA 2004)*, volume 3229 of *Lecture Notes in AI (LNAI)*, pages 200–212. Springer Verlag, September 2004.
7. 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.
8. Paolo Ferraris. Answer Sets for Propositional Theories. Available via the author's homepage at `http://www.cs.utexas.edu/users/otto/papers/proptheories.ps`, 2004.
9. Michael Gelfond. Representing Knowledge in A-Prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic. Logic Programming and Beyond*, volume 2408 of *LNCS*, pages 413–451. Springer, 2002.
10. Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
11. David B. Kemp and Peter J. Stuckey. Semantics of Logic Programs with Aggregates. In Vijay A. Saraswat and Kazunori Ueda, editors, *Proceedings of the International Symposium on Logic Programming (ISLP'91)*, pages 387–401. MIT Press, 1991.
12. Mauricio Osorio and Bharat Jayaraman. Aggregation and Negation-As-Failure. *New Generation Computing*, 17(3):255–284, 1999.
13. Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Partial stable models for logic programs with aggregates. In *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, volume 2923 of *Lecture Notes in AI (LNAI)*, pages 207–219. Springer, 2004.
14. 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.
15. Nikolay Pelov and Mirosław Truszczyński. Semantics of disjunctive programs with monotone aggregates - an operator-based approach. In *Proceedings of the 10th International Workshop on Non-monotonic Reasoning (NMR 2004), Whistler, BC, Canada*, pages 327–334, 2004.
16. Kenneth A. Ross and Yehoshua Sagiv. Monotonic Aggregation in Deductive Databases. *Journal of Computer and System Sciences*, 54(1):79–97, February 1997.
17. Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.
18. Allen Van Gelder. The Well-Founded Semantics of Aggregation. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS'92)*, pages 127–138. ACM Press, 1992.