Programming with CLINGO

Vladimir Lifschitz, University of Texas

Programmers usually solve computational problems by designing algorithms and encoding them in an implemented programming language. Research in artificial intelligence and computational logic has led to an alternative, "declarative" approach to programming, which does not involve encoding algorithms. A program in a declarative language only describes what is counted as a solution. Given such a description, a declarative programming system finds a solution by the process of automated reasoning.

This document explains how to use the declarative programming tool CLINGO, based on the programming method called answer set programming (ASP). This method is oriented towards combinatorial search problems, where the goal is to find a solution among a large, but finite, number of possibilities. The companion document, *Stable Models*, provides an introduction to the mathematical theory of answer set programming.

There are many exercises here, and you'll find answers to them at the end.

1 Introduction

Assume that we are given information on today's temperatures in several cities, for instance:

City	Austin	Dallas	Houston	San Antonio
Temperature	88°	95°	90°	85°

Table 1: Cities and temperatures.

We'll use ASP to find all cities where the temperature is higher than in Austin. Let's call such cities "warm."

ASP programs consist of *rules*. To find the warm cities, we need the rule

```
warm(C) :- t(C,T1), t(austin,T2), T1>T2.            (1)
```

It has two parts—the *head*

```
warm(C)
```

and the *body*

```
t(C,T1), t(austin,T2), T1>T2
```

—separated by the symbol `:-` that looks like the arrow $\leftarrow$ and reads "if." The end of a rule is indicated by a period. Capitalized identifiers in a rule (in this case, `C`, `T1`, and `T2`) are variables. Since `austin` is here the name of a specific object and not a variable, it is not capitalized. Both variables and names of specific objects can be collectively referred to as *terms*. (There are also other kinds of terms, as we will see in Section 2.) The symbol `warm` at the beginning of the head denotes the property of being a warm city; symbols like `warm`, which denote properties and relations, are called *predicate symbols*. The symbol `t` in the body is a predicate symbol also; it expresses a relation between a city and a temperature. Expressions consisting of a predicate symbol followed by a list of terms in parentheses, such as `warm(C)`, `t(C,T1)`, and `t(austin,T2)`, are called *atoms*. The expression `T1>T2` at the end of the body is a *comparison*. The relation symbols that can be used in comparisons are

```
=      !=      <      >      <=      >=
```
(2)

The head of a rule is usually an atom, and the body of a rule is often a list of atoms and comparisons, as in the example above.

The last four among the relation symbols (2) are usually applied to numbers, but CLINGO allows us to apply them to symbolic constants as well. It so happens that according to the total order used by CLINGO for such comparisons, the symbol `houston` is greater than the symbol `dallas`, and `dallas` is greater than `3`.

Rule (1) can be read as follows:

C is warm if
  the temperature in C is T_1,
  the temperature in Austin is T_2,
  and $T_1 > T_2$.

Note that this sentence is declarative—it does not describe an algorithm. It merely explains how we understand "warm."

Table 1 can be represented by the rules

```
t(austin,88). t(dallas,95). t(houston,90). t(san_antonio,85).
```
(3)

These rules don't have bodies, and accordingly there is no "if" symbol in them. They are also *ground*, that is, don't contain variables. Ground rules without bodies are called *facts*.

To instruct CLINGO to find all warm cities, we create a text file, say `warm.lp`, containing lines (1) and (3). (The names of files that consist of rules are often given the extension `.lp`, for "logic program.") If we execute the command

```
% clingo warm.lp
```

then CLINGO will produce a collection of ground atoms:

```
t(austin,88) t(dallas,95) t(houston,90) t(san_antonio,85)
warm(dallas) warm(houston)
```

and some additional information.

About this set of ground atoms we say that it is the *stable model* of the program `warm.lp`. The precise meaning of this concept is discussed in the companion document *Stable Models*, and in Section 5 of that document you will find a justification of the claim that the answer produced in this case by CLINGO is correct. But informally we can say that the stable model consists of the ground atoms that can be "derived" using the rules of the program. To see, for instance, why the atom `warm(dallas)` is included in the stable model of `warm.lp`, consider the instance

```
        warm(dallas) :- t(dallas,95), t(austin,88), 95>88.
```

of rule (1), which is obtained from it by substituting the values `dallas`, `95`, and `88` for the variables `C`, `T1`, and `T2`. Both atoms in the body of that instance are among the given facts (3), and the comparison `95>88` is true. Consequently this instance justifies including its head `warm(dallas)` in the stable model of `warm.lp`.

Exercise P.1.1. What instance of rule (1) justifies including `warm(houston)` in the stable model?

The last two atoms in the stable model generated by CLINGO answer our question: the warm cities are Dallas and Houston. We may wish to suppress the rest of the output, which merely reproduces the given facts. This can be accomplished by including the *directive*

```
#show warm/1.
```

in the file `warm.lp`. (When we refer to a predicate symbol used in an ASP program, it is customary to append its arity—the number of arguments—after a slash; in this case, the arity is 1.) This directive instructs CLINGO to show the atoms formed from the predicate symbol `warm` and one argument, instead of the whole stable model.

Exercise P.1.2. Instead of comparing with the temperature is Austin, let's define "warm" by the condition "the temperature is over 90°." Modify the file `warm.lp` accordingly.

In case of a syntax error, CLINGO usually produces a message specifying the place in the file where parsing stopped. Some error messages say that the program has "unsafe variables." Such a message usually indicates that the head of one of the rules includes a variable that does not occur in its body; stable models of such programs may be infinite.

Exercise P.1.3. Consider the rule

```
child(X,Y) :- parent(Y,X).                                    (4)
```

(a) How would you read this rule? (b) If we run CLINGO on the program consisting of rules (4) and

```
parent(ann,bob). parent(bob,carol). parent(bob,dan).          (5)
```

then what stable model do you think it will produce?

A group of facts that contain the same predicate symbol can be "pooled together" using semicolons. For instance, line (3) can be abbreviated as

```
t(austin,88; dallas,95; houston,90; san_antonio,85).
```

Exercise P.1.4. Use pooling to abbreviate line (5).

2 Arithmetic

The rule

```
p(N,N*N+N+41) :- N=0..10.                                     (6)
```

reads:

N and $N^2 + N + 41$ are in the relation p/2 if
N is one of the numbers $0, \dots, 10$.

The stable model of this one-rule program shows the values of the polynomial $N^2 + N + 41$ for all N from 0 to 10:

```
p(0,41) p(1,43) p(2,47) p(3,53) p(4,61) p(5,71) p(6,83) p(7,97)
p(8,113) p(9,131) p(10,151)
```

Rule (6) contains terms that are more complex than variables and constants: the polynomial `N*N+N+41` and the interval `0..10`.

Exercise P.2.1. For each of the given one-rule programs, predict what stable model CLINGO is going to produce.

(a) `p(N,N*N+N+41) :- N+1=1..11.`

(b) `p(N,N*N+N+41) :- N=-10..10, N>=0.`

The only numbers that CLINGO knows about are integers. The stable model of the one-rule program

```
p(M,N) :- N=1..4, N=2*M.
```

consists of only two atoms

```
  p(1,2) p(2,4)
```

because no integer can be multiplied by 2 to produce an odd number.

In addition to the symbols + and *, terms in a CLINGO program can include the symbols

```
**        /        \        | |
```

for exponentiation, integer division, remainder, and absolute value.

Exercise P.2.2. Write a one-rule program that has the stable model

```
  p(1,2) p(2,4) p(4,8) p(8,16)
```

Intervals may be used not only in the bodies of rules, as in the examples above, but in the heads as well. For instance, a program may include the fact

```
  p(1..3).
```

which has the same meaning as the set of three facts

```
  p(1). p(2). p(3).
```

The stable model of the one-rule program

```
  square(1..8,1..8).
```

consists of the 64 atoms corresponding to the squares of the 8×8 chessboard:

```
square(1,1)    ···    square(1,8)
. . . . . . . . . . . . . . . .
square(8,1)    ···    square(8,8)
```

Exercise P.2.3. Consider the program consisting of two facts:

```
  p(1..2,1..4).  p(1..4,1..2).
```

How many atoms do you expect to see in its stable model?

Exercise P.2.4. Write a one-rule program that has the stable model

```
p(1,1)
p(2,1)  p(2,2)
p(3,1)  p(3,2)  p(3,3)
p(4,1)  p(4,2)  p(4,3)  p(4,4)
```

Rule (6), which describes the table of values of the polynomial $N^2 + N + 41$ for all N from 0 to 10, can be made more general:

```
p(N,a*N*N+b*N+c) :- N=0..n.                    (7)
```

Rule (7) uses *placeholders* `a`, `b`, `c`, `n` to describe the table of values of an arbitrary quadratic polynomial $aN^2 + bN + c$ with integer coefficients for the values of N from 0 to an arbitrary upper bound n. The values of the placeholders can be specified on the command line:

```
  % clingo -c a=1 -c b=1 -c c=41 -c n=10 quadratic.lp
```

where `quadratic.lp` is the name of the file containing rule (7). Alternatively, their values can be represented by including the directives

```
#const a=1. #const b=1. #const c=41. #const n=10.
```

in the file `quadratic.lp`.

To give another example, we can make rule (1) more general using a placeholder:

```
  warm(C) :- t(C,T1), t(c,T2), T1>T2.
```

Exercise P.2.5. Write a directive specifying that `c` stands for `austin`.

3 Definitions

Many ASP rules can be thought of as definitions. We can say, for instance, that rule (1) defines the predicate `warm/2` in terms of the predicate `t/2`, rule (4) defines `child/2` in terms of `parent/2`, and rule (6) defines `p/2`.

Exercise P.3.1. (a) How would you define the predicate `grandparent/2` in terms of `parent/2`? (b) If you run CLINGO on your definition, combined with facts (5), what stable model do you think it will produce?

Exercise P.3.2. (a) How would you define the predicate `sibling/2` in terms of `parent/2`? (b) If you run CLINGO on your definition, combined with facts (5), what stable model do you think it will produce?

Sometimes the definition of a predicate consists of several rules. For instance, the pair of rules

```
  parent(X,Y) :- father(X,Y).
  parent(X,Y) :- mother(X,Y).
```

defines `parent/2` in terms of `father/2` and `mother/2`.

A predicate can be defined recusively. In a recursive definition, the defined predicate occurs not only in the heads of the rules but also in some of the bodies. The definition of `ancestor/2` in terms of `parent/2` is an example:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- ancestor(X,Y), ancestor(Y,Z).
```
(8)

Exercise P.3.3. If we run CLINGO on the program consisting of rules (8) and (5), what stable model do you expect it to produce?

Sometimes a predicate can't be defined in one step, and some auxiliary predicates have to be introduced first. Consider, for instance, the property of being a prime number from the set $\{1, \ldots, n\}$. It's easier to define the opposite property of being a composite number from that set:

```
composite(N) :- N=1..n, I=2..N-1, N\I=0.
```
(9)

(N is composite if it is a multiple of a number I from the set $\{2, \ldots, N-1\}$.) Then `prime/1` can be defined in terms of `composite/1` by the rule

```
prime(N) :- N=2..n, not composite(N).
```
(10)

(N is prime if it is different from 1 and not composite.)

Rule (10) is our first example of the use of negation in a CLINGO program. In what sense can this rule be used to "derive" the atom `prime(7)` (assuming, for example, that $n = 10$)? About the instance

```
prime(7) :- 7=2..10, not composite(7).
```

of that rule we can say that the expression `not composite(7)` in its body is justified in the sense that any attempt to use rule (9) to "derive" `composite(7)` will fail. The idea of negation as failure plays an important part in answer set programming, and it is discussed in Sections 6 and 7 of *Stable Models*. To emphasize this understanding of negation, we can read rule (10) as follows:

N is prime if
  it is one of the numbers $2, \ldots, n$
  and there is no evidence that it is composite.

Listing 1 shows a program consisting of rules (9) and (10), a `#show` directive, and a few comments. The comment in Line 3 tells us that `n` is a placeholder for a positive integer; the value of `n` must be specified when the program is executed. The comment in Lines 6 and 7 tells us which instances of the atom `composite(N)` we expect to see in the stable model of the one-line program (9). (These instances

Listing 1: Prime numbers

```
 1  % Prime numbers from 1 to n
 2
 3  % input: positive integer n.
 4
 5  composite(N) :- N=1..n, I=2..N-1, N\I=0.
 6  % achieved: composite(N) iff N is a composite number
 7  %           from {1,...,n}.
 8
 9  prime(N) :- N=2..n, not composite(N).
10  % achieved: prime(N) iff N is a prime number
11  %           from {1,...,n}.
12
13  #show prime/1.
```

are the same, of course, as in the stable model of the program that includes both rules.) Comments explaining what has been "achieved" by a group of rules at the beginning of a program allow us to start debugging at an early stage, when only a part of the program has been written. For example, after writing Lines 1–7 of the program in Listing 1 we may wish to check whether the stable model produced by CLINGO for the first rule with $n = 10$ is indeed

```
composite(4) composite(6) composite(8) composite(9) composite(10)
```

Listing 2 gives another example of a pair of definitions, one on top of the other. Before defining the property `fac/1` of being a factorial, we give a recursive definition of the binary relation `fac/2`, "the factorial of N is F." The underscore in Line 9 is an "anonymous variable"; underscores can be used instead of identifiers for the variables that occur in the rule only once.

The comment in Line 7 of Listing 2 says, "`fac/2` is $\{(0, 0!), \dots, (n, n!)\}$" instead of "the set of pairs of numbers that are in the relation `fac/2` is $\{(0, 0!), \dots, (n, n!)\}$." In mathematics, it is customary to identify a binary relation with the corresponding set of ordered pairs. The meaning of Line 10 is similar.

There is no "achieved" comment after Line 5. Nothing of interest is achieved in the middle of a definition.

Exercise P.3.4. Consider the part of the program shown in Listing 2 that precedes the comment in Line 7. What atoms do you expect to see in its stable model if $n = 4$?

Listing 2: Factorials

```
% Beginning of the sequence of factorials

% input: nonnegative integer n.

fac(0,1).
fac(N+1,F*(N+1)) :- fac(N,F), N<n.
% achieved: fac/2 = {(0,0!),...,(n,n!)}

fac(F) :- fac(_,F).
% achieved: fac/1 = {0!,...,n!}

#show fac/1.
```

4 Choice Rules and Constraints

Each of the ASP programs discussed above has a single stable model, but in answer set programming we more often deal with programs that have many stable models. Some programs have no stable models. This is as usual as equations with many roots, or no roots, in algebra.

In CLINGO programs with several stable models we often see *choice rules*, which describe several alternative ways to form a stable model. The head of a choice rule includes an expression in braces, for instance:

`{p(a);q(b)}.` (11)

This choice rule says: choose which of the atoms `p(a)`, `q(b)` to include in the model. There are 4 possible combinations, so that one-rule program (11) has 4 stable models. The number of stable models that we would like CLINGO to display can be specified on the command line; 1 is the default, and 0 means “find all.” For instance, if rule (11) is saved in the file `choice.lp` then the command

```
  % clingo choice.lp 0
```

will produce a list of 4 stable models:

```
Answer: 1

Answer: 2
q(b)
Answer: 3
p(a)
```

```
Answer: 4
p(a) q(b)
```

Choice rules may include intervals and pools. For instance, the rule

```
{p(1..3)}.
```

has the same meaning as

```
{p(1);p(2);p(3)}.
```

The rule

```
{p(a;b;c)}.
```

has the same meaning as

```
{p(a);p(b);p(c)}.
```

Each of these one-rule programs has 8 stable models.

Exercise P.4.1. For each of the given programs, what do you think is the number of its stable models?

(a) `{p(1..3)}.  {q(1..3)}.`

(b) `{p(1..3,1..3)}.`

Before and after an expression in braces we can put integers, which express bounds on the cardinality (number of elements) of the stable models described by the rule. The number on the left is the lower bound, and the number on the right is the upper bound. For instance, the one rule program

```
1 {p(1..3)} 2.                    (12)
```

describes the subsets of $\{1, 2, 3\}$ that consist of 1 or 2 elements:

```
Answer: 1
p(2)
Answer: 2
p(3)
Answer: 3
p(2) p(3)
Answer: 4
p(1)
Answer: 5
p(1) p(3)
Answer: 6
p(1) p(2)
```

Exercise P.4.2. For each of the given programs, what do you think is the number of its stable models?

(a) `1 {p(1..6)}.`

(b) `3 {p(1..6)} 3.`

Exercise P.4.3. For each of the given rules, find a simpler rule that has the same meaning.

(a) `0 {p(a)}.`

(b) `1 {p(a)}.`

(c) `{p(a)} 1.`

Choice rules may also include variables. Consider, for instance, the one-rule program

```
1 {p(X);q(X)} 1 :- X=1..n.
```

where `n` is a placeholder for a nonnegative integer. Each of its stable models includes one of the atoms `p(1)`, `q(1)`, one of the atoms `p(2)`, `q(2)`, and so on. The program has 2^n stable models; each of them describes a partition of the set $\{1, ..., n\}$ into subsets `p/1`, `q/1` (possibly empty). For $n = 2$ CLINGO produces 4 stable models:

```
Answer: 1
q(1) p(2)
Answer: 2
q(1) q(2)
Answer: 3
p(1) p(2)
Answer: 4
p(1) q(2)
```

The rule

```
1 {p(X,1..2)} 1 :- X=1..n.
```

is similar: each of its 2^n stable models includes one of the atoms `p(1,1)`, `p(1,2)`, one of the atoms `p(2,1)`, `p(2,2)`, and so on.

ASP programs containing choice rules often contain also *constraints*—rules that weed out "undesirable" stable models, for which the constraint is "violated." A constraint is a rule that has no head, for instance

```
:- p(1).
```
(13)

By adding this constraint to a program, we eliminate its stable models that contain `p(1)`. We have seen, for example, that program (12) has 6 stable models. Adding rule (13) to it eliminates the last three of them—those that contain `p(1)`. Adding the "opposite" constraint

```
:- not p(1).
```

to (12) eliminates the first three solutions—those that don't contain `p(1)`. Adding both constraints to (12) will give a program that has no stable models. Combining choice rule (12) with the constraint

```
:- p(1), p(2).
```

will eliminate the only stable model among the 6 that includes both `p(1)` and `p(2)`—the last one.

Exercise P.4.4. Consider the program consisting of choice rule (12) and the constraint

```
:- p(1), not p(2).
```

How many stable models do you think it has?

Cardinality bounds in a choice rule can be sometimes replaced by constraints. For instance, the rule

```
{p(a);q(b)} 1.
```

has the same meaning as the pair of rules

```
{p(a);q(b)}.
:- p(a), q(b).
```

Exercise P.4.5. Find a similar transformation for the rule

```
1 {p(a);q(b)}.
```

5 Combinatorial Search

In a combinatorial search problem, the goal is to find a solution among a finite number of candidates. The ASP approach is to represent such a problem by a program whose stable models correspond to solutions, and then use a solver, such as CLINGO, to find a stable model. ASP encodings of search problems usually combine rules of two types discussed above—choice rules and constraints, and sometimes they also include definitions.

In this section, we give a few examples illustrating this approach to search.

Listing 3: Schur numbers

```
% Partition {1,..,n} into r sum-free sets

% input: positive integers n, r.

1 {in(I,1..r)} 1 :- I=1..n.
% achieved: set {1,...,n} is partitioned into
%           subsets {I:in(I,1)},...,{I:in(I,r)}.

:- in(I,S), in(J,S), in(I+J,S).
% achieved: these subsets are sum-free.
```

5.1 Schur Numbers

A set S of numbers is called *sum-free* if the sum of two elements of S never belongs to S. For instance, the set $\{5,\dots,9\}$ is sum-free; the set $\{4,\dots,9\}$ is not ($4+4=8$, $4+5=9$).

Can we partition the set $\{1,\dots,n\}$ into 2 sum-free subsets? This is possible if $n=4$: both $\{1,4\}$ and $\{2,3\}$ are sum-free. But if $n=5$ then such a partition doesn't exist.

Exercise P.5.1. Prove this claim.

What about partitioning $\{1,\dots,n\}$ into 3 sum-free subsets? This can be done for values of n that are much larger than 4. For instance, if $n=9$ then we can take the subsets $\{1,4\}$, $\{2,3\}$, and $\{5,\dots,9\}$.

Exercise P.5.2. Partition $\{1,\dots,10\}$ into 3 sum-free subsets.

The CLINGO program shown in Listing 3 solves the general problem of partitioning $\{1,\dots,n\}$ into r sum-free subsets whenever this is possible. The largest value of n for which such a partition exists is denoted by $S(r)$. For instance, $S(2)=4$, and the assertion of Exercise P.5.2 shows that $S(3)\geq 10$. The numbers $S(r)$ are called *Schur numbers*. So we can say that by running this program we estimate the Schur numbers.

The program represents partitions using the predicate `in/2`: `in(I,1)` means that `I` belongs to the first subset, `in(I,2)` means that `I` belongs to the second subset, and so on. For example, the stable model

```
in(1,1) in(2,2) in(3,2) in(4,1)
```

for $r=2$ and $n=4$ represents the partition of $\{1,\dots,4\}$ into $\{1,4\}$ and $\{2,3\}$.

Exercise P.5.3. How many models do you think this program has for $r=2$ and $n=4$?

The program consists of two rules: the choice rule in Line 5 and the constraint in Line 9. Stable models of the choice rule correspond to arbitrary partitions of $\{1, \dots, n\}$ into r subsets, possibly empty. The constraint eliminates the partitions in which some subsets are not sum-free. This division of labor between choice rules and constraints is typical for ASP solutions to search problems: the collection of stable models of the choice rule includes not only solutions to the problem but also some "bad" candidates, which are eliminated by the constraints.

Exercise P.5.4. What is the number of stable models of the rule in Line 5?

If we run the program in Listing 3 for $r = 3$ and various values of n then we will see that the largest n for which stable models exist is 13. In other words, $S(3) = 13$. In a similar way we can determine that $S(4) = 44$. The runtime of CLINGO for $r = 4$ and $n = 45$ may be large, and there is a useful trick that will speed up the process. CLINGO can use several search strategies, and we can instruct it to try several strategies in parallel. For example, if we save the program in the file `schur.lp` and issue the command

```
clingo -c r=4 -c n=45 schur.lp -t4
```

then CLINGO will try solving the problem with 4 "threads," and the output may look like this:

```
clingo version 5.1.0
Reading from schur.lp
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 35.813s
CPU Time     : 143.070s
Threads      : 4        (Winner: 2)
```

Information on the "winner" tells us which of the threads 0, 1, 2, 3 reached the goal first.

Exercise P.5.5. Use CLINGO to verify that $S(5) \geq 130$.

The exact value of $S(5)$ is not known.

5.2 Seating Arrangements

We are making arrangements for a big sit-down party. There will be n tables in the room, with m chairs around each table. The guests are numbered from 1 to mn, and we'd like to choose a table for each of them so that two conditions are

Listing 4: Seating arrangements

```
% There are n tables in the room, with m chairs around
% each table.  Choose a table for each of m*n guests
% so that guests who like each other sit at the same
% table, and guests who don't like each other sit at
% different tables.

% input: positive integers m and n; set like/2 of pairs
%        of guests who like each other; set dislike/2 of
%        pairs of guests who dislike each other.

m {at(1..m*n,T)} m :- T=1..n.
% achieved: for each table T there are exactly m
%           guests G such that at(G,T).

T1=T2 :- at(G,T1), at(G,T2).
% achieved: no guest is assigned two different tables.

T1=T2 :- at(G1,T1), at(G2,T2), like(G1,G2).
% achieved: guests who like each other sit at the same
%           table.

:- at(G1,T), at(G2,T), dislike(G1,G2).
% achieved: guests who don't like each other don't sit
%           at the same table.

#show at/2.
```

satisfied. First, some guests like each other and want to sit together; accordingly, we are given a set A of two-element subsets of $\{1,\dots,mn\}$, and, for every $\{i,j\}$ in A, guests i and j should sit at the same table. Second, some guests dislike each other and want to sit at different tables; accordingly, we are given a set B of two-element subsets of $\{1,\dots,mn\}$, and, for every $\{i,j\}$ in B, guests i and j should sit at different tables.

Listing 4 shows a CLINGO program that finds an assignment of tables to guests satisfiying these conditions, if it exists. The input expected by this program includes not only the values of placeholders but also two predicates, and an input of this kind can't be specified on a command line; a separate file is required, such as the one shown in Listing 5.

The command line will include the names of two files (in any order)—one

Listing 5: Sample input for the seating arrangements program

```
#const n=3.
#const m=4.

like(1,2; 2,3; 4,5; 5,6; 7,8; 8,9).
dislike(1,10; 1,11; 4,11; 4,12).
```

containing the program, the other containing the input.

The choice rule in Line 11 of Listing 4 selects a group of m guests for each table, and the role of the rule in Line 15 is to ensure that these groups are pairwise disjoint. The head of this rule is not an atom, as in definitions, and not an atom in braces, as in choice rules; it is a comparison. Rules with a comparison in the head are constraints in disguise. In particular, the rule in Line 15 has the same meaning as the constraint

```
:- at(G,T1), at(G,T2), T1!=T2.
```

(discard a stable model if a guest `G` is included in two different groups).

The rule in Line 18 is a constraint in disguise as well.

Exercise P.5.6. If $n = 2$ then what is the number of stable models

(a) of the first rule of the seating arrangements program?

(b) of the first two rules?

5.3 Independent Sets

In ASP, graphs are often described by two predicates, `vertex/1` and `edge/2`, as in Listing 6.

A set S of vertices in a graph is *independent* if no two vertices from S are adjacent. For instance, $\{B, D, E, G\}$ is an independent set in the graph from Listing 6.

The program shown in Listing 7 describes independent sets of vertices of a given size. The choice rule in Line 6 says: include exactly n vertices in the set `in/1`. The atoms that can be chosen for a stable model are characterized in this rule indirectly, in terms of the condition `vertex(X)`. If, for example, the predicate `vertex/1` is defined as in Listing 6 then the expression `in(X) : vertex(X)` is understood as

```
in(a);in(b);in(c);in(d);in(e);in(f);in(g);in(h)
```

Listing 6: Graph with 8 vertices and 13 edges

```
%     A ----- B
%     |\     / |
%     | E--F   |
%     | | /|   |
%     | |/ |   |
%     | H--G   |
%     |/     \ |
%     D-------C

vertex(a;b;c;d;e;f;g;h).

edge(a,b; b,c; c,d; d,a;
     e,f; f,g; g,h; h,e;
     a,e; b,f; c,g; d,h; f,h).
```

Listing 7: Independent sets of a given size

```
% Find independent sets of vertices of size n

% input: set vertex/1 of vertices of a graph G;
%        set edge/2 of edges of G; positive integer n.

n {in(X) : vertex(X)} n.
% achieved: in/1 is a set consisting of n vertices of G.

adjacent(X,Y) :- edge(X,Y).
adjacent(X,Y) :- edge(Y,X).
% achieved: adjacent(X,Y) iff X, Y are adjacent vertices
%           of G.

:- in(X), in(Y), adjacent(X,Y).
% achieved: in/1 has no pairs of adjacent vertices.

#show in/1.
```

The use of the variable `X` in this choice rule is different from the uses of variables that we saw earlier, in the sense that substituting new values for `X` does not produce new instances of the rule. In fact, this choice rule is its only instance. Variables like this are said to be *local*. All occurrences of a local variable in a rule are in a part of the rule that is restricted by braces: `{ . . . }`. Variables that are not local are called *global*.

The rules in Lines 9 and 10 define the auxiliary predicate `adjacent/2`, which is used in the constraint in Line 14.

Exercise P.5.7. A set S of vertices in a graph is called a *clique* if every two distinct vertices in it are adjacent. How will you modify the program in Listing 7 if your goal is to describe cliques of size n, instead of independent sets?

5.4 Number Snake

In the logic puzzle known as Hidato, or Number Snake, you are given a grid partially filled with numbers, for instance:

6		
	2	8
1		

Table 2: A Number Snake puzzle

The goal is to fill the grid so that consecutive numbers connect horizontally, vertically, or diagonally.

Exercise P.5.8. Solve the puzzle in Table 2.

The program shown in Listing 8 solves the Number Snake puzzles in which the grid is a square of size n, and the numbers go from 1 to n^2. The inequality in Line 16 expresses that the distance between the points `(R1,C1)` and `(R2,C2)` is less than or equal to $\sqrt{2}$; this is equivalent to saying that the square in row `R1`, column `C1` and the square in row `R2`, column `C2` connect horizontally, vertically, or diagonally.

Exercise P.5.9. Assuming that the input is

```
#const n=2.
given(1,1,1).
```

how many stable models you expect to see for (a) the first rule in Listing 8? (b) the first 3 rules? (c) the first 4 rules? (d) the full program?

Listing 8: Number Snake

```
% Number Snake puzzle

% input: positive integer n; set given/3 of triples
%        R,C,X such that the given n-by-n grid has
%        number X in row R, column C.

1 {filled(R,C,1..n*n)} 1 :- R=1..n, C=1..n.
% achieved: every square of the grid is filled with
%           a number between 1 and n^2.

R1=R2 :- filled(R1,_,X), filled(R2,_,X).
C1=C2 :- filled(_,C1,X), filled(_,C2,X).
% achieved: different squares are filled with different
%           numbers.

(R1-R2)**2+(C1-C2)**2<=2 :- filled(R1,C1,X),
                            filled(R2,C2,X+1).
% achieved: consecutive numbers connect horizontally,
%           vertically, or diagonally.

:- given(R,C,X), not filled(R,C,X).
% achieved: given/3 is a subset of filled/3.

#show filled/3.
```

Listing 9: Degrees of vertices and the number of edges

```
 1 % Number of edges and degrees of vertices
 2
 3 % input: set vertex/1 of vertices of a graph G; set
 4 %        edge/2 of edges of G.
 5
 6 adjacent(X,Y) :- edge(X,Y).
 7 adjacent(X,Y) :- edge(Y,X).
 8 % achieved: adjacent(X,Y) iff X, Y are adjacent vertices
 9 %           of G.
10
11 number_of_edges(N/2) :- N = #count{X,Y : adjacent(X,Y)}.
12 % achieved: number_of_edges(N) iff N is the number of
13 %           edges of G.
14
15 degree(X,D) :- vertex(X), D = #count{Y: adjacent(X,Y)}.
16 % achieved: degree(X,D) iff D is the degree of the
17 %           vertex X of G.
18
19 #show degree/2.  #show number_of_edges/1.
```

6 Aggregates

An *aggregate* is a function that can be applied to sets. The language of CLINGO has symbols for several aggregates, including `#count` for the number of elements of a set, and `#sum` for the sum of a set of numbers.

Listing 9 shows how `#count` can be used to calculate the number of edges in a graph, and also the degree of every vertex (the number of adjacent vertices). Given this program and the input from Listing 6, CLINGO will respond:

```
number_of_edges(13) degree(a,3) degree(b,3) degree(c,3)
degree(d,3) degree(e,3) degree(f,4) degree(g,3) degree(h,4)
```

In Line 11, variable `N` is global, and `X`, `Y` are local (see Section 5.3). Division by 2 is needed because the number of pairs `X,Y` of adjacent vertices is greater than the number of edges by a factor of 2: either endpoint can be chosen as `X`. Including `number_of_edges(13)` in the stable model is justified by the instance

```
number_of_edges(26/2) :- 26 = #count{X,Y : adjacent(X,Y)}.
```

of that rule. In Line 15, on the other hand, `Y` is the only local variable: `X` occurs not only between the braces, but also outside. Including `degree(a,3)` in the stable model is justified by the instance

```
degree(a,3) :- vertex(a), 3 = #count{Y: adjacent(a,Y)}.
```

of that rule.

Aggregates can be used in inequalities. For instance, the set of vertices of degree greater than 3 can be defined by the rule

```
many_neighbors(X) :- vertex(X), #count{Y: adjacent(X,Y)} > 3.
```

Exercise P.6.1. Consider the program

```
p(a,1).  p(b,1).  p(c,2).
q(N) :- N = #count{A,X : p(A,X)}.
r(N) :- N = #count{A : p(A,X)}.
s(N) :- N = #count{X : p(A,X)}.
```

What is its stable model, in your opinion?

The aggregate symbol `#count` can be used to rewrite the constraint in Line 15 of Listing 4 as follows:

```
:- #count{T : at(G,T)} > 1, G=1..m*n.
```

(The comparison at the end is needed to make the variable `G` global.)

To give another example of the use of `#count`, consider the rule

```
n {in(X) : vertex(X)} n.
```

from Listing 7. It can be equivalently replaced by a choice rule that has neither cardinality bounds nor local variables, combined with a constraint:

```
{in(X)} :- vertex(X).
:- #count{X : in(X), vertex(X)} != n.
```

Exercise P.6.2. Find a similar transformation for the rule

```
5 {p(X) : q(X)} 7.
```

The program in Listing 10 describes magic squares of size n. (A magic square is an $n \times n$ square grid filled with distinct integers in the range $1, \ldots, n^2$ so that the sum of numbers in each row, each column, and each of the two diagonals equals the same "magic constant.") The first 3 rules are similar to the rules that we saw in Listing 8, and the constraints that follow use the aggregate symbol `#sum`.

Exercise P.6.3. (a) In Line 17 of Listing 10, which variables are local, and which are global? Which instance of that rule expresses that Row 1 sums up to the magic constant?

If the expression between `#sum{` and the colon is a tuple, rather than a single variable, then summation is applied to the values of the first member of the tuple. For instance, the stable model of the program

Listing 10: Magic squares

```
% Magic squares of size n

% input: positive integer n.

1 {num(R,C,1..n*n)} 1 :- R=1..n, C=1..n.
% achieved: every square of the grid is filled with
%           a number between 1 and n^2.

R1=R2 :- num(R1,_,X), num(R2,_,X).
C1=C2 :- num(_,C1,X), num(_,C2,X).
% achieved: different squares are filled with different
%           numbers.

% Magic constant: (1+2+...+n^2)/n.
#const magic=(n**3+n)/2.

:- #sum{X : num(R,_,X)} != magic, R=1..n.
% achieved: every row sums up to magic.

:- #sum{X : num(_,C,X)} != magic, C=1..n.
% achieved: every column sums up to magic.

:- #sum{X : num(R,R,X)} != magic.
:- #sum{X : num(R,n+1-R,X)} != magic.
% achieved: both diagonals sum up to magic.
```

```
  p(1,10; 2,20).
  q(S) :- S = #sum{X,Y : p(X,Y)}.
  r(S) :- S = #sum{Y,X : p(X,Y)}.
```

includes `q(3)`, because $1 + 2 = 3$, and `r(30)`, because $10 + 20 = 30$.

Exercise P.6.4. What is the stable model of the program

```
p(S) :- S = #sum{N*N, N : N=-2..2}.
q(S) :- S = #sum{N*N : N=-2..2}.
```

in your opinion?

Exercise P.6.5. You wrote a test that includes several questions, and the atom `score(Q,P)` expresses that you received `P` points for your answer to question `Q`. Define `total(S)` ("`S` is the total score") in terms of `score/2`.

Exercise P.6.6. Find an expression that is equivalent to `#sum{1,X : p(X)}` but is simpler.

7 Optimization

The directives `#maximize` and `#minimize` instruct CLINGO to look for the "best" stable model of the given program. The numeric parameter that is maximized or minimized is the value of the corresponding `#sum` aggregate.

Consider, for instance, the modification of the independent set program (Listing 7) shown in Listing 11. It uses `#maximize` to find the largest independent set of vertices in a graph. The parameter that is maximized in this example, `#sum{1,X : in(X)}`, is the cardinality of `in/1` (see Exercise P.6.6). The output produced by CLINGO for this program and the graph from Listing 6 shows the sequence of larger and larger independent sets found by CLINGO until the largest is found:

```
  Answer: 1

  Optimization: 0
  Answer: 2
  in(d)
  Optimization: -1
  Answer: 3
  in(b) in(d)
  Optimization: -2
  Answer: 4
  in(b) in(d) in(e)
  Optimization: -3
```

Listing 11: Maximal independent sets

```
 1 % Find largest independent sets of vertices
 2 
 3 % input: set vertex/1 of vertices of a graph G;
 4 %        set edge/2 of edges of G
 5 
 6 {in(X)} :- vertex(X).
 7 % achieved: in/1 is a set consisting of n vertices of G.
 8 
 9 adjacent(X,Y) :- edge(X,Y).
10 adjacent(X,Y) :- edge(Y,X).
11 % achieved: adjacent(X,Y) iff X, Y are adjacent vertices
12 %           of G.
13 
14 :- in(X), in(Y), adjacent(X,Y).
15 % achieved: in/1 has no pairs of adjacent vertices.
16 
17 #maximize{1,X : in(X)}.
18 % achieved: the number of elements of in/1 is maximal
19 
20 #show in/1.
```

```
Answer: 5
in(b) in(d) in(e) in(g)
Optimization: -4
OPTIMUM FOUND
```

In the second example of the use of `#maximize` (Listing 12), CLINGO is used to encode the problem of finding, for a given set of positive integers, a subset for which the sum of its elements doesn't exceed a given upper bound but is as large as possible, given that restriction. For instance, if the given set of numbers is

$$\{1, 3, 5, 10, 20\}$$

and the upper bound is 22, the solution is given by the subset $\{1, 20\}$.

Exercise P.7.1. If we replace Line 15 in this program by

```
#maximize{1,X : in(X)}.
```

then how do you think the optimum found by CLINGO will change?

Listing 12: Subset sum

```
% Among the subsets of a given set of numbers for which
% the sum doesn't exceed the given upper bound find the
% one for which this sum is maximal.

% input: a set number/1 of positive integers; a positive
%        integer m.

{in(X)} :- number(X).
% achieved: in/1 is a subset of number/1.

:- #sum{X : in(X)} > m.
% achieved: the sum of the elements of in/1 doesn't
%           exceed m.

#maximize{X : in(X)}.
% achieved: the sum is maximal.

#show in/1.
```

Answers to Exercises

In some exercises, you are asked about the output that you expect CLINGO to produce for a given program. Answers to these exercises are not shown here, check your answer by running CLINGO.

P.1.1. `warm(houston) :- t(houston,90), t(austin,88), 90>88.`

P.1.2. Replace rule (1) in `warm.lp` by `warm(C) :- t(C,T), T>90.`

P.1.3. (a) X is a child of Y if Y is a parent of X.

P.1.4. `parent(ann,bob; bob,carol; bob,dan).`

P.2.2. `p(2**N,2**(N+1)) :- N=0..3.`

P.2.4. `p(M,N) :- M=1..4, N=1..4, M>=N.`

P.2.5. `#const c=austin.`

P.3.1. (a) `grandparent(X,Z) :- parent(X,Y), parent(Y,Z).`

P.3.2. (a) `sibling(X,Y) :- parent(Z,X), parent(Z,Y), X!=Y.`

P.3.4. `fac(0,1) fac(1,1) fac(2,2) fac(3,6) fac(4,24)`

P.4.3. (a) `{p(a)}.` (b) `p(a).` (c) `{p(a)}.`

P.4.5.

```
{p(a);q(b)}.
:- not p(a), not q(b).
```

P.5.1. Numbers 1 and 2 can't be included in the same set, because $1 + 1 = 2$. Denote the set containing 1 by A, and the set containing 2 by B. Number 4 can't belong to B, because $2 + 2 = 4$. It follows that 4 belongs to A. Then 3 can't belong to A, because $1 + 3 = 4$. It follows that 3 belongs to B. But then 5 can belong neither to A $(1 + 4 = 5)$ nor to B $(2 + 3 = 5)$; contradiction.

P.5.2. $\{1, 3, 8\}$, $\{2, 9, 10\}$, $\{4, 5, 6, 7\}$.

P.5.4. There are r ways to decide which subset will include 1, r ways to decide which subset will include 2, and so on. The total number of possibilities is r^n.

P.5.6. (a) $\binom{2m}{m}^2$, because there are $\binom{2m}{m}$ ways to choose each of the 2 groups. (b) $\binom{2m}{m}$, because the composition of the first group uniquely determines the composition of the second.

P.5.7. Replace the rules in Lines 9, 10, 14 by the rule

```
:- in(X), in(Y), X!=Y, not edge(X,Y), not edge(Y,X).
```

P.5.8.

6	7	9
5	2	8
1	4	3

P.6.2.

```
{p(X)} :- q(X).
:- #count{X : p(X), q(X)} < 5.
:- #count{X : p(X), q(X)} > 7.
```

P.6.3. (a) `X` and the anonymous variable are local; `R` is global.
(b) `:- #sum{X : num(1,_,X)} != magic, 1=1..n.`

P.6.5. `total(S) :- S = #sum{P,Q : score(Q,P)}.`

P.6.6. `#count{X : p(X)}.`