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

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Log</td>
<td>3</td>
</tr>
<tr>
<td>Language Syntax</td>
<td>4</td>
</tr>
<tr>
<td>1.1 Terms</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Atoms and Naf-Literals</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Aggregate Literals</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Rules</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Weak Constraints</td>
<td>5</td>
</tr>
<tr>
<td>1.6 Queries</td>
<td>5</td>
</tr>
<tr>
<td>1.7 Programs</td>
<td>5</td>
</tr>
<tr>
<td>Semantics</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Herbrand Interpretation</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Ground Instantiation</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Term Ordering and Satisfaction of Naf-Literals</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Satisfaction of Aggregate Literals</td>
<td>6</td>
</tr>
<tr>
<td>2.5 Answer Sets</td>
<td>7</td>
</tr>
<tr>
<td>2.6 Optimal Answer Sets</td>
<td>7</td>
</tr>
<tr>
<td>2.7 Ground Queries</td>
<td>7</td>
</tr>
<tr>
<td>2.8 Non-Ground Queries</td>
<td>8</td>
</tr>
<tr>
<td>Syntactic Shortcuts</td>
<td>8</td>
</tr>
<tr>
<td>3.1 Anonymous Variables</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Choice Rules</td>
<td>8</td>
</tr>
<tr>
<td>3.3 Aggregate Relations</td>
<td>9</td>
</tr>
<tr>
<td>EBNF Grammar</td>
<td>10</td>
</tr>
<tr>
<td>Lexical Matching Table</td>
<td>12</td>
</tr>
<tr>
<td>Using ASP-Core-2 in Practice – Restrictions</td>
<td>13</td>
</tr>
<tr>
<td>6.1 Safety</td>
<td>13</td>
</tr>
<tr>
<td>6.2 Programs with Function Symbols and Integers</td>
<td>13</td>
</tr>
<tr>
<td>6.3 Aggregate Literals</td>
<td>13</td>
</tr>
<tr>
<td>6.4 Non-Recursiveness of Aggregates and Conditional Literals</td>
<td>13</td>
</tr>
<tr>
<td>6.5 Restrictions on Disjunction</td>
<td>14</td>
</tr>
<tr>
<td>6.6 Invariance under Undefined Arithmetics</td>
<td>14</td>
</tr>
<tr>
<td>6.7 Predicate Arities</td>
<td>14</td>
</tr>
</tbody>
</table>
Change Log

Current version: 2.02.

2.01 Nov 21th, 2012. Explicit support of negative integers in grammar table.
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 \((t \circ u)\) for terms \(t\) and \(u\) with \(\circ \in \{+, -, \times, /\}\); 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 \(\prec \in \{<, \leq, =, \neq, >, \geq\}\). Built-in atoms \(a\) as well as the expressions \(a\) and \(\text{not } a\) 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

\[ \#aggr[e_1; \ldots; e_n] \prec u \]

where \(e_1, \ldots, e_n\) are aggregate elements for \(n \geq 0\), \#aggr \(\in \{\#\text{count}, \#\text{sum}, \#\text{max}, \#\text{min}\}\) is an aggregate function name, \(\prec \in \{<, \leq, =, \neq, >, \geq\}\) is an aggregate relation and \(u\) is a term. Given an aggregate atom \(a\), the expressions \(a\) and \(\text{not } a\) 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

\[ h_1 | \ldots | h_m \leftarrow b_1, \ldots, b_n. \]

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
\[ b_1, \ldots, b_n [w@l, t_1, \ldots, t_m] \]
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 \( Q \) 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, \( \{ q, \neg q \} \not\subseteq I \) must hold for each predicate atom \( q \in B_P \).

2.2 Ground Instantiation.

A substitution \( \sigma \) 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 \( \sigma \) for \( r \) (or substitution \( \sigma \) for \( e \)) is well-formed if the arithmetic evaluation, performed
in the standard way, of any arithmetic subterm \((-t)\) or \((t \circ u)\) with \(\circ \in \{\"+\", \"-\", \"\times\", \"/\"\}\) 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:

\[
\text{inst}(\{e_1; \ldots; e_n\}) = \bigcup_{\sigma \in \text{SIS}} e_i \sigma \mid \sigma \text{ is a well-formed substitution for } e_i
\]

A ground instance of a rule or weak constraint \(r\) is obtained in two steps: (1) a well-formed global substitution \(\sigma\) for \(r\) is applied to \(r\); (2) for every aggregate atom \(\text{aggr}(e_1; \ldots; e_n) < u\) 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 \(\text{grnd}(P)\), 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 \(t < u\) (with \(\in \{\"<\", \"\leq\", \"\neq\", \"\geq\", \">\", \"\geq\"\}\)) holds, we rely on a total order \(\leq\) on terms in \(U_P\) defined as follows:

\[
\begin{align*}
&- t \leq u \text{ for integers } t \text{ and } u \text{ if } t \leq u; \\
&- t \leq u \text{ for any integer } t \text{ and any symbolic constant } u; \\
&- t \leq u \text{ for symbolic constants } t \text{ and } u \text{ if } t \text{ is lexicographically smaller than or equal to } u; \\
&- t \leq u \text{ for any symbolic constant } t \text{ and any string constant } u; \\
&- t \leq u \text{ for string constants } t \text{ and } u \text{ if } t \text{ is lexicographically smaller than or equal to } u; \\
&- t \leq u \text{ for any string constant } t \text{ and any functional term } u; \\
&- t \leq u \text{ for functional terms } t = f(t_1, \ldots, t_m) \text{ and } u = g(u_1, \ldots, u_n) \text{ if } \\
&\quad \bullet m < n \text{ (the arity of } t \text{ is smaller than the arity of } u), \\
&\quad \bullet m \leq n \text{ and } g \neq f \text{ (the functor of } f \text{ is smaller than the one of } u, \text{ while arities coincide) or } \\
&\quad \bullet m \leq n, f \neq g \text{ and, for any } j = 1, \ldots, m \text{ such that } t_j \neq u_j, \text{ there is some } i = 1, \ldots, j-1 \text{ such that } u_i \neq t_i \text{ (the tuple of arguments of } t \text{ is smaller than or equal to arguments of } u). \\
\end{align*}
\]

Then, \(t < u\) is true w.r.t. \(I\) if \(t \leq u\) for \(\leq = "\leq"\); \(u \leq t\) for \(\leq = "\geq"\); \(t \leq u\) and \(u \neq t\) for \(\leq = "\neq"\); \(t \leq u\) and \(u \lneq t\) for \(\leq = "<"\); \(u \lneq t\) and \(t \lneq u\) for \(\leq = ">)\); \(t \leq u\) and \(u \lneq t\) for \(\leq = "\geq"\); \(t \lneq u\) and \(u \lneq t\) for \(\leq = "\neq"\); \(t \neq u\) and \(u \neq t\) for \(\leq = "\neq"\). 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:

\[
\begin{align*}
&- \#\text{count}(T) = |T|; \\
&- \#\text{sum}(T) = \sum_{(t_1, \ldots, t_m) \in T} t_i \text{ if } t_i \text{ is an integer } t_1; \\
&- \#\text{max}(T) = \max\{t_1 \mid (t_1, \ldots, t_m) \in T\}; \\
&- \#\text{min}(T) = \min\{t_1 \mid (t_1, \ldots, t_m) \in T\}.
\end{align*}
\]

\(^1\) 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 \( \# \max(\emptyset) \leq u \) and \( u \leq \# \min(\emptyset) \) for every term \( u \in U_P \). An expression \( \# \text{aggr}(T) < u \) is true (or false) for \( \# \text{aggr} \in \{ \# \text{count}, \# \text{sum}, \# \max, \# \min \} \), an aggregate relation \( \prec \in \{ \prec \leq, \preceq, =, \# <, \# >, \# \geq \} \) and a term \( u \) if \( \# \text{aggr}(T) < u \) 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:

\[
eval(E, I) = \{(t_1, \ldots, t_m) | t_1, \ldots, t_m \text{ occur in } E \text{ and } l_1, \ldots, l_n \text{ are true w.r.t. } I\}
\]

A positive aggregate literal \( a = \# \text{aggr}(e_1; \ldots; e_n) < u \) is true (or false) w.r.t. \( I \) if \( \# \text{aggr}( \text{eval}(e_1; \ldots; e_n), I) < u \) 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. \( I \).

### 2.5 Answer Sets.

Given a program \( P \) and a (consistent) interpretation \( I \subseteq B_P \), a rule \( h_1 | \ldots | h_m \leftarrow b_1, \ldots, b_n \) in \( \text{grnd}(P) \) 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 \( \text{grnd}(P) \) is satisfied w.r.t. \( I \). The reduct of \( P \) w.r.t. \( I \), denoted by \( P_I \), consists of the rules \( h_1 | \ldots | h_m \leftarrow b_1, \ldots, b_n \) in \( \text{grnd}(P) \) such that \( b_1, \ldots, b_n \) are true w.r.t. \( I \); \( I \) is an answer set of \( P \) if \( I \) is a \( \preceq \)-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 \( \text{AS}(P) \).

### 2.6 Optimal Answer Sets.

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

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

For any integer \( l \), let

\[
P_I^l = \sum_{(w@l, t_1, \ldots, t_m) \in \text{weak}(P, I), w \text{ is an integer}^W} p
\]

denote the sum of integers \( w \) over tuples with \( w@l \) in \( \text{weak}(P, I) \). Then, an answer set \( I \) of \( P \) is dominated by an answer set \( I' \) of \( P \) if there is some integer \( l \) such that \( P_I^l < P_I'^l \) and \( P_I^u = P_I'^u \) 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 \( \forall I \in \text{AS}(P) \ q \) is true w.r.t. \( I \). Otherwise, \( Q \) is false. Note that, if \( \text{AS}(P) = \emptyset \), 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 \( \leq \) to \( \# \max(\emptyset) \) and \( \# \min(\emptyset) \), the truth values of \( \# \min(\emptyset) < u \) and \( \# \max(\emptyset) < u \) are well-defined (solely relying on \( \prec \in \{ \prec \leq, \preceq, =, \# <, \# >, \# \geq \} \) for\( \# \min(\text{eval}(0 : p, \not p), I) \) and \( \# \max(\text{eval}(0 : p, \not p), I) \) 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, \( \# \max(\text{eval}(0 : p, \not p), I) = 0 \) as well as \( \# \max(\text{eval}(0 : p, \not p), I) < 0 \) 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 \( \text{Ans}(Q, P) \) be the set of all substitutions \( \sigma \) for \( Q \) such that \( Q\sigma \) is true. The set \( \text{Ans}(Q, P) \) constitutes the set of answers to \( Q \). Note that, if \( \text{AS}(P) = \emptyset \), \( \text{Ans}(Q, P) \) 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

\[ \{e_1; \ldots; e_m\} < u \]

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

A choice rule has form

\[ \{e_1; \ldots; e_m\} < u \leftarrow b_1, \ldots, b_n. \]

where \( \{e_1; \ldots; e_m\} < u \) 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 \( \overline{q} = \overline{p(1, t_1, \ldots, t_n)} \) and \( \overline{\overline{q}} = \overline{\overline{p(0, t_1, \ldots, t_n)}} \), where \( \overline{p} \neq p \) 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 \( \overline{q} \), respectively.\(^3\)

Then, a choice rule stands for the rules

\[ a_i \mid \overline{a}_i \leftarrow b_1, \ldots, b_n, l_1, \ldots, l_k. \]

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

\[ \leftarrow b_1, \ldots, b_n, \text{not} \#\text{count} [\overline{a}_i : a_1, l_1, \ldots, l_k; \ldots; \overline{a}_m : a_m, l_1, \ldots, l_k] < u. \]

The first group of rules expresses that the classical atom \( a_i \) in a choice element \( a_i : l_1, \ldots, l_k \) for \( 1 \leq i \leq m \) can be chosen as true (or false) if \( b_1, \ldots, b_n \) and \( l_1, \ldots, l_k \) 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 \( \{e_1; \ldots; e_m\} < u \) to hold if \( b_1, \ldots, b_n \) are true.\(^4\)

For illustration, consider the choice rule

\[
(p(a) : q(2); \neg p(a) : q(3)) \leq 1 \iff q(1).
\]

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

\[
\begin{align*}
& p(a) \mid \hat{p}(1, a) \iff q(1), q(2), \\
& \neg p(a) \mid \hat{p}(0, a) \iff q(1), q(3).
\end{align*}
\]

\( \iff q(1), \not\#\text{count}(\hat{p}(1, a) : p(a), q(2); \hat{p}(0, a) : \neg p(a), q(3)) \leq 1. \)

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

\[
\#\text{aggr}[e_1; \ldots; e_m] < u \quad \text{or} \quad \{e_1; \ldots; e_m\} < u
\]

may be written as

\[
u <^{-1} \#\text{aggr}[e_1; \ldots; e_m] \quad \text{or} \quad u <^{-1} \{e_1; \ldots; e_m\}
\]

where: “\( <^{-1} = “>” \), “\( \leq^{-1} = “\geq” \), “\( =^{-1} = “=” \), “\( \#^{-1} = “\#” \), “\( “^{-1} = “<” \), “\( \geq^{-1} = “\leq” \).”

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

\[
u_1 <_1 \#\text{aggr}[e_1; \ldots; e_m] <_2 u_2 \quad \text{or} \quad u_1 <_1 \{e_1; \ldots; e_m\} <_2 u_2
\]

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

- \( u_1 <_1 \{e_1; \ldots; e_m\} <_2 u_2 \iff b_1, \ldots, b_n \) stands for
  \[
u_1 <_1 \{e_1; \ldots; e_m\} \iff b_1, \ldots, b_n.
  \]

- \( h_1; \ldots; h_k \iff b_1, \ldots, b_{n-1}, u_1 <_1 \#\text{aggr}[e_1; \ldots; e_m] <_2 u_2, b_{i+1}, \ldots, b_{n} \) stands for
  \[
h_1; \ldots; h_k \iff b_1, \ldots, b_{n-1}, \#\text{aggr}[e_1; \ldots; e_m] <_2 u_2, b_{i+1}, \ldots, b_{n}.
  \]

- \( h_1; \ldots; h_k \iff b_1, \ldots, b_{n-1}, \not u_1 <_1 \#\text{aggr}[e_1; \ldots; e_m] <_2 u_2, b_{i+1}, \ldots, b_{n} \) stands for
  \[
h_1; \ldots; h_k \iff b_1, \ldots, b_{n-1}, \not\#\text{aggr}[e_1; \ldots; e_m] <_2 u_2, b_{i+1}, \ldots, b_{n}.
  \]

- \( b_1, \ldots, b_{n-1}, u_1 <_1 \#\text{aggr}[e_1; \ldots; e_m] <_2 u_2, b_{i+1}, \ldots, b_{n}, \lnot w@l, t_1, \ldots, t_k \) stands for
  \[
b_1, \ldots, b_{n-1}, u_1 <_1 \#\text{aggr}[e_1; \ldots; e_m] <_2 u_2, b_{i+1}, \ldots, b_{n}, \lnot w@l, t_1, \ldots, t_k\]

- \( b_1, \ldots, b_{n-1}, \not u_1 <_1 \#\text{aggr}[e_1; \ldots; e_m] <_2 u_2, b_{i+1}, \ldots, b_{n}, \lnot w@l, t_1, \ldots, t_k \) stands for
  \[
b_1, \ldots, b_{n-1}, \not u_1 <_1 \#\text{aggr}[e_1; \ldots; e_m] <_2 u_2, b_{i+1}, \ldots, b_{n}, \lnot w@l, t_1, \ldots, t_k\]

\( \)

\( ^4 \) In disjunctive heads of rules of the first form, an occurrence of \( \hat{a}_i \) denotes an (auxiliary) atom that is linked to the original atom \( a_i \). Given the relationship between \( a_i \) and \( \hat{a}_i \), the latter is reused as a term in the body of a rule of the second form. That is, we overload the notation \( \hat{a}_i \) 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]
          <aggregate>
          ([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>
<thead>
<tr>
<th>Token Name</th>
<th>Mathematical Notation used within this document (exemplified)</th>
<th>Lexical Format (Flex Notation)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID</strong></td>
<td>$p, P_q_1,...$</td>
<td>$[^A-Za-z]([^A-Za-z]</td>
</tr>
<tr>
<td><strong>SYMBOLIC_CONSTANT</strong></td>
<td>$a, b, anna,...$</td>
<td>$[^a-z]([^A-Za-z]</td>
</tr>
<tr>
<td><strong>VARIABLE</strong></td>
<td>$X, Y.Name,...$</td>
<td>$[^A-Z]([^A-Za-z]</td>
</tr>
<tr>
<td><strong>STRING</strong></td>
<td>&quot;http : //bit.ly/cw6lDS&quot;, “Peter”,...</td>
<td>&quot;([&quot;]</td>
</tr>
<tr>
<td><strong>ANON_VAR</strong></td>
<td>-</td>
<td>&quot;$&quot; $</td>
</tr>
<tr>
<td><strong>NUMBER</strong></td>
<td>1, 0, 100000,...</td>
<td>$[0-9]*$</td>
</tr>
<tr>
<td><strong>DOT</strong></td>
<td>.</td>
<td>&quot;$&quot; $</td>
</tr>
<tr>
<td><strong>BODY_SEPARATOR</strong></td>
<td>,</td>
<td>&quot;$&quot; $</td>
</tr>
<tr>
<td><strong>TERM_SEPARATOR</strong></td>
<td>,</td>
<td>&quot;$&quot; $</td>
</tr>
<tr>
<td><strong>QUERY_MARK</strong></td>
<td>?</td>
<td>&quot;$?&quot; $</td>
</tr>
<tr>
<td><strong>COLON</strong></td>
<td>:</td>
<td>&quot;$:&quot; $</td>
</tr>
<tr>
<td><strong>SEMICOLON</strong></td>
<td>;</td>
<td>&quot;$;&quot; $</td>
</tr>
<tr>
<td><strong>HEAD_SEPARATOR</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NEG</strong></td>
<td>¬</td>
<td>&quot;$&quot; $</td>
</tr>
<tr>
<td><strong>NAF</strong></td>
<td>not</td>
<td>&quot;$not&quot; $</td>
</tr>
<tr>
<td><strong>CONS</strong></td>
<td>←</td>
<td>&quot;$-&quot; $</td>
</tr>
<tr>
<td><strong>PLUS</strong></td>
<td>+</td>
<td>&quot;$+&quot; $</td>
</tr>
<tr>
<td><strong>MINUS</strong></td>
<td>-</td>
<td>&quot;$-&quot; $</td>
</tr>
<tr>
<td><strong>TIMES</strong></td>
<td>*</td>
<td>&quot;$*&quot; $</td>
</tr>
<tr>
<td><strong>DIV</strong></td>
<td>/</td>
<td>&quot;$/&quot; $</td>
</tr>
<tr>
<td><strong>PARAM_OPEN</strong></td>
<td>(</td>
<td>&quot;$(&quot; $</td>
</tr>
<tr>
<td><strong>PARAM_CLOSE</strong></td>
<td>)</td>
<td>&quot;)&quot; $</td>
</tr>
<tr>
<td><strong>SQUARE_OPEN</strong></td>
<td>[</td>
<td>&quot;[&quot; $</td>
</tr>
<tr>
<td><strong>SQUARE_CLOSE</strong></td>
<td>]</td>
<td>&quot;]&quot; $</td>
</tr>
<tr>
<td><strong>CURLY_OPEN</strong></td>
<td>{</td>
<td>&quot;{&quot; $</td>
</tr>
<tr>
<td><strong>CURLY_CLOSE</strong></td>
<td>}</td>
<td>&quot;}&quot; $</td>
</tr>
<tr>
<td><strong>EQUAL</strong></td>
<td>=</td>
<td>&quot;$=&quot; $</td>
</tr>
<tr>
<td><strong>UNEQUAL</strong></td>
<td>≠</td>
<td>&quot;&lt;&gt;&quot; $</td>
</tr>
<tr>
<td><strong>LESS</strong></td>
<td>&lt;</td>
<td>&quot;&lt;&quot; $</td>
</tr>
<tr>
<td><strong>GREATER</strong></td>
<td>&gt;</td>
<td>&quot;&gt;&quot; $</td>
</tr>
<tr>
<td><strong>LESS_OR_EQ</strong></td>
<td>≤</td>
<td>&quot;=&quot; $</td>
</tr>
<tr>
<td><strong>GREATER_OR_EQ</strong></td>
<td>≥</td>
<td>&quot;&gt;=&quot; $</td>
</tr>
<tr>
<td><strong>AGGR_COUNT</strong></td>
<td>#count</td>
<td>&quot;#count&quot; $</td>
</tr>
<tr>
<td><strong>AGGR_MAX</strong></td>
<td>#max</td>
<td>&quot;#max&quot; $</td>
</tr>
<tr>
<td><strong>AGGR_MIN</strong></td>
<td>#min</td>
<td>&quot;#min&quot; $</td>
</tr>
<tr>
<td><strong>AGGR_SUM</strong></td>
<td>#sum</td>
<td>&quot;#sum&quot; $</td>
</tr>
<tr>
<td><strong>COMMENT</strong></td>
<td>% this is a comment</td>
<td>&quot;%&quot; &quot;$</td>
</tr>
<tr>
<td><strong>MULTI_LINE_COMMENT</strong></td>
<td>% this is a comment %/</td>
<td>&quot;/%&quot;,&quot;%/&quot;</td>
</tr>
<tr>
<td><strong>BLANK</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</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 $X = Y \circ Z$ 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 in $P$;
– 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 \( p \leftarrow q \).

### 6.5 Restrictions on Disjunction.

Arbitrary usage of disjunction in programs might cause a shift in complexity towards \( F - \Sigma^P_2 \) [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 \( p \leftarrow \text{not} \ q(0/0) \) 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, \( \text{grnd}(P) \) 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 \( p \leftarrow \text{not} \ q(0) \)), while the semantics of the alternative program \( p \leftarrow r, \text{not} \ q(0/0) \) 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


