Third ASP Competition
File and language formats

The Competition Organizing Committee

Università della Calabria

Change Log

- **V.1.01**
  - Modified lexical matching table (Sec. 8): now predicate names can start with an uppercase letter.
- **V.1.00**
  - First draft.

1 Standard Language Principles

The System competition will be held over the two language formats ASP-Core and ASP-RfC. The two languages have been conceived according to the following goals:

1. Include no less than the constructs appearing in the original A-Prolog language as formulated in [7], and be compliant with the LPNMR 2004 core language draft [1].
2. Include, as an extension, a reduced number of features which are seen both as highly desirable and have now maturity for entering a standard language for ASP;
3. The above extensions should be appropriately chosen, in a way such that the cost of alignment of the input format would be fair enough to allow existing and future ASP solvers to comply with.
4. Have a non-ambiguous semantics over which widespread consensus has been reached;

2 Language Overview

According to goal 1, ASP-Core includes a language with disjunctive heads and strong and NAF negation, and does not require domain predicates; according to goals 2 and 3, ASP-RfC includes ASP-Core as a fragment, with the conservative addition, as native features, of non-recursive aggregates, both with set and multiset semantics, and function symbols. The chosen aggregates are `#sum`, `#count`, `#max` and `#min`. Choices on the design of the ASP-RfC format allow also to comply with goal 4.

ASP-Core is a conservative extension to the non-ground case of the SCore language adopted in the First ASP Competition, complies with the core language draft specified at LPNMR 2004 [1], and includes constructs which are nowadays common in current ASP parsers.

The ASP-RfC format comes in the form of a "Request for Comments" from the ASP community, and extends ASP-Core with function symbols and a limited number of predefined aggregate functions.

A limited number of problems specified in ASP-RfC will be selected for the System competition. We do expect the ASP-RfC format will foster discussion in the community and feed useful material to the foreseen forthcoming constitution of an ASP standard language working group.

3 ASP-Core and ASP-RfC Language Syntax

We define in the following programs written in ASP-Core: additions in the ASP-RfC format are explicitly described in framed boxes.

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 prescribed for participants is provided in Section 8.

An ASP-Core program P is constituted by a set of *rules*.

Rules. A rule r is in the form

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_k, o_1, \ldots, o_l, \textbf{not}\ n_1, ..., \textbf{not}\ n_m.$$

where $n, k, m, l \geq 0$, and at least one of n,k and m is greater than 0.

$a_1, \ldots, a_n$, $b_1, \ldots, b_k$, and $n_1, \ldots, n_m$ are *classical literals*, while $o_1, \ldots, o_l$ are *builtin atoms*.

$a_1 \vee \ldots \vee a_n$ constitutes the *head* of r, while $b_1, \ldots, b_k, \textbf{not } n_1, ..., \textbf{not } n_m$ is the *body* of r. As usual, whenever $k = m = 0$, we omit the "$\leftarrow$" sign. We call r a *fact* if $n = 1, k = m = 0$ or a *constraint* if $n = 0$.

Literals. A classical literal is either $-a$ (*negative classical literal*) or a (*positive classical literal*) for a a *predicate atom*. A *naf-literal* is either a *positive naf-literal* a or a *negative naf-literal* **not** a, for a a classical literal.

ASP-RfC

In ASP-RfC, the notion of naf-literal is redefined to include *aggregate literals*. An *ASP-RfC naf-literal* is either an ASP-Core naf-literal or an *aggregate literal*. An *aggregate literal*, is either **not** a or a, for a an *aggregate atom*.

■ ASP-RfC

Atoms. An atom is either

– a *predicate atom* in the form $p(X_1, \ldots, X_n)$ for p a *predicate name* and $X_1, \ldots, X_n$ *terms*, for n ($n \geq 0$) the fixed arity associated to p[1], or
– a *built-in atom* in any of the two forms $X \prec Y \diamond Z$ and $X \prec Y$, for X, Y and Z *terms*, "$\prec$" one of "$<$", "$\leq$", "$=$", "$\neq$", "$>$" and "$\geq$", and "$\diamond$" one of "$+$","$-$","$*$" and "$/$".

ASP-RfC

Aggregate atoms. An *aggregate atom* a is in the form $\#aggr\, S \prec v$ or $v \prec \#aggr\, S$, where:

– S is either a *set term* or a *multi-set* term; accordingly, we call a a *set aggregate* if S is a set term, and a *multiset aggregate* otherwise;
– "$\prec$" is one among "$<$", "$\leq$", "$=$", "$\neq$", "$>$" and "$\geq$";
– $\#aggr$ is an *aggregate function name*: allowed values for $\#aggr$ are $\#sum$, $\#count$, $\#max$ and $\#min$, and
– v is either a variable or an integer constants.

A *set term* is in the form $\{s\}$, while a *multiset term* is in the form $[s]$. In both cases, s is either a symbolic set or a ground set. A *symbolic set* is a pair $Vars : Conj$, where $Vars$ is a list of variables and $Conj$ is a conjunction of predicate and builtin atoms.

A *ground set* is a list of pairs of the form $\langle \overline{t} : Conj \rangle$, where $\overline{t}$ is a list of constants and $Conj$ is a ground (variable free) conjunction of predicate atoms.

Syntactic shortcuts. An aggregate atom in the form $l \prec_1 \#aggr\, S \prec_2 u$ is a syntactic shortcut for the conjunction $l \prec_1 \#aggr\, S, \#aggr\, S, \prec_2 u$, where $\prec_1$ and

[1] Atoms referring to a predicate q of arity 0, can be stated either in the form $q()$ or q. Negative literals and naf-literals cannot be built on top of built-in atoms.

$\prec_2$ are both either *leftward* operators or *rightward* operators. A *rightward* operator is either "$>$" or "$\geq$"; a *leftward* operator is either "$<$" or "$\leq$".

Non-normative syntactic shortcuts. If omitted, "$\prec_1$" and "$\prec_2$" are assumed to be both set to "$\leq$". If $\#aggr$ is omitted, it is assumed to be $\#sum$ for multiset aggregates, and $\#count$ for set aggregates.

A symbolic set in the form $Conj$ is a syntactic shortcut for $\{Vars : Conj\}$ in which $Vars$ is the list of all the variables appearing in $Conj$.

■ ASP-RfC

Terms, constants, variables. Terms are either *constants* or *variables*.

Constants can be either *symbolic constants* (strings starting with lowercase letter), *strings* (quoted sequences of characters) or integers; variables are denoted as strings starting with an uppercase letter (for the exact lexical matching of constants and variables, see Sec. 8). As a syntactic shortcut, the special variable _ is intended as replaced by a fresh variable name in the context of the rule at hand.

ASP-RfC

In ASP-RfC terms can be *functional*. A *functional term* is either a term or a structure in the form $f(t_1, \ldots, t_m)$ for f a *functor* (the *function name*), and $t_1, \ldots, t_m$ functional terms.

■ ASP-RfC

Queries. A program P can be coupled with a *ground query* in the form q?, where q is a ground atom.

ASP-RfC

In ASP-RfC queries can be built over non-ground atoms.

■ ASP-RfC

4 Semantics

As a reference, we herein give the full model-theoretic semantics of ASP-Core and ASP-RfC. As for non-ground programs, the semantics of both languages is mostly based on 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 [7] for ASP-Core, while it is based on [5] for what ASP-RfC is concerned. We understand that the semantics of aggregate atoms is currently subject of debate in the community: nonetheless, for the sake of the Competition, we recall that ASP-RfC programs are restricted to programs containing non-recursive aggregates (see Section 5), for which the general semantics herein presented is in substantial agreement with all other proposals for adding aggregates to ASP [8,15,9,13,4,6,14,3,12,10,11]. Other restrictions to the family of allowed programs apply: these are listed in Section 5.

Herbrand universe. Given a program P, the *Herbrand universe* of P, denoted by U_P, consists of all (ground) terms that can be built combining constants and functors appearing in P, and integers. The *Herbrand base* of P, denoted by B_P, is the set of all ground literals obtainable from the atoms of P by replacing variables with elements from U_P.

Substitutions and instances. A *consistent substitution* θ for a rule $r \in P$ is a mapping from the set of variables of r to the set U_P of ground terms, such that each built-in atom appearing in r is true with respect to the value assigned to variables by θ, according to Table 1. A *ground instance* of a rule r is obtained applying a consistent substitution to r and removing built-in atoms. Given a substitution θ and a object Obj (rule, set, etc.), we denote by $\theta(Obj)$ the object obtained by replacing each variable X in Obj by $\theta(X)$.

Built-in atoms consistency.

For a triple of values $x, y, z \in U_P$, "$\prec$" ranging over the operators "$<$", "$\leq$", "$=$", "$\neq$", "$>$" and "$\geq$" and $\diamond$ ranging over "$+$","$-$","$*$","$/$", we say that

- $x \prec y \diamond z$ is true if x, y and z are integers and $x \prec y \diamond z$ is satisfied in the canonical way over the domain of integers, for "/" being the division operation rounded to the lowest integer. $x \prec y \diamond z$ is false in all other cases;
- $x \prec y$ is true if x and y are of the same type (i.e. x and y are both integers, both constants or both quoted constants), and $x \prec y$ is true according to the respective domain; for strings and quoted strings we assume a total order given by the lexicographic precedence enforced by the character encoding of the input format (see Section 8 for details). $x \prec y$ is false in all other cases.

Table 1. Criteria for built-in atoms satisfaction.

ASP-RfC

Global and local variables. A *local* variable of a rule r is a variable appearing in a aggregate atom only; all other variables are *global* variables.

Instantiation. A consistent substitution from the set of global variables of a rule r (to U_P) is a *global substitution for* r; a substitution from the set of local variables of a symbolic set S (to U_P) is a *local substitution for* S.

Given a symbolic set without global variables $S = \{Vars : Conj\}$, the *instantiation of* S is the following ground set of pairs $inst(S)$:

$$\{\langle \gamma(Vars) : \gamma(Conj) \rangle \mid \gamma \text{ is a local substitution for } S\}$$

A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$.

■ ASP-RfC

Ground program. Given a program P the *instantiation (grounding)* $grnd(P)$ of P is defined as the set of all ground instances of its rules. Given a ground program P, an *interpretation* I for P is a subset of B_P. A *consistent* interpretation is such that $\{a, -a\} \not\subseteq I$ for each ground atom a. We deal in the following with consistent interpretations.

Satisfaction of literals. A positive naf-literal $l = a$ (resp., a naf-literal $l = \textbf{not}\ \ a$), for a a predicate atom, is true w.r.t. I if $a \in I$ (resp., $a \notin I$); it is false otherwise.

ASP-RfC

Aggregate functions. We associate to each aggregate function name $\#f$ a corresponding *aggregate function* f, mapping multisets to integer values. For a multiset S, let $\Pi(S)$ its corresponding set.

Let I be an interpretation. A standard ground conjunction is true (resp. false) w.r.t I if all its literals are true.

The valuation $I(S)$ of S w.r.t. I is the multiset of the first constant of the elements in S whose conjunction is true w.r.t. I. More precisely, let

$I(S)$ denote the multiset

$$[t_1 \mid \langle t_1, ..., t_n : Conj \rangle \in S \wedge\ Conj \text{ is true w.r.t. } I]$$

The *valuation* $V(I, S)$ of an instantiated set aggregate atom $A = v \prec \#f\{S\}$ is defined as $f(\Pi(I(S)))$. The valuation $V(I, S)$ of an instantiated multiset aggregate atom $v \prec \#f[S]$ is defined as $f(I(S))$.

An instantiated aggregate atom either in the form $A = v \prec \#f\{S\}$ or $A = v \prec \#f[S]$ is *true w.r.t. I* if:

– (i) $V(I, S) \neq \bot$, and,
– (ii) $v \prec V(I, S)$ holds; otherwise, A is false.

An instantiated aggregate literal **not** $A =$ **not** $v \prec \#f\{S\}$ or **not** $A =$ **not** $v \prec \#f[S]$ is *true w.r.t. I* if

– (i) $V(I, S) \neq \bot$, and,
– (ii) $v \prec V(I, S)$ does not hold; otherwise, **not** A is false.

Accordingly, for an instantiated aggregate literal **not** $A =$ **not** $l \prec_1 \#f\{S\} \prec_2 u$ or **not** $A =$ **not** $l \prec_1 \#f[S] \prec_2 u$

we have that **not** A is *true w.r.t. I* if

– (i) $V(I, S) \neq \bot$, and,
– (ii) either $l \prec_1 V(I, S)$ or $V(I, S) \prec_2 u$ do not hold; otherwise, **not** A is false.

Available aggregate functions. The available aggregate functions in ASP-RfC are defined as:

– $count(S) = |S|$;
– $sum(S) = \Sigma_{s \in S} s$. sum is defined only for multisets of integers;
– $max(S) = \max_{s \in S} s$.
– $min(S) = \min_{s \in S} s$.

Both *min* and *max* are defined over homogenous sets of integers, strings or quoted strings. For the latter two cases it is considered the total partial order enforced by the program character encoding (see Section 8 for details). If the multiset S is not in the domain of an aggregate function f, we conventionally set $f(S) = \bot$ (where $\bot$ is a fixed symbol not occurring in P).

■ ASP-RfC

Satisfaction of rules. Given a ground rule r, we say that r is satisfied w.r.t. I if some naf-literal appearing in the head of r is true w.r.t. I or some naf-literal appearing in the body of r is false w.r.t. I. Given a ground program P, we say that I is a *model* of P, iff all rules in $grnd(P)$ are satisfied w.r.t. I. A model M is *minimal* if there is no model N for P such that $N \subset M$.

Gelfond-Lifschitz reduct. The *Gelfond-Lifschitz reduct* [7] of P, w.r.t. an interpretation I, is the positive ground program P^I obtained from $grnd(P)$ by: (i) deleting all rules having a negative naf-literal false w.r.t. I; (ii) deleting all negative naf-literals from the remaining rules. $I \subseteq B_P$ is an *answer set* for a program P iff I is a minimal model for P^I. The set of all answer sets for P is denoted by $AS(P)$.

ASP-RfC

Generalized Gelfond-Lifschitz reduct [5]. The notion of Gelfond-Lifschitz reduct is replaced in ASP-RfC by the following. Note that for ASP-Core programs the two types of reduct coincide in the answer set they produce.

For a ASP-RfC ground program P and an interpretation I, let P^I denote the transformed program obtained from P by deleting all rules in which a body naf-literal is false w.r.t. I. I is an answer set of a program P if it is a minimal model of P^I.

■ ASP-RfC

5 Semantic Restrictions

A number of restrictions apply for ASP-Core and ASP-RfC encodings which will used for this Competition.

5.1 Safety

Programs written in ASP-Core are assumed to be *safe*. A program P is safe if all its rules are safe. A rule r is safe is all its variables are safe.

A variable X appearing in r is safe if either

- X appears in a positive naf-literal in the body of r, or
- X appears in a builtin atom $X = Y \diamond Z$ in the body of r, having X as its left-hand side, and Y and Z are safe.

ASP-RfC

For ASP-RfC programs, 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 builtin 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 $X = \#f\{Conj\}$ or $X = \#f[Conj]$ (or any equivalent one) and all other variables in the atom are safe.
2. if X is local to a symbolic set $\{\mathit{Vars} : \mathit{Conj}\}$ then it appears in an atom of Conj;

■ ASP-RfC

5.2 Programs with Function symbols and integers

Programs with function symbols and integers are in principle subject to no restriction. For the sake of Competition, and to facilitate implementors of ASP-RfC and ASP-Core it is prescribed that

- each selected problem encoding P must provably have finitely many finite answer sets for any of its benchmark instance I, that is $AS(P \cup I)$ 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 I and for any term t appearing in $AS(P \cup I)$, 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.

ASP-RfC

Non-recursiveness of aggregates. Recursive aggregates shall not appear in selected encodings for the competition. Formally, given a ASP-RfC program P we consider the labeled *dependency graph* $DG(P)$ between predicates of P, for which

- an arc $p \leftarrow q$ appears in $DG(P)$ if there is a rule $r \in P$ in which p appears in the head of r and q appears in a predicate atom in the body of r. If q appears in a set term in r, then we say that $p \leftarrow_a q$;
- 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 if there is no cycle (of arcs) in $DG(P)$ containing an edge in the form $p \leftarrow_a q$.

■ ASP-RfC

5.3 Restrictions on disjunction

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]. A converter from disjunctive head-cycle free ASP-Core programs to equivalent shifted versions is provided on the Competition web site.

6 Reasoning Tasks for the System competition

ASP-Core and ASP-RfC programs are subject to the following reasoning tasks.

Model generation (Search). Given a program P, to generate an answer set or to output UNSATISFIABLE.

Querying. If in the encoding of P it is trailed a ground query q? to output whether q is true in all the answer sets or not. l

ASP-RfC

In ASP-RfC, given a non ground query q?, it is requested to output all the ground instances of q which are true in all the answer sets.

■ ASP-RfC

Input and output formats for the above tasks are defined next.

7 EBNF Grammar for ASP-Core and ASP-RfC

The following is the EBNF grammar for ASP-Core:

```
<program>         ::= | <rules> | <rules> <query>
<rules>           ::= | <rules> | <rule>
<rule>            ::= <head> [CONS] DOT
                      | [<head>] CONS <body> DOT
<head>            ::= [<head> HEAD_SEPARATOR] <classic_literal>
<body>            ::= [<body> BODY_SEPARATOR] ( <naf_literal>
                      | <builtin_atom>)
(a) <naf_literal> ::= [NAF] <classic_literal>
<classic_literal> ::= [NEG] <atom>
<atom>            ::= <predicate_name> [PARAM_OPEN]
```

```
                               [<terms>] PARAM_CLOSE]
<terms>            ::= [<terms> TERM_SEPARATOR] <term>
<builtin_atom>     ::= <term> <binop> <term> [<arithop> <term>]
<binop>            ::= EQUAL | UNEQUAL | LESS | GREATER
                       | LESS_OR_EQ | GREATER_OR_EQ
<arithop>          ::= PLUS | MINUS | TIMES | DIV
(b) <term>         ::= <ground_term> | VARIABLE | ANON_VAR
<ground_term>      ::= SYMBOLIC_CONSTANT | STRING | NUMBER
<predicate_name>   ::= ID | STRING
(c) <query>        ::= <ground_atom> QUERY_MARK
<ground_atom>      ::= <predicate_name> [PARAM_OPEN
                            <ground_terms> PARAM_CLOSE]
                       | <predicate_name> PARAM_OPEN PARAM_CLOSE
<ground_terms>     ::= [<ground_terms> TERM_SEPARATOR] <ground_term>
```

For ASP-RfC EBNF Grammar, rules `(a)`, `(b)` and `(c)` are replaced with the following versions. The newly introduced non-terminal symbols are defined accordingly in the following:

```
(a) <naf_literal>    ::= [NAF] <classic_literal> | [NAF] <aggregate>
(b) <term>           ::= <ground_term> | VARIABLE | ANON_VAR
                         | <function_term>
(c) <query>          ::= <classic_literal> QUERY_MARK
<aggregate>          ::= <term> <binop> <aggregate_atom>
                         | <aggregate_atom> <binop> <term>
                         | <term> <leftop> <aggregate_atom>
                              <leftop> <term>
                         | <term> <rightop> <aggregate_atom>
                              <rightop> <term>
<leftop>             ::= LESS | LESS_OR_EQ
<rightop>            ::= GREATER | GREATER_OR_EQ
<aggregate_atom>     ::= <aggregate_function> CURLY_OPEN
                              <variables> COLON
                              <conjunction> CURLY_CLOSE
                         | CURLY_OPEN <variables> COLON
                              <conjunction> CURLY_CLOSE
                         | SQUARE_OPEN <variables> COLON
                              <conjunction> SQUARE_CLOSE
<variables>          ::= [<variables> TERM_SEPARATOR] VARIABLE
<conjunction>        ::= [<conjunction> BODY_SEPARATOR] <atom>
<aggregate_function> ::= AGGR_COUNT | AGGR_MAX | AGGR_MIN | AGGR_SUM
<function_term>      ::= <predicate_name> PARAM_OPEN
                              <terms> PARAM_CLOSE
```

8 Lexical matching table

<table>
<tr><th>Token Name</th><th>Symbolic Value or Symbolic Example</th><th>Lexical Value</th></tr>
<tr><td>ID</td><td>p, P, q1, ...</td><td>[A-Za-z][A-Za-z_0-9]*</td></tr>
<tr><td>SYMBOLIC_CONSTANT</td><td>a, b, $anna$, ...</td><td>[a-z][A-Za-z_0-9]*</td></tr>
<tr><td>VARIABLE</td><td>X,Y, $Name$:, ...</td><td>[_A-Z][A-Za-z_0-9]*</td></tr>
<tr><td>STRING</td><td>‘‘http://bit.ly/cw6lDS", ‘‘Full name" , ...</td><td>\"[^\"*]\"</td></tr>
<tr><td>ANON_VAR</td><td>_</td><td>"_"</td></tr>
<tr><td>NUMBER</td><td>1, 0, 100000, ...</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>HEAD_SEPARATOR</td><td>$\vee$</td><td>"|" | ";" | "v"</td></tr>
<tr><td>NEG</td><td>$-$</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>"/" | "div"</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></td><td>\%.*$</td></tr>
<tr><td>BLANK</td><td></td><td>[\t\n]+</td></tr>
</table>

Lexical values are given in Flex[2] syntax. The COMMENT and BLANK tokens can be freely interspersed amidst other tokens and have no syntactical and semantic meaning.

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

9 Instance Input and Output Formats

Benchmark problems specifications have to clearly indicate the vocabulary of input and output predicates. Each ASP system (or solver script) will read an input instance (from standard input) and produce an output (to standard output) according to the formats described in the following paragraphs.

Input Specification. A solver script (or ASP system) will read each input instance from the standard input. Each input instance (both in case of search and optimization problems) is expected to be both:

- made of sequences of *Facts* (atoms followed by the dot "." character) with only predicates of the input vocabulary, possibly separated by spaces and line breaks; and
- entirely saved in a text file (only one instance per file is allowed).

A fact is syntactically defined as
<fact> ::= <ground_atom> DOT
for <ground_atom> defined as in the ASP-Core grammar.

Output Specification. A solver script (or ASP system) is expected to write to the standard output an output respecting the following specification:

- *Search problem output:*
  - A single row of text containing a sequence of facts from atoms of the output vocabulary, representing a "witness", i.e, a portion of the answer set representing a solution for the instance problem (if the instance is satisfiable) or an answer to a query. The string "ANSWER SET FOUND" should appear on a separate line following the witness;
  - the string "NO ANSWER SET FOUND", in case the instance has no solution;
  - the string "UNKNOWN", if the solver decides to give up before time-out.
- *Optimization problem output:*
  - A series of witnesses of the search problem (i.e., a sequence of facts from atoms of the output vocabulary), one per line and separated by the return character, in case of satisfiable instances. The keyword "OPTIMUM FOUND" should appear on a new line following the last (and optimal) witness if and only if the last produced witness is optimal. Only the last (and hopefully best) witness will be considered.
  - the string "NO ANSWER SET FOUND", in case the instance has no solution;
  - the string "UNKNOWN", if the solver decides to give up before time-out.

Samples of input and output are available in the competition web site.

References

1. Core language for asp solver competitions. Minutes of the steering committee meeting at LPNMR04. Available at https://www.mat.unical.it/aspcomp2011/files/Corelang2004.pdf.
2. Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

3. 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/`.
4. 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.
5. 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.
6. 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.
7. Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
8. 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.
9. Mauricio Osorio and Bharat Jayaraman. Aggregation and Negation-As-Failure. *New Generation Computing*, 17(3):255–284, 1999.
10. 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.
11. 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.
12. 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.
13. Kenneth A. Ross and Yehoshua Sagiv. Monotonic Aggregation in Deductive Databases. *Journal of Computer and System Sciences*, 54(1):79–97, February 1997.
14. Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.
15. 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.