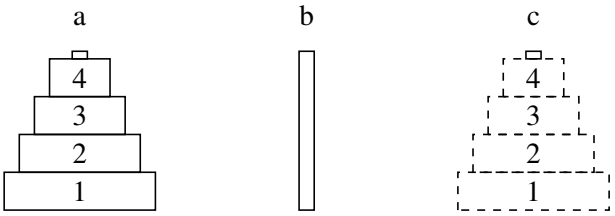

Figure 1: Towers of Hanoi: Initial and Goal Situation.

2 Quickstart

As an introductory example, we consider a simple Towers of Hanoi puzzle, consisting of three pegs and four disks of different size. As shown in Figure 1, the goal is to move all disks from the left peg to the right one, where only the topmost disk of a peg can be moved at a time. Furthermore, a disk cannot be moved to a peg already containing some disk that is smaller. Although there is an efficient algorithm to solve our simple Towers of Hanoi puzzle, we do not exploit it and below merely specify conditions for sequences of moves being solutions.

In ASP, it is custom to provide a *uniform* problem definition [68, 70, 77]. Following this methodology, we separately specify an instance and an encoding (applying to every instance) of the following problem: given an initial placement of the disks, a goal situation, and a number n, decide whether there is a sequence of n moves that achieves the goal. We will see that this problem can be elegantly described in ASP and solved by domain-independent tools like *gringo* and *clasp*. Such a declarative solution is now exemplified.

2.1 Problem Instance

We describe the pegs and disks of a Towers of Hanoi puzzle via facts over the predicates `peg`/1 and `disk`/1 (the number denotes the arity of the predicate). Disks are numbered by consecutive integers starting at `1`, where a disk with a smaller number is considered to be bigger than a disk with a greater number. The names of the pegs can be arbitrary; in our case, we use `a`, `b`, and `c`. Furthermore, the predicates `init_on`/2 and `goal_on`/2 describe the initial and the goal situation, respectively. Their arguments, the number of a disk and the name of a peg, determine the location of a disk in the respective situation. Finally, the predicate `moves`/1 specifies the number of moves in which the goal must be achieved. When allowing `15` moves, the Towers of Hanoi puzzle shown in Figure 1 is described by the following facts:

You can save this instance locally by clicking its file name: `toh_ins.lp`. Depending on your viewer, a right or double-click should do.

```
1 peg(a;b;c).
2 disk(1..4).
3 init_on(1..4,a).
4 goal_on(1..4,c).
5 moves(15).
```

Note that the ';' in the first line is syntactic sugar (detailed in Section 3.1.10) that expands the statement into three facts: `peg(a).`, `peg(b).`, and `peg(c).` Similarly, '`1..4`' used in Line 2–4 refers to an interval (detailed in Section 3.1.9). Here, it abbreviates distinct facts over four values: `1`, `2`, `3`, and `4`. In summary, the facts in Line 1–5 describe the Towers of Hanoi puzzle in Figure 1 along with the requirement that the goal ought to be achieved within `15` moves.

2.2 Problem Encoding

We now proceed by encoding Towers of Hanoi via schematic rules, i.e., rules containing variables (whose names start with uppercase letters) that are independent of a particular instance. Typically, an encoding can be logically partitioned into a *Generate*, a *Define*, and a *Test* part [63]. An additional *Display* part allows for restricting the output to a distinguished set of atoms, and thus, for suppressing auxiliary predicates. We follow this methodology and mark the respective parts via comment lines beginning with '`%`' in the following encoding:

You can also save the encoding by clicking this file name: toh_enc.lp.
We below explain how to run the saved files in order to solve our Towers of Hanoi puzzle.

```
1  % Generate
2  { move(D,P,T) : disk(D), peg(P) } = 1 :- moves(M),
     T = 1..M.
3  % Define
4  move(D,T)    :- move(D,_,T).
5  on(D,P,0)    :- init_on(D,P).
6  on(D,P,T)    :- move(D,P,T).
7  on(D,P,T+1)  :- on(D,P,T), not move(D,T+1),
     not moves(T).
8  blocked(D-1,P,T+1) :- on(D,P,T), not moves(T).
9  blocked(D-1,P,T)   :- blocked(D,P,T), disk(D).
10 % Test
11 :- move(D,P,T), blocked(D-1,P,T).
12 :- move(D,T), on(D,P,T-1), blocked(D,P,T).
13 :- goal_on(D,P), not on(D,P,M), moves(M).
14 :- { on(D,P,T) } != 1, disk(D), moves(M), T = 1..M.
15 % Display
16 #show move/3.
```

Note that the variables `D`, `P`, `T`, and `M` are used to refer to disks, pegs, the number of a move, and the length of the sequence of moves, respectively.

The Generate part, describing solution candidates, consists of the rule in Line 2. It expresses that, at each point `T` in time (other than `0`), exactly one move of a disk `D` to some peg `P` must be executed. The head of the rule (left of '`:-`') is a so-called cardinality constraint (see Section 3.1.12). It consists of a set of literals, expanded using the conditions behind the colon (detailed in Section 3.1.11), along with the guard '`= 1`'. The cardinality constraint is satisfied if the number of true elements is equal to one, as specified by the guard. Since the cardinality constraint occurs

as the head of a rule, it allows for deriving ("guessing") atoms over the predicate `move`/3 to be true. In the body (right of '`:-`'), we define (detailed in Section 3.1.8), `T = 1..M`, to refer to each time point `T` from `1` to the maximum time point `M`. We have thus characterized all sequences of `M` moves as solution candidates for Towers of Hanoi. Up to now, we have not yet imposed any further conditions, e.g., that a bigger disk must not be moved on top of a smaller one.

The Define part in Line 4–9 contains rules defining auxiliary predicates, i.e., predicates that provide properties of a solution candidate at hand. (Such properties will be investigated in the Test part described below.) The rule in Line 4 simply projects moves to disks and time points. The resulting predicate `move`/2 can be used whenever the target peg is insignificant, so that one of its atoms actually subsumes three possible cases. Furthermore, the predicate `on`/3 captures the state of a Towers of Hanoi puzzle at each time point. To this end, the rule in Line 5 identifies the locations of disks at time point `0` with the initial state (given in an instance). State transitions are modeled by the rules in Line 6 and 7. While the former specifies the direct effect of a move at time point `T`, i.e., the affected disk `D` is relocated to the target peg `P`, the latter describes inertia: the location of a disk `D` carries forward from time point `T` to `T+1` if `D` is not moved at `T+1`. Observe the usage of `not moves(T)` in Line 7, which prevents deriving disk locations beyond the maximum time point. Finally, we define the auxiliary predicate `blocked`/3 to indicate that a smaller disk, with a number greater than `D-1`, is located on a peg `P`. The rule in Line 8 derives this condition for time point `T+1` from `on(D,P,T)`, provided that `T` is not the maximum time point. The rule in Line 9 further propagates the status of being blocked to all bigger disks on the same peg. Note that we also mark `D-1 = 0`, not referring to any disk, as blocked, which is convenient for eliminating redundant moves in the Test part described next.

The Test part consists of the integrity constraints in Line 11–14, rules that eliminate unintended solution candidates. The first integrity constraint in Line 11 asserts that a disk `D` must not be moved to a peg `P` if `D-1` is blocked at time point `T`. This excludes moves putting a bigger disk on top of a smaller one and, in view of the definition of `blocked`/3, also disallows that a disk is put back to its previous location. Similarly, the integrity constraint in Line 12 expresses that a disk `D` cannot be moved at time point `T` if it is blocked by some smaller disk on the same peg `P`. Note that we use `move(D,T)` here because the target of an illegal move does not matter in this context. The fact that the goal situation, given in an instance, must be achieved at maximum time point `M` is represented by the integrity constraint in Line 13. The final integrity constraint in Line 14 asserts that, for every disk `D` and time point `T`, there is exactly one peg `P` such that `on(D,P,T)` holds. Although this condition is implied by the definition of `on`/3 in Line 6 and 7 with respect to the moves in a solution, making such knowledge explicit via an integrity constraint turns out to improve the solving efficiency.

Finally, the meta-statement (detailed in Section 3.1.15) of the Display part in Line 16 indicates that only atoms over the predicate `move`/3 ought to be printed, thus suppressing the predicates used to describe an instance as well as the auxiliary

predicates move/2, on/3, and blocked/3. This is for more convenient reading of a solution, given that it is fully determined by atoms over move/3.

2.3 Problem Solution

We are now ready to solve our Towers of Hanoi puzzle. To compute an answer set representing a solution, invoke one of the following commands:

clingo or *gringo* and *clasp* ought to be located in some directory in the system path. Also, toh_ins.lp and toh_enc.lp (click file name to save) should reside in the working directory.

```
clingo toh_ins.lp toh_enc.lp
gringo toh_ins.lp toh_enc.lp | clasp
```

The output of the solver, *clingo* in this case, should look somehow like this:

```
clingo version 4.4.0
Reading from toh_ins.lp ...
Solving...
Answer: 1
move(4,b,1) move(3,c,2) move(4,c,3) move(2,b,4) \
move(4,a,5) move(3,b,6) move(4,b,7) move(1,c,8) \
move(4,c,9) move(3,a,10) move(4,a,11) move(2,c,12) \
move(4,b,13) move(3,c,14) move(4,c,15)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 0.017s (Solving: 0.01s 1st Model: 0.01s \
                                      Unsat: 0.00s)
CPU Time     : 0.010s
```

The first line shows the *clingo* version. The following two lines indicate *clingo*'s state. *clingo* should print immediately that it is reading. Once this is done, it prints Solving... to the command line. The Towers of Hanoi instance above is so easy to solve that you will not recognize the delay, but for larger problems it can be noticeable. The line starting with Answer: indicates that the (output) atoms of an answer set follow in the next line. In this example, it contains the true instances of move/3 in the order of time points, so that we can easily read off the following solution from them: first move disk 4 to peg b, second move disk 3 to peg c, third move disk 4 to peg c, and so on. We use '\' to indicate that all atoms over move/3 actually belong to a single line. Note that the order in which atoms are printed does not bear any meaning (and the same applies to the order in which answer sets are found). Below this solution, we find the satisfiability status of the problem, which is reported as SATISFIABLE by the solver.[2] The '1+' in the line starting with Models tells us that one answer set has been found.[3] Calls to the solver

The given instance has just one solution. In fact, the '+' from '1+' disappears if you compute all solutions by invoking:

```
clingo toh_ins.lp \
toh_enc.lp 0
```

or alternatively:

```
gringo toh_ins.lp \
toh_enc.lp | clasp 0
```

[2] Other possibilities include UNSATISFIABLE and UNKNOWN, the latter in case of an abort.

[3] The '+' indicates that the solver has not exhaustively explored the search space (but stopped upon finding an answer set), so that further answer sets may exist.

are of interest in multi-shot solving (see Section 4). The final lines report statistics including total run-time (wall-clock `Time` as well as `CPU Time`) and the amount of time spent on search (`Solving`), along with the fractions taken to find the first solution (`1st Model`) and to prove unsatisfiability[4] (`Unsat`). More information about available options, e.g., to obtain extended statistics output, can be found in Section 7.

2.4 Summary

To conclude our quickstart, let us summarize some "take-home messages". For solving our Towers of Hanoi puzzle, we first provided facts representing an instance. Although we did not discuss the choice of predicates, an appropriate instance representation is already part of the modeling in ASP and not always as straightforward as here. Second, we provided an encoding of the problem applying to any instance. The encoding consisted of parts generating solution candidates, deriving their essential properties, testing that no solution condition is violated, and finally projecting the output to characteristic atoms. With the encoding at hand, we could use off-the-shelf ASP tools to solve our instance, and the encoding can be reused for any further instance that may arise in the future.

[4] No unsatisfiability proof is done here, hence, this time is zero. But for example, when enumerating all models, this is the time spent between finding the last model and termination.

3 Input Languages

This section provides an overview of the input languages of grounder *gringo*, combined grounder and solver *clingo*, and solver *clasp*. The joint input language of *gringo* and *clingo* is detailed in Section 3.1. Finally, Section 3.2 is dedicated to the inputs handled by *clasp*.

3.1 Input Language of *gringo* and *clingo*

The tool *gringo* [47] is a grounder capable of transforming user-defined logic programs (usually containing variables) into equivalent ground (that is, variable-free) programs. The output of *gringo* can be piped into solver *clasp* [39, 44], which then computes answer sets. System *clingo* internally couples *gringo* and *clasp*, thus, it takes care of both grounding and solving. In contrast to *gringo* outputting ground programs, *clingo* returns answer sets.

Usually, logic programs are specified in one or more (text) files whose names are provided as arguments in an invocation of either *gringo* or *clingo*. In what follows, we describe the constructs belonging to the input language of *gringo* and *clingo*.

3.1.1 Terms

Every (non-propositional) logic program includes *terms*, mainly to specify the arguments of atoms (see below). The grammar for *gringo*'s (and *clingo*'s) terms is shown in Figure 2.

The basic building blocks are simple terms: *integers*, *constants*, *strings*, and *variables* as well as the tokens '_', `#sup`, and `#inf`. An integer is represented by means of an arithmetic expression, further explained in Section 3.1.7. Constants and variables are distinguished by their first letters, which are *lowercase* and *uppercase*, respectively, where leading occurrences of '_' are allowed (may be useful to circumvent name clashes). Furthermore, a string is an arbitrary sequence of characters enclosed in double quotes (`"·"`), where any occurrences of '\', newline, and double quote must be escaped via '`\\`', '`\n`', or '`\"`', respectively.

While a constant or string represents itself, a variable is a placeholder for *all* variable-free terms in the language of a logic program.[5] Unlike a variable name whose recurrences within a rule refer to the same variable, the token '_' (not followed by any letter) stands for an *anonymous variable* that does not recur anywhere. (One can view this as if a new variable name is invented on each occurrence of '_'.) In addition, there are the special constants `#sup` and `#inf` representing the greatest and smallest element among all variable-free terms[6] , respectively; we illustrate their use in Section 3.1.12.

[5]The set of all terms constructible from the available constants and function symbols is called *Herbrand universe*.

[6]Their is a total order defined on variable-free terms; for details see Section 3.1.8.

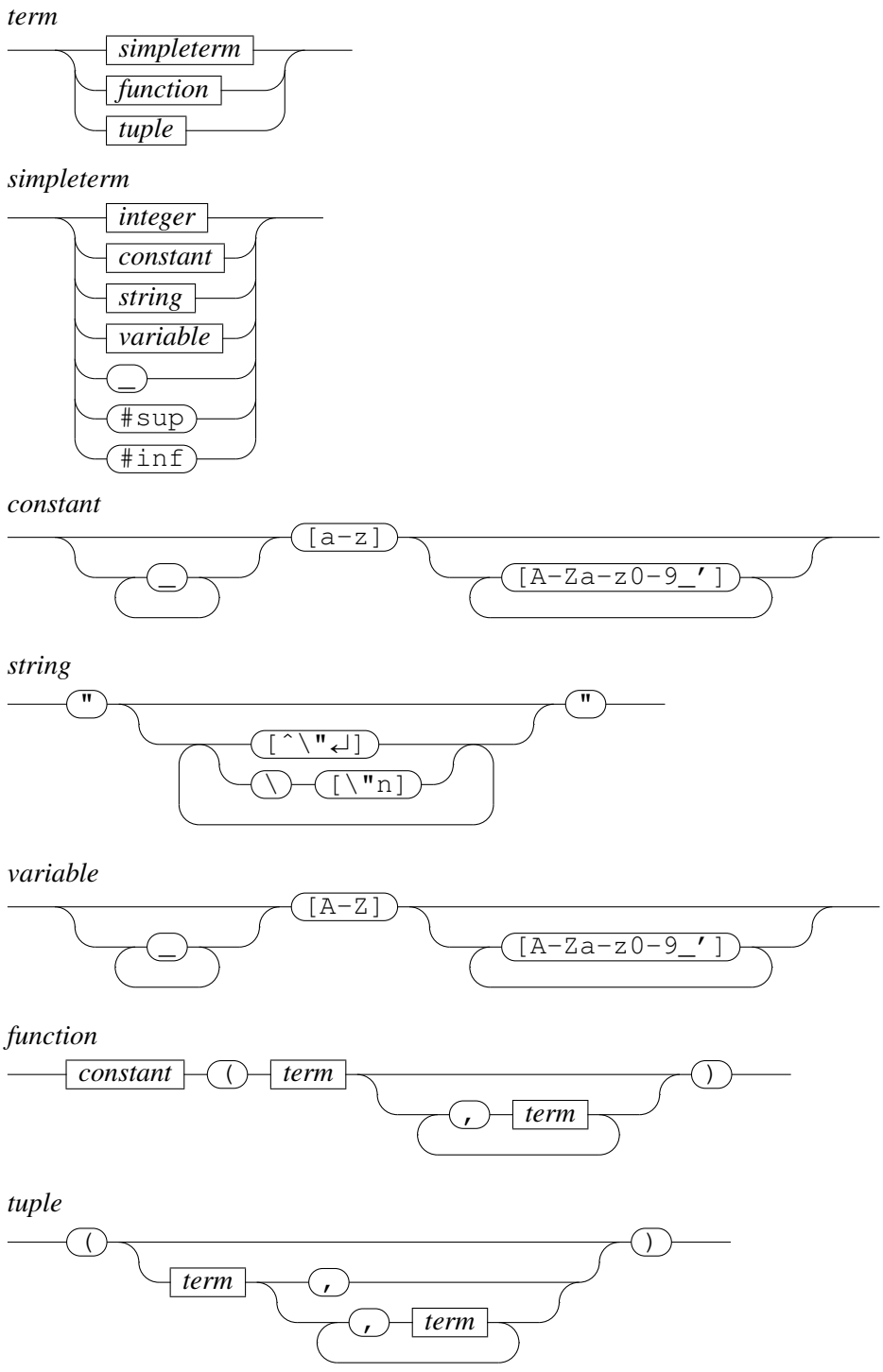

Figure 2: Grammar for Terms.

Next, (uninterpreted) *functions* are complex terms composed of a name (like a constant) and one or more terms as arguments. For instance, `at(peter,time(12),X)` is a function with three arguments: constant `peter`, another function `time(12)` with an integer argument, and variable `X`. Finally, there are *tuples*, which are similar to *functions* but without a name. Examples for tuples are: the empty tuple `()` and the tuple `(at,peter,time(12),X)` with four elements. Tuples may optionally end in a comma '`,`' for distinguishing one-elementary tuples. That is, $(t,)$ is a one-elementary tuple, while a term of form (t) is equivalent to t. For instance, `(42,)` is a one-elementary tuple, whereas `(42)` is not, and the above quadruple is equivalent to `(at,peter,time(12),X,)`.

3.1.2 Normal Programs and Integrity Constraints

Rules of the following forms are admitted in a *normal logic program* (with integrity constraints):

$$\begin{aligned} \textbf{Fact:} \quad & A_0\,\texttt{.} \\ \textbf{Rule:} \quad & A_0\ \texttt{:-}\ L_1\texttt{,}\ldots\texttt{,}L_n\,\texttt{.} \\ \textbf{Integrity Constraint:} \quad & \texttt{:-}\ L_1\texttt{,}\ldots\texttt{,}L_n\,\texttt{.} \end{aligned}$$

The *head* A_0 of a rule or a fact is an *atom* of the same syntactic form as a constant or function. In the *body* of a rule or an integrity constraint, every L_j for $1 \leqslant j \leqslant n$ is a *literal* of the form A or `not` A, where A is an atom and the connective `not` denotes default negation. We say that a literal L is *positive* if it is an atom, and *negative* otherwise. While the head atom A_0 of a fact must unconditionally be true, the intuitive reading of a rule corresponds to an implication: if all positive literals in the rule's body are true and all negative literals are satisfied, then A_0 must be true. On the other hand, an integrity constraint is a rule that filters solution candidates, meaning that the literals in its body must not jointly be satisfied.

A set of (propositional) atoms is called a *model* of a logic program if it satisfies all rules, facts, and integrity constraints. Atoms are considered true if and only if they are in the model. In ASP, a model is called an *answer set* if every atom in the model has an (acyclic) derivation from the program. See [51, 48, 64] for formal definitions of answer sets of logic programs.

To get the idea, let us consider some small examples.

Example 3.1. Consider the following logic program:

```
a :- b.
b :- a.
```

When `a` and `b` are false, the bodies of both rules are false as well, so that the rules are satisfied. Furthermore, there is no (true) atom to be derived, which shows that the empty set is an answer set. On the other hand, if `a` is true but `b` is not, then the first rule is unsatisfied because the body holds but the head does not. Similarly, the second rule is unsatisfied if `b` is true and `a` is not. Hence, an answer set cannot

contain only one of the atoms `a` and `b`. It remains to investigate the set including both `a` and `b`. Although both rules are satisfied, `a` and `b` cannot be derived acyclically: `a` relies on `b`, and vice versa. That is, the set including both `a` and `b` is not an answer set. Hence, the empty set is the only answer set of the logic program. We say that there is a *positive cycle* through `a` and `b` subject to minimization. ■

Consider the following logic program:

```
a :- not b.
b :- not a.
```

Here, the empty set is not a model because both rules are unsatisfied. However, the sets containing only `a` or only `b` are models. To see that each of them is an answer set, note that `a` is derived by the rule `a :- not b.` if `b` is false; similarly, `b` is derived by `b :- not a.` if `a` is false. Note that the set including both `a` and `b` is not an answer set because neither atom can be derived if both are assumed to be true: the bodies of the rules `a :- not b.` and `b :- not a.` are false. Hence, we have that either `a` or `b` belongs to an answer set of the logic program.

To illustrate the use of facts and integrity constraints, let us augment the previous logic program:

```
a :- not b.
b :- not a.
c.
:- c, not b.
```

Since `c.` is a fact, atom `c` must unconditionally be true, i.e., it belongs to every model. In view of this, the integrity constraint `:- c, not b.` tells us that `b` must be true as well in order to prevent its body from being satisfied. However, this kind of reasoning does not provide us with a derivation of `b`. Rather, we still need to make sure that the body of the rule `b :- not a.` is satisfied, so that atom `a` must be false. Hence, the set containing `b` and `c` is the only answer set of our logic program.

In the above examples, we used propositional logic programs to exemplify the idea of an answer set: a model of a logic program such that all its true atoms are (acyclically) derivable. In practice, logic programs are typically non-propositional, i.e., they include schematic rules with variables. The next example illustrates this.

Example 3.2. Consider a child from the south pole watching cartoons, where it sees a yellow bird that is not a penguin. The child knows that penguins can definitely not fly (due to small wingspread), but it is unsure about whether the yellow bird flies. This knowledge is generalized by the following schematic rules:

```
1      fly(X) :- bird(X), not neg_fly(X).
2 neg_fly(X) :- bird(X), not     fly(X).
3 neg_fly(X) :- penguin(X).
```

The first rule expresses that it is generally possible that a bird flies, unless the contrary, subject to the second rule, is the case. The definite knowledge that penguins cannot fly is specified by the third rule.

Later on, the child learns that the yellow bird is a chicken called "tweety", while its favorite penguin is called "tux". The knowledge about these two individuals is represented by the following facts:

```
4 bird(tweety).   chicken(tweety).
5 bird(tux).      penguin(tux).
```

When we instantiate the variable `X` in the three schematic rules with `tweety` and `tux`, we obtain the following ground rules:

```
fly(tweety)     :- bird(tweety), not neg_fly(tweety).
fly(tux)        :- bird(tux),    not neg_fly(tux).
neg_fly(tweety) :- bird(tweety), not     fly(tweety).
neg_fly(tux)    :- bird(tux),    not     fly(tux).
neg_fly(tweety) :- penguin(tweety).
neg_fly(tux)    :- penguin(tux).
```

Further taking into account that `tweety` and `tux` are known to be birds, that `tux` is a penguin, while `tweety` is not, and that penguins can definitely not fly, we can simplify the previous ground rules to obtain the following ones:

```
    fly(tweety) :- not neg_fly(tweety).
neg_fly(tweety) :- not     fly(tweety).
neg_fly(tux).
```

Now it becomes apparent that `tweety` may fly or not, while `tux` surely does not fly. Thus, there are two answer sets for the three schematic rules above, instantiated with `tweety` and `tux`. ■

The reader can reproduce these ground rules by invoking:

```
clingo --text \
bird.lp fly.lp
```

or alternatively:

```
gringo --text \
bird.lp fly.lp
```

To compute both answer sets, invoke:

```
clingo bird.lp \
fly.lp 0
```

or alternatively:

```
gringo bird.lp \
fly.lp | clasp 0
```

The above example illustrated how variables are used to represent all instances of rules with respect to the language of a logic program. In fact, grounder *gringo* (or the grounding component of *clingo*) takes care of instantiating variables such that an equivalent propositional logic program is obtained. To this end, rules are required to be *safe*, i.e., all variables in a rule must occur in some positive literal (a literal not preceded by `not`) in the body of the rule. For instance, the first two schematic rules in Example 3.2 are safe because they include `bird(X)` in their positive bodies. This tells *gringo* (or *clingo*) that the values to be substituted for `X` are limited to birds.

Up to now, we have introduced terms, facts, (normal) rules, and integrity constraints. Before we proceed to describe handy extensions to this simple core language, keep in mind that the role of a rule (or fact) is that an atom in the head can be derived to be true if the body is satisfied. Unlike this, an integrity constraint implements a test, but it cannot be used to derive any atom. This universal meaning still applies when more sophisticated language constructs, as described in the following, are used.

3.1.3 Classical Negation

The connective `not` expresses default negation, i.e., a literal `not` A is assumed to hold unless atom A is derived to be true. In contrast, the classical (or strong)

negation of an atom [52] holds only if it can be derived. Classical negation, indicated by symbol '-', is permitted in front of atoms. That is, if A is an atom, then $-A$ is an atom representing the complement of A. The semantic relationship between A and $-A$ is simply that they must not jointly hold. Hence, classical negation can be understood as a syntactic feature allowing us to impose an integrity constraint `:- A, -A.` without explicitly writing it in a logic program. Depending on the logic program at hand, it may be possible that neither A nor $-A$ is contained in an answer set, thus representing a state where the truth and the falsity of A are both unknown.

Example 3.3. Using classical negation, we can rewrite the schematic rules in Example 3.2 in the following way:

```
1   fly(X) :- bird(X), not -fly(X).
2  -fly(X) :- bird(X), not  fly(X).
3  -fly(X) :- penguin(X).
```

Given the individuals `tweety` and `tux`, classical negation is reflected by the following (implicit) integrity constraints:

```
4  :- fly(tweety), -fly(tweety).
5  :- fly(tux),    -fly(tux).
```

By invoking:
```
clingo --text \
bird.lp flycn.lp
```
or alternatively:
```
gringo --text \
bird.lp flycn.lp
```
the reader can observe that the integrity constraint in Line 4 is indeed part of the grounding. The second one in Line 5 is not printed; it becomes obsolete by a static analysis exhibiting that `tux` does surely not fly.

There are still two answer sets, containing `-fly(tux)` and either `fly(tweety)` or `-fly(tweety)`.

Now assume that we add the following fact to the program:

```
fly(tux).
```

Then, `fly(tux)` must unconditionally be true, and `-fly(tux)` is still derived by an instance of the third schematic rule. Since every answer set candidate containing both `fly(tux)` and `-fly(tux)` triggers the (implicit) integrity constraint in Line 5, there is no longer any answer set. ■

3.1.4 Disjunction

Disjunctive logic programs permit connective '`;`' between atoms in rule heads.[7]

Fact: $A_0\texttt{;}\ldots\texttt{;}A_m\texttt{.}$
Rule: $A_0\texttt{;}\ldots\texttt{;}A_m\ \texttt{:-}\ L_1\texttt{,}\ldots\texttt{,}L_n\texttt{.}$

A disjunctive head holds if at least one of its atoms is true. Answer sets of a disjunctive logic program satisfy a minimality criterion that we do not detail here (see [22, 36] for an implementation methodology in disjunctive ASP). We only mention that the simple disjunctive program `a;b.` has two answer sets, one containing `a` and another one containing `b`, while both atoms do not jointly belong to an answer

[7] Note that disjunction in rule heads was not supported by *clasp* and *clingo* versions before series 3 and 4, respectively.

set. After adding the rules of Example 3.1, a single answer set containing both `a` and `b` is obtained. This illustrates that disjunction in ASP is neither strictly exclusive or inclusive but subject to minimization.

In general, the use of disjunction may increase computational complexity [21]. We thus suggest to use "choice constructs" (detailed in Section 3.1.12) instead of disjunction, unless the latter is required for complexity reasons.

3.1.5 Double Negation and Head Literals

The input language of *gringo* also supports double default negated literals, written `not not` A. They are satisfied whenever their positive counterparts are. But like negative literals of form `not` A, double negated ones are also preceded by `not` and do thus not require an (acyclic) derivation from the program; it is sufficient that they are true in the model at hand.

Consider the logic program:

```
a :- not not b.
b :- not not a.
```

This program has an empty answer set, like the program in Example 3.1, as well as the additional answer set containing both `a` and `b`. This is because neither '`not not a`' nor '`not not b`' requires an acyclic derivation from the program. Note that, in contrast to Example 3.1, the above program does not induce mutual positive dependencies between `a` and `b`. Given this, `a` and `b` can thus be both true or false, just like in classical logic.

Also, negative literals are admitted in the head of rules. When disregarding disjunction, this offers just another way to write integrity constraints, putting the emphasis on the head literal. In fact, the rule `not` A_0 `:-` L_1`,` $\ldots$`,` L_n`.` is equivalent to `:-` L_1`,` $\ldots$`,` L_n`,` `not not` A_0`.`, and with double negation in the head, rule `not not` A_0 `:-` L_1`,` $\ldots$`,` L_n`.` is equivalent to `:-` L_1`,` $\ldots$`,` L_n`,` `not` A_0`.`

Example 3.4. Consider the logic program:

```
1      fly(X) :- bird(X), not not fly(X).
2  not fly(X) :- penguin(X).
```

To compute both answer sets, invoke:

```
clingo bird.lp \
flynn.lp 0
```

or alternatively:

```
gringo bird.lp \
flynn.lp | clasp 0
```

The possibility that a bird flies is expressed with a double negation in the first line. Solutions with flying penguins are filtered out in the second line. Like in Example 3.2 there are two answer sets, but without an explicit atom to indicate that a bird does not fly. Hence, the answer set where tweety does not fly contains no atoms over predicate `fly`/1. ■

Remark 3.1. Note that negative head literals are also supported in disjunctions. For more information see [65]. ■

3.1.6 Boolean Constants

Sometimes it is useful to have literals possessing a constant truth value. Literals over the two *Boolean constants* `#true` and `#false`, which are always true or false, respectively, have a constant truth value.

The unique answer set of the program, can be inspected by invoking:
```
clingo bool.lp 0
```
or alternatively:
```
gringo bool.lp \
| clasp 0
```
Note that this program simply produces an empty grounding:
```
clingo --text \
bool.lp
```
or alternatively:
```
gringo --text \
bool.lp
```

Example 3.5. Consider the following program:

```
1 #true.
2 not #false.
3 not not #true.
4 :- #false.
5 :- not #true.
6 :- not not #false.
```

The first rule uses `#true` in the head. Because this rule is a fact, it is trivially satisfied. Similarly, the rules in Line 2 and 3 have satisfied heads. The bodies of the last three integrity constraints are false. Hence, the constraints do not cause a conflict. Note that neither of the rules above derives any atom. Thus, we obtain the empty answer set for the program. ■

See Example 3.14 below for an application of interest.

3.1.7 Built-in Arithmetic Functions

Besides integers (constant arithmetic functions), written as sequences of the digits `0...9` possibly preceded by '−', *gringo* and *clingo* support a variety of arithmetic functions that are evaluated during grounding. The following symbols are used for these functions: `+` (addition), `-` (subtraction, unary minus), `*` (multiplication), `/` (integer division), `\` (modulo), `**` (exponentiation), `|·|` (absolute value), `&` (bitwise AND), `?` (bitwise OR), `^` (bitwise exclusive OR), and `~` (bitwise complement).

The unique answer set of the program, obtained after evaluating all arithmetic functions, can be inspected by invoking:
```
clingo --text \
arithf.lp
```
or alternatively:
```
gringo --text \
arithf.lp
```

Example 3.6. The usage of arithmetic functions is illustrated by the program:

```
1  left     (7).
2  right         (2).
3  plus    ( L +  R ) :- left(L), right(R).
4  minus   ( L -  R ) :- left(L), right(R).
5  uminus  (   -  R ) :-          right(R).
6  times   ( L *  R ) :- left(L), right(R).
7  divide  ( L /  R ) :- left(L), right(R).
8  modulo  ( L \  R ) :- left(L), right(R).
9  absolute(|  -  R|) :-          right(R).
10 power   ( L ** R ) :- left(L), right(R).
11 bitand  ( L &  R ) :- left(L), right(R).
12 bitor   ( L ?  R ) :- left(L), right(R).
13 bitxor  ( L ^  R ) :- left(L), right(R).
14 bitneg  (   ~  R ) :-          right(R).
```

Note that the variables `L` and `R` are instantiated to `7` and `2`, respectively, before arithmetic evaluations. Consecutive and non-separative (e.g., before '`(`') spaces can optionally be dropped. The four bitwise functions apply to signed integers, using two's complement arithmetic. ■

Remark 3.2. An occurrence of a variable in the scope of an arithmetic function only counts as positive in the sense of safety (cf. Page 21) for simple arithmetic terms. Such simple arithmetic terms are terms with exactly one variable occurrence composed of the arithmetic functions '+', '−', '⋆', and integers. Moreover, if multiplication is used, then the constant part must not evaluate to 0 for the variable occurrence to be considered positive. E.g., the rule `q(X) :- p(2*(X+1)).` is considered safe, but the rule `q(X) :- p(X+X).` is not. ■

3.1.8 Built-in Comparison Predicates

Grounder *gringo* (and *clingo*) feature a total order among variable-free terms (without arithmetic functions). The built-in predicates to compare terms are = (equal), `!=` (not equal), < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal). *Comparison literals* over the above *comparison predicates* are used like other literals (cf. Section 3.1.2) but are evaluated during grounding.

Example 3.7. The application of comparison literals to integers is illustrated by the following program:

```
1 num(1).  num(2).
2 eq (X,Y) :- X   =   Y, num(X), num(Y).
3 neq(X,Y) :- X   !=  Y, num(X), num(Y).
4 lt (X,Y) :- X   <   Y, num(X), num(Y).
5 leq(X,Y) :- X   <=  Y, num(X), num(Y).
6 gt (X,Y) :- X   >   Y, num(X), num(Y).
7 geq(X,Y) :- X   >=  Y, num(X), num(Y).
8 all(X,Y) :- X-1 < X+Y, num(X), num(Y).
9 non(X,Y) :- X/X > Y*Y, num(X), num(Y).
```

The simplified ground program obtained by evaluating built-ins can be inspected by invoking:

```
clingo --text \
arithc.lp
```

or alternatively:

```
gringo --text \
arithc.lp
```

The last two lines hint at the fact that arithmetic functions are evaluated before comparison literals, so that the latter actually compare the results of arithmetic evaluations. ■

Example 3.8. Comparison literals can also be applied to constants and functions, as illustrated by the following program:

```
1 sym(1).  sym(a).  sym(f(a)).
2 eq (X,Y) :- X =  Y, sym(X), sym(Y).
3 neq(X,Y) :- X != Y, sym(X), sym(Y).
4 lt (X,Y) :- X <  Y, sym(X), sym(Y).
5 leq(X,Y) :- X <= Y, sym(X), sym(Y).
6 gt (X,Y) :- X >  Y, sym(X), sym(Y).
7 geq(X,Y) :- X >= Y, sym(X), sym(Y).
```

As above, by invoking:

```
clingo --text \
symbc.lp
```

or alternatively:

```
gringo --text \
symbc.lp
```

one can inspect the simplified ground program obtained by evaluating built-ins.

Integers are compared in the usual way, constants are ordered lexicographically, and functions both structurally and lexicographically. Furthermore, all integers are smaller than constants, which in turn are smaller than functions. ■

The built-in comparison predicate '=' has another interesting use case. Apart from just testing whether a relation between two terms holds, it can be used to define shorthands (via unification) for terms.

The simplified ground program can be inspected by invoking:
```
clingo --text \
define.lp
```
or alternatively:
```
gringo --text \
define.lp
```

Example 3.9. This usage is illustrated by the following program:

```
1 num(1).  num(2).  num(3).  num(4).  num(5).
2 squares(XX,YY) :-
          XX = X*X, Y*Y = YY, Y'-1 = Y,
          Y'*Y' = XX+YY, num(X), num(Y), X < Y.
```

The body of the rule in Line 2 defines four comparison predicates over '=', which directly or indirectly depend on `X` and `Y`. The values of `X` and `Y` are obtained via instances of the predicate `num`/1. The first comparison predicate depends on `X` to provide shortcut `XX`. Similarly, the second comparison predicate depends on `Y` to provide shortcut `YY`. The third comparison predicate provides variable `Y'` because it occurs in a simple arithmetic term, which is solved during unification. The last comparison predicate provides no variables and, hence, is just a test, checking whether its left-hand and right-hand sides are equal. ■

The simplified ground program can be inspected by invoking:
```
clingo --text \
unify.lp
```
or alternatively:
```
gringo --text \
unify.lp
```

Example 3.10. This example illustrates how to unify with function terms and tuples:

```
1 sym(f(a,1,2)).  sym(f(a,2,4)).  sym(f(a,b)).
2 sym( (a,1,2)).  sym( (a,2,4)).  sym( (a,b)).
3 unify1(X) :- f(a,X,X+1) = F, sym(F).
4 unify2(X) :-  (a,X,X+1) = T, sym(T).
```

Here, `f(a,X,X+1)` or `(a,X,X+1)`, respectively, is unified with instances of the predicate `sym`/1. To this end, arguments of `sym`/1 with matching arity are used to instantiate the variable `X` occurring as the second argument in terms on the left-hand sides of =. With a value for `X` at hand, we can further check whether the arithmetic evaluation of `X+1`, occurring as the third argument, coincides with the corresponding value given on the right-hand side of '='. ■

Remark 3.3. Note that comparison literals can be preceded by `not` or `not not`. In the first case, this is equivalent to using the complementary comparison literal (e.g., '<' and '>=' complement each other). In the second case, the prefix has no effect on the meaning of the literal.

An occurrence of a variable in the scope of a built-in comparison literal over '!=', '<', '<=','>', or '>=' does not count as a positive occurrence in the sense of safety (cf. Page 21), i.e., such comparison literals are not considered to be positive.

Unlike with the built-in comparison literals above, comparisons predicates over '=' are considered as positive (body) literals in the sense of safety (cf. Page 21), so that variables occurring on one side can be instantiated. However, this only works when unification can be made directionally, i.e., it must be possible to instantiate one side without knowing the values of variables on the other side. For example, the rule `p(X):-X=Y, Y=X.` is not accepted by *gringo* (or *clingo*) because values for `X` rely on values for `Y`, and vice versa. Only simple arithmetic terms can be unified with (cf. Remark 3.2). Hence, variable `X` in literal `X*X=8` must be bound by some other positive literal. ■

3.1.9 Intervals

Line 1 of Example 3.9 contains five facts of the form `num(`k`).` over consecutive integers k. For a more compact representation, *gringo* and *clingo* support integer intervals of the form i`..`j. Such an interval, representing each integer k such that $i \leqslant k \leqslant j$, is expanded during grounding. An interval is expanded differently depending on where it occurs. In the head of a rule, an interval is expanded conjunctively, while in the body of a rule, it is expanded disjunctively. So we could have simply written `num(1..5).` to represent the five facts.

Example 3.11. Consider the following program:

```
1 size(3).
2 grid(1..S,1..S) :- size(S).
```

Because all intervals in the second rule occur in the rule head, they expand conjunctively. Furthermore, the two intervals expand into the cross product `(1..3)`×`(1..3)`, resulting in the following set of facts:

```
2 grid(1,1). grid(1,2). grid(1,3).
  grid(2,1). grid(2,2). grid(2,3).
  grid(3,1). grid(3,2). grid(3,3).
```

Similarly, intervals can be used in a rule body. Typically, this is done using comparison literals over '=', which expand disjunctively:

```
2 grid(X,Y) :- X = 1..S, Y = 1..S, size(S).
```

This rule expands into the same set of facts as before. But intervals in comparison literals have the advantage that additional constraints can be added. For example, one could add the comparison literals `X-Y!=0` and `X+Y-1!=S` to the rule body to exclude the diagonals of the grid. ■

Remark 3.4. An occurrence of a variable in the specification of the bounds of an integer interval, like `S` in Line 2 of Example 3.11, does not count as a positive occurrence in the sense of safety (cf. Page 21). Hence, such a variable must also have another positive occurrence elsewhere; here in `size(S)`. ■

The simplified ground program obtained from intervals can be inspected by invoking:
`clingo --text int.lp`
or alternatively:
`gringo --text int.lp`

3.1.10 Pooling

The token ';' admits pooling alternative terms to be used as arguments of an atom, function, or tuple. Argument lists written in the form `(...,X;Y,...)` abbreviate multiple options: `(...,X), (Y,...)`. Pools are expanded just like intervals, i.e., conjunctively in the head and disjunctively in the body of a rule. In fact, the interval `1..3` is equivalent to the pool `(1;2;3)`.[8]

The simplified ground program obtained from pools can be inspected by invoking:

```
clingo --text \
pool.lp
```

or alternatively:

```
gringo --text \
pool.lp
```

Example 3.12. The following program makes use of pooling. It is similar to Example 3.11 but with the difference that, unlike intervals, pools have a fixed size:

```
1 grid((1;2;3),(1;2;3)).
```

Because all pools in this rule occur in the head, they are expanded conjunctively. Furthermore, the two pools expand into the cross product `(1..3)` × `(1..3)`, resulting again in the following set of facts:

```
grid(1,1).  grid(1,2).  grid(1,3).
grid(2,1).  grid(2,2).  grid(2,3).
grid(3,1).  grid(3,2).  grid(3,3).
```

Like intervals, pools can also be used in the body of a rule, where they are expanded disjunctively:

```
1 grid(X,Y) :- X = (1;2;3), Y = (1;2;3).
```

This rule expands into the same set of facts as before. As in Example 3.11, additional constraints involving `X` and `Y` can be added. ■

For another example on pooling, featuring non-consecutive elements, see Section 6.1.1.

3.1.11 Conditions and Conditional Literals

A *conditional literal* is of the form

$$L_0 : L_1, \ldots, L_n$$

where every L_j for $0 \leqslant j \leqslant n$ is a *literal*, $L_1, \ldots, L_n$ is called *condition*, and ':' resembles mathematical set notation. Whenever $n = 0$, we get a regular literal and denote it as usual by L_0.

For example, the rule

```
a :- b : c.
```

[8] We make use of the fact that one-elementary tuples must be made explicit by a trailing ',' (cf. Section 3.1.1). E.g., `(1;1,)` expands into `(1)` and `(1,)`, where `(1)` is equal to the integer `1`. On the other hand, note that the rule `p(X) :- X = (1,2;3,4).` is expanded into `p((1,2)).` and `p((3,4)).`, given that `(1,2)` and `(3,4)` are proper tuples, and the same facts are also obtained from `p((1,2;3,4)).` Unlike that, `p(1,2;3,4).` yields `p(1,2).` and `p(3,4).` because ';' here splits an argument list, rather than a tuple.

yields `a` whenever either `c` is false (and thus no matter whether `b` holds or not) or both `b` and `c` are true.

Remark 3.5. Logically, L_0 and $L_1, \ldots, L_n$ act as head and body, respectively, which gives $L_0 : L_1, \ldots, L_n$ the flavor of a nested implication (see [57] for details). ■

Together with variables, conditions allow for specifying collections of expressions within a single rule or aggregate. This is particularly useful for encoding conjunctions (or disjunctions) over arbitrarily many ground atoms as well as for the compact representation of aggregates (detailed in Section 3.1.12).

Example 3.13. The following program uses, in Line 5 and 6, conditions in a rule body and in a rule head, respectively:

```
1 person(jane).  person(john).
2 day(mon).  day(tue).  day(wed).  day(thu).
  day(fri).
3 available(jane) :- not on(fri).
4 available(john) :- not on(mon), not on(wed).
5 meet :- available(X) : person(X).
6 on(X) : day(X) :- meet.
```

The rules in Line 5 and 6 are instantiated as follows:

```
meet :- available(jane), available(john).
on(mon); on(tue); on(wed); on(thu); on(fri) :- meet.
```

The conjunction in the body of the first ground rule is obtained by replacing `X` in `available(X)` with all ground terms t such that `person(`t`)` holds, namely, with $t =$ `jane` and $t =$ `john`. Furthermore, the condition in the head of the rule in Line 6 turns into a disjunction over all ground instances of `on(X)` such that `X` is substituted by terms t for which `day(`t`)` holds. That is, conditions in the body and in the head of a rule are expanded to different basic language constructs.[9] ■

The reader can reproduce these ground rules by invoking:

```
clingo --text \
cond.lp
```

or alternatively:

```
gringo --text \
cond.lp
```

Further following set notation, a condition can be composed by separating literals with a comma, viz. '`,`'. Note that commas are used to separate both literals in rule bodies as well as conditions. To resolve this ambiguity, a condition is terminated with a semicolon '`;`' (rather than '`,`') when further body literals follow.

Example 3.14. The following program uses a literal with a composite condition in the middle of the rule body. Note the semicolon '`;`' after the condition:

```
1 set(1..4).
2 next(X,Z) :- set(X), #false : X < Y, set(Y), Y < Z;
      set(Z), X < Z.
```

[9] Recall our suggestion from Section 3.1.4 to use "choice constructs" (detailed in Section 3.1.12) instead of disjunction, unless the latter is required for complexity reasons. This also means that conditions must not be used *outside of aggregates* in rule heads if disjunction is unintended.

The conditional literal in the second rule evaluates to false whenever there is an element `Y` between `X` and `Z`. Hence, all rule instantiations where `X` and `Z` are not direct successors are discarded because they have a false body. On the other hand, whenever `X` and `Z` succeed each other, the condition is false for all elements `Y`. This means that the literal with condition stands for an empty conjunction, which is true:

```
set(1). set(2). set(3). set(4).
next(1,2). next(2,3). next(3,4).
```

We obtain an answer set where the elements of `set`/1 are ordered via `next`/2. ■

The reader can reproduce these ground rules by invoking:
```
clingo --text \
sort.lp
```
or alternatively:
```
gringo --text \
sort.lp
```

Remark 3.6. There are three important issues about the usage of conditions:

1. Any variable occurring within a condition does not count as a positive occurrence outside the condition in the sense of safety (cf. Page 21). Variables occurring in atoms not subject to any condition are *global*. Each variable within an atom in front of a condition must be global or have a positive occurrence on the right-hand side of the condition.

2. During grounding, the instantiation of global variables takes precedence over non-global ones, that is, the former are instantiated before the latter. As a consequence, variables that occur globally are substituted by terms before a condition is further evaluated. Hence, the names of variables in conditions must be chosen with care, making sure that they do not *accidentally* match the names of global variables.

3. We suggest using *domain predicates* [80] or built-ins (both used in Line 3 of Example 3.14) in conditions. Literals over such predicates are completely evaluated during grounding. In a logic program, domain predicates can be recognized by observing that they are neither subject to negative recursion (through `not`) nor to disjunction or "choice constructs" (detailed in Section 3.1.12) in the head of any rule. The domain predicates defined in Example 3.14 are `set`/1 and `next`/1. Literals with such conditions expand to arbitrary length disjunctions or conjunctions in the head or body of a rule, respectively. Otherwise, conditions give rise to nested implications. For further details see [57].

■

3.1.12 Aggregates

Aggregates are expressive modeling constructs that allow for forming values from groups of selected items. Together with comparisons they allow for expressing conditions over these terms. For instance, we may state that the sum of a semester's course credits must be at least 20, or that the sum of prizes of shopping items should not exceed 30 Euros.

More formally, an aggregate is a function on a set of tuples that are normally subject to conditions. By comparing an aggregated value with given values, we can extract a truth value from an aggregate's evaluation, thus obtaining an aggregate atom. Aggregate atoms come in two variants depending on whether they occur in a rule head or body.

Body Aggregates The form of an *aggregate atom* occurring in a rule body is as follows:

$$s_1 \prec_1 \alpha \{ \boldsymbol{t}_1 : \boldsymbol{L}_1 ; \ldots ; \boldsymbol{t}_n : \boldsymbol{L}_n \} \prec_2 s_2$$

Here, all $\boldsymbol{t}_i$ and $\boldsymbol{L}_i$, forming *aggregate elements*, are tuples of terms and literals (as introduced in Section 3.1.1), respectively. If a literal tuple is empty and the corresponding term tuple is non-empty, then the colon can be omitted. α is the name of some function that is to be applied to the term tuples $\boldsymbol{t}_i$ that remain after evaluating the conditions expressed by $\boldsymbol{L}_i$. Finally, the result of applying α is compared by means of the comparison predicates $\prec_1$ and $\prec_2$ to the terms s_1 and s_2, respectively. Note that one of the *guards* '$s_1 \prec_1$' or '$\prec_2 s_2$' (or even both) can be omitted; left out comparison predicates $\prec_1$ or $\prec_2$ default to '<=', thus interpreting s_1 and s_2 as lower or upper bound, respectively.

Currently, *gringo* (and *clingo*) support the aggregates `#count` (the number of elements; used for expressing cardinality constraints), `#sum` (the sum of weights; used for expressing weight constraints), `#sum+` (the sum of positive weights), `#min` (the minimum weight), and `#max` (the maximum weight). The weight refers to the first element of a term tuple. Aggregate atoms, as described above, are obtained by writing either `#count`, `#sum`, `#sum+`, `#min`, or `#max` for α. Note that, unlike the other aggregates, the `#count` aggregate does not require weights.

For example, instances of the natural language examples for aggregates given at the beginning of this section can be expressed as follows.

```
20 <= #sum { 4 : course(db);      6 : course(ai);
             8 : course(project); 3 : course(xml) }

#sum { 3 : bananas; 25 : cigars; 10 : broom } <= 30
```

Both aggregate atoms can be used in the body of a rule like any other atom, possibly preceded by negation. Within both aggregate atoms, atoms like `course(ai)` or `broom` are associated with weights. Assuming that `course(db)`, `course(ai)` as well as `bananas` and `broom` are true, the aggregates inner sets evaluate to {`4;6`} and {`3;10`}, respectively. After applying the `#sum` aggregate function to both sets, we get `20 <= 10` and `13 <= 30`; hence, in this case, the second aggregate atom holds while the first one does not.

As indicated by the curly braces, the elements within aggregates are treated as members of a set. Hence, duplicates are not accounted for twice. For instance, the following aggregate atoms express the same:

```
#count { 42 : a;          t : not b            } = 2
```

```
#count { 42 : a; 42 : a; t : not b; t : not b } = 2
```

That is, if `a` holds but not `b`, both inner sets reduce to {`42;t`}; and so both aggregate atoms evaluate to true. However, both are different from the aggregate

```
#count { 42 : a; t : not b; s : not b } = 2
```

that holds if both `a` and `b` are false, yielding `#count{t;s} = 2`.

Likewise, the elements of other aggregates are understood as sets. Consider the next two summation aggregates:

```
#sum { 3 : cost(1,2,3); 3 : cost(2,3,3) } = 3
#sum { 3,1,2 : cost(1,2,3);
       3,2,3 : cost(2,3,3) } = 6
```

As done in Section 6.2.1, an atom like `cost(1,2,3)` can be used to represent an arc from node `1` to `2` with cost `3`. If both `cost(1,2,3)` and `cost(2,3,3)` hold, the first sum evaluates to `3`, while the second yields `6`. Note that all term tuples, the singular tuple `3` as well as the ternary tuples `3,1,2` and `3,2,3` share the same weight, viz. `3`. However, the set property makes the first aggregate count edges with the same cost only once, while the second one accounts for each edge no matter whether they have the same cost or not. To see this, observe that after evaluating the conditions in each aggregate, the first one reduces to `#sum{3}`, while the second results in `#sum{3,1,2;3,2,3}`. In other words, associating each cost with its respective arc enforces a multi-set property; in this way, the same cost can be accounted for several times.

Head Aggregates Whenever a rule head is a (single) aggregate atom, the derivable head literals must be distinguished. This is done by appending such atoms (or in general literals) separated by an additional ':' to the tuples of the aggregate elements:

$$s_1 \prec_1 \alpha \ \{ \ \boldsymbol{t}_1 : L_1 : \boldsymbol{L_1} ; \ldots ; \boldsymbol{t}_n : L_n : \boldsymbol{L_n} \ \} \prec_2 s_2$$

Here, all L_i are literals as introduced in Section 3.1.2, while all other entities are as described above. The second colon in $\boldsymbol{t}_i : L_i : \boldsymbol{L_i}$ is dropped whenever $\boldsymbol{L_i}$ is empty, yielding $\boldsymbol{t}_i : L_i$.

Remark 3.7. Aggregate atoms in the head can be understood as a combination of unrestricted choices with body aggregates enforcing the constraint expressed by the original head aggregate. In fact, when producing *smodels* format, all aggregate atoms occurring in rule heads are transformed away. For details consult [78, 33]. ■

Shortcuts There are some shorthands that can be used in the syntactic representation of aggregates. The expression

$$s_1 \prec_1 \ \{ \ L_1 : \boldsymbol{L_1} ; \ldots ; L_n : \boldsymbol{L_n} \ \} \prec_2 s_2$$

where all entities are defined as above is a shortcut for

$$s_1 \prec_1 \texttt{\#count}\ \{\ \boldsymbol{t}_1 : L_1 : \boldsymbol{L_1}\ ;\ \ldots\ ;\ \boldsymbol{t}_n : L_n : \boldsymbol{L_n}\ \}\ \prec_2 s_2$$

if it appears in the head of a rule, and it is a shortcut for

$$s_1 \prec_1 \texttt{\#count}\ \{\ \boldsymbol{t}_1 : L_1 \texttt{,} \boldsymbol{L_1}\ ;\ \ldots\ ;\ \boldsymbol{t}_n : L_n \texttt{,} \boldsymbol{L_n}\ \}\ \prec_2 s_2$$

if it appears in the body of a rule. In both cases, all $\boldsymbol{t}_i$ are pairwise distinct term tuples generated by *gringo* whenever the distinguished (head) literals L_i are different. Just like with aggregates, the guards '$s_1 \prec_1$' and '$\prec_2 s_2$' are optional, and the symbols '$\prec_1$' and '$\prec_2$' default to '<=' if omitted.

For example, the rule

```
{ a; b }.
```

is expanded to

```
#count { 0,a : a; 0,b : b }.
```

Here, *gringo* generates two distinct term tuples `0,a` and `0,b`. With *clingo*, we obtain four answer sets representing all sets over `a` and `b`.

Recurrences of literals yield identical terms, as we see next. The rule

```
{ a; a }.
```

is expanded to

```
#count { 0,a : a; 0,a : a }.
```

In fact, within the term tuple produced by *gringo*, the first term indicates the number of preceding default negations, and the second reproduces the atom as a term in order to make the whole term tuple unique. To see this, observe that the integrity constraint

```
:- { a; not b; not not c } > 0.
```

is expanded to

```
:- #count { 0,a : a;
            1,b : not b;
            2,c : not not c } > 0.
```

Remark 3.8. By allowing the omission of `#count`, so-called "cardinality constraints" [78] can almost be written in their traditional notation (without keyword, yet different separators), as put forward in the *lparse* grounder [80]. ■

Having discussed head and body aggregate atoms, let us note that there is a second way to use body aggregates: they act like positive literals when used together with comparison predicate '='. For instance, variable `X` is safe in the following rules:

```
cnt(X) :- X = #count { 2 : a; 3 : a }.
sum(X) :- X = #sum   { 2 : a; 3 : a }.
pos(X) :- X = #sum+  { 2 : a; 3 : a }.
min(X) :- X = #min   { 2 : a; 3 : a }.
max(X) :- X = #max   { 2 : a; 3 : a }.
```

Under the assumption that atom `a` holds, the atoms `cnt(2)`, `sum(5)`, `pos(5)`, `min(2)`, and `max(3)` are derived by the above rules. If `a` does not hold, we derive `cnt(0)`, `sum(0)`, `pos(0)`, `min(#sup)`, and `max(#inf)`. Here, the special constants `#sup` and `#inf` (introduced in Section 3.1.1), obtained by applying `#min` and `#max` to the empty set of weights, indicate the neutral elements of the aggregates. These constants can also be used as weights, subject to `#min` and `#max` (in order to exceed any other ground term):

```
bot :-      #min { #inf : a } -1000.
top :- 1000 #max { #sup : a }.
```

Assuming that atom `a` holds, the atoms `bot` and `top` are derived by the above rules because both `#inf <= -1000` and `1000 <= #sup` hold (cf. Section 3.1.8 for details how terms are ordered).

Remark 3.9. Although it seems convenient to use aggregates together with the '=' predicate, this feature should be used with care. If the literals of an aggregate belong to domain predicates (see Remark 3.6) or built-ins, the aggregate is evaluated during grounding to exactly one value. Otherwise, if the literals do not belong to domain predicates, the value of an aggregate is not known during grounding, in which case *gringo* or *clingo* unwraps all possible outcomes of the aggregate's evaluation. The latter can lead to a space blow-up, which should be avoided whenever possible. For instance, unwrapping the aggregate in

```
{ a; b; c }.
:- #sum { 1 : a; 2 : b; 3 : c } = N, N > 3.
```

yields three integrity constraints:

```
:- #sum { 1 : a; 2 : b; 3 : c } = 4.
:- #sum { 1 : a; 2 : b; 3 : c } = 5.
:- #sum { 1 : a; 2 : b; 3 : c } = 6.
```

Such duplication does not happen when we use a comparison predicate instead:

```
:- #sum { 1 : a; 2 : b; 3 : c } > 3.
```

Hence, it is advisable to rather apply comparison predicate '>' directly. In general, aggregates should only be used to bind variables if they refer solely to domain predicates and built-ins. ■

Non-ground Aggregates After considering the syntax and semantics of ground aggregate atoms, we now turn our attention to non-ground aggregates. Regarding

contained variables, only variable occurrences in the guards give rise to global variables. Hence, any variable in an aggregate element must be bound by either a positive global occurrence or a variable that occurs positively in its condition $\boldsymbol{L_i}$. Variable names in aggregate elements have to be chosen carefully to avoid clashes with global variables. Furthermore, pools and intervals in aggregate elements give rise to multiple aggregate elements; very similar to the disjunctive unpacking of pools and intervals in rules. The following example, making exhaustive use of aggregates, demonstrates a variety of features (note that it ignores Remark 3.9).

Example 3.15. Consider a situation where an informatics student wants to enroll for a number of courses at the beginning of a new term. In the university calendar, eight courses are found eligible, and they are represented by the following facts:

```
1 course( 1,1,5; 1,2,5                 ).
2 course( 2,1,4; 2,2,4                 ).
3 course( 3,1,6;         3,3,6         ).
4 course( 4,1,3;         4,3,3; 4,4,3  ).
5 course( 5,1,4;                5,4,4  ).
6 course(         6,2,2; 6,3,2         ).
7 course(         7,2,4; 7,3,4; 7,4,4  ).
8 course(                8,3,5; 8,4,5  ).
```

In an instance of `course`/3, the first argument is a number identifying one of the eight courses, and the third argument provides the course's contact hours per week. The second argument stands for a subject area: `1` corresponding to "theoretical informatics", `2` to "practical informatics", `3` to "technical informatics", and `4` to "applied informatics". For instance, atom `course(1,2,5)` expresses that course `1` accounts for `5` contact hours per week that may be credited to subject area `2` ("practical informatics"). Observe that a single course is usually eligible for multiple subject areas.

After specifying the above facts, the student starts to provide personal constraints on the courses to enroll. The student's first condition is to enroll in `3` to `6` courses:

```
9 3 { enroll(C) : course(C,S,H) } 6.
```

Instantiating the above `#count` aggregate yields the following ground rule:

```
3 <= #count { 0,enroll(1) : enroll(1);
              0,enroll(2) : enroll(2);
              0,enroll(3) : enroll(3);
              0,enroll(4) : enroll(4);
              0,enroll(5) : enroll(5);
              0,enroll(6) : enroll(6);
              0,enroll(7) : enroll(7);
              0,enroll(8) : enroll(8) } <= 6.
```

The full ground program is obtained by invoking:

```
clingo --text \
aggr.lp
```

or alternatively:

```
gringo --text \
aggr.lp
```

Observe that an instance of atom `enroll(C)` is included for each instantiation of `C` such that `course(C,S,H)` holds for some values of `S` and `H`. Duplicates resulting from distinct values for `S` are removed, thus obtaining the above set of ground atoms.

The next constraints of the student regard the subject areas of enrolled courses:

```
10 :-   #count { C,S :     enroll(C), course(C,S,H) } 10.
11 :- 2 #count { C,2 : not enroll(C), course(C,2,H) }.
12 :- 6 #count { C,3 :     enroll(C), course(C,3,H);
13               C,4 :     enroll(C), course(C,4,H) }.
```

Each of the three integrity constraints above contains a `#count` aggregate, in which '`,`' is used to construct composite conditions (introduced in Section 3.1.11). Recall that aggregates operate on sets and thus duplicates are removed; hence, we use term tuples to take into account courses together with their subject areas. Thus, the integrity constraint in Line 10 is instantiated as follows:[10]

```
10 :- 10 >= #count { 1,1 : enroll(1); 1,2 : enroll(1);
                     2,1 : enroll(2); 2,2 : enroll(2);
                     3,1 : enroll(3); 3,3 : enroll(3);
    4,1 : enroll(4); 4,3 : enroll(4); 4,4 : enroll(4);
                     5,1 : enroll(5); 5,4 : enroll(5);
                     6,2 : enroll(6); 6,3 : enroll(6);
    7,2 : enroll(7); 7,3 : enroll(7); 7,4 : enroll(7);
                     8,3 : enroll(8); 8,4 : enroll(8) }.
```

Note that courses `4` and `7` count three times because they are eligible for three subject areas, viz., there are three distinct instantiations for `S` in `course(4,S,3)` and `course(7,S,4)`, respectively. Comparing the above ground instance, the meaning of the integrity constraint in Line 10 is that the number of eligible subject areas over all enrolled courses must be more than `10`. Similarly, the integrity constraint in Line 11 expresses the requirement that at most one course of subject area `2` ("practical informatics") is not enrolled, while Line 12 stipulates that the enrolled courses amount to less than six nominations of subject area `3` ("technical informatics") or `4` ("applied informatics").

The remaining constraints of the student deal with contact hours. To express them, we first introduce an auxiliary rule and a fact:

```
14 hours(C,H) :- course(C,S,H).
15 max_hours(20).
```

The rule in Line 14 projects instances of `course`/3 to `hours`/2, thereby, dropping courses' subject areas. This is used to not consider the same course multiple times within the following integrity constraints:[11]

[10] Because contact hours are uniquely associated to a course, *gringo*'s shortcut expansion of `:- { course(C,S,H) : enroll(C) } 10.` is equivalent to the rule in Line 10 here. Similar equivalences hold for the other `#count` aggregates.

[11] Alternatively, we could also use `course(C,_,H)`.

```
16 :- not M-2 #sum { H,C : enroll(C), hours(C,H) } M,
      max_hours(M).
17 :-   #min { H,C : enroll(C), hours(C,H) } 2.
18 :- 6 #max { H,C : enroll(C), hours(C,H) }.
```

As illustrated in Line 16, we may use default negation via '`not`' in front of aggregate atoms, and bounds may be specified by terms with variables. In fact, by instantiating `M` to `20`, we obtain the following ground instance of the integrity constraint in Line 16:

```
16 :- not 18 <= #sum {
           5,1 : enroll(1); 4,2 : enroll(2);
           6,3 : enroll(3); 3,4 : enroll(4);
           4,5 : enroll(5); 2,6 : enroll(6);
           4,7 : enroll(7); 5,8 : enroll(8) } <= 20.
```

The above integrity constraint states that the `#sum` of contact hours per week must lie in-between `18` and `20`. Note that the `#min` and `#max` aggregates in Line 17 and 18, respectively, work on the same set of aggregate elements as in Line 16. While the integrity constraint in Line 17 stipulates that any course to enroll must include more than `2` contact hours, the one in Line 18 prohibits enrolling for courses of `6` or more contact hours. Of course, the last two requirements could also be formulated as follows:

```
17 :- enroll(C), hours(C,H), H <= 2.
18 :- enroll(C), hours(C,H), H >= 6.
```

Finally, the following rules illustrate the use of aggregates together with comparison predicate '='.

```
19 courses(N) :- N = #count { C : enroll(C) }.
20 hours(N) :- N = #sum { H,C : enroll(C), hours(C,H) }.
```

The role of aggregates here is different from before, as they now serve to bind an integer to variable `N`. The effect of Line 19 and 20, which do not follow the recommendation in Remark 3.9, is that the student can read off the number of courses to enroll and the amount of contact hours per week from instances of `courses`/1 and `hours`/1 belonging to an answer set. In fact, running *clingo* or *clasp* shows that a unique collection of `5` courses to enroll satisfies all requirements: the courses `1`, `2`, `4`, `5`, and `7`, amounting to `20` contact hours per week. ■

To compute the unique answer set of the program, invoke:

```
clingo aggr.lp 0
```

or alternatively:

```
gringo aggr.lp | \
clasp 0
```

Remark 3.10. Users familiar with *gringo* 3 may remember that conditions in aggregates had to be either literals over domain predicates or built-ins. This restriction does not exist anymore in *gringo* and *clingo* 4. ■

3.1.13 Optimization

Optimization statements extend the basic question of whether a set of atoms is an answer set to whether it is an optimal answer set. To support this reasoning mode,

gringo and *clingo* adopt *dlv*'s weak constraints [14]. The form of weak constraints is similar to integrity constraints (cf. Section 3.1.2) being associated with a term tuple:

$$\texttt{:\sim}\ L_1\texttt{,}\ldots\texttt{,}L_n\texttt{.}\ \ \texttt{[}w\texttt{@}p\texttt{,}t_1\texttt{,}\ldots\texttt{,}t_n\texttt{]}$$

The priority '$\texttt{@}p$' is optional. For simplicity, we first consider the non-prioritized case omitting '$\texttt{@}p$'. Whenever the body of a weak constraint is satisfied, it contributes its term tuple (as with aggregates, each tuple is included at most once) to a cost function. This cost function accumulates the integer weights w of all contributed tuples just like a `#sum` aggregate does (cf. Section 3.1.12). The semantics of a program with weak constraints is intuitive: an answer set is *optimal* if the obtained cost is minimal among all answer sets of the given program. Whenever there are different priorities attached to tuples, we obtain a (possibly zero) cost for each priority. To determine whether an answer set is optimal, we do not just compare two single costs but lexicographically compare cost tuples whose elements are ordered by priority (greater is more important). Note that a tuple is always associated with a priority; if it is omitted, then the priority defaults to zero. A weak constraint is safe if the variables in its term tuples are bound by the atoms in the body and the safety requirements for the body itself are the same as for integrity constraints.

As an alternative way to express an optimization problem, there are optimization statements. A minimize statement of the form

$$\texttt{\#minimize}\ \{\ w_1\texttt{@}p_1\texttt{,}\boldsymbol{t}_1\texttt{:}\boldsymbol{L}_1\texttt{,}\ldots\texttt{,}w_n\texttt{@}p_n\texttt{,}\boldsymbol{t}_n\texttt{:}\boldsymbol{L}_n\ \}\texttt{.}$$

represents the following n weak constraints:

$$\texttt{:\sim}\ \boldsymbol{L}_1\texttt{.}\ \ \texttt{[}w_1\texttt{@}p_1\texttt{,}\boldsymbol{t}_1\texttt{]}\quad\ldots\quad\texttt{:\sim}\ \boldsymbol{L}_n\texttt{.}\ \ \texttt{[}w_n\texttt{@}p_n\texttt{,}\boldsymbol{t}_n\texttt{]}$$

Moreover, maximize statements can be viewed as minimize statements with inverse weights. Hence, a maximize statement of the form

$$\texttt{\#maximize}\ \{\ w_1\texttt{@}p_1\texttt{,}\boldsymbol{t}_1\texttt{:}\boldsymbol{L}_1\texttt{,}\ldots\texttt{,}w_n\texttt{@}p_n\texttt{,}\boldsymbol{t}_n\texttt{:}\boldsymbol{L}_n\ \}\texttt{.}$$

represents the following n weak constraints:

$$\texttt{:\sim}\ \boldsymbol{L}_1\texttt{.}\ \ \texttt{[}-w_1\texttt{@}p_1\texttt{,}\boldsymbol{t}_1\texttt{]}\quad\ldots\quad\texttt{:\sim}\ \boldsymbol{L}_n\texttt{.}\ \ \texttt{[}-w_n\texttt{@}p_n\texttt{,}\boldsymbol{t}_n\texttt{]}$$

As with weak constraints, the priorities '$\texttt{@}p_i$' are optional and default to zero.

Example 3.16. To illustrate optimization, we consider a hotel booking situation where we want to choose one among five available hotels. The hotels are identified via numbers assigned in descending order of stars. Of course, the more stars a hotel has, the more it costs per night. As ancillary information, we know that hotel `4` is located on a main street, which is why we expect its rooms to be noisy. This knowledge is specified in Line 1–7 of the following program:

```
 1 { hotel(1..5) } = 1.
 2 star(1,5). cost(1,170).
 3 star(2,4). cost(2,140).
 4 star(3,3). cost(3,90).
 5 star(4,3). cost(4,75).  main_street(4).
 6 star(5,2). cost(5,60).
 7 noisy :- hotel(X), main_street(X).
 8 #maximize { Y@1,X : hotel(X), star(X,Y) }.
 9 #minimize { Y/Z@2,X : hotel(X), cost(X,Y), star(X,Z) }.
10 :~ noisy. [ 1@3 ]
```

Line 8–9 contribute optimization statements in inverse order of significance, according to which we want to choose the best hotel to book. The most significant optimization statement in Line 10 states that avoiding noise is our main priority. The secondary optimization criterion in Line 9 consists of minimizing the cost per star. Finally, the third optimization statement in Line 8 specifies that we want to maximize the number of stars among hotels that are otherwise indistinguishable. The optimization statements in Line 8–10 are instantiated as follows:

The full ground program is obtained by invoking:

```
clingo --text opt.lp
```

or alternatively:

```
gringo --text opt.lp
```

```
 8 :~ hotel(1). [-5@1,1]
   :~ hotel(2). [-4@1,2]
   :~ hotel(3). [-3@1,3]
   :~ hotel(4). [-3@1,4]
   :~ hotel(5). [-2@1,5]
 9 :~ hotel(1). [34@2,1]
   :~ hotel(2). [35@2,2]
   :~ hotel(3). [30@2,3]
   :~ hotel(4). [25@2,4]
   :~ hotel(5). [30@2,5]
10 :~ noisy. [ 1@3 ]
```

If we now use *clasp* or *clingo* to compute an optimal answer set, we find that hotel `4` is not eligible because it implies `noisy`. Thus, hotel `3` and `5` remain as optimal with respect to the second most significant optimization statement in Line 9. This tie is broken via the least significant optimization statement in Line 8 because hotel `3` has one star more than hotel `5`. We thus decide to book hotel `3` offering `3` stars to cost `90` per night. ■

To compute the unique optimal answer set, invoke:

```
clingo opt.lp 0
```

or alternatively:

```
gringo opt.lp | \
clasp 0
```

3.1.14 External Functions

Utilizing the scripting languages Lua or Python[12], *gringo*'s input language can be enriched by arbitrary functions. We focus on functions that are evaluated during grounding here. In Section 4, we explain how to take complete control of the grounding and solving process using the scripting API. We do not give an introduction to

[12] `http://lua.org` and `http://python.org`

Lua or Python here (there are numerous tutorials on the web), but give some examples showing the capabilities of this integration. In the following, we show code snippets for both scripting languages. Note that our precompiled binaries ship with Lua support and can be used to run the Lua examples. To enable Python support, *gringo* and *clingo* have to be compiled from source (cf. Section 1.1). A complete reference for the Python scripting API is available at: [13]

`http://potassco.org/clingo`

Example 3.17. The first example shows how to add a simple arithmetic function:

```
1  #script (lua)

3  clingo = require("clingo")
4  N = clingo.Number

6  function gcd(a, b)
7    if a.number == 0 then
8      return b
9    else
10     na = a.number
11     nb = b.number
12     nc = nb % na
13     return gcd(N(nc), a)
14   end
15 end

17 #end.
```

```
1  #script (python)

3  import clingo
4  N = clingo.Number

6  def gcd(a, b):
7    if a.number == 0:
8      return b
9    else:
10     na = a.number
11     nb = b.number
12     nc = nb % na
13     return gcd(N(nc), a)

17 #end.
```

In Line 6, we add a function that calculates the greatest common divisor of two numbers. Integers from a logic program are returned as objects of type `Symbol`[14] — a variant type capturing ground terms. The numeric value can be accessed using the `number` property in both Lua and Python. To construct a numeric term, the `Number` constructor is used. The `gcd` function can then be used in a logic program:

To inspect the unique answer set of the program, invoke:
```
gringo --text \
gcd.lp gcd-lua.lp
```
or:
```
gringo --text \
gcd.lp gcd-py.lp
```
Calls to *clingo* are similar.

```
1 p(210,213).
2 p(1365,385).
3 gcd(X,Y,@gcd(X,Y)) :- p(X,Y).
```

The function is called in Line 3 and the result stored in predicate `gcd`/3. Note that external function calls look like function terms but are preceded by '`@`'. As with non-simple arithmetic terms according to Remark 3.2, variable occurrences in arguments to external functions do not count as positive in the sense of safety (cf. Page 21). In

[13] The API of *clingo* series 4 is described at `http://potassco.sourceforge.net/gringo.html`.

[14] Strictly speaking there are no classes in Lua, the Userdata type together with a metatable is used to emulate classes.

Line 3, values for `X` and `Y` are thus obtained from `p(X,Y)` in order to apply the `gcd` function to them. ■

Example 3.18. This example shows how to return multiple values from a function:

```
1  #script (lua)

3  function rng(a, b)
4    ret = {}
5    na = a.number
6    nb = b.number
7    for i = na,nb do
8      table.insert(ret, i)
9    end
10   return ret
11 end

13 #end.
```

```
1  #script (python)

3  def rng(a, b):
4    na = a.number
5    nb = b.number

10   return range(na, na+1)

13 #end.
```

In Line 3, we add a function that emulates an interval. Instead of just returning one number, this function returns a table of numbers in Lua and a list of numbers in Python, respectively. The `rng` function can then be used in a logic program:

```
1 p(1,3).
2 p(5,10).
3 rng(X,Y,@rng(X,Y)) :- p(X,Y).
```

The function is called in Line 3 and the result stored in predicate `rng`/3. The values in the table or list returned from a call to `rng(X,Y)` are then successively inserted. In fact, this function behaves exactly like the interval `X..Y`.

An interesting use case for returning multiple values is to pull whole instances from external sources, like for example a database or some text file not already in fact format. ■

To inspect the unique answer set of the program, invoke:

```
gringo --text \
rng.lp rng-lua.lp
```

or:

```
gringo --text \
rng.lp rng-py.lp
```

Calls to *clingo* are similar.

As we have seen in the previous example, the `number` property is used to get the numeric representation of a term from objects of type `Symbol`. In fact, all terms are captured by the `Symbol` class. Similarly to numeric terms, the `string` property is used to get the representation of quoted strings. For constants and functions, there is the property `name` to access the string representation of the constant or the name of the function term. Furthermore, the arguments of a function term can be accessed using the `arguments` property. Note that constants as well as tuples are considered as special cases of function terms. The former have an empty argument list and the latter an empty name. Finally, the terms `#sup` and `#inf` are mapped to the constants `Sup` and `Inf`. Both are subclasses of class `Symbol`, too. Unlike other terms both are captured by unique objects.

To construct terms from within the scripting language, the global functions `Function(name, arguments)`, `Number(number)`, and

`String(string)` are used. All of them (and their advanced usage) are fully documented in the Python API documentation.

Example 3.19. This example shows how to inspect and create terms:

```
1  #script (lua)

3  clingo = require("clingo")
4  F = clingo.Function

6  function g(c, f)
7    n = c.name
8    r = F(n, f.arguments)
9    return r
10 end

12 #end.
```

```
1  #script (python)

3  import clingo
4  F = clingo.Function

6  def g(c, f):
7    n = c.name
8    r = F(n, f.arguments)
9    return r

12 #end.
```

In Line 6, we add a function `g` that takes a constant and a tuple as arguments and returns a function term with the name of the constant and the tuple as arguments. The `g` function can then be used in a logic program:

To inspect the unique answer set of the program, invoke:
```
gringo --text \
term.lp term-lua.lp
```
or:
```
gringo --text \
term.lp term-py.lp
```
Calls to *clingo* are similar.

```
1 p(f, (1,2)).
2 p(g, (a,b)).
3 g(X,Y,@g(X,Y)) :- p(X,Y).
```

The function is called in Line 3 and the result stored in predicate `g`/3. Using this scheme, new terms that cannot be constructed by means of plain ASP can be created during grounding. Another interesting application might be string concatenation. ■

Remark 3.11.

1. The grounder assumes that all external functions are deterministic. That is, when a function is called multiple times with the same arguments during grounding, then it should return the same values. Adding functions that do not comply with this assumption can lead to undesired results.

2. If an error occurs during the evaluation of an external function, a warning is printed and the current instance of a rule or condition is dropped. For example, this happens when the `gcd` function from Example 3.17 is applied to non-integer arguments.

■

3.1.15 Meta-Statements

After considering the language of logic programs, we now introduce features going beyond the contents of a program.

Comments To annotate the source code of a logic program, a logic program file may include comments. A comment until the end of a line is initiated by symbol '%', and a comment within one or over multiple lines is enclosed in '%*' and '*%'. As an abstract example, consider:

```
tos(jim).    %* enclosed comment *%  tos(spock).
tos(bones).  % comment till end of line
tos(scotty). tos(chekov).
%*
comment over multiple lines
*%
tos(uhura).  tos(sulu).
```

Show Statements Usually, only a subset of the atoms belonging to an answer set characterizes a solution. In order to suppress the atoms of "irrelevant" predicates from the output or even to show arbitrary terms, the `#show` directive can be used. There are three different kinds of such statements:

Show atoms: `#show` p/n`.`
Show terms: `#show` t`:`L_1`,...,`L_n`.`
Show nothing: `#show.`

The first `#show` statement is the most commonly used form. Whenever there is at least one statement of this form, all atoms are hidden, except for those over predicates p/n given by the respective `#show` statements. The second form can be used to show arbitrary terms. The term t is part of the output if the literals in the condition after the ':' hold. Unlike the previous form, this statement does not automatically hide all atoms. To hide all atoms in this case and only show selected terms, the last statement (mnemonic: show nothing) can be added to suppress all atoms in the output.

Example 3.20. This example illustrates the common use case to selectively show atoms:

```
1 p(1). p(2). p(3).
2 { q(X) : p(X) }.
3 a :- q(1).
4 #show a/0.
5 #show q/1.
```

To inspect the output, invoke:
`clingo showa.lp 0`
or alternatively:
`gringo showa.lp | \`
`clasp 0`

Only atoms over `q`/1 and `a` appear in the output here. ■

Example 3.21. This example illustrates how to show terms:

```
1 p(1). p(2). p(3).
2 { holds(q(X)) : p(X) }.
```

To inspect the output, invoke:
`clingo showt.lp 0`
or alternatively:
`gringo showt.lp | \`
`clasp 0`

```
3 holds(a) :- holds(q(1)).
4 #show.
5 #show X : holds(X).
```

When running this example, the same output as in the previous example is produced. This feature is especially handy when applying meta-programming techniques (cf. Section 9) where the signatures of the reified atoms are not fixed and `holds(·)` atoms would just clutter the output. ■

Remark 3.12. The second form of `#show` statements to show terms may contain variables. Regarding safety (cf. Page 21), it behaves similar to a rule, where the term t takes the role of the head and the condition after the colon the role of the body. ■

Const Statements Constants appearing in a logic program may actually be placeholders for concrete values provided by a user. An example of this is given in Section 6.1. Via the `#const` directive, one may define a default value to be inserted for a constant. Such a default value can still be overridden via command line option `--const` (cf. Section 7.1). Syntactically, `#const` must be followed by an assignment having a constant on the left-hand side and a term without variables, pools, and intervals on the right-hand side.

Example 3.22. This example is about using the grounder as a simple calculator:

```
1 #const x = 42.
2 #const y = f(x,z).
3 p(x,y).
```

Try running this example using the following calls:

```
gringo --text const.lp
gringo --text const.lp -c x=6   -c z=6
gringo --text const.lp -c x=6+6 -c y=6
gringo --text const.lp -c x="6+6*6"
```

Note that quotes have to be added to prevent the shell from expanding the '`*`' in the last call or from interpreting parentheses in functions. ■

External Statements External statements are used to declare atoms that should not be subject to certain simplifications. Namely, atoms marked external are not removed from the bodies of rules, conditions, etc., even if they do not appear in the head of any rule. The main use case is to implement extensions to plain ASP solving, like multi-shot solving detailed in Section 4. An `#external` statement has the following form:

$$\texttt{\#external}\ A\texttt{:}L_1\texttt{,}\ldots\texttt{,}L_n\texttt{.}$$

Here, A is an atom over some predicate and the part following the ':' is a condition. The condition is instantiated to obtain a set of external atoms. Note that the condition is discarded after grounding, hence, it is a good idea to use only domain predicates or built-ins after the colon.[15]

To inspect the instantiation of externals, invoke:
```
clingo --text ext.lp
```
or alternatively:
```
gringo --text ext.lp
```

Example 3.23. Consider the following example:

```
1 p(1). p(2). p(3).
2 #external q(X) : p(X).
3 q(1).
4 r(X) :- q(X).
```

The `#external` statement in Line 2 gives rise to three external atoms, which appear accordingly in the text output. With these three atoms, the rule in Line 4 yields three ground instantiations, where atoms `q(2)` and `q(3)` appear in the body. Because we have the fact `q(1)` in Line 3, the atom `q(1)` is still subject to simplification and removed from the body of the respective instantiation of the rule in Line 4. The idea here is that no matter how `q(1)` is supplied externally, there can never be an answer set that does not contain `q(1)`. ■

Remark 3.13. External statements that contain variables have very similar requirements regarding safety as rules (cf. Page 21). The atom A takes the role of the head and the condition after the colon the role of the body. ■

Program Parts A logic program can be organized in multiple program parts. To begin a new program part, we write a statement

$$\texttt{\#program } p(s_1,\ldots,s_n).$$

where p is the program part name and the parameters s_i are constants. If n is zero, then the parentheses can be omitted. All rules, external statements, and show statements for terms up to the next `#program` statement or the end of the file belong to the program part p/n. Rules that are not subject to any such directive are included in the `base`/0 part.

The default behavior of *gringo* is to ground (and solve in the case of *clingo*) the `base`/0 part. Using the scripting API (cf. Section 4), we can ground other parts than `base`/0, too. Occurrences of constants that are parameters of a part are replaced with ground terms when instantiating the program part.

Example 3.24. The following example shows how to instantiate program parts:

```
1 a.
2 #program a(s,t).
3 b(s,t).
4 #program base.
5 c.
```

[15] Non-domain predicates are supported, too, because in some situations it might be inconvenient to specify domain predicates.

The above program is organized in two parts, base/0 and a/2. Note that the fact in the first line is implicitly in the base/0 part. Solving the program as is results in answer set {a, c}, because the base/0 part is instantiated by default. Scripts to instantiate the a/2 part as well are as follows:

To inspect the instantiation of program parts, invoke:
```
gringo --text \
part.lp part-lua.lp
```
or:
```
gringo --text \
part.lp part-py.lp
```
Calls to *clingo* are similar.

```
1  #script (lua)

3  add = table.insert

5  function main(prg)
6    p = {}
7    add(p, {"base",{}})
8    add(p, {"a",{1,3}})
9    prg:ground(p)
10   prg:solve()
11 end

13 #end.
```

```
1  #script (python)

5  def main(prg):
6    p = []
7    p.append(("base",[]))
8    p.append(("a",[1,3]))
9    prg.ground(p)
10   prg.solve()

13 #end.
```

In Line 9, the script grounds the base/0 part (Line 7) as well as the a/2 part with parameters 1 and 3 (Line 8). The call in Line 10 is essential to solve the program with *clingo*, and even in *gringo* some post-processing happens, e.g., printing the symbol table of the *smodels* format [80]. ■

Remark 3.14. Program parts are mainly interesting for incremental grounding and solving of logic programs detailed in Section 4. For single-shot solving, program parts are not needed. The feature is merely listed for completeness here. ■

Include Statements Include statements allow for including files from within another file. They have the form

```
#include "file".
```

where *file* is a path to another encoding file. When including a file it is first looked up relative to the current working directory. If it is not found there, then it is looked up relative to the file it was included from. Note that program part declarations do not affect the inclusion of files, that is, including a file is equivalent to passing it on the command line.

To inspect the instantiation, invoke:
```
clingo --text \
include.lp
```
or alternatively:
```
gringo --text \
include.lp
```

Example 3.25. Suppose that we have a file include.lp with the following statement:

```
1  #include "bird.lp".
```

We can simply pass the file on the command line to include file bird.lp from Example 3.2. Since files are included from the current working directory as well as relative to the file with the include statement, an invocation like ‘clingo examples/include.lp’ works with either of the following directory layouts:

```
.
|-- bird.lp
\-- examples
    \-- include.lp

.
\-- examples
    |-- bird.lp
    \-- include.lp
```

■

3.2 Input Language of *clasp*

Solver *clasp* [37] (or '*clingo* `--mode=clasp`') accepts logic programs in *aspif* format [62] and *smodels* format [80] (for backward compatibility), SAT and MaxSAT instances in DIMACS-cnf[16] and DIMACS-wcnf[17] format, and PB problems in OPB/WBO[18] format.

For ASP solving, *clasp* is typically invoked in a pipe reading a logic program output by *gringo* (or *clingo*):

```
gringo [ options | files ] | clasp [ options | number ]
clingo [ options | files | number ]
```

Note that `number` may be provided to specify a maximum number of answer sets to be computed, where `0` makes *clasp* compute all answer sets. This maximum number can also be set via option `--models` or its abbreviation `-n` (cf. Section 7.3). By default, *clasp* computes one (optimal) answer set (if it exists).

To solve a problem in one of the supported formats stored in a `file`, an invocation of *clasp* looks as follows:

```
clasp [ options | number ] file
clingo --mode=clasp [ options | number ] file
```

In general, *clasp* autodetects the input format. However, option `--opt-sat` is necessary to distinguish a MaxSAT instance in DIMACS-wcnf format from a plain SAT instance in DIMACS-cnf format.

[16] `http://www.satcompetition.org/2009/format-benchmarks2009.html`
[17] `http://www.maxsat.udl.cat/12/requirements/index.html`
[18] `http://www.cril.univ-artois.fr/PB12/format.pdf`

B Differences to the Language of *gringo* 3

This section is not yet ready for publishing and will be included in one of the forthcoming editions of this guide.

Information on differences between the languages of *gringo* 3 and 4 can be obtained here:

- `NOTES` in *gringo*/*clingo* distribution

Removed features

- `#hide` statements
- `#domain` statements
- `#compute` statements
- aggregates
  - multiset semantics
  - `#avg`
  - `#even`/`#odd`