Chapter 1

Introduction

1.1 Declarative Programming

When a programmer solves a computational problem, this is usually accomplished by finding or designing an algorithm and encoding it in an implemented programming language. This book is about 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. A program in a declarative language is an encoding of the problem itself, not of an algorithm.

The difference between traditional "imperative" programming and declarative programming is similar to the difference between imperative and declarative sentences in natural language. An imperative sentence is a command that can be obeyed or disobeyed: "Go to school." A declarative sentence, on the other hand, is a statement that can be true or false: "I am at school." A program in an imperative language is formed from commands: "multiply n by 2." When declarative constructs, such as and inequalities, are used in an imperative program, they always occur within a command: "multiply n by 2 until $n > m$." They are used also in program specifications.

A program in a declarative language, on the other hand, consists of conditions on the values of variables that characterize solutions to the problem. Assignments are out of place in a declarative program. Such a program can be thought of as an "executable specification." Declarative solutions to computational problems are sometimes surprisingly short, in comparison with imperative solutions. Compare, for instance, two programs solving the eight queens puzzle—the declarative program on page 53 and the imperative program on page 59.

Declarative programming is closely related to artificial intelligence. AI research is concerned with teaching computers to perform intellectually challenging tasks, such as recognizing visual images, natural language understanding, or commonsense reasoning. Turning specifications into algorithms is yet another task of this kind, and that is what declarative programming systems do for us. Work on declarative programming is related, in particular, to the theory of knowledge representation—the subarea of AI dedicated to representing knowledge in a form that computers can use.

Declarative programming languages come in several flavors. One flavor is *functional*, which includes languages such as Lisp and Haskell. A Haskell program is formed from equations, for instance:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

These two equations express properties of the factorial function that can be used to calculate its values.

This book is about declarative programming of another kind—*logic programming*.

1.2 Logic Programming

A logic program consists of *rules*, which are similar to formulas used in mathematical logic.

As an example, consider the following problem. We are given a table showing the population sizes of several countries, such as Table 1.1. The goal is to make the list of all

Country	France	Germany	Italy	United Kingdom
Population (mln)	65	83	61	64

Table 1.1: Population of European countries in 2015.

countries inhabited by more people than the United Kingdom. Let us call such countries "large." This list can be generated by the logic program consisting of a single rule:

```
large(C) :- size(C,S1), size(uk,S2), S1 > S2.        (1.1)
```

This rule has two parts—the *head*

```
large(C)
```

and the *body*

```
size(C,S1), size(uk,S2), S1 > S2
```

—separated by the "colon-dash" symbol, which looks a little 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`, `S1`, and `S2`) are variables. Since `uk` is the name of a specific object and not a variable, it is not capitalized. The symbol `size` in the body expresses the binary relation that holds between a country and its population size. Thus rule (1.1) can be translated into English as follows:

A country C is large
    if the population size of C is S_1, the population size of the UK is S_2, and $S_1 > S_2$.

This is not a command; it is a declarative sentence explaining how we understand "large country." Turning this sentence into rule (1.1) can be thought of as an exercise in knowledge representation.

The head and body of a rule are similar to the consequent and antecedent of an implication. But their order in a rule is different from what is common in logic: instead of

$$\text{if} \cdots \text{ then } C \text{ is large}$$

we say

$$C \text{ is large if} \cdots.$$

Commas separating the expressions in the body of a rule are similar to the conjunction symbol $\wedge$ ("and") in logical formulas.

To generate the list of large countries using rule (1.1), we encode the input—Table 1.1—as a collection of additional rules:

```
size(france,65).  size(germany,83).  size(italy,61).  size(uk,64).
```
(1.2)

A system implementing the logic programming language Prolog will load a file consisting of rules (1.1) and (1.2), in any order, and display the prompt `?-` that invites the user to submit "queries"—questions that can be answered on the basis of the given information. The query `large(C)` would be understood as the request to find a value of `C` that has the property `large`, and the system would respond:

```
C = france.
```

If the user requests another value of `C` with this property, the answer will be

```
C = germany.
```

To a request for a third solution the system will reply `no` (no more large countries).

To write or understand Prolog programs, one has to think not only about their meaning as specifications, but also about Prolog's search strategy. In that sense, Prolog is not fully declarative; it has an "operational semantics." Answer set programming (ASP)—the form of logic programming described in this book—is closer to the ideal of declarativism. ASP became possible after the invention of the concept of a stable model and the creation of software systems that generate stable models. These systems are called *answer set solvers*, and their operation is described in many articles, dissertations, and books. But if your goal is to *use* answer set solvers, rather than design a new system of this kind, then you do not need to know much about how they operate. This book, in fact, will tell you almost nothing about what happens "under the hood." (Section 3.4 and a passage in Section 2.8 are the only places where the operation of an answer set solver is discussed in any detail.) Answer set programming has no operational semantics.

1.3 Answer Set Solvers

An answer set solver can load program (1.1), (1.2) and return an answer without waiting for a query. The answer, the "stable model" of the program, consists of all facts that can be derived using the rules of the program:

```
size(france,65) size(germany,83) size(italy,61) size(uk,64)
large(france) large(germany)
```
(1.3)

The design of answer set solvers is based on computational methods somewhat similar to those employed by *satisfiability solvers*—systems that find a truth assignment satisfying a given set of propositional formulas. We will talk later about the relationship between solvers of these two kinds, and we will see that in some cases they can simulate each other. Satisfiability solvers are widely used as tools for solving combinatorial search problems, where the goal is to find a solution among a large, but finite, number of possibilities. Such problems are ubiquitous in science and technology. Many applications of answer set solvers are related to combinatorial search as well.

Input languages of answer set solvers provide many capabilities that are not available in Prolog. We will talk here about the art of programming for CLINGO, one of the best answer set solvers available today, and about the mathematics behind the design of its input language.

1.4 Bibliographical and Historical Remarks

Lisp was specified in 1958 and is now the second-oldest (after Fortran) high-level programming language still in widespread use. The first version of Haskell was defined in 1990.

The invention of logic programming was the result of collaboration between researchers in Edinburgh and Marseille [73]. Prolog was developed in 1972 at Aix-Marseille University and used for implementing natural language processing systems. Its name is an abbreviation for *programmation en logique* (*programming in logic*). Two monographs on logic programming [85, 113] were published in the 1980s. New work in this area is presented at annual International Conferences on Logic Programming and published in the journal *Theory and Practice of Logic Programming*.

Research on stable models started in the late 1980s [13, 39, 47, 50]; we will say more about that work in Section 5.12. Many extensions of the original definition of a stable model have been proposed, beginning with the paper [51] where the term "answer set" was suggested as an alternative to "stable model."

The long history of efforts to design efficient satisfiability solvers, which began with the invention of the DPLL procedure in 1962 [29], has led to the creation of sophisticated systems that can solve, in some cases, satisfiability problems with over a million atoms [41, 57].

The development in the late 1960s and early 1970s of the concept of NP-completeness and the proof of the NP-completeness of the propositional satisfiability problem [28] showed that satisfiability solvers may be used to solve many difficult combinatorial search problems. A convincing demonstration of the power of this approach [70] was provided by applying it to the problem of planning in artificial intelligence [55].

The first answer set solver SMODELS [98] was implemented in 1996. The system DERES [24], which became available around the same time, is not exactly an answer set solver, but its functionality is similar: it implements reasoning in default logic, which is closely related to stable models (see Sections 5.12 and 6.7).

That early work was followed by creating the answer set solver DLV [26, 31, 75], CLINGO, and several others. The computational methods that CLINGO uses for generating stable models are discussed in Chapters 4 and 6 of the book [44] written by its designers.

Two papers published in 1999 [88, 97] argued that stable models can serve as the basis of a new programming paradigm, and the term "answer set programming" was introduced in the volume where the first of those papers appeared. In that sense, ASP was born in 1999. But the fact that SMODELS can be used for planning was demonstrated two years earlier [30]. Biannual ASP competitions are organized now to assess the state of the art [45].

The relationship between ASP and artificial intelligence is emphasized in two books on ASP with the words "knowledge representation" in their titles [10, 49]. The *Handbook of Knowledge Representation* includes a chapter on answer set programming [48], and the *AI Magazine* has published a special issue on ASP [1].

Chapter 2

Input Language of CLINGO

CLINGO is the centerpiece of the collection of ASP-related tools created at the University of Potsdam in Germany, called Potassco (for *Potsdam Answer Set Solving Collection*). Useful documentation and teaching materials, including information on downloading the latest CLINGO release and on running CLINGO in your browser, are available at the website of the Potassco project, `https://potassco.org`.

The description of the language of CLINGO in this chapter is sufficient for understanding and writing many interesting programs, but it is informal and incomplete. A precise definition of a number of CLINGO constructs is given in Chapters 4 and 5. In Chapter 6 we talk about several elements of the language that are not described in this chapter.

Files containing logic programs are usually given the extension `lp`. The command

```
% clingo myfile.lp
```

instructs CLINGO to find a stable model of the program `myfile.lp`.

Exercise 2.1. (a) Save program (1.1), (1.2) in a file and instruct CLINGO to find its stable model. (b) The population of Russia in 2015 was 142 million. Add this fact to your file and check how this affects the output of CLINGO. (c) Instead of comparing countries with the United Kingdom, let us define "large" as having more than 500 million inhabitants. Modify your file accordingly, and check how this change affects the output of CLINGO.

Exercise 2.2. Consider the rule

```
child(X,Y) :- parent(Y,X).                                    (2.1)
```

(a) How would you translate this rule into English? (b) If we run CLINGO on the program consisting of rule (2.1) and the rules

```
parent(ann,bob).  parent(bob,carol).  parent(bob,dan).        (2.2)
```

then what stable model do you think it will produce?

2.1 Rules

As discussed in Section 1.2, a typical rule, such as (1.1) or (2.1), consists of a head and a body separated by the "if" symbol `:-` and with a period at the end. Rules (1.2) and (2.2) do not contain "if"; such a rule is viewed as the head without a body.

The heads and bodies of rules (1.1), (1.2) are formed from *atoms*

```
large(C),  size(C,S1),  size(uk,S2),
size(france,65),  size(germany,83),  size(italy,61),  size(uk,64)
```

and one expression of another kind—a *comparison*, `S1 > S2`. Within an atom or comparison, we see elements of three types: *symbolic constants*, *numeric constants*, and *variables*. They can be distinguished by looking at the first character. A numeric constant is an integer in decimal notation, so that its first character is a digit or the minus sign. A symbolic constant is a string of letters, digits, and underscores that begins with a lower-case letter. A variable is a string of letters, digits, and underscores that begins with an upper-case letter.

An atom consists of a *predicate symbol*—a symbolic constant representing a property or a relation—and an optional list of arguments in parentheses. A comparison consists of two arguments separated by one of the symbols

```
=     !=     <     >     <=     >=
```
(2.3)

Expressions that can serve as arguments in an atom or comparison are called *terms*. The terms that we see in rules (1.1), (1.2), (2.1), (2.2) are constants and variables, but in Section 2.3 we will encounter also complex terms that are formed from constants and variables using arithmetic operations. In Section 6.6 we will talk about one more way of forming terms—the use of symbolic functions.

An atom, a rule, or another syntactic expression is *ground* if it does not contain variables. We talked above about "facts" informally; now we can say that a *fact* is a ground atom.

In Section 1.3 we explained why CLINGO produces facts (1.3) in response to rules (1.1), (1.2) by saying that these are the facts that can be derived using these rules. The first four of these facts are simply part of the program, but what about the other two—in what sense can they be "derived"?

This can be clarified by considering *instances* of rule (1.1)—the ground rules that can be obtained from it by substituting constants for variables. The presence of the atom `large(france)` in the stable model generated by CLINGO can be justified by the instance

```
large(france) :- size(france,65), size(uk,64), 65 > 64.
```

of rule (1.1), which is obtained from it by substituting the terms

`france`, `65`, and `64`

for the variables

`C`, `S1`, and `S2`

respectively. Both atoms in the body of this instance are among the given facts, and the comparison in the body is true. Consequently this instance justifies including its head `large(france)` in the stable model.

Exercise 2.3. (a) Which instance of rule (1.1) justifies including `large(germany)` in the stable model of the program? (b) Which instance of rule (2.1) justifies including `child(dan,bob)` in the stable model of program (2.1), (2.2)?

Exercise 2.4. Which of the following ground rules are instances of rule (1.1)?

(a) `large(france) :- size(france,65), size(italy,61), 65 > 61.`

(b) `large(italy) :- size(italy,61), size(uk,64), 61 > 64.`

(c) `large(italy) :- size(italy,83), size(uk,64), 83 > 64.`

(d) `large(7) :- size(7,7), size(uk,7), 7 > 7.`

The last four among the relation symbols (2.3) 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 `abracadabra` is greater than `7`. We can verify this assertion by running CLINGO on the one-rule program

```
p :- abracadabra > 7.
```

The stable model of this program, according to CLINGO, includes the atom `p`. The stable model of

```
p :- abracadabra < 7.
```

is empty.

The total order chosen by the designers of CLINGO has a minimal element and a maximal element. They are denoted by `#inf` and `#sup`.

Stable models of some programs are infinite. Consider, for instance, the one-rule program

```
p(X) :- X > 7.                                    (2.4)
```

The instance

```
p(8) :- 8 > 7.
```

of this rule justifies including `p(8)` in the stable model; the instance

```
p(9) :- 9 > 7.
```

justifies including `p(9)`; and so on. The algorithms used by answer set solvers for generating stable models are not applicable to programs that contain rules like this. In response to rule (2.4) CLINGO produces an error message saying that there are "unsafe variables" in it—an indication of the fact that a program containing this rule is likely to have an infinite stable model. But in Section 4.5 we will apply the mathematical definition of a stable model to program (2.4), and we will see what its stable model consists of: it is the set of all atoms of the form `p(`v`)`, where v is an integer or symbolic constant that is greater than 7.

This conclusion does not reflect the functionality of CLINGO, but it is in agreement with the behavior of Prolog systems. As discussed in Section 1.2, Prolog does not generate all elements of a stable model at one blow. Infinite stable models are not problematic for it, and Prolog does not reject rules like (2.4). Given this one-rule program, Prolog will answer `yes`, for instance, to the query `?- p(10)`, and `no` to the query `?- p(5)`.

When a program contains a group of facts with the same predicate symbol, these facts can be "pooled together" using semicolons. For instance, line (1.2) can be abbreviated as

```
size(france,65; germany,83; italy,61; uk,64).
```

Exercise 2.5. Use pooling to abbreviate line (2.2).

Exercise 2.6. If you run CLINGO on the one-rule program

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

then what stable model do you think it will produce?

2.2 Directives and Comments

In addition to rules, a logic program may contain *directives*, which tell CLINGO how to process the rules, and *comments*, which are intended for humans and are disregarded by CLINGO.

A `#show` directive instructs CLINGO to show some elements of the stable model and suppress the others, which is often useful. For example, in the output (1.3) of program (1.1), (1.2) the first four atoms are irrelevant—they simply repeat the facts included in the program. The output that we want the program to produce—the list of countries inhabited by more people than the UK—is given by the last two atoms. We can instruct CLINGO to "hide" all atoms that do not begin with the predicate symbol `large` by including the directive

```
#show large/1.
```

In `#show` directives, and in other cases when we refer to a predicate symbol used in a logic program, we append its *arity*—the number of arguments—after a slash; in this case, the predicate is unary, and its arity is 1. Specifying the arity is needed because the language of CLINGO allows us to use the same character string to represent several predicate symbols of different arities. This is sometimes convenient; we will see examples in Exercise 2.13 and Listing 2.6. For instance, if we run CLINGO on the program

```
p.  p(a).  p(a,b).
#show p/0.  #show p/2.
```

Listing 2.1: Large countries

```
1 % Countries with the population larger than the population
2 % of c0.
3
4 % input: country c0; the set p/2 of pairs (c,n) such that n
5 %        is the population of country c.
6
7 large(C) :- size(C,S1), size(c0,S2), S1 > S2.
8 #show large/1.
```

Listing 2.2: Input for the program in Listing 2.1

```
1 #const c0=uk.
2 size(france,65; germany,83; italy,61; uk,64).
```

then CLINGO will drop the atom `p(a)` from the output, because its predicate symbol `p/1` is different from both `p/0` and `p/2`.

A `#const` directive allows us to use a symbolic constant as a placeholder for another constant, symbolic or numeric (or for a more complex expression). For example, the directive

`#const c0=uk.` (2.5)

instructs CLINGO to substitute `uk` for `c0` in the rest of the file. In the presence of directive (2.5), the rule

```
large(C) :- size(C,S1), size(c0,S2), S1 > S2.
```

has the same meaning as (1.1).

In principle, using `#const` directives can always be avoided, because the command line option `-c` can be used instead. For instance, instead of including directive (2.5) in the file we can add

```
-c c0=uk
```

to the command line.

Any text between the symbol `%` and the end of a line is a comment, disregarded by CLINGO. Many programs that you will see in this book include comments describing the input that the program expects. The input is often provided in a separate file that consists of facts and/or `#const` directives. An example is given by Listings 2.1 and 2.2. The phrase

`the set p/2 of pairs (c,n) such that n is` ...

in the comment on Lines 3, 4 of Listing 2.1 has the same meaning as the longer phrase

`the binary relation p/2 that holds between c and n whenever n is` ...

As customary in mathematics, we identify a binary relation with the set of pairs of objects for which that relation holds. Similarly, we will identify a property with the set of objects with that property. We can say, for instance, that `large/1` is a set of countries.

If the file `large.lp` contains the program in Listing 2.1, and the file `large_input.lp` contains the program in Listing 2.2, then the command

```
% clingo large.lp large_input.lp                (2.6)
```

will cause CLINGO to concatenate the two files and produce the answer

```
large(france) large(germany)
```

Instead of adding the name of an input file to the command line, we can specify it in the program file using an `#include` directive. For instance, we can put the line

```
#include "large_input.lp".
```

anywhere in the file `large.lp` and then drop `large_input.lp` from command line (2.6).

2.3 Arithmetic

In the language of CLINGO, complex terms can be built from constants and variables using the symbols

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

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

The symbol `..` is used to form intervals. For instance, the expression `0..3` denotes the set $\{0, 1, 2, 3\}$. To express that the value of `N` belongs to this set we write `N = 0..3`. (Note that in this case the symbol `=` plays the same role as the symbol $\in$ in standard mathematical notation. In Section 4.6 we will say more about comparisons that contain intervals.) For example, the rule

```
p(N,N*N+N+41) :- N = 0..3.                (2.7)
```

expresses that the pair of integers $(x, x^2 + x + 41)$ belongs to `p/2` whenever x a number between 0 and 3. The stable model of this one-rule program is

```
p(0,41) p(1,43) p(2,47) p(3,53)
```

Exercise 2.7. 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..4.`

(b) `p(N,N*N+N+41) :- N = -3..3, N >= 0.`

Exercise 2.8. Write a one-rule program that does not contain pooling and has the same stable model as the program from Exercise 2.6.

Exercise 2.9. For each of the given sets of ground atoms, write a one-rule program that does not contain pooling and has that set as its stable model.

(a)

```
p(0,1) p(1,-1) p(2,1) p(3,-1) p(4,1)
```

(b)

```
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)
```

Intervals may be used not only in the bodies of rules, as in (2.7), but in the heads as well. For instance, a program may include the fact

```
p(0..3).
```

which has the same meaning as the set of 4 facts

```
p(0).  p(1).  p(2).  p(3).
```

This group of facts can be also abbreviated using pooling:

```
p(0; 1; 2; 3).
```

Each of these two constructs, intervals and pooling, has its advantages and limitations. Intervals are sets of *numbers*; we cannot replace pooling in Line 2 of Listing 2.2 by an interval. On the other hand, it is not practical to replace a long interval, such as `p(1..100)`, by pooling.

Another example of intervals in the head is given by the one-rule program

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

Its stable model consists of the 64 atoms corresponding to the squares of the chessboard:

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

Exercise 2.10. 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?

2.4 Definitions

Many rules in a logic program can be thought of as definitions. We can say, for instance, that rule (1.1) defines the predicate `large/1` in terms of the predicate `p/2`, rule (2.1) defines `child/2` in terms of `parent/2`, and rule (2.7) defines `p/2`.

Exercise 2.11. (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 (2.2), what stable model do you think it will produce?

Exercise 2.12. (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 (2.2), what stable model do you think it will produce?

Exercise 2.13. Assuming that the atom `enrolled(S,C)` expresses that student `S` is enrolled in class `C`, how would you define the set `enrolled/1` of all students who are enrolled in at least one class?

Exercise 2.14. Assuming that the atom `lives_in(X,C)` expresses that person `X` lives in city `C`, and that the atom `same_city(X,Y)` expresses that `X` and `Y` live in the same city, how would you define `same_city/2` in terms of `lives_in/2`?

Exercise 2.15. Assuming that the atom `age(X,N)` expresses that person `X` is `N` years old, and that the atom `older(X,Y)` expresses that `X` is older than `Y`, how would you define `older/2` in terms of `age/2`?

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 recursively. 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).                (2.8)
```

Exercise 2.16. If we run CLINGO on the program consisting of rules (2.2) and (2.8), what stable model do you expect it to produce?

Sometimes a predicate cannot be defined in one step, and auxiliary predicates have to be defined first. Consider, for instance, the property of being a prime number between 1 and some upper bound, say 5. It is easier to define the opposite property of being a composite number between 1 and 5:

```
composite(N) :- N = 1..5, I = 2..N-1, N\I = 0.            (2.9)
```

(Recall that a positive integer N is called composite if it is evenly divided by a number between 2 and $N - 1$.) Then `prime/1` can be defined in terms of `composite/1` by the rule

```
prime(N) :- N = 2..5, not composite(N).            (2.10)
```

(A positive integer N is called prime if it is different from 1 and not composite.)

Rule (2.10) is an example of the use of negation in a CLINGO program. Recall that an atom is included in a stable model of a program, informally speaking, if it can be derived using its rules (Section 1.3). But in what sense can rule (2.10) be used to derive the atom `prime(3)`? About the instance

```
prime(3) :- 3 = 2..5, not composite(3)
```

of that rule we can say that the expression

```
not composite(3)
```

in its body is justified in the sense that any attempt to use rule (2.9) to derive the atom `composite(3)` would fail. The negation symbol `not`, which is often used in logic programs, is said to represent "negation as failure." To emphasize this understanding of negation, we can read rule (2.10) as follows:

> N is a prime number between 1 and 5 if
> it is one of the numbers $2, \ldots, 5$
> and *there is no evidence* that it is composite.

Negation as failure is an important and difficult subject, and it is discussed in more detail in Section 5.1. We show how to characterize it by a mathematical definition in Section 5.2, and in Exercise 5.39 that definition is used to calculate the stable model of program (2.9), (2.10). In Section 6.5 we will talk about another kind of negation used in logic programs, "classical negation."

The definitions of `composite/1` and `prime/1` above, with 5 replaced by a placeholder and with a few comments added, are reproduced in Listing 2.3. If this program is saved in file `primes.lp` then we can instruct CLINGO to find all primes between 1 and 5 by issuing the command

```
% clingo primes.lp -c n=5
```

In Line 6 of the listing, "iff" is shortened "if and only if"—a standard abbreviation in mathematical literature. The comment in Lines 6 and 7 tells us which atoms containing the predicate symbol `composite/1` we expect to see in the stable model of the rule in Line 5. Including the second rule (Line 9) will not add any new atoms with this predicate symbol,

Listing 2.3: 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 from
 7  %           {1,...,n}.
 8
 9  prime(N) :- N = 2..n, not composite(N).
10  % achieved: prime(N) iff N is a prime number from {1,...,n}.
11
12  #show prime/1.
```

of course. Comments explaining what has been "achieved" by a group of rules at the beginning of a program express properties of stable models that the programmer expects to hold in the future, when more rules are added. Such comments help us understand the design of the program, the programmer's intentions. They also help the programmer 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 2.3 we may wish to check whether the stable model produced by CLINGO for the first rule with `n` equal to 5 is indeed

```
composite(4)
```

Exercise 2.17. Two integers are said to be *coprime* if the only positive integer that divides both of them is 1. We would like to generate the list of all integers from the set $\{1, \dots, n\}$ that are coprime with an integer k. For example, if we save such a program in the file `coprimes.lp` then the command line

```
% clingo coprimes.lp -c n=10 -c k=12
```

is expected to generate the output

```
coprime(1) coprime(5) coprime(7)
```

What rules would you place in Lines 5 and 10 of Listing 2.4 to get this result?

Exercise 2.18. Every nonnegative integer can be represented as the sum of 4 complete squares, for instance:

$$7 = 2^2 + 1^2 + 1^2 + 1^2;\ 10 = 3^2 + 1^1 + 0^2 + 0^2.$$

But if we want to represent a given number as the sum of only 3 complete squares, that may be impossible. The two smallest numbers that require 4 complete squares are 7 and

Listing 2.4: Coprime numbers (Exercise 2.17)

```
% Numbers from 1 to n that are coprime with k.

% input: positive integer n; integer k.

_______________________________________________________
% achieved: noncoprime(N) iff N is a number from {1,...,n}
%           such that N and k have a common divisor greater
%           than 1.

_______________________________________________________
% achieved: coprime(N) iff N is a number from {1,...,n}
%           that is coprime with k.

#show coprime/1.
```

15. We would like to generate the list of all integers from the set $\{1, \ldots, n\}$ that cannot be represented as the sum of 3 complete squares. What rules would you place in Lines 6 and 10 of Listing 2.5 to get such a program?

Listing 2.6 gives yet 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." Note that there is no "achieved" comment after Line 5. Nothing of interest is achieved in the middle of a definition.

Exercise 2.19. Consider the part of the program shown in Listing 2.6 that precedes the comment in Line 7. What atoms do you expect to see in its stable model if the value of `n` is 4?

2.5 Choice Rules

Each of the logic programs discussed so far 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 common 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:

$$\texttt{\{p(a); q(b)\}.} \tag{2.11}$$

This choice rule describes all possible ways to choose which of the atoms `p(a)`, `q(b)` are included in the model. There are 4 possible combinations, so that one-rule program (2.11)

Listing 2.5: Three squares are not enough (Exercise 2.18)

```
% Numbers from 1 to n that cannot be represented as the sum
% of 3 complete squares.

% input: positive integer n.

_______________________________________________________
% achieved: three/1 is the set of numbers from {1,...,n} that
%           can be represented as the sum of 3 squares.

_______________________________________________________
% achieved: more_than_three/1 is the set of numbers from
%           {1,...,n} that can't be represented as the sum
%           of 3 squares.

#show more_than_three/1.
```

Listing 2.6: Factorials

```
% Factorials of numbers from 0 to n.

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

#show fac/1.
```

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 (2.11) 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)
SATISFIABLE

Models       : 4
```

In response to the command line

```
% clingo choice.lp 2
```

CLINGO will respond:

```
  Answer: 1

  Answer: 2
  q(b)
  SATISFIABLE

  Models       : 2+
```

The plus after 2 indicates that the process of generating stable models has not been completed, so that the program may have other stable models.

Choice rules may contain pooling and intervals. For instance, the rule

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

has the same meaning as

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

and the rule

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

has the same meaning as

```
{p(1); p(2); p(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.
```
(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 2.20. For each of the given programs, what do you think is the number of its stable models?

(a) `1 {p(1..10)}.`

(b) `3 {elected(ann; bob; carol; dan; elaine; fred)} 3.`

Exercise 2.21. 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.`

If the lower and upper bound in a choice rule are equal to each other then the rule can be rewritten in a different format, using the equal sign. For instance, the rule from Exercise 2.20(b) can be written as

```
{elected(ann; bob; carol; dan; elaine; fred)} = 3.
```
(2.13)

2.6 Global and Local Variables

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

```
{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

```
{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.

Variables can be also used in a choice rule *locally*, for the purpose of specifying the list of atoms in braces in terms of predicates defined earlier. For instance, if we defined the predicate `person/1` by the rule

```
person(ann; bob; carol; dan; elaine; fred).
```

then choice rule (2.13) can be rewritten as

```
{elected(X) : person(X)} = 3.                    (2.14)
```

Local variables, such as `X` in this example, are syntactically distinguished by the fact that all their occurrences are between braces.

Variables that are not local are said to be *global.* Substituting new values for a global variable produces new instances of the rule; substituting values for a local variable does not. For example, rule (2.14) has one instance—itself, just like rule (2.13) that has no variables at all.

Exercise 2.22. (a) Rewrite the last rule of the program

```
p(a; b).
{q(X,Y) : p(X), p(Y)} = 1.
```

without the use of local variables. (b) How many stable models do you think this program has?

A choice rule may contain both local and global variables. For instance, in the rule

```
{elected(X,C) : person(X)} = 3 :- committee(C).
```

the variable `X` is local, and the variable `C` is global.

Exercise 2.23. (a) Rewrite the last rule of the program

```
p(a; b).
q(1..4).
1 {r(X,Y) : p(X)} :- q(Y).
```

without the use of local variables. (b) How many stable models do you think this program has?

2.7 Constraints

Logic programs containing choice rules often contain also *constraints*—rules that weed out the stable models for which the constraint is "violated." A constraint is a rule with the empty head, for instance

```
:- p(1).                                    (2.15)
```

By adding this constraint to a program, we eliminate its stable models that contain `p(1)`. We have seen, for example, that program (2.12) has 6 stable models. Adding rule (2.15) to it eliminates the last three of them—those that contain `p(1)`. Adding the "opposite" constraint

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

to (2.12) eliminates the first three solutions—those that do not contain `p(1)`. Adding both constraints to (2.12) will give a program that has no stable models. Combining choice rule (2.12) 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 2.24. Consider the program consisting of choice rule (2.12) 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 2.25. Find a similar transformation for the rule

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

Constraints may contain variables. For instance, the constraint

```
:- p(X), q(X).
```

expresses that the set `p/1` is disjoint from `q/1`. Adding this constraint to a program eliminates the stable models in which `p/1` and `q/1` have a common element. The constraint

```
:- f(X,Y1), f(X,Y2), Y1! = Y2.
```

expresses that the binary relation `f/2` is functional: for every `X` there is at most one `Y` such that `f(X,Y)`.

Any comparison in the body of a constraint can be replaced by the opposite comparison in the head. For instance, the last constraint can be rewritten as

```
Y1 = Y2 :- f(X,Y1), f(X,Y2).
```

If the head of a rule is a comparison then that rule is not part of a definition; it is a constraint in disguise. We will return to this example in Section 5.8.

2.8 Anonymous Variables

Imagine a rectangular grid filled with numbers. If the atom `filled(R,C,X)` expresses that the number in row `R` and column `C` is `X` then the constraints

```
R1 = R2 :- filled(R1,C1,X), filled(R2,C2,X).
C1 = C2 :- filled(R1,C1,X), filled(R2,C2,X).
```

express that the numbers in the grid are pairwise distinct. In the first of these rules, each of the variables `C1`, `C2` occurs only once. Consequently the choice of variables in these positions is irrelevant, as long as they are different from the other variables occurring in the rule and from each other. The language of CLINGO allows us to make such variables "anonymous" and replace each of them by an underscore:

```
R1 = R2 :- filled(R1,_,X), filled(R2,_,X).        (2.16)
```

In the second rule, underscores can be used instead of `R1` and `R2`.

To give another example, an underscore can be used in place of the variable `C` in the answer to Exercise 2.13.

Exercise 2.26. Find a place in Listing 2.6 (page 26) where an anonymous variable can be used.

Underscores can be eliminated from a rule with anonymous variables by replacing them with distinct new variables. For instance, rule (2.16) can be rewritten as

```
R1 = R2 :- filled(R1,Var1,X), filled(R2,Var2,X).
```

But the process implemented in CLINGO is different: it "projects out" anonymous variables using auxiliary predicates. We can project out the anonymous variables in rule (2.16) by rewriting it as

```
R1 = R2 :- aux(R1,X), aux(R2,X).
aux(R,X) :- filled(R,Var,X).
```

Auxiliary predicates, such as `aux/2` in this example, are not shown in the output of CLINGO, so that they remain invisible to the user.

It is important to keep this detail in mind when an anonymous variable is used in the scope of negation. Consider, for instance, the program

```
{p(1..2)}.
:- not p(_).
```
(2.17)

The corresponding program with the auxiliary variable projected out is

```
{p(1..2)}.
aux :- p(Var).
:- not aux.
```

This program has 3 stable models:

```
Answer: 1
p(2) aux
Answer: 2
p(1) aux
Answer: 3
p(1) p(2) aux
```

The same answers will be produced by CLINGO in response to program (2.17), except that the atom `aux` will be hidden. If, on the other hand, we replace the underscore in (2.17) with `Var` then the response of CLINGO will be different: it will tell us that the program is unsafe.

Example (2.17) illustrates a general fact: adding the constraint

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

to any program weeds out its stable models in which `p/1` is empty.

Exercise 2.27. Given the program

```
p(1,1).
q(X) :- X = 1..2, not p(X,_).
```

what solutions do you think CLINGO is going to produce?

2.9 Bibliographical and Historical Remarks

The oldest answer set solver SMODELS [98] produced error messages similar to the "unsafe" messages of CLINGO in many cases that CLINGO justifiably considers safe: these programs have finite stable models. But even CLINGO is sometimes unnecessarily careful and rejects rules as unsafe even though they cannot possibly cause a stable model to be infinite. For example, the current version of CLINGO rejects the rule

```
p(X) :- X > 7, X < 13.
```

as unsafe; to avoid getting an error message, we have to rewrite it as

```
p(X) :- X = 8..12.
```

The rule

```
noncoprime(N) :- N = 1..n, I = 2..N, N\I = 0, k\I = 0.
```

from the answer to Exercise 2.17 has the same meaning as the shorter rule

```
noncoprime(N) :- N = 1..n, I > 1, N\I = 0, k\I = 0.
```

because the conditions `N = 1..n, I > 1, N\I = 0` entail `I = 2..N`; however, the shorter rule is rejected by CLINGO as unsafe. There are reasons to believe that an algorithm that would correctly identify all "truly safe" rules is impossible [80].

The polynomial $x^2 + x + 41$, used as an example in Section 2.3, is interesting for two reasons. First, it is a "prime number generator": if you start calculating its values for $x = 0, 1, 2, \dots$, you will get a long sequence of primes. The first composite number in this sequence corresponds to $x = 40$: $40^2 + 40 + 41 = 41^2$. Second, this is the polynomial that Charles Babbage chose, many years ago, to illustrate the idea of using his Difference Engine for evaluating polynomials [56].

There are no choice rules and no constraints in Prolog, and the first version of SMODELS did not have them either. Whenever a logic program with several stable models was needed, SMODELS programmers achieved that result using a "nonstratified" combination of rules with negation as failure, as in the program

```
p :- not q.
q :- not p.
```

It has two stable models: one of them includes `p` but not `q`; the other includes `q` but not `p`. (Incidentally, such combinations of rules are never found in Prolog programs. They would cause Prolog to go into infinite loop.) We will return to this example in Chapter 5.

The answer set solver DLV [75] uses a different mechanism for describing multiple stable models—*disjunctive* rules, which have more than one atom in the head. Disjunctive rules are available in the language of CLINGO as well, and we will see examples in Sections 4.4 and 4.5. A book [87] about logic programs with disjunctive rules (but without negation as failure) was published in 1992.

In the absence of constraints in the early days of ASP, the programmer who wanted to eliminate some of the stable models of a program would achieve that by adding a carefully constructed group of rules with nonstratified negation. With choice rules and constraints incorporated in the second version of SMODELS [99], ASP programs became more concise and easier to understand.

The tradition of using underscores for anonymous variables, as much else in the syntax of ASP programs, came from Prolog.

Chapter 3

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 encode such a problem as a logic program whose stable models correspond to solutions, and then use an answer set solver, such as CLINGO, to find a stable model.

In this chapter we discuss a few examples illustrating this approach to search.

3.1 Seating Arrangements

There are n chairs around the table. We want to choose a chair for each of n guests, numbered from 1 to n, so that two conditions are satisfied. First, some guests like each other and want to sit together; accordingly, we are given a set A of two-element subsets of $\{1, \dots, n\}$, and, for every $\{i, j\}$ in A, guests i and j should sit next to each other. Second, some guests dislike each other and do not want to sit together; accordingly, we are given a set B of two-element subsets of $\{1, \dots, n\}$, and, for every $\{i, j\}$ in B, guests i and j should be separated by at least one chair.

Listing 3.1 shows a CLINGO program that finds an assignment of chairs to guests satisfying these conditions, if it exists; a sample input for that program is shown in Listing 3.2. The program begins with a choice rule describing all possible ways to assign a chair to every guest (Line 12). Three constraints weed out the stable models of this choice rule that do not solve the problem, either because they assign the same chair to two different guests (Line 15), or because they separate guests who like each other (Line 23), or because they do not separate guests who dislike each other (Line 26). The rules in Lines 18 and 19 define the auxiliary predicate `adj/2`, which is used in the constraints in Lines 23 and 26.

We can think of the choice rule in this program as a description of "candidate solutions" to the seating arrangements problem; the constraints weed out all "bad" candidates. The definition introduces a predicate that is used in some of the constraints. This division of labor between choice rules, constraints, and definitions is typical for applications of ASP to combinatorial search. Whenever a constraint is added to an emerging program, the number of stable models decreases. When a definition is added, the number of stable models does

Listing 3.1: Seating arrangements

```
% There are n chairs around a table.  Choose a chair for
% each of n guests so that guests who like each other sit
% next to each other, and guests who don't like each other
% sit at least one chair away.

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

% at(G,C) means that guest G is assigned chair C.

{at(G,1..n)} = 1 :- G = 1..n.
% achieved: each guest is assigned a chair.

G1 = G2 :- at(G1,C), at(G2,C).
% achieved: different guests are assigned different chairs.

adj(X,Y) :- X = 1..n, Y = 1..n, |X-Y| = 1.
adj(1,n; n,1).
% achieved: adj(X,Y) iff chair X is adjacent to chair Y.

:- like(G1,G2), at(G1,C1), at(G2,C2), not adj(C1,C2).
% achieved: guests who like each other sit next to each
%           other.

:- dislike(G1,G2), at(G1,C1), at(G2,C2), adj(C1,C2).
% achieved: guests who don't like each other don't sit next
%           to each other.

#show at/2.
```

Listing 3.2: Sample input for the program in Listing 3.1

```
#const n=6.

like(1,2; 3,4).
dislike(2,3; 1,3).
```

Listing 3.3: Seating arrangements with many tables (Exercise 3.2)

```
 1 % There are n tables in the room, with m chairs around each
 2 % table. Choose a table for each of m*n guests so that
 3 % guests who like each other sit at the same table, and
 4 % guests who don't like each other sit at different tables.
 5
 6 % input: positive integers m, n; set like/2 of pairs of
 7 %        guests who like each other; set dislike/2 of pairs
 8 %        of guests who dislike each other.
 9
10 % at(G,T) means that guest G is assigned table T.
11
12 {at(1..m*n,T)} = m :- T = 1..n.
13 % achieved: for each table, a group of m guests is selected.
14
15 ____________________________________________________________
16 % achieved: the groups are pairwise disjoint.
17
18 ____________________________________________________________
19 % achieved: guests who like each other sit at the same table.
20
21 ____________________________________________________________
22 % achieved: guests who don't like each other sit at different
23 %           tables.
24
25 #show at/2.
```

not change, but each stable model is enriched by atoms that involve the newly defined predicate.

Exercise 3.1. What is the number of stable models (a) of the first rule of the seating arrangements program? (b) of the first two rules? (c) of the first three rules?

Exercise 3.2. There are n tables in the room, with m chairs around each table. The guests are numbered from 1 to mn, and we would like to choose a table for each of them so that guests who like each other sit at the same table, and guests who dislike each other sit at different tables. What rules would you place in Lines 15, 18, 21 of Listing 3.3 to get a CLINGO program that does it?

3.2 Who Owns the Jackal?

Each of four men owns a different species of exotic pet. Here is what we know about them:

1. Mr. Engels (whose pet is named Sparky), Abner and Mr. Foster all belong to a club for owners of unusual pets.
2. The iguana is not owned by either Chuck or Duane.
3. Neither the jackal nor the king cobra is owned by Mr. Foster.
4. The llama does not belong to Duane (whose pet is named Waggles).
5. Abner, who does not own the king cobra, is not Mr. Gunter.
6. Bruce and Mr. Foster are neighbors.
7. Mr. Halevy is afraid of iguanas.

We would like to find the full name of the person who owns the jackal.

Clues 2–5 are clearly relevant to the task of determining the full names of the men and the species of their pets. The other clues provide useful information indirectly. From Clue 1, for instance, we can conclude that Abner's last name is neither Engels nor Foster. From Clue 6 we can see that Bruce's last name is not Foster either, and Clue 7 shows that Mr. Halevy's pet is not iguana. Furthermore, the information on the names of pets in Clues 1 and 4 shows that Duane's last name is not Engels.

Listings 3.4 and 3.5 show a CLINGO encoding of this puzzle. It defines the set of candidate solutions by two choice rules, not just one as in the seating arrangements program. Another difference between these examples is that the defined predicates `first_name/1`, `last_name/1`, `pet/1`, and `answer/2` are not used here for formulating constraints. The first three are used in choice rules, and the last in a `#show` directive.

The program has a unique stable model, as could be expected.

Exercise 3.3. If we run CLINGO on the first 12 lines of this program, how many stable models do you think it will produce? What if we run it on the first 16 lines?

3.3 Schur Numbers

A set X of numbers is called *sum-free* if the sum of two elements of X never belongs to X. 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 two 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 does not exist.

Exercise 3.4. Prove this claim.

What about partitioning $\{1, \dots, n\}$ into three 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 $\{5, \dots, 9\}$.

Listing 3.4: Exotic pets puzzle, Part 1

```
% Exotic pets puzzle, Part 1

first_name(abner; bruce; chuck; duane).
last_name(engels; foster; gunter; halevy).
pet(iguana; jackal; king_cobra; llama).
% achieved: first_name/1, last_name/1, pet/1 are the sets
%           of first names, last names, and pet species.

{full_name(F,L) : last_name(L)} = 1 :- first_name(F).
{owns(F,P) : pet(P)} = 1 :- first_name(F).
% achieved: a unique last name and unique pet species are
%           chosen for each first name.

F1 = F2 :- full_name(F1,L), full_name(F2,L).
F1 = F2 :- owns(F1,P), owns(F2,P).
% achieved: the chosen names and pets are pairwise distinct.

:- full_name(abner,engels).
:- full_name(abner,foster).
% achieved: Abner's last name is neither Engels nor Foster.

:- owns(chuck,iguana).
:- owns(duane,iguana).
% achieved: iguana belongs neither to Chuck nor to Duane.

:- full_name(X,foster), owns(X,jackal).
:- full_name(X,foster), owns(X,king_cobra).
% achieved: Mr. Foster owns neither jackal nor king cobra.

:- owns(duane,llama).
% achieved: Duane's pet is not llama.

:- full_name(duane,engels).
% achieved: Duane's last name is not Engels.

:- owns(abner,king_cobra).
% achieved: Abner's pet is not king cobra.

:- full_name(abner,gunter).
% achieved: Abner's last name is not Gunter.
```

Listing 3.5: Exotic pets puzzle, Part 2

```
% Exotic pets puzzle, Part 2

:- full_name(bruce,foster).
% achieved: Bruce's last name is not Foster.

:- full_name(X,halevy), owns(X,iguana).
% achieved: Mr. Halevy's pet is not iguana.

answer(X,Y) :- full_name(X,Y), owns(X,jackal).

#show answer/2.
```

Listing 3.6: Schur numbers

```
 1  % Partition {1,..,n} into r sum-free subsets.
 2
 3  % input: positive integers n, r.
 4
 5  % in(I,K) means that I belongs to the K-th subset.
 6
 7  {in(I,1..r)} = 1 :- I = 1..n.
 8  % achieved: set {1,...,n} is partitioned into r subsets.
 9
10  :- in(I,K), in(J,K), in(I+J,K).
11  % achieved: the subsets are sum-free.
```

Exercise 3.5. Partition $\{1, \dots, 10\}$ into three sum-free subsets.

The CLINGO program shown in Listing 3.6 solves the general problem of partitioning $\{1, \dots, n\}$ into r sum-free subsets (possibly empty) whenever this is possible. The largest value of n for which such a partition exists is traditionally denoted by $S(r)$. For instance, $S(2) = 4$, and by solving Exercise 3.5 you proved that $S(3) \geq 10$. The numbers $S(r)$ are called *Schur numbers*. We can say that by running this program we estimate Schur numbers.

Exercise 3.6. How many models do you think this program has for $r = 2$ and $n = 4$?

Exercise 3.7. What is the number of stable models of the rule in Line 7?

Exercise 3.8. About a set X of numbers we say that it is *almost* sum-free if the sum of two *different* elements of X never belongs to X. For instance, the set $\{1, 2, 4\}$ is almost

Listing 3.7: Subsets without arithmetic progressions of length 3 (Exercise 3.10)

```
 1 % Partition {1,...,n} into r subsets that do not contain
 2 % arithmetic progressions of length 3.
 3
 4 % input: positive integers n, r.
 5
 6 % in(I,J) means that I belongs to the J-th subset.
 7
 8 {in(I,1..r)} = 1 :- I = 1..n.
 9 % achieved: set {1,...,n} is partitioned into r subsets.
10
11 ____________________________________________________
12 % achieved: these subsets do not contain arithmetic
13 %           progressions of length 3.
```

sum-free. If we want to describe partitions of $\{1,\dots,n\}$ into r *almost* sum-free sets, how will you change the program in Listing 3.6?

If we run the program in Listing 3.6 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, CLINGO can tell us that $S(4) = 44$. In the next section we say more about using CLINGO for calculating Schur numbers.

Exercise 3.9. Use CLINGO to verify that $S(5) \geq 130$.

Exercise 3.10. About a set of numbers we say that it *contains an arithmetic progression of length 3* if it has three different elements a, b, c such that $b - a = c - b$. We would like to write a CLINGO program that partitions $\{1,\dots,n\}$ into r subsets that do not contain arithmetic progressions of length 3, if this is possible. What rule would you place in Line 11 of Listing 3.7 to get this result?

3.4 Digression on Grounding and Solving

If we save the program from Listing 3.6 in the file `schur.lp` and execute the command

```
clingo schur.lp -c r=4 -c n=44
```

then the output of CLINGO may look like this:

```
Answer: 1
in(1,1) in(2,3) ...
. . .
Time         : 0.189s (Solving: 0.16s 1st Model: 0.16s Unsat: 0.00s)
```

The difference between the total time, 0.189 *sec*, and the solving time, 0.16 *sec*, is explained by the fact that the operation of CLINGO begins with *grounding*—generating and simplifying the instances of rules that are essential for finding the stable models. In this case, the grounding time was around 0.03 *sec*; grounding was followed by the solving phase, which took 1.16 *sec*.

There are cases when the solving time of a CLINGO program is so large that the program is unusable. This happens, for instance, with the program in Listing 3.6 for large values of r and n. This is not surprising, because answer set solvers (and satisfiability solvers) are often used to solve intrinsically difficult, NP-hard problems. The runtimes of all (known) algorithms solving NP-hard problems grow exponentially with the size of the problem. But if large solving time is an issue then it may be useful to remember that CLINGO has several solving strategies, and we can instruct it to try several strategies in parallel. For example, the command

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

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

```
UNSATISFIABLE
. . .
Threads       : 4        (Winner: 2)
```

Information on the "winner" tells us which thread reached the goal first.

There are also cases when the *grounding* time of a program is unacceptably large. In the case of the program from Exercise 2.18, the solving time is negligible, but the grounding time grows quickly as n becomes larger. To understand why, look at the rule

```
three(N) :- N = 1..n, I = 0..n, J = 0..n, K = 0..n, N = I**2+J**2+K**2.
```

suggested in Appendix A for Line 6 of the program (page 155). The instances of this rule that are identified by CLINGO as essential for generating the stable model are obtained by substituting arbitrary values between 0 and n for each of the variables `I`, `J`, `K`, and the number of such instances is $(n+1)^3$, it grows quickly with n. Significant grounding times are typical for ASP programs containing a rule with many variables, and we try to avoid including such rules in an encoding when possible.

The grounding time of the program from Exercise 2.18 can be improved using the fact that the summands in the expression $i^2+j^2+k^2$ can be always reordered so that $i \leq j \leq k$. Rewriting the definition of `three/1` in the form

```
three(N) :- N = 1..n, I = 0..n, J = I..n, K = J..n, N = I**2+J**2+K**2.
```

decreases the number of instances identified by CLINGO as essential by approximately a factor of 6, and the grounding time goes down accordingly. Using the symmetry of a problem to improve the performance of an algorithm is known as "symmetry breaking." In Section 6.3 we will see how symmetry breaking can be used to speed up search in the Schur numbers example.

The program from Exercise 2.18 can be further optimized by exploiting the fact that the values of `I`, `J`, and `K` cannot be greater than $\sqrt{n}$:

```
sqrt(S) :- S = 1..n, S**2 <= n, (S+1)**2 > n.
three(N) :- sqrt(S), N = 1..n, I = 0..S, J = I..S, K = J..S,
            N = I**2+J**2+K**2.
```

For some CLINGO programs, the process of grounding does not terminate at all. Given the program

```
p(0).
p(N+2) :- p(N).
```
(3.1)

(the current version of) CLINGO dies and has to be restarted. We will return to this example in Section 4.7.

3.5 Permutation Pattern Matching

In the permutation matching problem, we are given a permutation T of the numbers $1, \dots, n$ ("text") and a permutation P of the numbers $1, \dots, k$ ("pattern"), where $k \leq n$. The goal is to find a k-element subsequence of T whose elements are ordered according to the permutation P. In other words, if the i-th element of the pattern is less than its j-th element then the i-th element of the subsequence should be less that the j-th element of the subsequence, and the other way around. For instance, if the text is 53142 and the pattern is 231 then the only solution is given by the subsequence 342.

A CLINGO program solving the permutation matching problem is shown in Listing 3.8, and a sample input for it in Listing 3.9.

Exercise 3.11. Assuming that the length of the text T is n and the length of the pattern P is k, what is the number of stable models (a) of the first rule of this program? (b) of the first two rules?

3.6 Sequence Covering Arrays

In some cases, faults in the design of a software system are triggered by events that occur in a particular order. Sequence covering arrays are used for designing tests that can detect faults of this kind.

This concept is defined as follows. About a permutation x of a set S of symbols and a permutation y of a subset of S we say that x *covers* y if y can be obtained from x by deleting some of its members. For instance, the permutation 52314 of $\{1 \dots, 5\}$ covers the permutation 531 of $\{1, 3, 5\}$: it can be obtained by deleting 2 and 4. A set M of permutations of S is a *sequence covering array of strength t* if every permutation of every t-element subset of S is covered by a permutation from M. The *size* of an array is the

Listing 3.8: Permutation pattern matching

```
% Permutation pattern matching.

% input: for some permutation T of 1,...,n, the set t/2 of
%        pairs (I,E) such that E is the I-th element of T;
%        for some permutation P of 1,...,k, the set p/2 of
%        pairs (I,E) such that E is the I-th element of P;
%        the length k of P.

{subseq(K,I,E) : t(I,E)} = 1 :- K = 1..k.
% achieved: for some functions i, e from {1,...,k} to
%           {1,...,n}, the numbers e(1),...,e(k) are the
%           elements of T in positions i(1),...,i(k).

I1 < I2 :- subseq(K1,I1,_), subseq(K2,I2,_), K1 < K2.
% achieved: e(1),...,e(k) is a subsequence of T.

EP1 < EP2 :- subseq(K1,_,ET1), subseq(K2,_,ET2),
           p(K1,EP1), p(K2,EP2), ET1 < ET2.
% achieved: this subsequence is ordered according to P.

subseq(E) :- subseq(_,_,E).
% achieved: subseq/1 = {e(1),...,e(k)}.

#show subseq/1.
```

Listing 3.9: Sample input for the program in Listing 3.8

```
% T = (5,3,1,4,2), P = (2,3,1).

t(1,5; 2,3; 3,1; 4,4; 5,2).
p(1,2; 2,3; 3,1).

#const k=3.
```

number of permutations in it. For instance, if $S = \{1, \dots, 5\}$ then the set

$$\begin{aligned}\{&52314,\\ &32541,\\ &15432,\\ &34512,\\ &42513,\\ &24315,\\ &12345\}\end{aligned}$$

is a sequence covering array of strength 3; its size is 7.

The CLINGO program shown in Listing 3.10 constructs a sequence covering array of strength 3 and size n for the symbols $1, \dots, s$.

Exercise 3.12. What is the number of stable models of the program in Lines 1–14 of Listing 3.10?

3.7 Partner Units Problem

In the partner units problem, we are given a set of sensors grouped into zones. A sensor may be attached to several zones. Figure 3.1, for instance, shows a set of zones with 3 sensors attached to each of them. The goal is to connect the sensors and zones to a given number of control units and to define which pairs of control units are "partners," so that two conditions are satisfied:

(i) if a sensor is attached to a zone, but the sensor and the zone are assigned to different control units, then these two control units are partners;

(ii) the number of sensors assigned to a unit, the number of zones assigned to a unit, and the number of partners of a unit do not exceed given upper bounds.

For example, the graph in Figure 3.2 shows a solution for the configuration in Figure 3.1 with 4 control units, assuming that each of the upper bounds (ii) equals 2. The horizontal edges represent the partnership relation between units.

A CLINGO program solving the partner units problem is shown in Listing 3.11, and Listing 3.12 is an input for that program corresponding to the example above.

3.8 Set Packing

In the set packing problem, we are given a list of n sets, and the goal is to find m members of the list that are pairwise disjoint (that is, have no common elements). A CLINGO program that solves this problem, if a solution exists, is shown in Listing 3.13, and a sample input for it in Listing 3.14.

Listing 3.10: Sequence covering arrays

```
% Construct a sequence covering array of strength 3 and size n
% for symbols 1,...s.

% input: positive integers n, s.

{hb(N,X,Y); hb(N,Y,X)} = 1 :- N = 1..n, X = 1..s, Y = 1..s,
                              X != Y.
% For every row N, let hb_N be the binary relation on the set
% of symbols defined by the condition: X hb_N Y iff hb(N,X,Y).
% achieved: each relation hb_N is irreflexive; each pair of
% distinct symbols satisfies either X hb_N Y or Y hb_N X.

:- hb(N,X,Y), hb(N,Y,Z), not hb(N,X,Z).
% achieved: each relation hb_N is transitive.

covered(X,Y,Z) :- hb(N,X,Y), hb(N,Y,Z).
% For every row N and every symbol X, let M_{N,X} be the
% symbol that is the X-th smallest w.r.t. hb_N.
% achieved: for any symbols X, Y, Z, covered(X,Y,Z) iff
%           for some row N, (X,Y,Z) is a subsequence of
%           (M_{N,1},...,M_{N,s}).

:- not covered(X,Y,Z), X = 1..s, Y = 1..s, Z = 1..s,
   X != Y, Y != Z, X != Z.
% achieved: covered(X,Y,Z) for any pairwise distinct symbols
%           X, Y, Z.

#show hb/3.
```

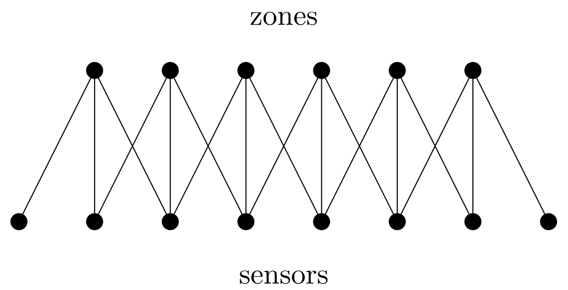

Figure 3.1: A sensor/zone configuration for the partner units problem.

Listing 3.11: Partner units problem

```
% Partner units problem.

% input: set zone2sensor/2 of pairs (z,s) such that z is a
%        zone and s is a sensor attached to it; number n
%        of units; upper bound unitCap on the number of
%        sensors and the number of zones connected to a
%        unit; upper bound interUnitCap on the number of
%        partners of a unit.

% A unit number is an integer between 1 and n.

{unit2zone(1..n,Z)} = 1 :- zone2sensor(Z,_).
% achieved: a unique unit number is assigned to each zone.

{unit2sensor(1..n,S)} = 1 :- zone2sensor(_,S).
% achieved: a unique unit number is assigned to each sensor.

partner(U1,U2) :- unit2zone(U1,Z), zone2sensor(Z,S),
                  unit2sensor(U2,S), U1 != U2.
partner(U1,U2) :- partner(U2,U1).
% achieved: partner(U1,U2) iff U1, U2 are unit numbers such
%           that one of them is assigned to a zone and the
%           other to a sensor not attached to that zone.

:- #count{Z : unit2zone(U,Z)} > unitCap, U = 1..n.
% achieved: the number of zones connected to a unit doesn't
%           exceed unitCap.

:- #count{S : unit2sensor(U,S)} > unitCap, U = 1..n.
% achieved: the number of sensors connected to a unit doesn't
%           exceed unitCap.

:- #count{U2 : partner(U1,U2)} > interUnitCap, U1 = 1..n.
% achieved: the number of partners of a unit doesn't exceed
%           interUnitCap.

#show unit2zone/2.  #show unit2sensor/2.  #show partner/2.
```

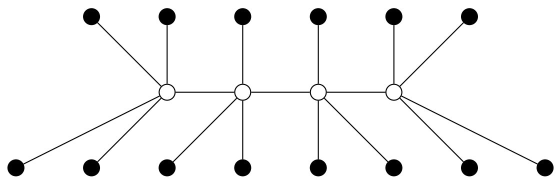

Figure 3.2: A solution to the partner units problem in Figure 3.1.

Listing 3.12: Input for the program in Listing 3.11

```
zone2sensor(z(N),s(N..N+2)) :- N = 1..6.
#const n=4.
#const unitCap=2.
#const interUnitCap=2.
```

Listing 3.13: Set packing

```
% Given a list of finite sets and a positive integer m,
% find m pairwise disjoint members of the list.

% input: for a list S_1,...,S_n of sets, its length n and
%        the set s/2 of pairs X,I such that X is in S_I;
%        a positive integer m.

% in(I) means that set S_I is included in the solution.

{in(1..n)} = m.
% achieved: in/1 is a set of m members of the list.

I = J :- in(I), in(J), s(X,I), s(X,J).
% achieved: the chosen sets are pairwise disjoint.

#show in/1.
```

Listing 3.14: Sample input for the program in Listing 3.13

```
% S_1 = {a,b,c}, S_2 = {b,c,d}, S_3 = {a,c}, S_4 = {b,d}.

#const n=4.
s(a,1; b,1; c,1;
  b,2; c,2; d,2;
  a,3; c,3;
  b,4; d,4).
#const m=2.
```

In Section 6.4 we will see how CLINGO can be instructed to solve the optimization version of the set packing problem—to find the largest possible number of pairwise disjoint members of a given list of sets.

Exercise 3.13. In the exact cover problem, we are given a list of finite sets, and the goal is to find a part of the list such that its members are pairwise disjoint, and their union is the same as the union of the entire list. What rules would you place in Lines 13, 16, 19 of Listing 3.15 to get a CLINGO program that finds a solution, if it exists?

3.9 Search in Graphs

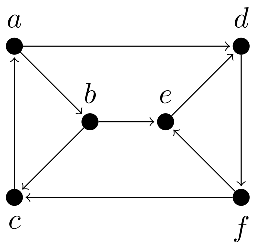

Figure 3.3: Graph with 6 vertices and 9 edges.

In a logic program, a directed graph can be described by two predicates, `vertex/1` and `edge/2`. For example, the graph shown in Figure 3.3 can be represented by the rules

```
vertex(a; b; c; d; e; f).
edge(a,b; b,c; c,a; d,f; f,e; e,d; a,d; f,c; b,e).
```
(3.2)

These rules can be viewed also as a representation of the corresponding undirected graph.

A *clique* in a graph is a subset of its vertices such that every two distinct vertices in it are adjacent. For example, the graph in Figure 3.3 has cliques $\{a, b, c\}$ and $\{d, e, f\}$, and

Listing 3.15: Exact cover (Exercise 3.13)

```
% Given a list of finite sets, find a part of the list
% such that its members are pairwise disjoint and their
% union is the same as the union of the entire list.

% input: for a list S_1,...,S_n of sets, its length n and
%        the set s/2 of pairs X, I such that X is in S_I.

% in(I) means that set S_I is included in the solution.

{in(1..n)}.
% achieved: several members of the list are chosen.

________________________________________________________
% achieved: the chosen sets are pairwise disjoint.

________________________________________________________
% achieved: covered/1 is the union of the chosen sets.

________________________________________________________
% achieved: the union of the chosen sets is the same as
%           the union of the entire list.

#show in/1.
```

Listing 3.16: Cliques of a given size

```
% Cliques of size n.

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

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

X = Y :- in(X), in(Y), not edge(X,Y), not edge(Y,X).
% achieved: in/1 is a clique.

#show in/1.
```

Listing 3.17: Vertex cover (Exercise 3.14)

```
 1  % Vertex cover of size n.
 2
 3  % input: set vertex/1 of vertices of a graph G;
 4  %        set edge/2 of edges of G; positive integer n.
 5
 6  --------------------------------------------------------
 7  % achieved: in/1 is a set consisting of n vertices of G.
 8
 9  --------------------------------------------------------
10  --------------------------------------------------------
11  % achieved: covered/2 is the set of edges of G that have
12  %           an endpoint in in/1.
13
14  --------------------------------------------------------
15  % achieved: every edge of G is in covered/2.
16
17  #show in/1.
```

many smaller cliques, but no cliques of size 4. The program shown in Listing 3.16 describes cliques of a given size.

Exercise 3.14. A *vertex cover* of a graph is a set of vertices such that every edge of the graph has at least one endpoint in that set. We would like to find a vertex cover of size n, if it exists. What rules would you place in Lines 6, 9, 10, 14 of Listing 3.17 to get a CLINGO program that produces this result?

Exercise 3.15. We would like to write a CLINGO program that finds a way to color the vertices of a given graph so that no two adjacent vertices share the same color. What rules would you place in Lines 8 and 12 of Listing 3.18?

By running the program from the last exercise a few times, with different sets of colors, we can estimate the chromatic number of a graph—the smallest number of colors needed to color its vertices. In Section 6.4 we will see how we can instruct CLINGO to calculate the chromatic number.

A *Hamiltonian cycle* in a directed graph is a cycle that visits every vertex of the graph exactly once. For example, (a, b, e, d, f, c, a) is a Hamiltonian cycle in the graph from Figure 3.3.

An encoding of Hamiltonian cycles in the language of CLINGO is shown in Listing 3.19. In the choice rule in Line 8, the variable `X` is global, and the variable `Y` is local (see Section 2.6). The stable models of this rule (with definitions of `vertex/1` and `edge/2` added) correspond to the sets of edges such that every vertex x of the graph is the tail of exactly one edge

Listing 3.18: Graph coloring (Exercise 3.15)

```
% Graph coloring.

% input: set vertex/1 of vertices of a graph G; set edge/2
%        of edges of G; set color/1 of colors.

% color(X,C) means that the color of vertex X is C.


_________________________________________________________
% achieved: for every vertex X there is a unique color C
%           such that color(X,C).


__________________________________________________________
% achieved: no two adjacent vertices share the same color.

#show color/2.
```

(x, y) from that set. Adding the choice rule in Line 9 ensures, in addition, that every vertex y of the graph is the head of exactly one edge from that set. Thus adding the second choice rule does what it usually accomplished by adding a constraint—it eliminates some of the unwanted candidate solutions.

Every Hamiltonian cycle is represented by one of the stable models of this pair of choice rules, but the converse does not hold—some of these stable models do not correspond to cycles. Consider, for instance, the edges

$$(a, b),\ (b, c),\ (c, a),\ (d, f),\ (f, e),\ (e, d)$$

of the graph in Figure 3.3. Every vertex of the graph is the head of exactly one edge from this set, and also the tail of exactly one edge from this set, but these edges do not form a cycle. To eliminate the unsuccessful candidates like this, we need to require that any vertex of the graph be reachable from any other vertex by a chain of edges from `in/2`. In the program, this is accomplished using the set `reachable/1` of all vertices that are reachable by such a chain from some fixed vertex `v0`.

3.10 Search in Two Dimensions

The eight queens puzzle is the problem of placing eight chess queens on a chessboard so that no two queens threaten each other. In other words, no two queens can share the same row, column, or diagonal. Solutions to this puzzle are represented by the stable models of the program in Listing 3.20.

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

Listing 3.19: Hamiltonian cycles in a directed graph

```
% Hamiltonian cycles in a directed graph.

% input: set vertex/1 of vertices of a graph G;
%        set edge/2 of edges of G; a vertex v0 of G.

% in(X,Y) is the set of edges included in the cycle.

{in(X,Y) : edge(X,Y)} = 1 :- vertex(X).
{in(X,Y) : edge(X,Y)} = 1 :- vertex(Y).
% achieved: a set in/2 of edges is selected such that every
%           vertex of G is the head of exactly one edge from
%           in/2, and the tail of exactly one edge from in/2.

reachable(X) :- in(v0,X).
reachable(Y) :- reachable(X), in(X,Y).
% achieved: reachable(X) iff there exists a path of non-zero
%           length from v0 to X such that all its edges
%           belong to in/2.

:- not reachable(X), vertex(X).
% achieved: all vertices of G belong to reachable/1.

#show in/2.
```

Listing 3.20: Eight queens

```
% Eight queens puzzle.

{q(1..8,1..8)} = 8.
% achieved: q/2 is a set of 8 squares on the chessboard.

:- q(R,C1), q(R,C2), C1 < C2.
:- q(R1,C), q(R2,C), R1 < R2.
% achieved: q/2 contains at most one square in each column
            and at most one in each row.

:- q(R1,C1), q(R2,C2), R1 < R2, |R1-R2| = |C1-C2|.
% achieved: q/2 contains at most one square in each diagonal.
```

6		
	2	8
1		

Table 3.1: A Number Snake puzzle

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

Exercise 3.16. Solve the puzzle in Table 3.1.

The program shown in Listing 3.21 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 14 expresses that the distance between the centers of the squares `(R1,C1)` and `(R2,C2)` is less than or equal to $\sqrt{2}$; this is equivalent to saying that the squares connect horizontally, vertically, or diagonally.

Exercise 3.17. Cover a chessboard by twenty-one 3×1 tiles and one 1×1 tile.

The program in Listing 3.22 solves the problem from Exercise 3.17. This encoding describes all possible ways to place twenty-one 3×1 tiles on the board without overlaps; it does not mention the 1×1 tile explicitly. The expressions `h(1..6,1..8)` and `v(1..8,1..6)` represent the 48 possible positions of horizontal tiles and the 48 possible positions of vertical tiles, so that the choice rule in Line 7 requires the total number of tiles, both horizontal and vertical, to be 21.

Addition in Line 10 is applied to a number and an interval—to `C` and `1..2`. In the language of CLINGO, this is understood as forming the set of all integers that can be obtained by adding `C` and a number from the interval `1..2`. In other words, this expression has the same meaning as `C+1..C+2`. Similarly, `R-(0..2)` in Line 16 has the same meaning as `R-2..R`. Applying arithmetic operations to intervals in the language of CLINGO is discussed in Section 4.6 in more detail.

3.11 Programming Exercises

Who Owns the Fish?

There are five houses of five different colors. In each house lives a person of a different nationality. Each of these five men drinks a certain beverage, smokes a certain brand of cigarettes, and keeps a certain pet. No two men have the same pet, drink the same drink or smoke the same brand. We also know the following:

1. The Brit lives in the red house.

2. The Swede keeps a dog.

Listing 3.21: 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.

{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.

:- not filled(_,_,X), X = 1..n*n.
% achieved: every number between 1 and n^2 is included.

(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 3.22: Tiling

```
% Cover the 8-by-8 chessboard by twenty-one 3-by-1 tiles and
% one 1-by-1 tile.

% h(R,C) means that there is a tile at (R,C), (R,C+1), (R,C+2).
% v(R,C) means that there is a tile at (R,C), (R+1,C), (R+2,C).

{h(1..8,1..6); v(1..6,1..8)} = 21.
% achieved: positions of 21 tiles are chosen.

:- h(R,C), h(R,C+(1..2)).
% achieved: no overlaps between horizontal tiles.

:- v(R,C), v(R+(1..2),C).
% achieved: no overlaps between vertical tiles.

:- h(R,C), v(R-(0..2),C+(0..2)).
% achieved: no overlaps between a horizontal tile and a
%           vertical tile.
```

3. The Dane drinks tea.
4. The green house is on the left of the white house.
5. The owner of the green house drinks coffee.
6. The person who smokes Pall Mall rears birds.
7. The owner of the yellow house smokes Dunhill.
8. The man living in the house right in the center drinks milk.
9. The Norwegian lives in the first house.
10. The man who smokes Blend lives next to the one who has cats.
11. The man who has horses lives next to the Dunhill smoker.
12. The man who smokes Bluemaster drinks beer.
13. The German smokes Princess.
14. The Norwegian lives next to the blue house.
15. The man who smokes Blend has a neighbor who drinks water.

Use CLINGO to determine who owns the fish. To test your encoding, verify that it has a unique stable model.

Clique Cover

Write a program for CLINGO to cover the set of vertices of a given graph by n cliques, if possible. To test your program, use the graph from Figure 3.3.

Bishops on a Chessboard

Use CLINGO to determine how many bishops can be placed on a chessboard so that they do not attack each other.

Filling a Grid with Letters

You are given a 5×5 grid partially filled with letters A, B, C, D, E, and with a question mark in one of the squares, for instance:

	?			
				A
		B		
D		C		
			E	

Write a program for CLINGO that determines what letter can replace the question mark if we fill the grid so that each letter is used once in each row, each column, and each of the two diagonals.

Packing Squares into a Rectangle

Write a program for CLINGO that packs a given set of squares into a given rectangular area without overlaps. For instance, if we want to pack the squares

$$A \text{ of size } 4, \quad B \text{ of size } 3, \quad C \text{ and } D \text{ of size } 2, \quad E \text{ of size } 1$$

into an area 5×8 then one of the solutions is

A	A	A	A	B	B	B	
A	A	A	A	B	B	B	
A	A	A	A	B	B	B	E
A	A	A	A	C	C	D	D
				C	C	D	D

Hitori

In a Hitori puzzle, you are given a square grid with integers appearing in all squares. The object is to darken some of the squares so that

- in undarkened squares, no number appears in any row or column more than once,
- darkened squares do not touch each other vertically or horizontally,
- all undarkened squares are connected to each other.

Write a CLINGO program that solves Hitori puzzles. To test your program, run it on the example from the Wikipedia article on Hitori (`http://en.wikipedia.org/wiki/Hitori`).

3.12 Bibliographical and Historical Remarks

Research on the use of automated reasoning for solving logic puzzles, such as the one in Section 3.2, has been conducted since the 1980s [120]. Constraint Lingo [40] is a high-level language for describing puzzles of this kind; a compiler translates Constraint Lingo representations into input languages of several answer set solvers and automated reasoning systems of other types. Work has been done also on generating ASP encodings of logic puzzles from their descriptions in English [94, 110].

Translating clues of a logic puzzle from unconstrained English into the input language of an automated reasoning system is a large step towards creating an AI system that solves puzzles. But that challenging task has also another component: the automation of commonsense reasoning steps involved in the use of the clues. The relationship between the sentences "Mr. Halevy is afraid of iguanas" and "Mr. Halevy's pet is not iguana" is that of *textual entailment* [71]: a human reading the former would be justified in inferring the latter. Writing programs capable of recognizing textual entailment is an important and difficult problem on the border of computational linguistics and artificial intelligence.

The investigation of Schur numbers started with the proof of the fact that no matter how the set of positive integers less than or equal to $\lfloor r!e \rfloor$ is partitioned into r classes, one of these classes is not sum-free [109]. (Here $\lfloor x \rfloor$ is the floor of x—the greatest integer less than or equal to x, and e is Euler's number $2.718\ldots$.) Thus $S(r) < \lfloor r!e \rfloor$. The fact that $S(4) = 44$ was established in 1965 [12]. The value of $S(5)$ is 160; it was computed in 2018 [65] using massively parallel satisfiability solving techniques. About the next two Schur numbers we know that $S(6) \geq 536$ and $S(7) \geq 1680$ [42].

The problem of detecting the cases when grounding a safe program does not terminate was studied by several researchers [20, 21, 77, 78]. The general problem is not solvable, but a partial algorithm is implemented in the answer set solver DLV. It warns the user when the termination of grounding is not guaranteed.

For any positive integers r and k there exists a positive integer n such that if the set $\{1, \ldots, n\}$ is patitioned into r subsets then at least one of these subsets contains k integers in arithmetic progression [116]. The smallest such n is called the *van der Waerden number* $W(r, k)$. The program from Exercise 3.10 allows us to estimate van der Waerden numbers of the form $W(3, r)$. For example, $W(3, 3) = 27$ [25]; this can be easily confirmed by CLINGO. It is known also that $W(3, 4) = 293$ [72].

Deciding whether a given instance of the permutation matching problem has a solution is NP-complete [14].

Sequence covering arrays, and the answer set approach to generating them, were introduced in 2012 [74, 15].

Applications of the partner units problem include traffic management and security. The program in Listing 3.11 is reproduced, with minor changes, from a 2011 paper [3].

The search problems discussed in Sections 3.8 and 3.9 belong to the list of 21 NP-complete problems in the classical paper [69]. To be precise, the concept of NP-completeness applies to the corresponding decision problems—determining whether a solution exists. The list includes the set packing problem, the exact cover problem, the clique problem, the vertex cover problem, graph coloring problem, and the Hamiltonian cycle problem.

Hidato is an invention of the computer scientist Gyora Benedek. Its name comes from the Hebrew word *hida* (*puzzle*). Hidato puzzles of different degrees of difficulty are available at `http://hidato.com`.

The eight queens puzzle was published by the chess composer Max Bezzel in 1848. It is often used as an example in introductory programming courses. Here is the Pascal program solving this puzzle from a classical book on programming methodology [118]:

```
program eightqueen1(output);

var i : integer; q : boolean;
    a : array[1 .. 8] of boolean;
    b : array[2 .. 16] of boolean;
    c : array[-7 .. 7] of boolean;
    x : array[1 .. 8] of integer;

procedure try(i : integer; var q : boolean);
    var j : integer;
    begin
    j := 0;
    repeat
        j := j + 1;
        q := false;
        if a[j] and b[i + j] and c[i - j] then
            begin
            x[i    ] := j;
            a[j    ] := false;
            b[i + j] := false;
            c[i - j] := false;
            if i < 8 then
                begin
        try(i + 1, q);
        if not q then
          begin
          a[j] := true;
          b[i + j] := true;
          c[i - j] := true;
          end
        end
      else
        q := true
      end
  until q or (j = 8);
  end;

begin
for i :=  1 to  8 do a[i] := true;
for i :=  2 to 16 do b[i] := true;
for i := -7 to  7 do c[i] := true;
try(1, q);
if q then
  for i := 1 to 8 do write( x[i]:4);
writeln
end.
```

Chapter 4

Propositional Programs and Minimal Models

In this chapter and the next, we discuss the mathematical definition of a stable model. In the process, we introduce a few syntactic features of the input language of CLINGO that have not been mentioned earlier.

The first step is to review a few concepts related to propositional formulas. The presentation here is different from standard in that we write implications "backwards":

$$q \leftarrow p \quad (q \text{ if } p)$$

instead of

$$p \rightarrow q \quad (\text{if } p \text{ then } q).$$

This makes propositional formulas syntactically closer to rules in a logic program.

As common in the literature on logic programming, truth assignments are represented here by sets of atomic formulas. For instance, assigning *true* to p and q and *false* to all other atomic formulas will be represented by the set $\{p, q\}$. This is convenient because of the role of sets of atoms in logic programming: a stable model is a set of ground atoms.

4.1 Propositional Formulas

Assume that we are given a *vocabulary*—a set of symbols called *atomic formulas*. *Formulas* are built from atomic formulas and the logical constants $\bot$ (false) and $\top$ (true) using the connectives $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), and $\leftarrow$ (implication).

Implication binds weaker than the other connectives; for instance,

$$p \leftarrow q \wedge r \tag{4.1}$$

is understood as shorthand for the formula $p \leftarrow (q \wedge r)$.

If one of the truth values *false* or *true* is assigned to each atomic formula then the truth values of other formulas are defined as follows. The truth value of $\bot$ is *false*; the truth value of $\top$ is *true*. For propositional connectives, we use the truth tables:

F	$\neg F$
false	*true*
true	*false*

F	G	$F \wedge G$	$F \vee G$	$F \leftarrow G$
false	*false*	*false*	*false*	*true*
false	*true*	*false*	*true*	*false*
true	*false*	*false*	*true*	*true*
true	*true*	*true*	*true*	*true*

Sets of atomic formulas will be called *interpretations*. An interpretation I can be thought of as an assignment of truth values to atomic formulas: those that belong to I get the value *true*, and all others get the value *false*. If a formula F gets the value *true* for an interpretation I then we say that I *satisfies* F, or that I is a *model* of F. For instance, the empty set $\emptyset$ satisfies formula (4.1), because that formula gets the value *true* when all atomic formulas p, q, r get the value *false*.

Exercise 4.1. Assuming that the vocabulary is $\{p, q, r\}$, find all interpretations that do not satisfy formula (4.1).

Exercise 4.2. Find a formula that is satisfied by $\{p\}$ and by $\{q\}$ but is not satisfied by $\{p, q\}$.

We say that an interpretation I is a *model* of a set Γ of formulas if I satisfies all formulas in Γ.

Exercise 4.3. Assuming that the vocabulary is $\{p, q, r\}$, find all models of the set

$$\{p \leftarrow q \wedge r,\ q \leftarrow p,\ r \leftarrow p\}.$$

Exercise 4.4. Assuming that the vocabulary is $\{p, q, r, s\}$, find all models of the set

$$\{p \leftarrow q,\ q \leftarrow r,\ r \leftarrow s,\ s \leftarrow p\}.$$

Exercise 4.5. Assuming that the vocabulary is $\{p_1, p_2, \dots, p_8\}$, find the number of models of

(a) $p_1 \wedge p_2$,

(b) $p_1 \leftarrow p_2$.

Exercise 4.6. Assuming that vocabulary is the infinite set $\{p_0, p_1, p_2, \dots\}$, find all models of the infinite sets

(a) $\{p_1, \neg p_2, p_3, \neg p_4, \dots\}$,

(b) $\{p_0 \leftarrow p_1, p_1 \leftarrow p_2, p_2 \leftarrow p_3, \dots\} \cup \{\neg p_2\}$.

The formulas in the examples and exercises above use the implication symbol $\leftarrow$ in a limited way. Some of them do not contain implication at all. Others have the form $F \leftarrow G$, where F and G do not contain implication. Formulas of these two kinds will be called *propositional rules*, and sets of propositional rules will be called *propositional programs*. We will drop "propositional" when it is clear from the context that we are not talking about rules and programs in the language of CLINGO. About a propositional rule $F \leftarrow G$ we say that F is its *head* and G is its *body*. If there are no implications in F then we say that the whole formula F is the head of the rule.

Our plan is to define the concept of a stable model for propositional programs, and use that definition as the basis for the study of stable models of CLINGO programs.

Note that a propositional program is a *set* of rules, not a list. The collection of stable models of a propositional program, as we are going to define it, will not depend on how the rules are ordered. This fact reflects a property of CLINGO: if we change the order of rules of a CLINGO program, its stable models will remain the same, although the runtime may change, as well as the order in which the stable models are generated. It contrasts with properties of sequential composition in imperative programming, where the order of components (for instance, assignments) is essential. The order of rules in a CLINGO program is only important to humans who read the program: like the availability of comments, it may affect our ability to understand the program's design.

Furthermore, a model of a propositional program is a *set* of atoms, not a list. The order in which the elements of a stable model are displayed by an answer set solver has no special significance. It may change from one version of the system to another.

4.2 Equivalence

Two formulas or sets of formulas are *equivalent* to each other if they have the same models. For instance, the set $\{p, q \leftarrow p\}$ is equivalent to the formula $p \wedge q$.

When two formulas or sets of formulas are not equivalent, this fact can be demonstrated by a counterexample—an interpretation that satisfies one of them but not the other. For instance, the formula $p \wedge q \wedge r$ is not equivalent to $p \wedge q$ because the interpretation $\{p, q\}$ satisfies the latter but not the former.

Exercise 4.7. Determine whether the given formulas or sets of formulas are equivalent. If they are not, give a counterexample.

(a) $\{p \leftarrow q,\ q \leftarrow r\}$ and $p \leftarrow r$.

(b) $p \wedge q \leftarrow r$ and $\{p \leftarrow r,\ q \leftarrow r\}$.

(c) $p \leftarrow q \vee r$ and $\{p \leftarrow q,\ p \leftarrow r\}$.

Formula	$F \wedge \top$	$F \wedge \bot$	$F \vee \top$	$F \vee \bot$
Simplification	F	$\bot$	$\top$	F

Formula	$\top \leftarrow F$	$\bot \leftarrow F$	$F \leftarrow \top$	$F \leftarrow \bot$
Simplification	$\top$	$\neg F$	F	$\top$

Set of formulas	$\Gamma \cup \{\top\}$	$\Gamma \cup \{\bot\}$
Simplification	Γ	$\bot$

Table 4.1: Simplifying formulas and sets of formulas containing $\top$ and $\bot$.

(d) $p \leftarrow p$ and $q \vee \neg q$.

(e) $p \leftarrow \neg q$ and $q \leftarrow \neg p$.

To simplify a propositional formula F means to find a propositional formula that is simpler than F and equivalent to F. (This terminology is useful, but it is not mathematically precise, because we did not define "simpler.") For instance, we can say that p is the result of simplifying $p \wedge p$. In a similar way, we can talk about simplifying a set of formulas. The art of simplifying formulas is an important part of logic, like the art of simplifying algebraic expressions in algebra, or simplifying trigonometric expressions in trigonometry. Several ways to simplify formulas and sets of formulas containing $\top$ and $\bot$ are shown in Table 4.1.

Exercise 4.8. Use simplification steps from Table 4.1 to simplify the given formulas.

(a) $p \vee q \leftarrow r \wedge \bot$.

(b) $p \vee q \leftarrow r \vee \top$.

(c) $p \vee \bot \leftarrow q \wedge \top$.

A set of formulas containing an atomic formula p can be simplified by replacing p with $\top$ in the other formulas that contain p, and then simplifying each of these formulas. Similarly, if a set of formulas contains $\neg p$ then it can be simplified by replacing p in other formulas by $\bot$.

Exercise 4.9. Simplify each of the given sets of formulas.

(a) $\{p,\ p \vee q \leftarrow r\}$.

(b) $\{\neg p,\ p \vee q \leftarrow r\}$.

A *tautology* is a formula that is satisfied by all interpretations. In other words, a tautology is a formula equivalent to $\top$. If a set of formulas contains a tautology then it can be simplified by removing it.

Exercise 4.10. Determine which of the given formulas are tautologies.

(a) $p \vee \top \leftarrow \neg q$.

(b) $p \leftarrow q \wedge r \wedge \neg q$.

A formula of the form

$$(F \leftarrow G) \wedge (G \leftarrow F)$$

can be abbreviated as $F \leftrightarrow G$ ("F if and only if G"). This formula is a tautology if and only if F is equivalent to G.

Exercise 4.11. Simplify the formula $p \leftrightarrow (p \wedge q)$.

If a set Γ of formulas contains a formula of the form $F \leftrightarrow G$ then replacing F by G or G by F in any other formula from Γ gives a set of formulas that is equivalent to Γ. Sometimes this fact can be used for simplification.

Exercise 4.12. Simplify the set $\{p \leftrightarrow q,\ p \leftarrow q \wedge r\}$.

If Γ is a finite set of formulas then we can form the conjunction and the disjunction of all elements of Γ. We will use this terminology even when Γ is empty; the conjunction of the empty set of formulas is understood as $\top$, and the disjunction as $\bot$. There is a good reason for this convention. For any list $F_1, \dots, F_n$ of formulas of length greater than 1,

$$\begin{array}{ll} F_1 \wedge \cdots \wedge F_n & \text{is equivalent to } (F_1 \wedge \cdots \wedge F_{n-1}) \wedge F_n, \\ F_1 \vee \cdots \vee F_n & \text{is equivalent to } (F_1 \vee \cdots \vee F_{n-1}) \vee F_n. \end{array}$$

If we understand the empty conjunction as $\top$ and the empty disjunction as $\bot$ then these properties will hold for $n = 1$ as well. This is similar to the convention adopted in algebra: the sum of the empty set of numbers is understood as 0, and the product of the empty set of numbers (for instance, 2^0 and $0!$) is understood as 1.

It is clear that any finite set of formulas is equivalent to the conjunction of all its elements.

4.3 Minimal Models

Definition. About a model I of a formula F we say that it is *minimal* if no other model of F is a subset of I.

For instance, if the vocabulary is $\{p, q\}$ then the formula $p \vee q$ has three models:

$$\{p\},\ \{q\},\ \{p, q\}.$$

The first two are minimal, and the third is not.

For sets of formulas, the definition of a minimal model is similar: a model I of Γ is called *minimal* if no other model of Γ is a subset of I.

Exercise 4.13. (a) Assuming that the vocabulary is $\{p, q, r, s\}$, find all models of the program

$$\{p \vee q,\ r \leftarrow p,\ s \leftarrow q\}.$$

(b) Which of them are minimal?

Exercise 4.14. Find all minimal models of the program

$$\{p \leftarrow q,\ q \lor r\}.$$

Note that in the last exercise the vocabulary is not specified. Since the question is about *minimal* models, there is no need to know whether the vocabulary includes any atomic formulas other than p, q, and r: such an atomic formula, if it is available, cannot belong to a minimal model.

Exercise 4.15. Describe (a) the models of the empty set of formulas, (b) the minimal models of the empty set.

Exercise 4.16. For the case when all elements of Γ are atomic formulas, describe (a) the models of Γ, (b) the minimal models of Γ.

Exercise 4.17. Find a propositional rule that has exactly four minimal models.

Exercise 4.18. It is clear that if two formulas or sets of formulas are equivalent then they have the same minimal models. But the converse is not true: two formulas may have the same minimal models even though they are not equivalent to each other. Find such a pair of formulas.

Definition. A propositional rule is *definite* if

(i) its head is the conjunction of a non-empty set of atomic formulas (for instance, a single atomic formula), and

(ii) its body (if it has one) does not contain negation.

A propositional program is *definite* if all its rules are definite.

A definite program has a unique minimal model, and we can construct it by accumulating, step by step, the atomic formulas that need to be included to satisfy all rules of the program. For instance, consider the program

$$p, \tag{4.2}$$
$$q \leftarrow p \land r, \tag{4.3}$$
$$r \leftarrow p \lor t, \tag{4.4}$$
$$s \leftarrow r \land t. \tag{4.5}$$

Which atomic formulas need to be included in any model of (4.2)–(4.5)? To satisfy (4.2), we must include p. Once p is included, the body of (4.4) becomes true, and to satisfy that rule we must include r. Now the body of (4.3) became true, and to satisfy that rule we must include q. The set of atomic formulas that we have accumulated, $\{p, q, r\}$, satisfies all formulas (4.2)–(4.5). This is the minimal model of the program.

Exercise 4.19. Describe the step-by-step process of constructing the minimal model of the program

$$\begin{aligned}&p_1 \wedge p_2 \leftarrow q_1 \vee q_2,\\&q_1 \leftarrow r_1 \vee r_2,\\&r_1.\end{aligned}$$

Exercise 4.20. Let Π be the program

$$\{p_1 \leftarrow p_2 \wedge p_3,\ p_2 \leftarrow p_3 \wedge p_4,\ \ldots,\ p_8 \leftarrow p_9 \wedge p_{10}\}.$$

For each of the following programs, describe the step-by-step process of constructing its minimal model.

(a) Π,

(b) $\Pi \cup \{p_5\}$,

(c) $\Pi \cup \{p_5, p_6\}$.

It is clear that condition (i) in the definition of a definite rule is essential: if we drop it then the claim that every definite program has a unique minimal model will become incorrect. Rule $p \vee q$ can serve as a counterexample. Indeed, it satisfies condition (ii), because it has no body, but it has two minimal models.

Exercise 4.21. Condition (ii) in the definition of a definite rule is essential also. Give a counterexample illustrating this fact.

Exercise 4.22. Consider the infinite programs

$$\begin{aligned}\Pi &= \{p_1 \leftarrow p_2,\ p_2 \leftarrow p_3,\ p_3 \leftarrow p_4,\ p_4 \leftarrow p_5, \ldots\},\\ \Sigma &= \{p_2 \leftarrow p_1,\ p_3 \leftarrow p_2,\ p_4 \leftarrow p_3,\ p_5 \leftarrow p_4, \ldots\}.\end{aligned}$$

For each of the following programs, describe the step-by-step process of constructing its minimal model.

(a) $\Pi \cup \{p_3\}$,

(b) $\Sigma \cup \{p_3\}$.

CLINGO rules	Propositional rules
`:-`	$\leftarrow$
comma in the body	$\wedge$
comma in the head	$\vee$
`not`	$\neg$
`#false`	$\bot$
`#true`	$\top$

Table 4.2: Correspondence between symbols in CLINGO rules and in propositional formulas.

4.4 Stable Models of Positive Propositional Programs

About a propositional rule or program we say that it is *positive* if it does not contain negation. For example, all definite programs are positive, as well as the programs from Exercises 4.13 and 4.14.

In application to a positive propositional program, the expression "stable model" has the same meaning as "minimal model." We can say that by doing Exercises 4.13(b), 4.14, 4.19, 4.20, 4.22 we found the stable models of the given propositional programs.

In view of the close relationship between stable models of propositional programs and the functionality of CLINGO, we can use CLINGO to generate the stable/minimal models of positive propositional programs. Table 4.2 relates some of the symbols found in CLINGO programs to the corresponding symbols in propositional programs. Note that the comma sometimes corresponds to conjunction and sometimes to disjunction, depending on where it occurs in the rule. The line that shows how negation is represented in a CLINGO program is not relevant now, because we are talking here about positive programs, but it will be needed in the next chapter.

To find the stable models of the propositional program from Exercise 4.13 using CLINGO, we rewrite the program as

```
p, q.
r :- p.
s :- q.
```

and instruct CLINGO to find its stable models. In the output we will see:

```
Answer: 1
s q
Answer: 2
r p
```

Note that the first rule of the program above is disjunctive—it has two atoms in the head.

According to Table 4.2, the counterpart of the propositional rule $\bot \leftarrow p \wedge q$ in the language of CLINGO is

```
#false :- p, q.
```

In a CLINGO program, this rule has the same meaning as the constraint

```
:- p, q.
```

In other words, adding this rule to a program eliminates its stable models that contain both `p` and `q`.

Rewriting rule (4.4) in the syntax of CLINGO is less straightforward, because the body of this rule is a disjunction, and in the language of CLINGO there is no symbol for disjunction in the body. But this rule can be replaced by an equivalent pair of simpler rules; see Exercise 4.7(c).

Exercise 4.23. (a) Rewrite program (4.2)–(4.5) in the syntax of CLINGO. (b) Use CLINGO to find its stable model.

Exercise 4.24. (a) Do the same for the program from Exercise 4.19.

Exercise 4.25. Use CLINGO to find the number of stable models of the program

$$\begin{aligned}&p \lor q,\\&r \lor s,\\&s_1 \lor s_2 \leftarrow s,\\&\bot \leftarrow p \land s_1.\end{aligned}$$

Answer set solvers can be used to generate arbitrary models of a propositional program, not only minimal/stable models as in the examples above. In other words, they can function as satisfiability solvers. We will talk about it in Section 5.4. The other way around, a satisfiability solver can be sometimes used for generating stable models, as we will see in Section 5.9.

4.5 Propositional Image of a CLINGO Program

As discussed at the end of Section 4.1, our plan is to define what we mean by a stable model of a CLINGO program using the simpler concept of a stable model of a propositional program. We are ready now to do this for CLINGO programs satisfying two conditions. First, we assume that each rule of the program has the form

$$H_1, \dots, H_m \texttt{ :- } B_1, \dots, B_n. \tag{4.6}$$

($m, n \geq 0$) or the simpler form

$$H_1, \dots, H_m. \tag{4.7}$$

($m \geq 1$), where the members $H_1, \dots, H_m$ of the head and the members $B_1, \dots, B_n$ of the body are atoms and comparisons. That means, in particular, that the program does not contain the negation as failure symbol. Second, we assume that the program does not contain arithmetic operations, intervals, pooling, placeholders, and anonymous variables (see Sections 2.1–2.3 and 2.8). Every term occurring in such a program is a symbolic constant, an integer, or a variable. For instance, rule (1.1) is a rule of type (4.6): here

$m = 1$ and H_1 is `large(C)`;
$n = 3$, B_1 is `size(C,S1)`, B_2 is `size(uk,S2)`, and B_3 is `S1 > S2`.

Facts (1.2) are rules of type (4.7) with $m = 1$.

For any program Π consisting of such rules we will describe a positive propositional program called the propositional image of Π. Then we will define the stable models of a program as the stable models of its propositional image. Since the propositional image is a positive propositional program, "stable" in this case means "minimal"(Section 4.4).

The vocabulary of the propositional image is the set of ground atoms of the form $p(v_1, \dots, v_k)$, where p is a symbolic constant, and each v_i is a symbolic constant or an integer. We will denote the set of all symbolic constants by $\mathbf{S}$, and the set of all integers by $\mathbf{Z}$.

Recall that an instance of a rule is any rule that can be obtained from it by substituting constants for all its (global) variables (Section 2.1). Since the set $\mathbf{S} \cup \mathbf{Z}$ is infinite, the set of instances of any rule containing variables is infinite. For example, the instances of rule (1.1) are the rules

$$\texttt{large(}v_0\texttt{) :- size(}v_0\texttt{,}v_1\texttt{), size(uk,}v_2\texttt{), } v_1 \texttt{ > } v_2\texttt{.} \tag{4.8}$$

for all v_0, v_1, v_2 from $\mathbf{S} \cup \mathbf{Z}$.

Definition. The *propositional image* of a program consists of the instances of its rules transformed into propositional formulas

(i) by replacing the symbol `:-` and all commas in the head and the body by propositional connectives as shown in Table 4.2, and dropping the period at the end of the rule;

(ii) by replacing each comparison $t_1 \prec t_2$ in the head and in the body by $\top$ if it is true, and by $\bot$ if it is false;

(iii) by replacing the head of the rule by $\bot$ if it is empty, and replacing the body by $\top$ if it is empty.

Note that this definition is limited to CLINGO programs satisfying the two conditions stated at the beginning of this section. A more general definition of the propositional image will be given in Section 4.7. Then it will be generalized again in Sections 5.6 and 5.7, and at that point we will see that the propositional images of some CLINGO programs are not positive.

Definition. A *stable model* of a CLINGO program is a stable model of its propositional image.

For instance, the propositional images of facts (1.2) are the atomic formulas

$$size(france, 65),\ \ size(germany, 83),\ \ size(italy, 61),\ \ size(uk, 64). \tag{4.9}$$

Rule (4.8), transformed into a formula as described above, is

$$large(v_0) \leftarrow size(v_0, v_1) \wedge size(uk, v_2) \wedge \top \tag{4.10}$$

if $v_1 > v_2$, and

$$large(v_0) \leftarrow size(v_0, v_1) \wedge size(uk, v_2) \wedge \bot \tag{4.11}$$

otherwise. Consequently the propositional image of CLINGO program (1.1), (1.2) consists of propositional rules (4.9)–(4.11) for arbitrary v_0, v_1, v_2 from $\mathbf{S} \cup \mathbf{Z}$.

This program is definite (Section 4.3), so that is has a unique minimal model, and that model can be constructed by accumulating the atoms that need to be included to satisfy all its rules. Before doing this calculation, we will simplify the program. Formulas (4.11) are tautologies and can be dropped, and in formula (4.10) the conjunctive term $\top$ can be dropped. Consequently program (4.9)–(4.11) is equivalent to the propositional program consisting of rules (4.9) and the rules

$$large(v_0) \leftarrow size(v_0, v_1) \wedge size(uk, v_2) \qquad (v_0, v_1, v_2 \in \mathbf{S} \cup \mathbf{Z};\;\; v_1 > v_2). \tag{4.12}$$

To satisfy these rules, we need to include, first of all, atomic formulas (4.9). After that, among the infinitely many rules (4.12) there will be two that are not satisfied:

$$large(france) \leftarrow size(france, 65) \wedge size(uk, 64)$$

and

$$large(germany) \leftarrow size(germany, 83) \wedge size(uk, 64).$$

Once the heads

$$large(france),\;\; large(germany) \tag{4.13}$$

of these rules are added, the construction of the minimal model will be completed.

This calculation justifies the assertion in Section 1.3 that atoms (4.9), (4.13) form the only stable model of CLINGO program (1.1), (1.2).

Exercise 4.26. (a) Find the propositional image of the program

```
p(0,1).
p(1,2).
q(X,Y) :- p(X,Y), X > 0, Y > 0.
```

(b) Simplify it. (c) Describe the step-by-step process of constructing its minimal model. (d) Check whether your answer is in agreement with the output of CLINGO.

Exercise 4.27. (a) Find the propositional image of the program consisting of rules (2.1), (2.2), (2.8) (pages 15 and 22). (b) Describe the step-by-step process of constructing its minimal model.

Exercise 4.28. (a) Find the propositional image of the program

```
p(1), p(2), p(3).
:- p(X), X > 2.
```

(b) Simplify it. (c) Find all its minimal models. (c) Check whether your answer is in agreement with the output of CLINGO.

The above definition of a stable model is applicable even the program is unsafe. For instance, the propositional image of program (2.4) on page 17 consists of the rules

$$\begin{array}{ll} p(v) \leftarrow \top & \text{for all } v \in \mathbf{S} \cup \mathbf{Z} \text{ such that } v > 7, \\ p(v) \leftarrow \bot & \text{for all } v \in \mathbf{S} \cup \mathbf{Z} \text{ such that } v \leq 7. \end{array}$$

Its only minimal model is obviously the set of all atoms $p(v)$ with $v > 7$, as we claimed in Section 2.1.

Exercise 4.29. (a) Find the propositional image of the unsafe program

```
p(a).
q(X,Y) :- p(X).
```

(b) Describe the step-by-step process of constructing its minimal model.

4.6 Values of a Ground Term

In the next section, the definition of the propositional image is extended to programs that may contain arithmetic operations and intervals. We will see there that in this more general setting the process of constructing the propositional image includes an additional step—replacing terms by their values. For instance, the propositional image of the fact

```
p(2*2).
```

is the atomic formula $p(4)$.

But first we need to talk about values of ground terms that may occur in such programs. The exact meaning of this concept is not so easy to define, for several reasons. First, a ground term may have many values; for instance, the values of `1..3` are `1`, `2`, `3`. Second, a ground term may have no values: for instance, `1/0` has no values, and the term `abracadabra+7` has no values either. Third, as we saw in Section 3.10, the language of CLINGO allows us to apply arithmetic operations to intervals. In fact, when we construct a ground term from constants, applying artithmetic operations and forming intervals may follow each other any number of times in any order.

We can clarify this issue by defining the set of *values* of a ground term t that does not contain placeholders. The definition is recursive, and it consists of 4 clauses:

Definition.

1. If t is a symbolic constant or an integer then the only value of t is t itself.

CLINGO terms	Algebraic notation
m`*`n	$m \cdot n$
m/n	$\lfloor m/n \rfloor$
$m\backslash n$	$m - n \cdot \lfloor m/n \rfloor$
m`**`n	$\lfloor m^n \rfloor$

Table 4.3: Correspondence between symbols for arithmetic operations in CLINGO terms and in standard algebraic notation. The symbol $\lfloor x \rfloor$ denotes the floor of a real number x, that is, the largest integer less than or equal to x. The use of this symbol in the last line is needed because n can be negative; then m^n is a fraction.

2. If t is $t_1 \circ t_2$, where $\circ$ is an arithmetic operation, then the values of t are integers of the form $n_1 \circ n_2$, where the integer n_1 is a value of t_1, and the integer n_2 is a value of t_2. (Table 4.3 shows how to translate symbols for arithmetic operations in CLINGO into standard algebraic notation.)

3. If t is $|t_1|$ then the values of t are integers of the form $|n_1|$, where the integer n_1 is a value of t_1.

4. If t is $t_1..t_2$ then the values of t are the integers n for which there exist integers n_1 and n_2 such that

- n_1 is a value of t_1,
- n_2 is a value of t_2,
- $n_1 \leq n \leq n_2$.

For instance, the only value of `2*2` is `4`, because the only value of `2` is `2`, and $2 \cdot 2 = 4$. The set of values of `2/0` is empty, because division by 0 is undefined. The set of values of `2*a` is empty as well, because the only value of the symbolic constant `a` is not an integer. The same goes for `2..a`.

It is clear that every value of a ground term is either a symbolic constant (if the term itself is a symbolic constant) or an integer.

Exercise 4.30. Determine for which of the following terms the set of values is empty.

(a) `6..5`.

(b) `a..(a+1)`.

(c) `2**(-2)`.

Exercise 4.31. Find all values of the term `(2..4)*(2..4)`.

Exercise 4.32. Find a ground term (a) with the values `1`, `3`, `9`; (b) with the values `22`, `32`, `42`.

Expression	Propositional image
atom $p(t_1,\dots,t_k)$ in the head	conjunction of all formulas of the form $p(v_1,\dots,v_k)$ where v_i is a value of t_i $(i=1,\dots,k)$
atom $p(t_1,\dots,t_k)$ in the body	disjunction of all formulas of the form $p(v_1,\dots,v_k)$ where v_i is a value of t_i $(i=1,\dots,k)$
comparison $t_1 \prec t_2$ in the head	$\top$ if for every value v_1 of t_1 and every value v_2 of t_2, $v_1 \prec v_2$; $\bot$ otherwise
comparison $t_1 \prec t_2$ in the body	$\top$ if for some value v_1 of t_1 and some value v_2 of t_2, $v_1 \prec v_2$; $\bot$ otherwise

Table 4.4: Propositional images of ground atoms and comparisons. Here p is a symbolic constant, each t_i is a ground term, and $\prec$ is a comparison symbol.

4.7 More on Propositional Images

We will define now the concept of a stable model for CLINGO programs consisting of rules of forms (4.6) and (4.7) that may contain arithmetic operations and intervals (but no pooling, placeholders, or anonymous variables). The *propositional image* of such a program is formed by the process described in Section 4.5, with clause (ii) modified as follows:

(ii′) by replacing each atom and each comparison in every rule by its propositional image as described in Table 4.4.

According to Table 4.4, the same expression may correspond to different formulas depending on whether it occurs in the head or in the body. For instance,

- the propositional image of the atom `p(1..2)` in the head of a rule is the conjunction $p(1) \wedge p(2)$, but the propositional image of the same atom in the body of a rule is the disjunction $p(1) \vee p(2)$;
- the propositional image of the atom `p(1/0)` in the head is the empty conjunction $\top$, but the propositional image of the same atom in the body is the empty disjunction $\bot$ (see the discussion of the empty conjunction and empty disjunction at the end of Section 4.2);
- the propositional image of the comparison `1..2 = 2..3` in the head is $\bot$ (because it is not true that every element of one of the sets $\{1,2\}$, $\{2,3\}$ is an element of the other), but the propositional image of the same comparison in the body is $\top$ (because sets $\{1,2\}$ and $\{2,3\}$ have a common element).

However, if each of the terms $t_1,\dots,t_k$ is a symbolic constant or an integer, as in Section 4.5, then the only value of t_i is t_i itself, so that the conjunction in the first line of Table 4.4 is the single atom $p(t_1,\dots,t_k)$, and the disjunction in the second line of the table is the same atom. Similarly, if each of the terms t_1, t_2 is a symbolic constant or an integer then the truth value described in the third line of the table is $\top$ or $\bot$ depending on whether the condition $t_1 \prec t_2$ is true or false, and the value in the fourth line is calculated in the same way. We conclude that the definition of the propositional image above has the same meaning as the definition in Section 4.5 when the program contains neither arithmetical operations nor intervals.

If each of the terms $t_1, \dots, t_k$ is formed from integers using addition, subtraction, and multiplication, then t_i has a unique value v_i, so that the conjunction in the first line of Table 4.4 is the atom $p(v_1, \dots, v_k)$, and so is the disjunction in the second line. For instance, the propositional image of `p(2*2)` is $p(4)$ no matter whether this atom occurs in the head or in the body.

Consider, for instance, one-rule program (2.7) (page 20). Its propositional image consists of the rules

$$\begin{array}{rcll} p(0,41) & \leftarrow & \top, & \\ p(1,43) & \leftarrow & \top, & \\ p(2,47) & \leftarrow & \top, & \\ p(3,53) & \leftarrow & \top, & \\ p(n,n^2+n+41) & \leftarrow & \bot & \text{for all } n \in \mathbf{Z} \setminus \{0,\dots,3\}, \\ \top & \leftarrow & \bot. & \end{array} \tag{4.14}$$

(The last formula corresponds to the instances of (2.7) obtained by substituting symbolic constants for `N`.) The rules in the last two lines are tautologies, and the other rules can be equivalently written as

$$p(0,41),\ p(1,43),\ p(2,47),\ p(3,53).$$

Consequently these atoms form the only stable model of (2.7), in accordance with the claim about this program made in Section 2.3.

The propositional image of the program

```
p(2).
p(a).
q(X+1) :- p(X).
```

consists of the formulas

$$\begin{array}{ll} p(2), & \\ p(a), & \\ q(m) \leftarrow p(n) & \text{for all } n \in \mathbf{Z},\ \text{where } m \text{ is the value of } n+1, \\ \top \leftarrow q(v) & \text{for all } v \in \mathbf{S}. \end{array}$$

The rules in the last line are tautologies and can be dropped. The stable model of the program is $\{p(2), p(a), q(3)\}$.

Exercise 4.33. For each of the given rules, find its propositional image. Which of the propositional images are tautologies?

(a) `square(1..2,1..2).`

(b) `q :- square(1..2,1..2).`

(c) `square(abra..cadabra,abra..cadabra).`

(d) `q :- square(abra..cadabra,abra..cadabra).`

Exercise 4.34. (a) Find the propositional image of the program

```
p(1).
q :- p(1..3).
```

(b) Find the minimal model of the propositional image. (c) Check whether your answer is in agreement with the output of CLINGO.

Exercise 4.35. Do the same for the one-rule program

```
p(1/N) :- N = 0..1.
```

Exercise 4.36. Find the propositional image of the program

```
p(1..3).
q(X) :- p(X), X = 2..4.
```

(b) Simplify it. (c) Describe the step-by-step process of constructing its minimal model. (d) Check whether your answer is in agreement with the output of CLINGO.

Exercise 4.37. Do the same for the program

```
p(1,1..2).
q(X,Y) :- p(X,Y), X != Y.
q(X,Y) :- q(Y,X).
```

Exercise 4.38. Do the same for the program

```
p(1..3).
q(N-1..N+1) :- p(N).
```

Exercise 4.39. (a) Find the propositional image of the program

```
p(1..3).
X = 1 :- p(X).
```

(b) Find all its minimal models.

Exercise 4.40. (a) Find the propositional image of the unsafe program

```
p(1).
q(X) :- p((-1)**X).
```

(b) Describe the step-by-step process of constructing its minimal model.

We have seen that the mathematical definition of a stable model is applicable to unsafe programs, such as (2.4) and the program from Exercise 4.40. It is applicable also to safe programs for which grounding does not terminate, such as program (3.1) on page 43. The propositional image of that program consists of the rules

$$\begin{array}{ll} p(0), & \\ p(m) \leftarrow p(n) & \text{for all } n \in \mathbf{Z},\ \text{where } m \text{ is the value of } n+2, \\ \top \leftarrow p(v) & \text{for all } v \in \mathbf{S}. \end{array}$$

The rules in the last line are tautologies and can be dropped. The result is a definite program, and the process of constructing its minimal model involves infinitely many steps: we include $p(0)$, then add $p(2)$, then add $p(4)$, and so on. This is similar to what we saw in Exercise 4.22(b) on page 67.

Programs (2.4) and (3.1) are similar to each other in the sense that each of them has a unique infinite stable model, but CLINGO responds to these programs in different ways: it rejects the former as unsafe, but it tries—unsuccessfully—to ground the latter. The difference can be explained in terms of the process of accumulating the ground atoms that need to be included in the minimal model of the propositional image. With (2.4) and similar unsafe programs, the set of accumulated atoms becomes infinite after several steps. In case of (3.1), the set of accumulated atoms remains finite at every step, and the minimal model turns out to be infinite only because the number of steps is infinite. Such programs are treated by CLINGO as safe.

4.8 Bibliographical and Historical Remarks

The use of truth tables for defining the semantics of propositional connectives was invented in the 1920s [105, 119]. The truth tables given in Section 4.1 are considered now standard, or "classical" (except that we use here the left arrow for implication).

The truth table for implication may look puzzling: why is the value *false* assigned to $F \leftarrow G$ when the value of F is *false* and the value of G is *true*, whereas the value *true* appears in the other three lines of the table? This choice can be justified by the fact that in mathematics, to give a counterexample to a claim of the form "F if G" we need to make sure that F is false and G is true; we need to show, in other words, that this implication is in the "false" line of the truth table. This semantics makes implication similar to disjunction—the other connective that has *false* in one line of the truth table out of four. According to the classical semantics, the two connectives can be expressed in terms of each other using negation:

$$\begin{array}{rcl} F \leftarrow G & \text{is equivalent to} & F \vee \neg G, \\ F \vee G & \text{is equivalent to} & F \leftarrow \neg G. \end{array}$$

The classical truth table for implication is the best possible choice as long as we intend to characterize the meaning of implication in terms of two truth values. But it describes only one of many meanings of the word “if” [61].

Logical minimization of the kind described in this chapter plays an important part in the theory of “nonmonotonic reasoning.” This term describes the situations when reasoners draw tentative conclusions that may be retracted on the basis of further evidence. Reasoning found in mathematical proofs is monotonic: when a theorem is derived from a set of axioms, adding more axioms will not invalidate the proof. (This is so even if extending the set of axioms makes it inconsistent: in this case, every proposition becomes a theorem, including the theorems proved earlier.) But commonsense reasoning, practiced in everyday life, is often nonmonotonic. For example, we often deal with lists that are presumed to be complete. Conclusions based on such an assumption may need to be retracted when we get additional information. Something like this happened when Prolog and CLINGO told us, in response to rules (1.1), (1.2), that in 2015 there were two countries in Europe populated by more people than the United Kingdom. We realized that the number of such countries is actually three when we took into account the fact that Russia is a European country (Exercise 2.1 on page 15).

Theory of nonmonotonic reasoning became an active area of research after the publication of the special issue of the journal *Artificial Intelligence* on nonmonotonic reasoning in 1980 [2]. One of the mathematical models of nonmonotonic reasoning described in that journal used the syntactic transformation called *circumscription* [90, 91]. To give an example, the result of applying this transformation to the formula

$$p(a) \wedge p(b) \wedge q(c) \tag{4.15}$$

is a formula that can be equivalently written as the first-order sentence

$$\forall X(p(X) \leftrightarrow X = a \vee X = b) \wedge \forall X(q(X) \leftrightarrow X = c). \tag{4.16}$$

Formula (4.15) expresses that a and b are elements of the set p, and c is an element of q; the result (4.16) of “circumscribing p and q” in that formula expresses that these lists of elements are complete, so that a and b are the only elements of p, and c is the only element of q. This is similar to what happens when we apply the definition of the minimal model given in this chapter to formula (4.15): in the minimal model

$$\{p(a), p(b), q(c)\}$$

of that formula, the atoms $p(c)$, $q(a)$ and $q(b)$ are false, so that c does not belong to p, and a, b do not belong to q.

Interaction between the semantics of logic programs and the theory of nonmonotonic reasoning is the theme of biannual international conferences on logic programming and nonmonotonic reasoning that have been held in America and Europe since 1991. We will return to this subject in connection with the semantics of negation as failure in the next chapter, and in connection with the frame problem in Chapter 7.

The discussion of ground terms in Section 4.6 follows the account of the semantics of CLINGO published in 2015 [43]. The explanation of the difference between two kinds of programs with infinite stable models is based on the theory of safe rules proposed a year later [80].

Chapter 5

Programs with Negation

Our next goal is to extend the definition of a stable model from Chapter 4 to propositional programs that contain negation. We begin with an informal discussion of a few examples.

5.1 Examples

The program

$$\begin{aligned} &p, \\ &q, \\ &r \leftarrow p, \\ &s \leftarrow q \end{aligned}$$

is positive definite, and its stable model can be formed by accumulating the atomic formulas that are needed to satisfy all rules: we include p and q to satisfy the first two rules, and then add r and s to satisfy the other two. The stable model is $\{p, q, r, s\}$. Consider the modification of that program in which the conjunctive term $\neg s$ is added to the body of the third rule:

$$p, \tag{5.1}$$
$$q, \tag{5.2}$$
$$r \leftarrow p \wedge \neg s, \tag{5.3}$$
$$s \leftarrow q. \tag{5.4}$$

We think of this conjunctive term as a restriction on the process of accumulating atomic formulas in the process of constructing the stable model. Rule (5.3) instructs us to add r to the model if p is included under the condition that

s is not included in the model and will not be included in the future. (5.5)

Since the program contains rules (5.2) and (5.4), the model that we are building will include s, so that the conjunctive term $\neg s$ "disables" the third rule of the program. Since r

is not added in the process of accumulating atomic formulas, the stable model of program (5.1)–(5.4) is $\{p, q, s\}$.

Consider now the program consisting of only three rules (5.1)–(5.3). Since the heads of the rules of this program do not contain s, condition (5.5) is satisfied, and, in accordance with rule (5.3), we add r. The stable model of program (5.1)–(5.3) is $\{p, q, r\}$.

These conclusions about the stable models of programs (5.1)–(5.4) and (5.1)–(5.3) are in agreement with the behavior of CLINGO.

The idea that a negation in the body of a rule represents a restriction that can "disable" the use of that rule for accumulating atomic formulas is related to the expression "negation as failure" (Section 2.4). Condition (5.5) says, for example, that the attempt to justify including s in the model will fail.

Failure conditions of this kind are somewhat vague, because they are circular: which rules are disabled depends on which atomic formulas are going to be included in the model, and the other way around. The definition of a stable model in the next section makes the idea of negation as failure precise. But there are many cases when the meaning of failure conditions like (5.5) is sufficiently clear for predicting what output is going to be produced by an answer set solver.

Exercise 5.1. (a) Use the process described above to find the stable model of program (5.1), (5.3), (5.4). (b) Check whether your answer is in agreement with the output of CLINGO.

Exercise 5.2. (a) Use the process described above to find the stable model of the program

$$p \leftarrow \neg q, \tag{5.6}$$
$$q \leftarrow \neg r. \tag{5.7}$$

(b) Check whether your answer is in agreement with the output of CLINGO.

The last example we want to look at in this section is the program

$$p \leftarrow \neg q, \tag{5.8}$$
$$q \leftarrow \neg p. \tag{5.9}$$

We saw this program, written in the syntax of CLINGO, in Section 2.9, and asserted there that it has two stable models. That assertion can be explained by saying that there are two ways to decide which of the atomic formulas p, q are included in the process of constructing a stable model. We can include p; then rule (5.9) is disabled, so that q is not included, and rule (5.8) justifies the presence of p in the model. On the other hand, not including p is a reasonable option as well; then rule (5.9) justifies including q, rule (5.8) is disabled, and this fact justifies not including p. So it appears that program (5.8), (5.9) has the stable models $\{p\}$ and $\{q\}$.

In the next section we give a precise definition of the concept of a stable model applicable to all propositional programs, and then we will be able to support this claim by a proof.

5.2 Stable Models of Programs with Negation

A *critical part* of a propositional formula F is a subformula of F that begins with negation but is not part of any other subformula that begins with negation. For instance, the only critical part of rule (5.3) is $\neg s$. Rules (5.1), (5.2), (5.4) have no critical parts, because they do not contain negation. The rule

$$\neg p \leftarrow \neg(q \wedge \neg r)$$

has two critical parts: the head $\neg p$ and the body $\neg(q \wedge \neg r)$. (The subformula $\neg r$ is not critical because it is contained in a larger subformula that begins with negation.)

Definition. Let F be a propositional rule, and let I be an interpretation. The *reduct* of F relative to I is the positive rule obtained from F by substituting $\top$ for each critical part that is satisfied by I, and substituting $\bot$ for every other critical part. The reduct of a propositional program Π relative to I is the positive program obtained from Π by the same process.

Definition. Let Π be a propositional program, and let I be an interpretation. If I is a minimal model of the reduct of Π relative to I then we say that I is a *stable model* of Π.

For instance, let Π be program (5.1)–(5.4), and let I be $\{p, q, s\}$. Since the critical part $\neg s$ of the third rule is not satisfied by I, the reduct is

$$\begin{array}{l} p, \\ q, \\ r \leftarrow p \wedge \bot, \\ s \leftarrow q. \end{array} \tag{5.10}$$

This program is definite, and its only minimal model is $\{p, q, s\}$—exactly the interpretation I that we started with. Consequently I is a stable model of (5.1)–(5.4).

The interpretation $\{p, q\}$ is not a stable model of (5.1)–(5.4). Indeed, the reduct of the program relative to this interpretation is

$$\begin{array}{l} p, \\ q, \\ r \leftarrow p \wedge \top, \\ s \leftarrow q, \end{array} \tag{5.11}$$

and $\{p, q\}$ is not a model of this reduct—it does not satisfy the last two rules.

Interpretation $\{p, q, r, s\}$ is not a stable model of (5.1)–(5.4) either. Indeed, the reduct of the program relative to this interpretation is (5.10), and $\{p, q, r, s\}$ is not minimal among the models of (5.10).

It is easy to show, in fact, that program (5.1)–(5.4) has no stable models other than $\{p, q, s\}$. Take any interpretation I different from $\{p, q, s\}$, and consider two cases. *Case 1:* $s \in I$. The reduct of the program relative to I is (5.10). The only minimal model of the reduct is $\{p, q, s\}$, and it is different from I. Consequently I is not stable. *Case 2:* $s \notin I$.

The reduct of the program relative to I is (5.11), and the only minimal model of the reduct is $\{p, q, r, s\}$. Since this model contains s, it is different from I, so that I is not stable.

Exercise 5.3. (a) Prove that $\{p, q, r\}$ is a stable model of program (5.1)–(5.3). (b) Prove that program (5.1)–(5.3) has no other stable models.

Let us find now all stable models of the program

$$p \vee q, \tag{5.12}$$
$$r \leftarrow \neg p. \tag{5.13}$$

Take an arbitrary interpretation I, and consider two cases. *Case 1:* $p \in I$. The reduct of the program relative to I is

$$\begin{aligned} &p \vee q, \\ &r \leftarrow \bot. \end{aligned}$$

It has two minimal models, $\{p\}$ and $\{q\}$. The former satisfies the condition $p \in I$ that characterizes Case 1, so that it is a stable model of (5.12), (5.13). *Case 2:* $p \notin I$. The reduct of the program relative to I is

$$\begin{aligned} &p \vee q, \\ &r \leftarrow \top. \end{aligned}$$

It has two minimal models, $\{p, r\}$ and $\{q, r\}$. The latter satisfies the condition $p \notin I$ that characterizes Case 2, so that it is a stable model of (5.12), (5.13). We conclude that this program has two stable models, $\{p\}$ and $\{q, r\}$.

Exercise 5.4. (a) Find all stable models of the program

$$p \leftarrow \neg p.$$

(b) Check that the result of your calculation is in agreement with the output of CLINGO.

Exercise 5.5. Do the same for the program

$$\begin{aligned} &p \vee q, \\ &\bot \leftarrow p \wedge \neg q. \end{aligned}$$

Program (5.8), (5.9) contains negations of two atoms, so that four cases need to be considered in the process of calculating its stable models. Take any interpretation I. *Case 1:* $p, q \in I$. The reduct is

$$\begin{aligned} &p \leftarrow \bot, \\ &q \leftarrow \bot. \end{aligned}$$

The minimal model $\emptyset$ does not satisfy the condition characterizing this case. *Case 2:* $p \in I$, $q \notin I$. The reduct is

$$\begin{aligned} p &\leftarrow \top, \\ q &\leftarrow \bot. \end{aligned}$$

The minimal model $\{p\}$ satisfies the condition characterizing this case, so that $\{p\}$ is a stable model of the given program. *Case 3:* $p \notin I$, $q \in I$. A similar calculation gives the stable model $\{q\}$. *Case 4:* $p, q \notin I$. The reduct is

$$\begin{aligned} p &\leftarrow \top, \\ q &\leftarrow \top. \end{aligned}$$

The minimal model $\{p, q\}$ does not satisfy the condition characterizing this case. Consequently the program has two stable models.

Exercise 5.6. Prove that your answer to Exercise 5.2 is correct.

Exercise 5.7. (a) Find all stable models of the program

$$p \leftarrow \neg q, \tag{5.14}$$
$$q \leftarrow \neg p, \tag{5.15}$$
$$r \leftarrow p, \tag{5.16}$$
$$r \leftarrow q. \tag{5.17}$$

(b) Check that the result of your calculation is in agreement with the output of CLINGO.

Exercise 5.8. Do the same for the program

$$p \leftarrow \neg q, \tag{5.18}$$
$$q \leftarrow \neg p, \tag{5.19}$$
$$p \leftarrow q, \tag{5.20}$$
$$q \leftarrow p. \tag{5.21}$$

The process of calculating stable models used in the examples above becomes impractical when the program contains many negated atomic formulas, because the number of reducts of a program grows exponentially with the number of critical parts of its rules. Fortunately, the study of stable models has led to the discovery of theorems that allow us to approach the problem of calculating stable models in other ways. In the next section, for instance, we will see how the stable models of the program from Exercise 5.5 can be calculated without looking at reducts at all. Program completion, defined in Section 5.9, allows us to describe and calculate the stable models of many logic programs without referring to reducts.

The use of the term "stable *model*" is justified by the fact that every stable model of a propositional program is indeed one of its models. In other words, every stable model satisfies all rules of the program. This property of stable models is not immediately obvious: the definition only shows that a stable model I of a program satisfies the *reduct* of the program relative to I. But any stable model of a program satisfies the program itself as well, in view of the following theorem:

Theorem on Reducts. *An interpretation I satisfies a propositional program Π if and only if it satisfies the reduct of Π relative to I.*

For example, the interpretation $\{q\}$ satisfies both the one-rule program $p \leftarrow \neg q$ and its reduct $p \leftarrow \bot$ (which is a tautology); the interpretation $\emptyset$ satisfies neither $p \leftarrow \neg q$ nor its reduct $p \leftarrow \top$.

From Theorem on Reducts we can conclude that the stable models of a propositional program can be characterized as follows: an interpretation I is a stable model of Π if and only if

(i) I is a model of Π, and

(ii) no proper subset of I is a model of the reduct of Π relative to I.

It is clear that in the special case when the given program is positive, its stable models in the sense of the definition above are exactly its minimal models. This is because the reduct of a positive program Π relative to any interpretation is Π itself, so that the condition "I is a minimal model of the reduct of Π relative to I" means simply that I is a minimal model of Π. This observation justifies the claim in Section 4.4 that in application to models of a positive propositional program, "stable" has the same meaning as "minimal."

5.3 Stable Models as Fixpoints

In this section we talk about propositional programs satisfying the following condition:

(α) the head of every rule of the program is an atomic formula.

Stable models of such programs are related to the concept of a fixpoint. A *fixpoint* of a function is an element of the function's domain that is mapped by the function to itself. In other words, fixpoints of a function f are solutions of the equation $f(x) = x$. For example, 2 is a fixpoint of the function $f(x) = 3x - 4$, because it is a solution of the equation $3x - 4 = x$. The function $f(x) = x^2$ has two fixpoints, 0 and 1.

The concept of a fixpoint is not limited to functions that operate on numbers. Consider, for instance, the function f with the domain consisting of all subsets of the set $\{a, b, c\}$, which is defined by the equation

$$f(x) = x \cup \{a, b\}. \tag{5.22}$$

Table 5.1 shows the values of $f(x)$ for all values of x. It is clear that this function has two fixpoints.

x	$f(x)$	fixpoint?
$\emptyset$	$\{a,b\}$	
$\{a\}$	$\{a,b\}$	
$\{b\}$	$\{a,b\}$	
$\{c\}$	$\{a,b,c\}$	
$\{a,b\}$	$\{a,b\}$	✓
$\{a,c\}$	$\{a,b,c\}$	
$\{b,c\}$	$\{a,b,c\}$	
$\{a,b,c\}$	$\{a,b.c\}$	✓

Table 5.1: Fixpoints of function (5.22).

I	reduct relative to I	$s(I)$	fixpoint?
$\emptyset$	$\{p \leftarrow \top,\ q \leftarrow \top,\ r \leftarrow p,\ r \leftarrow q\}$	$\{p,q,r\}$	
$\{p\}$	$\{p \leftarrow \top,\ q \leftarrow \bot,\ r \leftarrow p,\ r \leftarrow q\}$	$\{p,r\}$	
$\{q\}$	$\{p \leftarrow \bot,\ q \leftarrow \top,\ r \leftarrow p,\ r \leftarrow q\}$	$\{q,r\}$	
$\{r\}$	$\{p \leftarrow \top,\ q \leftarrow \top,\ r \leftarrow p,\ r \leftarrow q\}$	$\{p,q,r\}$	
$\{p,q\}$	$\{p \leftarrow \bot,\ q \leftarrow \bot,\ r \leftarrow p,\ r \leftarrow q\}$	$\emptyset$	
$\{p,r\}$	$\{p \leftarrow \top,\ q \leftarrow \bot,\ r \leftarrow p,\ r \leftarrow q\}$	$\{p,r\}$	✓
$\{q,r\}$	$\{p \leftarrow \bot,\ q \leftarrow \top,\ r \leftarrow p,\ r \leftarrow q\}$	$\{q,r\}$	✓
$\{p,q,r\}$	$\{p \leftarrow \bot,\ q \leftarrow \bot,\ r \leftarrow p,\ r \leftarrow q\}$	$\emptyset$	

Table 5.2: Fixpoints of the stability operator s of the program from Exercise 5.7.

If a propositional program Π satisfies condition (α) then its reduct relative to any interpretation is a definite program (Section 4.3), so that it has a unique minimal model. We define the *stability operator* of Π as the function that maps every interpretation I to the minimal model of the reduct of I relative to Π. It is clear that the stable models of a program with property (α) can be characterized as the fixpoints of its stability operator.

Table (5.2), for instance, shows the values of the stability operator of the program from Exercise 5.7 and its fixpoints—the stable models of the program.

Exercise 5.9. Make the table of values of the stability operator of the one rule program

$$q \leftarrow \neg p$$

and mark its fixpoints.

Exercise 5.10. Describe the stability operator of program (4.2)–(4.5) on page 66.

5.4 Excluded Middle and Double Negation

The one rule program

$$p \vee \neg p \tag{5.23}$$

has two stable models, $\emptyset$ and $\{p\}$. Indeed, the reduct of (5.23) relative to $\emptyset$ is $p \vee \top$, which is equivalent to $\top$; the minimal model of the reduct is empty. The reduct of (5.23) relative to $\{p\}$ is $p \vee \bot$, which is equivalent to p; the minimal model of the reduct is $\{p\}$.

In rule (5.23), negation occurs in the head, and not in the body as in the examples that we have seen before. CLINGO does not object against negation in the head. It will accept program (5.23) if we rewrite it as

```
p, not p.
```

and it will produce two stable models for it, in accordance with the calculation above. In other words, CLINGO views this rule as synonymous to the choice rule `{p}`.

Exercise 5.11. (a) Find all stable models of the program

$$\begin{array}{l} p \vee q, \\ r \vee \neg s \leftarrow p. \end{array}$$

(b) Check that the result of your calculation is in agreement with the output of CLINGO.

Formula (5.23) is known in logic as *the law of excluded middle.* (Any proposition p is either true or false, there is nothing in the middle.) This formula plays an important role in the theory of stable models because of its close relation to choice rules, and also because of the following fact:

Theorem on Excluded Middle. *An interpretation I is a model of a propositional program Π if and only if I is a stable model of the program obtained from Π by adding the rules $p \vee \neg p$ for all atomic formulas p.*

For example, an interpretation I is a model of the propositional program from Exercise 4.3 (page 62) if and only if it is a stable model of the program obtained from it by adding the rules

$$p \vee \neg p,\ q \vee \neg q,\ r \vee \neg r.$$

That means that the question from Exercise 4.3 can be answered by running CLINGO on the program

```
p :- q, r.
q :- p.
r :- p.
p, not p.
q, not q.
r, not r.
```

I	reduct relative to I	$s(I)$	fixpoint?
$\emptyset$	$\{p \leftarrow \bot\}$	$\emptyset$	✓
$\{p\}$	$\{p \leftarrow \top\}$	$\{p\}$	✓

Table 5.3: Fixpoints of the stability operator s of program (5.24).

We will see in Section 5.7 that the choice rule

```
{p; q; r}.
```

has the same meaning as the last three rules of the program above. Consequently it can be used instead of those rules.

In the special case when Π is empty, Theorem on Excluded Middle describes the stable models of a set of rules the form $p \vee \neg p$: they are arbitrary subsets of the set of atomic propositions occurring in the rules.

Exercise 5.12. Use CLINGO to verify your answers to Exercises 4.3, 4.4, 4.5(a), 4.5(b), 4.13(a).

The formula

$$p \leftarrow \neg\neg p \tag{5.24}$$

is known as *the law of double negation*. From Table 5.3 we see that it has the same stable models as the law of excluded middle: $\emptyset$ and $\{p\}$. Thus there is an essential difference between rule (5.24) and the positive rule $p \leftarrow p$, which is obtained from it by dropping the double negation. The latter has only one stable model—the minimal model $\emptyset$. The fact that an equivalent transformation, such as dropping a double negation in a propositional rule, may affect the set of stable models of a program is important. When we want to find models, or minimal models, of a set of formulas, we often start by simplifying it—replacing it by an equivalent set of formulas that is simpler. But if our goal is to find *stable* models of a propositional program then simplifying the program may lead to a wrong answer. Some transformations of rules are "strongly" equivalent, in the sense that they do not affect the set of stable models; but dropping a double negation in front of an atomic formula is not one of them. (Simplifying the reduct before calculating its minimal models is permissible, of course; we are talking here about simplifying the program itself.)

It is often useful to know which equivalent transformations of propositional rules are strongly equivalent, and which are not, and we will talk about this in Section 5.8. For the time being, it is sufficient to remember two facts. First, the set of stable models of a program is not affected by the simplifications shown in Table 4.1 (page 64). Second, the set of stable models does not change if we break a rule with a conjunction in the head into several rules, corresponding to the conjunctive terms. For instance, replacing $p \wedge q \wedge r$ by three rules p, q, r is a strongly equivalent transformation, and so is replacing $p \wedge q \wedge r \leftarrow s$ by

$$p \leftarrow s,\ q \leftarrow s,\ r \leftarrow s.$$

The input language of CLINGO allows rules with two negations in a row. It will accept, for instance, program (5.24) if we rewrite it as

```
p :- not not p.
```

and it will produce the two stable models shown in Table 5.3.

The designers of CLINGO decided, however, against permitting *three* negations in a row. There is a good reason for that, as we will see in Section 5.8.

5.5 Theorem on Constraints

Recall that a rule in the input language of CLINGO is called a constraint if its head is empty (Section 2.7). By adding a constraint to a CLINGO program we eliminate its stable models that violate the constraint. This useful observation is somewhat vague, because we have not defined the precise meaning of "violate."

In the world of propositional programs, a *constraint* is a propositional rule of the form $\bot \leftarrow F$. The propositional image of a CLINGO constraint, as defined in Sections 4.5, 4.7, is a constraint in the sense of this definition, because the empty disjunction is $\bot$ (Section 4.2).

The vague observation on the effect of constraints on the behavior of CLINGO turns, in the world of propositional programs, into a theorem:

Theorem on Constraints. *Let Π be a propositional program, and let Σ be a set of constraints. An interpretation I is a stable model of $\Pi \cup \Sigma$ if and only if I is a stable model of Π and a model of Σ.*

This theorem shows, for instance, that an interpretation I is a stable model of the program from Exercise 5.5 if and only if it is a stable model of its first rule that does not satisfy the body $p \wedge \neg q$ of the constraint. The first rule is positive, and its stable models are its minimal models $\{p\}$, $\{q\}$. The first of them satisfies $p \wedge \neg q$, and the second does not. We can conclude that $\{q\}$ is the only stable model of the program.

Exercise 5.13. (a) Use Theorem on Constraints to find the stable models of the program

$$\begin{aligned} &p \vee q, \\ &r \leftarrow p, \\ &s \leftarrow q, \\ &\bot \leftarrow \neg p, \\ &\bot \leftarrow \neg r \wedge \neg s. \end{aligned}$$

(b) Check whether your answer is in agreement with the output of CLINGO.

In the special case when program Π is definite (Section 4.3), Theorem on Constraints shows that the result of adding a constraint $\bot \leftarrow F$ to Π will have either no stable models or the same stable model as Π, depending on whether the stable model of Π satisfies F. We

Expression	Propositional image
`not` $p(t_1,\dots,t_k)$ in the head	conjunction of all formulas of the form $\neg p(v_1,\dots,v_k)$ where v_i is a value of t_i $(i=1,\dots,k)$
`not` $p(t_1,\dots,t_k)$ in the body	disjunction of all formulas of the form $\neg p(v_1,\dots,v_k)$ where v_i is a value of t_i $(i=1,\dots,k)$
`not not` $p(t_1,\dots,t_k)$ in the head	conjunction of all formulas of the form $\neg\neg p(v_1,\dots,v_k)$ where v_i is a value of t_i $(i=1,\dots,k)$
`not not` $p(t_1,\dots,t_k)$ in the body	disjunction of all formulas of the form $\neg\neg p(v_1,\dots,v_k)$ where v_i is a value of t_i $(i=1,\dots,k)$

Table 5.4: Propositional images of negated ground atoms.

saw, for example, that the stable model of program (4.2)–(4.5) (page is $\{p, q, r\}$. Adding the constraint $\bot \leftarrow \neg s$ to that program has no stable models; adding the constraint $\bot \leftarrow \neg r$ has the stable model $\{p, q, r\}$.

5.6 CLINGO Programs with Negation

The definition of the propositional image in Sections 4.5 and 4.7 applies to CLINGO programs that consist of rules of forms (4.6) and (4.7), where each H_i and B_j is an atom or a comparison. To extend that definition to the case when H_i and B_j can be negated atoms, we need to extend Table 4.4 by lines showing how to form the propositional images of negated atoms. These additional lines are shown in Table 5.4. In this more general setting, the propositional image is a propositional program that may contain negation, and stable models of such programs are defined in Section 5.2. Thus we have now a precise definition of a stable model for sets of rules (4.6) and (4.7) in which every H_i and B_j is an atom, a comparison, or an atom prefixed with one or two negations.

Let us check, for example, that

$$\{p(a), q(a)\} \tag{5.25}$$

is a stable model of the program

```
p(a).
q(a).                                   (5.26)
r(X) :- p(X), not q(X).
```

The propositional image Π of this program consists of the rules

$$\begin{array}{ll} p(a), & \\ q(a), & \\ r(v) \leftarrow p(v) \wedge \neg q(v) & \text{for all } v \text{ in } \mathbf{S} \cup \mathbf{Z}. \end{array} \tag{5.27}$$

The reduct of Π relative to (5.25) is

$$\begin{array}{ll} p(a), & \\ q(a), & \\ r(a) \leftarrow p(a) \wedge \bot, & \\ r(v) \leftarrow p(v) \wedge \top & \text{for all } v \text{ in } \mathbf{S} \cup \mathbf{Z} \setminus \{a\}, \end{array}$$

or, equivalently,

$$\begin{array}{ll} p(a), & \\ q(a), & \\ r(v) \leftarrow p(v) & \text{for all } v \text{ in } \mathbf{S} \cup \mathbf{Z} \setminus \{a\}. \end{array}$$

This is a definite program, and (5.25) is its only minimal model.

The calculation above accomplished less than we would like to, in two ways. First, it did not lead us from program (5.26) to its stable model; rather, we verified the *guess* that a certain interpretation is stable. Second, it did not tell us whether the program has stable models other than (5.25). In Section 5.10 we will see how to overcome these limitations.

Exercise 5.14. (a) Find the propositional image of the program

```
p(a).
q(b).
r(X) :- p(X), not q(X).
```

(b) Find the reduct of this propositional image relative to the interpretation

$$\{p(a), q(b), r(a), r(b)\}. \tag{5.28}$$

(c) Simplify the reduct. (d) Find the minimal model of the reduct to determine whether interpretation (5.28) is a stable model of the program.

Exercise 5.15. Do the same for the program

```
p(1).
q :- not p(1..3).
```

and the interpretation

$$\{p(1), q\}. \tag{5.29}$$

Exercise 5.16. Find the propositional images of the rules

(a) `q :- (5/0).`

(b) `q :- not p(5/0).`

(c) `q :- not not p(5/0).`

(d) `(5/0) :- q.`

(e) `not p(5/0) :- q.`

(f) `not not p(5/0) :- q.`

Consider now the program consisting of rules (2.9) and (2.10) on page 23. We will check that the interpretation

$$\{prime(2), prime(3), composite(4), prime(5)\} \tag{5.30}$$

is a stable model of this program.

The propositional image Π of the program consists of the rules

$$\begin{aligned} composite(4) &\leftarrow \top \wedge \top \wedge \top, \\ prime(2) &\leftarrow \top \wedge \neg composite(2), \\ prime(3) &\leftarrow \top \wedge \neg composite(3), \\ prime(4) &\leftarrow \top \wedge \neg composite(4), \\ prime(5) &\leftarrow \top \wedge \neg composite(5) \end{aligned} \tag{5.31}$$

and infinitely many propositional rules containing ⊥ as a conjunctive term in the body. Simplification steps from Table 4.1 turn it into the finite program

$$\begin{aligned} &composite(4), \\ prime(2) &\leftarrow \neg composite(2), \\ prime(3) &\leftarrow \neg composite(3), \\ prime(4) &\leftarrow \neg composite(4), \\ prime(5) &\leftarrow \neg composite(5). \end{aligned} \tag{5.32}$$

The reduct of (5.32) relative to (5.30) consists of the rules

$$\begin{aligned} composite(4) &\leftarrow \top \wedge \top \wedge \top, \\ prime(2) &\leftarrow \top \wedge \top, \\ prime(3) &\leftarrow \top \wedge \top, \\ prime(4) &\leftarrow \top \wedge \bot, \\ prime(5) &\leftarrow \top \wedge \top, \end{aligned}$$

and is equivalent to (5.30). The only minimal model of that set of atoms is (5.30) itself.

Exercise 5.17. (a) Find the propositional image of the program

```
p(1..3).
q(X) :- X = 2..4, not p(X).
```

(b) Simplify the propositional image by steps from Table 4.1. (c) Find the reduct of the simplified propositional image relative to the interpretation

$$\{p(1), p(2), p(3), q(3)\}. \tag{5.33}$$

(d) Simplify the reduct. (e) Find the minimal model of the reduct and determine whether interpretation (5.33) is a stable model of the program.

Exercise 5.18. Do the same for the program

```
p(1..4).
q(X) :- X = 1..4, not p(X**2).
```

and the interpretation

$$\{p(1), p(2), p(3), p(4), q(3), q(4)\}. \tag{5.34}$$

5.7 CLINGO Programs with Choice

Now we will extend the definition of propositional image to CLINGO rules of the forms

$$\texttt{\{}A_1\texttt{;}\dots\texttt{;}A_m\texttt{\}.} \tag{5.35}$$

and

$$\texttt{\{}A_1\texttt{;}\dots\texttt{;}A_m\texttt{\} :- } B_1, \dots, B_n. \tag{5.36}$$

where $A_1, \dots, A_m$ are atoms, and $B_1, \dots, B_n$ are as in the previous section—atoms, possibly prefixed with one or two negations, and comparisons.

As noted in Section 5.4, the law of excluded middle $p \vee \neg p$ has two stable models, $\emptyset$ and $\{p\}$. Consequently it can serve as the propositional image of the choice rule `{p}.` We will apply this idea to arbitrary rules of forms (5.35) and (5.36) using the following notation. For any set Ω of ground atoms, $V(\Omega)$ will stand for the set of all ground atoms $p(v_1, \dots, v_k)$ such that for some atom $p(t_1, \dots, t_k)$ from Ω, $v_1, \dots, v_k$ are values of $t_1, \dots, t_k$, respectively. The propositional image of the head of ground rules (5.35), (5.36) is defined as the conjunction of the formulas $A \vee \neg A$ for all atoms A in $V(\{A_1, \dots, A_m\})$.

For example,

$$V(\{p(1), q(1..3)\}) = \{p(1), q(1), q(2), q(3)\}.$$

It follows that the propositional image of the rule

```
{p(1); q(1..3)} :- not r(5).
```

is

$$(p(1) \vee \neg p(1)) \wedge (q(1) \vee \neg q(1)) \wedge (q(2) \vee \neg q(2)) \wedge (q(3) \vee \neg q(3)) \leftarrow \neg r(5).$$

Exercise 5.19. (a) Find the propositional image of the program

```
{p(a)}.
q(X) :- p(X).
```

(b) Find the reduct of this propositional image relative to the interpretation

$$\{q(a)\}. \tag{5.37}$$

(c) Simplify the reduct. (d) Find the minimal model of the reduct and determine whether interpretation (5.37) is a stable model of the program.

Exercise 5.20. Do the same for the program

```
p(a).
{q(X)} :- p(X).
```

and the interpretation

$$\{p(a)\}. \tag{5.38}$$

In Section 5.4 we asserted that replacing the group of rules

```
p, not p.
q, not q.        (5.39)
r, not r.
```

with the choice rule

```
{p; q; r}.
```

does not affect the stable models of a program. We can now explain why: the propositional image

$$(p \vee \neg p) \wedge (q \vee \neg q) \wedge (r \vee \neg r) \tag{5.40}$$

of the choice rule is the conjunction of the propositional images of rules (5.39).

Now we will extend the definition of the propositional image to choice rules with cardinality bounds. For any finite set Π of formulas, $\Pi^\wedge$ stands for the conjunction of all elements of Π. For any finite set Ω of atomic formulas and any nonnegative integer z, by $\Omega^{>z}$ we denote the disjunction of the formulas $\Pi^\wedge$ over all subsets Π of Ω that have $z+1$ elements. This formula expresses that the number of atomic formulas that are assigned the value *true* is greater than z. For example, the expression

$$\{p, q, r\}^{>1}$$

stands for the formula

$$(p \wedge q) \vee (p \wedge r) \vee (q \wedge r).$$

We define the propositional image of a ground choice rule with an upper bound

$$\{A_1; \dots; A_m\}\ \mathtt{u}.$$

as the pair of propositional rules: the propositional image of choice rule (5.35) as defined above and the constraint

$$\bot \leftarrow V(\{A_1, \dots, A_m\})^{>u}. \tag{5.41}$$

For example, the propositional image of the rule

```
{p; q; r} 1.
```

consists of rule (5.40) and constraint

$$\bot \leftarrow (p \wedge q) \vee (p \wedge r) \vee (q \wedge r).$$

Exercise 5.21. Find the propositional image of the rule

```
{p(1..2,1..2)} 2.
```

For any finite set Ω of atomic formulas and any nonnegative integer z, $\Omega^{<z}$ will denote the conjunction of the formulas $\neg\Pi^{\wedge}$ over all subsets Π of Ω that have z elements. This formula expressed that the number of atomic formulas that are assigned the value *true* is less than z. For example, the expression

$$\{p, q, r\}^{<2}$$

stands for the formula

$$\neg(p \wedge q) \wedge \neg(p \wedge r) \wedge \neg(q \wedge r).$$

We define the propositional image of a ground choice rule with a lower bound

$$\texttt{l}\ \{A_1; \dots; A_m\}.$$

as the propositional image of choice rule (5.35) plus the constraint

$$\bot \leftarrow V(\{A_1, \dots, A_m\})^{<l}. \tag{5.42}$$

For example, the propositional image of the rule

```
2 {p; q; r}.
```

consists of rule (5.40) and constraint

$$\bot \leftarrow \neg(p \wedge q) \wedge \neg(p \wedge r) \wedge \neg(q \wedge r).$$

The propositional image of a ground choice rule with two bounds

$$\texttt{l}\ \{A_1; \dots; A_m\}\ \texttt{u}.$$

consists of the propositional image of choice rule (5.35) and two constraints (5.41), (5.42).

Exercise 5.22. Find the propositional image of the rule

```
{p(1..2,1..2)} = 2.
```

For a ground choice rule of one of the forms

$\{A_1;\dots;A_m\}$ `u :-` $B_1,\dots,B_n$.
`l` $\{A_1;\dots;A_m\}$ `:-` $B_1,\dots,B_n$.
`l` $\{A_1;\dots;A_m\}$ `u :-` $B_1,\dots,B_n$.

the propositional image is defined in a similar way: it consists of the propositional image of choice rule (5.36) and one or two constraints. The constraints are (5.41) and (5.42) modified by appending the propositional image of $B_1,\dots,B_n$ to their bodies.

As an example, consider the program

```
letter(a).
letter(b).                              (5.43)
{p(X,1..2)} = 1 :- letter(X).
```

We would like to determine whether the interpretation

$$\{\mathit{letter}(a),\ \mathit{letter}(b),\ p(a,2),\ p(b,1)\} \tag{5.44}$$

is one of its stable models. The propositional image of the program consists of the rules

$$\begin{array}{l}
\mathit{letter}(a),\\
\mathit{letter}(b),\\
(p(v,1)\vee\neg p(v,1))\wedge(p(v,2)\vee\neg p(v,2))\leftarrow \mathit{letter}(v),\\
\bot\leftarrow p(v,1)\wedge p(v,2)\wedge \mathit{letter}(v),\\
\bot\leftarrow \neg p(v,1)\wedge\neg p(v,2)\wedge \mathit{letter}(v)
\end{array} \tag{5.45}$$

for all v in $\mathbf{S}\cup\mathbf{Z}$. One way to proceed is to find the reduct of (5.45) relative to interpretation (5.44). Alternatively, we can use here Theorem on Constraints: an interpretation is a stable model of program (5.45) if and only if it is a stable model of the first three lines of the program and does not satisfy the bodies of the constraints in the last two lines. The use of Theorem on Constraints makes the calculation a little easier, because it does not require computing the reducts of the constraints. The reduct of the first three lines of (5.45) is

$$\begin{array}{ll}
\mathit{letter}(a), & \\
\mathit{letter}(b), & \\
(p(a,1)\vee\top)\wedge(p(a,2)\vee\bot)\leftarrow \mathit{letter}(a), & \\
(p(b,1)\vee\bot)\wedge(p(b,2)\vee\top)\leftarrow \mathit{letter}(b), & \\
(p(v,1)\vee\top)\wedge(p(v,2)\vee\top)\leftarrow \mathit{letter}(v) & (v\in\mathbf{S}\cup\mathbf{Z}\setminus\{a,b\}),
\end{array}$$

or, after simplification,

$$\begin{array}{l} \mathit{letter}(a), \\ \mathit{letter}(b), \\ p(a,2) \leftarrow \mathit{letter}(a), \\ p(b,1) \leftarrow \mathit{letter}(b). \end{array}$$

This is a definite program, and its minimal model is (5.44). Consequently (5.44) is a stable model of the first three lines of (5.45). Since this interpretation does not satisfy the bodies of the constraints in the last two lines of (5.45), it is a stable model of the entire program (5.45) as well.

Exercise 5.23. Consider the program obtained from (5.43) by adding the rule

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

Determine whether interpretation (5.44) is a stable model of the extended program.

5.8 Strong Equivalence

Definition. About sets Π_1, Π_2 of propositional rules we say that they are *strongly equivalent* to each other if, for every propositional program Π, the program $\Pi \cup \Pi_1$ has the same stable models as $\Pi \cup \Pi_2$.

From this definition we see that replacing any group of rules within a propositional program by a strongly equivalent group of rules does not affect the set of stable models of the program. For example, in Section 5.4 we claimed that the set of stable models of a propositional program does not change if we break a rule with a conjunction in the head into several rules. That claim can be expressed by saying that a rule of the form

$$F_1 \wedge \cdots \wedge F_n \leftarrow G \tag{5.46}$$

is strongly equivalent to the set of n rules

$$F_1 \leftarrow G, \ldots, G_n \leftarrow G. \tag{5.47}$$

The observation, in the same section, that the law of double negation $p \leftarrow \neg\neg p$ has a stable model that is not a stable model of $p \leftarrow p$ shows that these two propositional rules are not strongly equivalent.

In the last example we do not even need to combine Π_1 and Π_2 with any additional rules to see that they are not strongly equivalent. In other words, the empty set Π can serve in this case as a counterexample. Similarly, we can see that the three equivalent rules

$$p \leftarrow \neg q, \quad q \leftarrow \neg p, \quad p \vee q \tag{5.48}$$

are not strongly equivalent to each other by observing that these rules have different stable models if we think of each of them as a one-rule program. (The only stable model of the first rule is $\{p\}$; the only stable model of the second is $\{q\}$; the third has two stable models.) On the other hand, the one-rule programs $q \leftarrow p$ and $s \leftarrow r$ are not strongly equivalent, even though they have the same stable model $\emptyset$. To see why, combine each of them with the atomic formula p. The stable model of the former will turn into $\{p, q\}$; the stable model of the latter into $\{p\}$.

The last of rules (5.48) has the same stable models as the combination of the first two rules, but it is not strongly equivalent to that combination. Indeed, adding the rules

$$\begin{array}{l} p \leftarrow q, \\ q \leftarrow p \end{array}$$

to the rule $p \vee q$ gives a program with the stable model $\{p, q\}$, but adding the same rules to $p \leftarrow \neg q$, $q \leftarrow \neg p$ gives a program without stable models (Exercise 5.8 on page 85).

Exercise 5.24. Prove that $\neg\neg p$ is not strongly equivalent to p.

Exercise 5.25. Prove that $\neg q \leftarrow \neg p$ is not strongly equivalent to $p \leftarrow q$.

The use of the term "*strongly* equivalent" is justified by the fact that any two sets of propositional rules that are strongly equivalent to each other are also equivalent—that is, have the same models. To prove this assertion, assume that Π_1 is strongly equivalent to Π_2, and take Π to be the set of the excluded middle formulas $p \vee \neg p$ for all atomic formulas p. Then $\Pi \cup \Pi_1$ and $\Pi \cup \Pi_2$ have the same stable models. But by Theorem on Excluded Middle, the set of stable models of $\Pi \cup \Pi_1$ is the set of all models of Π_1, and the set of stable models of $\Pi \cup \Pi_2$ is the set of all models of Π_2. It follows that Π_1 and Π_2 have the same models.

We can assert that propositional programs Π_1 and Π_2 are strongly equivalent to each other whenever we can show that the reduct of Π_1 relative to any interpretation is equivalent to the reduct of Π_2 relative to the same interpretation. Indeed, let Π', Π'_1, and Π'_2 be the reducts of programs Π, Π_1, and Π_2, respectively, relative to the same interpretation I. Whether I is a stable model of $\Pi \cup \Pi_i$ $(i = 1, 2)$ is completely determined by the set of all models of $\Pi' \cup \Pi'_i$. If Π'_1 and Π'_2 have the same models then $\Pi' \cup \Pi'_1$ and $\Pi' \cup \Pi'_2$ have the same models as well.

For example, this reasoning can be used to prove that (5.46) is strongly equivalent to (5.47). Indeed, if the reduct of (5.46) relative to some interpretation I is

$$F'_1 \wedge \cdots \wedge F'_n \leftarrow G' \tag{5.49}$$

then the reduct of (5.47) relative to I is

$$F'_1 \leftarrow G', \ldots, F'_n \leftarrow G'. \tag{5.50}$$

It is clear that (5.49) is equivalent to (5.50).

In a similar way, we can check that any rule of the form $F \leftarrow G \wedge \top$ is strongly equivalent to $F \leftarrow G$. Indeed, the reducts of these rules relative to the same interpretation are the equivalent formulas $F' \leftarrow G' \wedge \top$ and $F' \leftarrow G'$.

Exercise 5.26. Prove that if two sets of positive propositional rules are equivalent to each other then they are strongly equivalent.

Let us check now that $\neg\neg\neg F$ is strongly equivalent to $\neg F$. Consider any interpretation I. The reduct of each of the two formulas relative to an interpretation I is $\bot$ if I satisfies F, and $\top$ otherwise. In either case, both formulas have the same reduct.

A similar calculation shows that $G \leftarrow \neg\neg\neg F$ is strongly equivalent to $G \leftarrow \neg F$. More generally, removing two of three successive negations anywhere within a propositional rule is a strongly equivalent transformation. This is why allowing more than two negations in a row within a CLINGO rule would be pointless.

In Section 2.7 we claimed that replacing the CLINGO constraint

```
:- f(X,Y1), f(X,Y2), Y1! = Y2.
```

by the rule

```
Y1 = Y2 :- f(X,Y1), f(X,Y2).
```

does not affect the stable models of a program. More generally, replacing a constraint of the form

$$\texttt{:-}\ \ldots,\ t_1\ \texttt{!=}\ t_2\ . \tag{5.51}$$

by

$$t_1\ \texttt{=}\ t_2\ \texttt{:-}\ \ldots\ . \tag{5.52}$$

within any CLINGO program does not affect its stable models. To see why, note that the propositional image of a ground rule of form (5.51) has the form

$$\bot \leftarrow F \wedge \bot$$

if the set of values of t_1 is the same as the set of values of t_2, and

$$\bot \leftarrow F \wedge \top$$

otherwise. The propositional image of (5.52) is

$$\top \leftarrow F$$

or

$$\bot \leftarrow F$$

depending on the same condition. In either case, the propositional images are strongly equivalent to each other.

Exercise 5.27. Replacing a rule of the form

$$\ldots\ \texttt{:-}\ t_1\ \texttt{!=}\ t_2\ . \tag{5.53}$$

by

$$\ldots,\ t_1\ \texttt{=}\ t_2\ . \tag{5.54}$$

within any CLINGO program does not affect its stable models. True or false?

In Section 5.4 we observed that the rule $p \leftarrow \neg\neg p$ has the same stable models as $p \vee \neg p$. In fact, these two rules are strongly equivalent to each other. More generally, any rule of the form $F \leftarrow \neg\neg F$ is strongly equivalent to $F \vee \neg F$. Indeed, if the reduct of the former relative to some interpretation is $F' \leftarrow \top$ then the reduct of the latter relative to the same interpretation is $F' \vee \bot$; if the reduct of the former is $F' \leftarrow \bot$ then the reduct of the latter is $F' \vee \top$. Either way, the two reducts are equivalent to each other.

Exercise 5.28. Any rule of the form

$$F \leftarrow \neg\neg F \wedge G \tag{5.55}$$

is strongly equivalent to

$$F \vee \neg F \leftarrow G. \tag{5.56}$$

True or false?

In Section 5.4 we claimed that all simplification steps listed in Table 4.1 preserve the stable models of a program. For one of these steps, the validity of this claim is more difficult to establish than for the others: it is not immediately clear that any constraint $\bot \leftarrow F$ is strongly equivalent to $\neg F$. This assertion is, however, a special case of the general theorem stated below.

A propositional rule is *negative* if every occurrence of every atomic formula in its head is in the scope of negation. For example, all constraints are negative—the head of a constraint does not contain atomic formulas. If the head of a rule has the form $\neg F$ or $\neg F \vee \neg G$ then the rule is negative.

Theorem on Negative Rules. *If two sets of negative propositional rules are equivalent to each other then they are strongly equivalent.*

The fact that $\bot \leftarrow F$ is strongly equivalent to $\neg F$ immediately follows, because these formulas are equivalent. Theorem on Negative Rules shows also that every negative rule is strongly equivalent to a constraint, because $F \leftarrow G$ is equivalent to $\bot \leftarrow \neg F \wedge G$.

A propositional rule is *strongly tautological* if the set of stable models of a program never changes when this rule is removed. In other words, a rule is strongly tautological rule if it is strongly equivalent to the empty program.

We can assert that a propositional rule is strongly tautological if we can show that all its reducts are tautologies. For instance, any rule of the form $F \leftarrow G \wedge \bot$ is strongly tautological, because its reducts have the form $F' \leftarrow G' \wedge \bot$ and consequently are tautological.

If a rule is a positive tautology or a negative tautology then it is strongly tautological. This fact follows from the assertion of Exercise 5.26 and from Theorem on Negative Rules.

Exercise 5.29. For each of the given expressions, determine whether every propositional rule of this form is strongly tautological.

(a) $F \vee \neg F$,

(b) $\neg F \vee \neg\neg F$,

(c) $\bot \leftarrow F \wedge \neg F$,

(d) $F \leftarrow F \wedge G$,

(e) $F \vee G \leftarrow F$.

5.9 Program Completion

The process of completion is defined for propositional programs satisfying condition (α) from Section 5.3 (which says that the heads of all rules are atomic formulas) and the condition

(β) for every atomic formula p, the set of rules of the program with the head p is finite.

This additional condition is trivially satisfied if the program has finitely many rules. But there are also infinite programs with property (β). For instance, the propositional image of the CLINGO rule

```
q(X) :- p(X).
```

consists of infinitely many propositional rules

$$q(v) \leftarrow p(v) \qquad (v \in \mathbf{S} \cup \mathbf{Z}),$$

but every atom $q(v)$ is the head of only one of these rules. On the other hand, condition (β) is violated for the propositional image of the rule

```
q(X) :- p(X,Y).                                                    (5.57)
```

It consists of the rules

$$q(v_1) \leftarrow p(v_1, v_2) \qquad (v_1, v_2 \in \mathbf{S} \cup \mathbf{Z}), \tag{5.58}$$

and every atom of the form $q(v)$ is the head of infinitely many of them.

Definition. Let Π be a propositional program Π satisfying conditions (α) and (β). For every atomic formula p from the vocabulary of Π, by B_p we denote

- $\top$, if p is one of the rules of Π,
- the disjunction of the bodies of all rules of Π with the head p, otherwise.

(Condition (β) guarantees that this disjunction is finite.) The *completion* of Π is the set of formulas

$$p \leftrightarrow B_p \tag{5.59}$$

for all atomic formulas p.

Consider, for instance, program (5.1)–(5.4), and assume that the vocabulary (the set of all atomic formulas) is $\{p, q, r, s\}$. The completion of the program consists of the formulas

$$\begin{aligned} & p \leftrightarrow \top, \\ & q \leftrightarrow \top, \\ & r \leftrightarrow p \land \neg s, \\ & s \leftrightarrow q. \end{aligned} \tag{5.60}$$

If the vocabulary includes any atomic formulas other than p, q, r, s, then the completion will include also the formulas

$$x \leftrightarrow \bot$$

for every such atomic formula x. (Recall that the disjunction of the empty set set is $\bot$.)

In the examples and exercises in the rest of this section we assume that the vocabulary includes only the atomic formulas that occur in the given program.

The completion of the program from Exercise 5.2 (page 82) is

$$\begin{aligned} & p \leftrightarrow \neg q, \\ & q \leftrightarrow \neg r, \\ & r \leftrightarrow \bot. \end{aligned} \tag{5.61}$$

The completion of the program from Exercise 5.7 (page 85) is

$$\begin{aligned} & p \leftrightarrow \neg q, \\ & q \leftrightarrow \neg p, \\ & r \leftrightarrow p \lor q. \end{aligned} \tag{5.62}$$

The intuition behind the concept of completion is that the bodies of rules with the head p can be viewed as sufficient conditions for the validity of p, and formula (5.59) says that, collectively, these sufficient conditions are also necessary. For instance, the last two rules of the program from Exercise 5.7 tell us that p is a sufficient condition for r, and so is q; the last of formulas (5.62) says that r is true if *and only if* at least one of these sufficient conditions holds. It says, in other words, that the disjunction of these sufficient conditions is necessary and sufficient.

In many cases, the set of stable models of a propositional program is exactly the same as the set of all models of its completion. As an example, let us find all models of the completion (5.60) of program (5.1)–(5.4). The completion can be easily simplified, because the first two formulas in (5.60) allow us to replace p and q in the last two formulas by $\top$:

$$\begin{aligned} & p \leftrightarrow \top, \\ & q \leftrightarrow \top, \\ & r \leftrightarrow \top \land \neg s, \\ & s \leftrightarrow \top. \end{aligned}$$

Then the last line allows us to replace s in the third line by $\top$:

$$r \leftrightarrow \top \land \neg\top.$$

This formula is equivalent to

$$r \leftrightarrow \bot.$$

It is clear now that (5.60) has one model, $\{p, q, s\}$. As we know, this is the only stable model of program (5.1)–(5.4).

Exercise 5.30. Check that the stable model of the program from Exercise 5.7 is the only stable model of the completion (5.61) of that program.

Exercise 5.31. (a) Form the completion of program (5.1)–(5.3) on page 81. (b) Check that the stable model of that program is the only stable model of its completion.

Exercise 5.32. Do the same for program (5.1), (5.3), (5.4).

Consider now the completion (5.62) of the program from Exercise 5.7. The first of formulas (5.62) allows us to replace p by $\neg q$ in the other two formulas:

$$\begin{aligned} &p \leftrightarrow \neg q, \\ &q \leftrightarrow \neg\neg q, \\ &r \leftrightarrow \neg q \lor q. \end{aligned}$$

This set of formulas is equivalent to

$$\begin{aligned} &p \leftrightarrow \neg q, \\ &r \leftrightarrow \top. \end{aligned}$$

It follows that (5.62) has two models, corresponding to two ways to assign a truth value to q: $\{p, r\}$ (q is false) and $\{q, r\}$ (q is true). As in the other examples, the set of all models of the program's completion coincides with the set of stable models.

There are cases, however, when these sets are different from each other. The program

$$\begin{aligned} &p \leftarrow q, \\ &q \leftarrow p \end{aligned} \tag{5.63}$$

is positive definite, and its only stable model is the minimal model $\emptyset$. But the completion of this program

$$\begin{aligned} &p \leftrightarrow q, \\ &q \leftrightarrow p \end{aligned}$$

has two models, $\emptyset$ and $\{p, q\}$.

A condition that eliminates cases when completion and stable models do not match is discussed in the next section.

5.10 Theorem on Completion

About a propositional program Π satisfying condition (α) and atomic formulas p, q we will say that p *depends* on q in Π if Π contains a rule $p \leftarrow F$ such that F contains an occurrence of q that is not in the scope of negation. In program (5.1)–(5.4), for example, p and q do not depend on any atoms; r depends on p but not on s; s depends on q.

The statement of the theorem below refers to the following additional condition on propositional programs:

(γ) it is possible to assign a nonnegative integer, called *the rank*, to every atomic formula occurring in the program, so that the rank of each atomic formula is greater than the ranks of the atomic formulas that it depends on.

For instance, program (5.1)–(5.4) satisfies not only conditions (α) and (β), but also condition (γ): we can assign rank 0 to p and q, and rank 1 to r and s.

On the other hand, condition (γ) does not hold for any program containing rules (5.63): in application to such a program, this condition requires that the rank of p be both greater than and less than the rank of q. Another example when condition (γ) is not satisfied is given by the propositional image of the second rule in the recursive definition (2.8) of `ancestor/2`:

$$anc(v_1, v_3) \leftarrow anc(v_1, v_2) \wedge anc(v_2, v_3) \qquad (v_1, v_2, v_3 \in \mathbf{S} \cup \mathbf{Z}).$$

For a rule of this form with $v_1 = v_2 = v_3$, condition (γ) requires that the rank of $anc(v_1, v_1)$ be greater than itself.

Exercise 5.33. Determine whether program (4.2)–(4.5) on page 66 satisfies condition (γ).

Theorem on Completion. *For any propositional program Π satisfying conditions (α) and (β), every stable model of Π is a model of the completion of Π. If Π satisfies also condition (γ) then the converse holds as well: every model of the completion of Π is a stable model of Π.*

We observed, for instance, in Section 5.4 that the one-rule program $p \leftarrow \neg\neg p$ has two stable models, $\emptyset$ and $\{p\}$. This can be confirmed by a reference to Theorem on Completion. This program satisfies conditions (α)–(γ), so that its stable models are identical to models of its completion. The completion is the tautology $p \leftrightarrow \neg\neg p$, and its models are both $\emptyset$ and $\{p\}$.

The completion of the one-rule program $p \leftarrow p$ is a tautology also, and it has the same two models, $\emptyset$ and $\{p\}$. But this program does not satisfy condition (γ); the former is a model of the completion, and the latter is not.

Theorem on Completion can help us find the stable models of a propositional program satisfying conditions (α) and (β). We form a completion of such a program, and then find its models—either manually, as in the examples above, or by running a satisfiability solver. If the program satisfies condition (γ) then the result is the collection of stable models of the program. If not then some models of the completion may not be stable; they can be sifted out using the definition of a stable model in terms of reducts, as in Section 5.2.

Exercise 5.34. (a) Use Theorem on Completion to find the stable models of program (5.14)–(5.16) on page 85. (b) Check that the result of your calculation is in agreement with the output of CLINGO.

Exercise 5.35. Do the same for the program

$$\begin{aligned}p &\leftarrow \neg q,\\ q &\leftarrow \neg r,\\ r &\leftarrow \neg p.\end{aligned}$$

Exercise 5.36. Use Theorem on Completion to confirm your answer to Exercise 5.8 (page 85).

Theorem on Completion is not limited to finite programs. Consider, for instance, two infinite definite programs:

$$\begin{aligned}p_1 &\leftarrow p_0,\\ p_2 &\leftarrow p_1,\\ &\cdots\end{aligned} \tag{5.64}$$

and

$$\begin{aligned}p_0 &\leftarrow p_1,\\ p_1 &\leftarrow p_2,\\ &\cdots .\end{aligned} \tag{5.65}$$

Each of them has one stable model—the minimal model $\emptyset$. The former satisfies conditions (α)–(γ). (To verify (γ), assign rank n to p_n.) Its completion consists of the formulas

$$\begin{aligned}p_0 &\leftrightarrow \bot,\\ p_1 &\leftrightarrow p_0,\\ p_2 &\leftrightarrow p_1,\\ &\cdots ,\end{aligned}$$

and $\emptyset$ is the only model of the completion, in accordance with the theorem. Program (5.65), on the other hand, does not satisfy condition (γ). Indeed, in application to that program condition (γ) requires that the ranks of the atomic formulas $p_0, p_1, \dots$ form a decreasing infinite sequence of nonnegative integers, which is impossible. The completion of this program

$$\begin{aligned}p_0 &\leftrightarrow p_1,\\ p_1 &\leftrightarrow p_2,\\ &\cdots\end{aligned}$$

has two models: $\emptyset$, which is stable, and $\{p_0, p_1, \dots\}$, which is not.

In Section 5.6 we used the definition of a stable model in terms of reducts to verify that interpretation (5.25) is a stable model of program (5.26). The problem of calculating the stable models of that program can be approached also on the basis of Theorem on Completion. The completion of the propositional image (5.27) of the program is

$$\begin{array}{ll}p(a) \leftrightarrow \top, & \\ p(v) \leftrightarrow \bot & \text{for all } v \text{ other than } a,\\ q(a) \leftrightarrow \top, & \\ q(v) \leftrightarrow \bot & \text{for all } v \text{ other than } a,\\ r(v) \leftrightarrow p(v) \wedge \neg q(v) & \text{for all } v.\end{array}$$

The first four lines allow us to replace the last line by

$$\begin{array}{ll} r(a) \leftrightarrow \top \wedge \neg\top, & \\ r(v) \leftrightarrow \bot \wedge \neg\bot & \qquad \text{for all } v \text{ other than } a, \end{array}$$

or, equivalently,

$$r(v) \leftrightarrow \bot \qquad \text{for all } v.$$

Consequently the only model of the completion is $\{p(a), q(a)\}$. Since (5.27) satisfies conditions (α)–(γ), it follows that $\{p(a), q(a)\}$ is the only stable model of program (5.26).

Exercise 5.37. (a) Use Theorem on Completion to find all stable models of the program from Exercise 5.14 on page 92. (b) Check whether your answer is in agreement with the output of CLINGO.

Exercise 5.38. Do the same for the program from Exercise 5.15.

Exercise 5.39. Use Theorem on Completion to find all stable models of program (2.9), (2.10) on page 23.

Theorem on Completion can help us find stable models of a program that does not satisfy condition (α) if we can find a strongly equivalent program that does. For instance, we can find the stable model of the program

$$\begin{array}{l} p \wedge q, \\ r \leftarrow \neg p \end{array}$$

by breaking the first rule into p, q and then calculating the completion. The stable models of the program

$$\begin{array}{l} p \vee \neg p, \\ q \leftarrow p \end{array}$$

can be found using Theorem on Completion if we rewrite the first rule as

$$p \leftarrow \neg\neg p.$$

Exercise 5.40. (a) Use Theorem on Completion to find all stable models of the program from Exercise 5.17 on page 93. (b) Check whether your answer is in agreement with the output of CLINGO.

Exercise 5.41. Do the same for the program from Exercise 5.18.

Exercise 5.42. Do the same for the program from Exercise 5.19.

Sometimes a propositional program can be represented as the union of a program Π satisfying conditions (α)–(γ) and a set Σ of constraints. The stable models of such a program can be characterized as the models of the union of the completion of Π with Σ. This fact follows from Theorem on Completion and Theorem on Constraints.

Exercise 5.43. (a) Use Theorem on Completion to find all stable models of the program

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

(b) Check whether your answer is in agreement with the output of CLINGO.

5.11 Local Variables and Infinitary Formulas

From examples in Chapter 3 we know that many CLINGO programs use local variables. Yet this construct is not covered by the definition of the propositional image above; its meaning is only explained in this book by the informal discussion in Section 2.6 and by examples of its use. The reason is that the class of formulas that is defined and investigated here is too narrow for this more difficult subject.

One generalization of propositional rules that can be used to explain the precise meaning of local variables is rules with quantifiers, such as

$$q \leftarrow \forall X\, p(X) \quad \text{and} \quad q \leftarrow \exists X\, p(X).$$

Another possibility is to allow propositional rules to contain infinitely long conjunctions and disjunctions, for instance

$$q \leftarrow p_1 \wedge p_2 \wedge \dots \quad \text{and} \quad q \leftarrow p_1 \vee p_2 \vee \dots\ .$$

Infinitary propositional formulas are often denoted by expressions containing the "big conjunction" and "big disjunction" symbols, similar to the symbol $\sum$ used in algebra to represent the sum of a set of numbers:

$$q \leftarrow \bigwedge_{n\geq 1} p_n, \quad q \leftarrow \bigvee_{n\geq 1} p_n. \tag{5.66}$$

In some cases, an infinitary propositional formula has the same meaning as a set of finite formulas. For instance, the second of formulas (5.66) is equivalent to the infinite set of finite implications

$$q \leftarrow p_1,\ q \leftarrow p_2,\ \dots\ .$$

But the first of formulas (5.66) is not equivalent to any set of finite propositional formulas.

Infinitary formulas can help us extend the definition of the propositional image to choice rules with local variables and cardinality bounds, such as

```
1 {p(X) : q(X)}.
```
(5.67)

In the absence of cardinality bounds, local variables in a choice rule are not problematic. For instance, the rule

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

has the same meaning as the rule

```
{p(X)} :- q(X).
```

in which the variable `X` is global, and we can define its propositional image as the set of rules

$$p(v) \vee \neg p(v) \leftarrow q(v)$$

for all v in $\mathbf{S} \cup \mathbf{Z}$. But the propositional image of rule (5.67), with its lower bound 1, should include also a constraint eliminating the stable models that do not include any of the atoms $p(v)$ for the values of v for which $q(v)$ is included. The constraint that we need should eliminate, in other words, the sets `p/1` that are disjoint from `q/1`. That can be expressed by the infinitary propositional rule

$$\bot \leftarrow \bigwedge_{v \in \mathbf{S} \cup \mathbf{Z}} \neg(p(v) \wedge q(v)),$$

which is not equivalent to any set of finite formulas.

We do not go here into further details concerning propositional images of rules with local variables.

Infinitary formulas are used in the theory of logic programming in other ways as well. In particular, they allow us to extend the process of completion (Section 5.9) to programs that do not satisfy condition (β). We can say, for example, that the completion of program (5.58) is the set of formulas

$$q(v_1) \leftrightarrow \bigvee_{v_2 \in \mathbf{S} \cup \mathbf{Z}} p(v_1, v_2)$$

for all v_1 in $\mathbf{S} \cup \mathbf{Z}$. In this more general setting, the assertion of Theorem on Completion can be strengthened by dropping condition (β). In other words, for any propositional program Π satisfying condition (α), every stable model of Π is a model of the completion of Π; if Π satisfies also condition (γ) then the converse holds as well.

To show how infinitary completion can help us reason about stable models, assume that we are given a finite directed acyclic graph G, such as the one shown in Figure 5.1, and

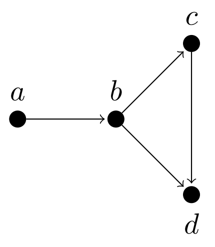

Figure 5.1: Acyclic directed graph.

consider the following game. We place a token on one of the vertices. Then two players take turns moving the token to another vertex along one of the edges of the graph. When the player who has to move finds the token on a vertex without outgoing edges, that player has lost. Whether the player who starts can win is determined by the vertex where the token was placed initially. For instance, in the graph shown in Figure 5.1, vertices b and c are winning (move the token to d), and vertices a and d are not.

The set `winning/1` of winning vertices can be described by the rule

```
winning(X) :- edge(X,Y), not winning(Y).
```
(5.68)

(a vertex is winning if there is an edge leading from it to a vertex that is not winning). The set of winning vertices can be determined by running CLINGO on the program Π_G obtained by adding this rule to the definition of the predicate `edge/2` for graph G.

The propositional image of Π_G satisfies condition (α), but not (β): among the rules

$$\mathit{winning}(v_1) \leftarrow \mathit{edge}(v_1, v_2) \wedge \neg\mathit{winning}(v_2) \tag{5.69}$$

in the propositional image of (5.68) there are infinitely many rules with the same head $\mathit{winning}(v_1)$. Condition (γ) is satisfied, however: we can assign rank 0 to all atoms $\mathit{edge}(v_1, v_2)$ and rank 1 to all atoms $\mathit{winning}(v)$. (Note that in rule (5.69), $\mathit{winning}(v_1)$ does not depend on $\mathit{winning}(v_2)$, because the latter is in the scope of negation.) Consequently stable models of program Π_G can be characterized as models of the infinitary completion of its propositional image, which consists of the formulas

$$\begin{array}{rll}
\mathit{edge}(v_1, v_2) & \leftrightarrow \top & \text{for every edge } (v_1, v_2),\\
\mathit{edge}(v_1, v_2) & \leftrightarrow \bot & \text{for every other pair } (v_1, v_2),\\
\mathit{winning}(v_1) & \leftrightarrow \bigvee_{v_2 \in \mathbf{S} \cup \mathbf{Z}} (\mathit{edge}(v_1, v_2) \wedge \neg\mathit{winning}(v_2)) & \text{for all } v_1.
\end{array}$$

In the presence of the formulas in the first two lines, the infinite disjunction in the last line can be equivalently replaced by the disjunction of the formulas $\neg\mathit{winning}(v_2)$ over all v_2 such that (v_1, v_2) is an edge of the graph, and that disjunction is finite. After that, a sequence of simplification steps will lead us to the unique stable models of Π_G. In the case of the graph in Figure 5.1, for instance, the last line of the completion is equivalent to

$$\begin{array}{ll}
\mathit{winning}(a) \leftrightarrow \neg\mathit{winning}(b), & \\
\mathit{winning}(b) \leftrightarrow \neg\mathit{winning}(c) \vee \neg\mathit{winning}(d), & \\
\mathit{winning}(c) \leftrightarrow \neg\mathit{winning}(d), & \\
\mathit{winning}(v) \leftrightarrow \bot & \text{for all } v \text{ other than } a, b, c.
\end{array}$$

This set of formulas is equivalent to

$$\begin{array}{ll}
\mathit{winning}(a) \leftrightarrow \neg\mathit{winning}(b), & \\
\mathit{winning}(b) \leftrightarrow \top, & \\
\mathit{winning}(c) \leftrightarrow \top, & \\
\mathit{winning}(v) \leftrightarrow \bot & \text{for all } v \text{ other than } a, b, c,
\end{array}$$

and consequently to

$$\begin{array}{ll}
\mathit{winning}(b) \leftrightarrow \top, & \\
\mathit{winning}(c) \leftrightarrow \top, & \\
\mathit{winning}(v) \leftrightarrow \bot & \text{for all } v \text{ other than } b, c.
\end{array}$$

It follows that the only stable model of Π_G in this example is

$$\{edge(a,b),\ edge(b,c),\ edge(b,d),\ edge(c,d),\ winning(b),\ winning(c)\}.$$

5.12 Bibliographical and Historical Remarks

The idea of using a fixpoint construction to describe nonmonotonic reasoning is older than the definition of a stable model. We find it, in particular, in the definition of default logic [107]. This is an extension of classical logic in which the set of postulates is allowed to contain *defaults*—postulates of the form

$$\frac{F_1,\dots,F_m\ :\ G_1,\ \dots,G_n}{H}. \tag{5.70}$$

Formulas $F_1,\dots,F_m$ are the *premises* of the default, formulas $G_1,\ \dots,G_n$ are its *justifications*, and formula H is its *conclusion*. Such a default is understood, informally speaking, as permission to derive the conclusion from the premises if the justifications can be consistently assumed. Propositional programs of simple syntactic form can be viewed as alternative notation for default theories [13]. For example, rule (5.3) on page 81 can be identified with the default

$$\frac{p\ :\neg s}{r}.$$

The condition "$\neg s$ can be consistently assumed" is similar to condition (5.5) on page 81, and in the definition of default logic such conditions are made precise using a fixpoint construction.

Autoepistemic logic [95] is a nonmonotonic logic that allows formulas to contain, in addition to propositional connectives, the logical operator L. The formula LF is read "F is believed," and the precise meaning of this operator is defined in terms of fixpoints. The autoepistemic counterpart of rule (5.3) is

$$r \leftarrow p \wedge \neg \mathrm{L}s.$$

From this perspective, negation as failure in logic programming has the same meaning as the combination $\neg$L in autoepistemic logic [47].

Stable models can be characterized in many different ways [79]. They were first defined for rules of the form

$$A_0 \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n,$$

where each A_i is an atomic formula [39, 50], then for rules with a disjunction in the head [52], and then for rules in which conjunction, disjunction, and negation can be nested arbitrarily in heads and bodies [82], as in propositional rules in the sense of Section 4.1. The original definition of a stable model for programs with choice rules [99] treated choice as an independent construct; a few years later [37] it became clear that `{p}` can be viewed also as shorthand for

```
p ; not p.
```

Equivalence relations between propositional formulas that are stronger than classical equivalence and do not treat, for instance, dropping a double negation as an equivalent transformation have been studied in logic for a long time. This work was initially related to the intuitionist philosophy of mathematics. The idea that mathematics is the creation of the mind has led intuitionists to the view that the law of excluded middle and "the principle that for every system the correctness of a property follows from the impossibility of the impossibility of the property" may be invalid in application to infinite systems [18]. The logical consequences of this view were clarified by the invention of intuitionistic propositional logic—a formal system that does not sanction the law of excluded middle and other tautologies that intuitionists find objectionable [66].

The claim that the law of excluded middle is not intuitionstically provable was justified using an auxiliary logical system, which is often called (for reasons that we do not go into in this book) "the logic of here-and-there" and denoted by HT. The logic of here-and-there is based on three truth values, instead of the two classical values *false*, *true*, and it is intermediate in strength between intuitionistic logic and classical. Since HT is closely related to the idea of strong equivalence in the theory of logic programming, we discuss it here in some detail.

The three truth values can be represented by numbers 0, $\frac{1}{2}$, and 1; the new truth value $\frac{1}{2}$ is reserved, informally speaking, for propositions "that cannot be false but whose correctness is not proved" [66]. Once one of these truth values is assigned to every atomic formula, the values of other formulas are calculated according to the following rules. The value of $\neg F$ is 1 if the value of F is 0, and it is 0 otherwise. The values of $F \wedge G$ and $F \vee G$ are, respectively, the minimum and maximum of the values of F and G. Finally, if the value of F is x, and the value of G is y, then

$$\text{the value of } F \leftarrow G \text{ is 1 if } x \geq y, \text{ and } x \text{ otherwise.}$$

It turns out that two propositional rules F, G are strongly equivalent if and only if they are equivalent in the logic of here-and-there [81]; in other words, if and only if the truth value of F for any assignment of truth values 0, $\frac{1}{2}$, 1 to atomic formulas is the same as the truth value of G for the same assignment of truth values. Furthermore, a propositional rule is strongly tautological if and only if its truth value is 1 for every assignment of truth values 0, $\frac{1}{2}$, 1 to atomic formulas. For instance, the rule $p \vee \neg p$ is not strongly tautological; accordingly, it gets the truth value $\frac{1}{2}$, different from 1, if the value of p is $\frac{1}{2}$. The rule $p \leftarrow p$, on the other hand, is strongly tautological; its truth value is 1 for each of the three ways to choose a truth value for p.

The proof of these theorems uses the fact that stable models can be characterized in terms of equilibrium logic—a nonmonotonic logic based on the logic of here-and-there [101].

Program completion was proposed as an explanation of the meaning of negation as failure in 1978 [27], long before the invention of stable models, and generalized in 1984 [86]. The definitions in these papers apply to rules with variables directly and do not require that specific values be substituted for variables, as the version in this book. In the style of the original definition, the completion of program (5.57) would be written as

$$\forall X(q(X) \leftrightarrow \exists Y\, p(X,Y)).$$

The first result on the relationship between completion and stable models was published in 1994 [35] and later extended to other classes of programs [32, 33, 64]. These papers define properties of programs similar to our condition (γ), and programs with these properties are usually referred to as "tight." The definitions of tightness in these papers are not expressed in terms of assigning ranks to atomic formulas; they require instead that it should be impossible to find an infinite sequence of atomic formulas such that every member of the sequence depends on its successor. Tightness conditions of this kind are more general than condition (γ). Consider, for instance, the infinite program

$$\begin{array}{ll} q \leftarrow p_0, & p_1 \leftarrow p_0, \\ q \leftarrow p_1, & p_2 \leftarrow p_1, \\ q \leftarrow p_2, & p_3 \leftarrow p_2, \\ \multicolumn{2}{c}{\cdots .} \end{array}$$

In this case, there is no infinite sequence of atomic formulas such that every member of the sequence depends on its successor. But finite sequences with this property that start with q can be arbitrarily long:

$$q, p_n, p_{n-1}, \dots, p_1, p_0.$$

It follows that this program does not satisfy condition (γ). Programs like this will be covered if we modify condition (γ) by allowing ranks to be infinite ordinals $\omega, \omega + 1, \dots$.

If a program is not tight then its stable models can be described as the models of its completion that satisfy additional "loop formulas" [84]. This fact has been used in the design of answer set solvers [76, 84].

In some cases, stable models of a program can be calculated by "splitting"—dividing the rules of the program into two parts, calculating the stable models of one part, and then extending each of them to find stable models of the entire program [83]. Splitting can be applied, for instance, to the program from Exercise 5.7 (page 85). The part consisting of the first two rules of the program has two stable models, $\{p\}$ and $\{q\}$. The last two rules turn the first of them into $\{p, r\}$, and the second into $\{q, r\}$.

Infinitary formulas have been studied in logic since the 1950s [68, 111], but allowing infinite rules in a logic program is a more recent idea [115]. Infinite conjunctions and disjunctions have been added not only to classical logic, but also to intuitionistic logic [96] and to the logic of here-and-there [63].

The encoding of two-person games in Section 5.11 is similar to encodings written earlier for Prolog [117].

Chapter 6

More about the Language of CLINGO

The programming constructs described below significantly extend the expressive possibilities of the language used so far. The first three sections are about *aggregates*—functions that apply to sets. Then we show how CLINGO can be used to solve combinatorial optimization problems and discuss CLINGO programs with symbolic functions and classical negation.

6.1 Counting

The aggregate `#count` calculates the number of elements of a set. For example, the expression

```
#count{X,Y : edge(X,Y)}
```

represents the number of elements of the set `edge/2`. Expressions like this are used in the bodies of rules as one side of a comparison, with a term on the other side:

```
number_of_edges(N) :- N = #count{X,Y : edge(X,Y)}.            (6.1)
```

The stable model of the program obtained by adding this rule to the definition (3.2) of a graph contains the atom `number_of_edges(9)`: the graph has 9 edges.

The part of an aggregate expression to the right of the colon may be a list of several atoms, negated atoms, and comparisons. For example, the expression

```
#count{X,Y,Z : edge(X,Y), edge(Y,Z)}
```

denotes the number of paths of length 2, and the expression

```
#count{X,Y : edge(X,Y), X != a}
```

denotes the number of edges with the tail different from a.

The part of an aggregate expression to the left of the colon may include not only variables, but also more complex terms. For example, if `n` is a placeholder for a positive integer then the expression

```
#count{X*Y : X = 2..n, Y = 2..n, X*Y <= n}
```

calculates the number of composite numbers between 1 and `n`.

In rule (6.1), the variables `X` and `Y` are local: all their occurrences are between braces (see Section 2.6). An aggregate expression can contain also global variables. For instance, the outdegrees of vertices of a graph can be defined by the rule

```
outdegree(X,N) :- vertex(X), N = #count{Y : edge(X,Y)}.
```

In this rule, the variables `X` and `N` are global.

Aggregate expressions can be used in inequalities. For instance, the set of vertices of outdegree greater than 1 can be defined by the rule

```
branching_vertex(X) :- vertex(X), #count{Y : edge(X,Y)} > 1.
```

Note that a comparison cannot contain aggregate expressions on both sides. When we want to compare the values of two aggregate expressions, this can be accomplished by comparing each of them with the value of a variable. For instance, call a vertex *balanced* if its indegree equals its outdegree; the set of balanced vertices can be defined by the rule

```
balanced(X) :- vertex(X), #count{Y : edge(X,Y)} = N,
                          #count{Y : edge(Y,X)} = N.
```

Exercise 6.1. The choice rule

```
{in(1..n)} = m.
```

(found, for example, in Line 10 of Listing 3.13, page 48) can be replaced by the combination of the rule

```
{in(1..n)}.
```

with a constraint. What constraint is that?

Exercise 6.2. We would like to write a CLINGO program that calculates the number of classes taught today on each floor of a classroom building. What rule would you place in Line 6 of Listing 6.1?

Exercise 6.3. Rule (1.1) defines a large country as a country inhabited by more people than the United Kingdom. How will you modify that rule if “large country” is understood as a country with a population that places it

(i) among the top k countries on the given list?

Listing 6.1: Number of classes (Exercise 6.2)

```
% Number of classes taught on each floor.

% input: number k of floors; set where/2 of all pairs (c,i)
%        such that class c is taught on the i-th floor.

-----------------------------------------------------------
% achieved: howmany(I,N) iff the number of classes taught
%           on the I-th floor is N.

#show howmany/2.
```

(ii) in the top half of the countries on the given list?

We will see now how the `#count` aggregate is used in a CLINGO program that solves skyscraper puzzles. In such a puzzle, the goal is to fill an $n \times n$ grid with numbers from 1 to n. No number may be repeated in any row or column. The numbers represent the heights of skyscrapers, and they should be chosen according to clues—numbers placed around the perimeter of the grid. (See Figure 6.1 for an example.) Each clue tells us how many buildings can be seen from that point, assuming that higher skyscrapers block the view of lower skyscrapers located behind them.

In Listing 6.2, the position of each clue is represented by the row and column where the clue is located. The clues to the left and above the grid are in row 0 and column 0, respectively; the clues to the right and below the grid are in row $n+1$ and column $n+1$. For example, the set `clue/3` representing the puzzle in Figure 6.1 is defined by the rule

```
clue(1,0,4; 2,0,2; 4,0,2; 0,4,1; 4,5,2; 5,3,3; 5,4,2).
```

The constraint in Line 27 of the program says that the number of skyscrapers blocked from

				1	
4	1	2	3	4	
2	3	1	4	2	
	4	3	2	1	
2	2	4	1	3	2
			3	2	

Figure 6.1: Solution to a skyscraper puzzle.

view cannot be different from $n - N$, where N is the value of the corresponding clue.

6.2 Summation

The aggregate `#sum` calculates the sum of a set of integers. For example, the set `q/1` defined by the rule

```
q(N) :- N = #sum{X*X : p(X)}.
```

is a singleton, and its only element is the sum of the squares of all integers in the set `p/1`. The program consisting of that rule and the rule

```
p(a; 1; 2).
```

has the stable model

$$\{p(a), p(1), p(2), q(5)\},$$

because $1^2 + 2^2 = 5$. Since the element `a` of `p/1` is not an integer, it is disregarded in the process of adding squares.

If `#sum` is applied to an expression containing several terms to the left of the colon then the value of `#sum` can be described in terms of "weights." The *weight* of a tuple consisting of integers and symbolic constants is the first member of the tuple. What `#sum` calculates in application to a set of tuples is the sum of the weights of all its elements that have integer weights. For example, if the predicate `size/2` is defined by rules (1.2) on page 11 then the expression

```
#sum{S,C : size(C,S)}                                  (6.2)
```

represents the total population of the countries in Table 1.1.

There is subtle—but important—difference between aggregate expression (6.2) and the shorter expression

```
#sum{S : size(C,S)}.                                   (6.3)
```

The former denotes the sum of the numbers s over all pairs (c, s) that belong to the set `size/2`. The latter denotes the sum of all numbers s such that for some c, (c, s) is an element of `size/2`. These sums may be different if the set includes two pairs with the same second member. For instance, if `size/2` is defined by the rule

```
size(a,1; b,2; c,2).
```

then the value of (6.2) is 5, and the value of (6.3) is 3. Since the population sizes of all countries in Table 1.1 are different from each other, expression (6.3) has in this case the same value as (6.2).

Exercise 6.4. What is the stable model of the program

```
p(N) :- N = #sum{X*X : X = -1..1}.
q(N) :- N = #sum{X*X, X : X = -1..1}.
```

Listing 6.2: Skyscrapers puzzle

```
% Skyscrapers puzzle.

% input: size n of the grid; set clue/3 of triples r,c,n
%        such that clue n is placed in row r, column c
%        around the perimeter of the grid.

% h(R,C,H) means that the height of the skyscraper in row R
% and column C is H.

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

:- not h(R,_,H), R = 1..n, H = 1..n.
:- not h(_,C,H), C = 1..n, H = 1..n.
% achieved: every number between 1 and n occurs in every row
%           and every column.

  blocked(R,C,0,C) :- h(R,C,H), h(R1,C,H1), R1 < R, H1 > H.
blocked(R,C,n+1,C) :- h(R,C,H), h(R1,C,H1), R1 > R, H1 > H.
  blocked(R,C,R,0) :- h(R,C,H), h(R,C1,H1), C1 < C, H1 > H.
blocked(R,C,R,n+1) :- h(R,C,H), h(R,C1,H1), C1 > C, H1 > H.
% achieved: blocked(R,C,R0,C0) iff the skyscraper in row R,
%           column C is not visible from the observation
%           point in row R0, column C0.

:- clue(R0,C0,N), #count{R,C : blocked(R,C,R0,C0)} != n-N.
% achieved: clues match the numbers of visible skyscrapers.

#show h/3.
```

Listing 6.3: Magic squares (Exercise 6.6)

```
 1  % Magic squares of size n
 2
 3  % input: positive integer n.
 4
 5  1 {filled(R,C,1..n*n)} 1 :- R = 1..n, C = 1..n.
 6  % achieved: every square of the grid is filled with a number
 7  %           between 1 and n^2.
 8
 9  :- not filled(_,_,X), X = 1..n*n.
10  % achieved: every number between 1 and n^2 is included.
11
12  #const magic=(n**3+n)/2.
13
14  _______________________________________________________
15  % achieved: every row sums up to magic.
16
17  _______________________________________________________
18  % achieved: every column sums up to magic.
19
20  _______________________________________________________
21  _______________________________________________________
22  % achieved: both diagonals sum up to magic.
```

in your opinion?

Exercise 6.5. Simplify the aggregate expression `#sum{1,X : p(X)}`.

Exercise 6.6. A *magic square* of size n 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." It is clear that the value of the magic constant is determined by the size of the square: it is $(1 + 2 + \cdots + n^2)/n$, which equals $(n + n^3)/2$. We would like to write a CLINGO program that generates all magic squares of a given size. The rules in Lines 5 and 9 of Listing 6.3 are the same as in Listing 3.21 (page 55). What rules would you place in Lines 14, 17, 20, 21?

The next example of using the `#sum` aggregate is motivated by the problem of determining the winner when voters rank several options or candidates in order of preference. One possibility is to calculate the number of points assigned to each candidate by each voter, as follows: the candidate ranked last gets zero points; next to last gets one point, and so on. Once all votes have been counted, the candidate with the most points is the winner. There are several winners in case of a tie.

For example, assume that Andy, Bob and Charlie run for the same office; the ranking $C > B > A$ is chosen by 400 voters, $B > C > A$ by 200 voters, and $A > B > C$ by 300 voters. Then

Andy gets $300 \times 2 = 600$ points,
Bob gets $400 \times 1 + 200 \times 2 + 300 \times 1 = 1100$ points,
Charlie gets $400 \times 2 + 200 \times 1 = 1000$ points;

Bob is the winner.

The program in Listing 6.4 calculates the winner in accordance with this procedure. The example above can be represented by the following input:

```
#const m=3.
votecount(1,400; 2,200; 3,300).
p(1,1,3; 1,2,2; 1,3,1;
  2,1,2; 2,2,3; 2,3,1;
  3,1,1; 3,2,2; 3,3,3).
```

Note that in the aggregate expression

```
#sum{S,R : posScore(R,C,S)}
```

(Line 14 of the program) the number `S` of points to the left of the colon is followed by the number `R` of the ranking that contributed these points. This is essential when a candidate receives the same number of points from different rankings; without `R` appended to `S` in this expression, the number of points earned by such a candidate would be calculated incorrectly. This would happen, in fact, in the example above, because Bob gets 400 points from the voters who chose the ranking $C > B > A$, and the same number of points from the voters who chose $B > C > A$. If we replace `S,R` in Line 14 by `S` then CLINGO will tell us, incorrectly, that Charlie is the winner.

6.3 Maximum and Minimum

The aggregates `#max` and `#min` represent the largest and the smallest elements of a set. For instance, the expression

```
#max{X : p(X)}                                   (6.4)
```

calculates the largest element of the set `p/1`, and the expression

```
#min{|X-Y| : p(X), q(Y)}                         (6.5)
```

calculates the distance between sets `p/1` and `q/1` of integers (that is, the smallest among the distances from elements of `p/1` to elements of `q/1`).

Recall that the total order chosen by the designers of CLINGO for evaluating comparisons is defined not only on numbers, but also on symbolic constants, and that it has the minimal element `#inf` and maximal element `#sup` (Section 2.1). For this reason, expression (6.4) can be used to choose one among the elements of any set `p/1`, no matter whether its elements

Listing 6.4: Choosing winners in an election with multiple candidates

```
% Choose the winner according to the number of points earned
% by each candidate.

% input: the number m of candidates 1,...,m; the set
%        votecount/2 of pairs (R,N) such that ranking R
%        was selected by N voters; the set p/3 of triples
%        (R,Pos,C) such that Pos is the position of
%        candidate C in ranking R.

posScore(R,C,X*N) :- p(R,Pos,C), X = m-Pos, votecount(R,N).
% achieved: posScore(R,C,S) iff the voters who chose
%           ranking R contributed S points to candidate C.

score(C,N) :- C = 1..m, N = #sum{S,R : posScore(R,C,S)}.
% achieved: score(C,N) iff candidate C earned N points.

loser(C) :- score(C,N), score(C1,N1), N1 > N.
% achieved: loser(C) iff candidate C earned fewer points
%           than another candidate.

winner(C) :- C = 1..m, not loser(C).
% achieved: winner(C) iff C is a winner.

#show winner/1.
```

are numbers or symbolic constants. In application to the empty set, `#max` returns `#inf`, and `#min` returns `#sup`. This is similar to the convention mentioned in Section 4.2: the sum of the empty set of numbers is 0, and the product is 1.

Exercise 6.7. Adding the constraint

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

to a CLINGO program eliminates the stable models in which the set `p/1` is empty (Section 2.8). Find constraints with the same property that use, instead of an anonymous variable, (a) the aggregate `#count`, (b) the aggregate `#max`, (c) the aggregate `#min`.

The encoding of Hamiltonian cycles in Listing 3.19 (page 53) expects an input that specifies, in addition to the predicates `vertex/1` and `edge/2`, one of the vertices `v0` of the graph. The symbol `v0` is used in the first rule

```
reachable(X) :- in(v0,X).
```

of the recursive definition of `reachable/1`. The need to specify `v0` will be eliminated if we rewrite that rule as

```
reachable(X) :- in(V0,X), V0 = #max{V : vertex(V)}.
```

The aggregate `#min` would work here as well, of course.

Exercise 6.8. The program from Exercise 6.2 (page 116) determines the number of classes taught on each floor of a building using two pieces of information: the number k of floors and the set of pairs (c, i) such that class c is taught on floor i. Under the assumption that at least one class is taught on the top floor, there is no need to include the value of k in the input. Under this assumption, what rule will you place in Line 6 of Listing 6.1 if the value of k is not given?

The runtime of CLINGO on the encoding of Schur numbers in Listing 3.6 (page 40) can be improved by including a symmetry breaking constraint that forces the first sum-free subset to include 1, the second to include the smallest number that does not belong to the first, and so on. This can be expressed using the `#min` aggregate:

```
:- K = 1..r-1, M = #min{I : in(I,K)}, M > #min{I : in(I,K+1)}.
```

6.4 Optimization

When a logic program has several stable models, we may be interested in finding its stable model that is good, or even the best possible, according to some measure of quality. A measure of quality can be specified by an aggregate expression. For instance, the expression

```
#sum{X : p(X)}                                                    (6.6)
```

Listing 6.5: Knapsack problem

```
% Knapsack problem.

% input: set weight/3 of pairs (i,w) such that w is the weight
%        of item 1; set value/3 of pairs (i,v) such that v is
%        the value of item i; limit maxweight on the total
%        weight.

{in(I)} :- weight(I,W).
% achieved: in/1 is a subset of the set of items.

:- #sum{W,I : in(I), weight(I,W)} > maxweight.
% achieved: the total weight of items in this subset doesn't
%           exceed maxweight.

% optimize the selection.
#maximize{V,I : in(I), value(I,V)}.

#show in/1.
```

measures the quality of a stable model by the sum of the integers in the set `p/1`; the model in which this sum is the largest or the smallest would be considered the best.

The directives `#maximize` and `#minimize` instruct CLINGO to improve the first stable model that it generated using a `#sum` aggregate expression as the measure of quality, and to keep looking for better and better stable models until the best model is found. For example, the directive

```
#maximize{X : p(X)}
```

will cause CLINGO to generate stable models in which the values of aggregate expression (6.6) are larger and larger. These directives allow us to apply CLINGO to combinatorial optimization problems, where the goal is to find the best among several alternatives.

Consider, for instance, the knapsack problem, where we are given a set of items, each with a weight and a value, and the goal is to determine which items to include in a collection so that the total weight does not exceed a given limit and the total value is as large as possible. This problem is encoded in Listing 6.5. Given this program and the input

```
weight(a,12; b,1; c,4; d,2; e,1).
value(a,4; b,2; c, 10; d,2; e,1).
#const maxweight=15.
```

CLINGO will produce an output like this:

```
Answer: 1
```

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

. . . . . . . . . .

Answer: 11
in(b) in(c) in(d) in(e)
Optimization: -15
OPTIMUM FOUND
```

In every answer, the number after the word `Optimization` shows the total weight of the items in the set `in/1`. (The minus sign is prepended by CLINGO whenever the `#maximize` directive is used, rather than `#minimize`.)

Exercise 6.9. We would like to restrict not only the total weight of the selected items, but also their combined volume. The set `volume/2` consists of the pairs (i, vol) such that vol is the volume of item i; `maxvolume` is the upper bound on the combined volume. How would you modify the program in Listing 6.5 to encode this enhancement of the basic knapsack problem?

Exercise 6.10. In the "unbounded" version of the knapsack problem, an unlimited number of copies of each kind of item is available. What rules would you place in Lines 10 and 15 of Listing 6.6, and what directive in Line 20, to solve this version of the problem?

The program in Listing 6.7 is a modification of the coloring program from Exercise 3.15 (page 51) that colors a graph using the smallest possible number of colors, and thus calculates its chromatic number. Integers between 1 and the number N of vertices of the graph are used as colors. (It is clear that the chromatic number never exceeds N.) The directive in Line 19 exploits the fact that the aggregate expression

```
#sum{1,C : color(X,C)}
```

has the same meaning as

```
#count{C : color(X,C)}
```

(see Exercise 6.5). If we replace `1,C` in that line by `C` then CLINGO will minimize the *sum* of the numbers used as colors. As a result, the colors that we will see in an optimal solution

Listing 6.6: Unbounded knapsack problem (Exercise 6.10)

```
% Unbounded knapsack problem.

% input: set weight/3 of pairs (i,w) such that w is the weight
%        of item 1; set value/3 of pairs (i,v) such that v is
%        the value of item i; limit maxweight on the total
%        weight.

% in(I,N) means that N copies of item I are selected.

_____________________________________________________________
% achieved: in/2 is a set of pairs(i,n) such that i is an item
%           and n is a nonnegative integer such that the
%           weight of n copies of i doesn't exceed maxweight.

_____________________________________________________________
% achieved: the total weight of selected items doesn't exceed
%           maxweight.

% optimize the selection.
_____________________________________________________________

#show in/2.
```

Listing 6.7: Graph coloring using the minimal number of colors

```
% Color the vertices of a graph using the minimal number
% of colors.

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

% color(X,C) means that the color of vertex X is C.

{color(X,1..N)} = 1 :- vertex(X),
                       N = #count{Y : vertex(Y)}.
% achieved: for every vertex X there is a unique C from
%           {1,...,N} such that color(X,C), where N is
%           the number of vertices of G.

:- edge(X,Y), color(X,C), color(Y,C).
% achieved: no two adjacent vertices share the same color.

% minimize the numbers of colors.
#minimize{1,C : color(X,C)}.

#show color/2.
```

Listing 6.8: Optimization version of set packing (Exercise 6.11)

```
 1  % Find the largest possible number of pairwise disjoint
 2  % members of a given list of finite sets.
 3
 4  % input: for a list S_1,...,S_n of sets, its length n and
 5  %        the set s/2 of pairs X,I such that X is in S_I.
 6
 7  % in(I) means that set S_I is included in the solution.
 8
 9  {in(1..n)}.
10  % achieved: in/1 is a set of members of the list.
11
12  I = J :- in(I), in(J), s(X,I), s(X,J).
13  % achieved: the chosen sets are pairwise disjoint.
14
15  % maximize the number of chosen sets
16  ---------------------------------------------------------
17
18  #show in/1.
```

will be small integers, between 1 and the chromatic number of the graph. This modification will also cause CLINGO to terminate faster, because of its symmetry breaking effect.

Exercise 6.11. We would like to solve the optimization version of the set packing problem (Section 3.8), which calls for finding the largest possible number of pairwise disjoint members of a given list of finite sets. What directive would you place in Line 16 of Listing 6.8?

Exercise 6.12. Recall that a clique in a graph is a subset of its vertices such that every two distinct vertices in it are adjacent (Section 3.9). We would like to write a CLINGO program that finds the largest clique in a given graph. What rule would you place in Line 6 of Listing 6.9, and what directive in Line 13, to achieve this goal?

Exercise 6.13. Research papers submitted to a technical conference are reviewed by the conference program committee. Every paper is read and discussed by a group of committee members chosen by the chair of the committee, and this group decides if the paper can be accepted for presentation. To help the chair find a good match between papers and referees, every committee member submits a bid that classifies all papers that need to be reviewed into three categories: "yes" (I want to review this paper), "maybe" (I do not mind reviewing it), and "no" (do not assign it to me). We would like to write a program for CLINGO that automates this part of the work of the chair. Using a list of bids, it should assign each submitted paper for review to a specific number k of committee members so that

Listing 6.9: The largest clique (Exercise 6.12)

```
% Find the largest clique.

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

-------------------------------------------------
% achieved: in/1 is a set of vertices of G.

X = Y :- in(X), in(Y), not edge(X,Y), not edge(Y,X).
% achieved: in/1 is a clique.

% maximize the size of the clique.
-------------------------------------------------

#show in/1.
```

- the workloads of committee members are approximately equal—do not differ by more than 1;
- no committee member is asked to review a submission that he placed in the "no" group;
- the total number of cases when a submission is assigned to a reviewer who placed it in the "yes" group is as large as possible.

What rules would you place in Lines 7, 11, 17, 22, and 26 of Listing 6.10, and what directive in Line 31, to achieve this goal?

6.5 Classical Negation

Recall that an atom is a symbolic constant that may be followed by a list of arguments in parentheses (Section 2.1). A *negated atom* is an atom preceded by the minus sign. When the minus sign is used to form a negated atom, it is called *classical* (or *strong*) *negation*. About an atom and its classical negation we say that they are *complementary* to each other. For example, the atom `p(a)` and the negated atom `-p(a)` form a complementary pair.

Syntactically, a negated atom can be used in a CLINGO program wherever an atom is allowed. Like an atom without classical negation, it can be preceded by one or two negation as failure symbols. When CLINGO generates stable models, it treats each classical negation as if it were an extra character at the beginning of a predicate symbol, except that it does not display stable models that contain a complementary pair. For example, CLINGO's answer to the program

Listing 6.10: Assigning papers to referees (Exercise 6.13)

```
% Assigning papers to referees.

% input: set bid/3 of triples (r,p,b) such that b is bid
%        ("yes", "no", or "maybe") submitted by referee r
%        for paper p; positive integer k.

------------------------------------------------------------
% achieved: referee/1 is the set of referees who submitted
%           bids.

------------------------------------------------------------
% achieved: paper/1 is the set of papers for which bids are
%           submitted.

% review(R,P) means that paper P is assigned to referee R.

------------------------------------------------------------
% achieved: for every paper P there are exactly k referees R
%           such that review(R,P); the bids submitted by
%           these referees for P are different from "no".

------------------------------------------------------------
% achieved: workload(R,N) iff N is the number of papers
%           assigned for review to referee R.

------------------------------------------------------------
% achieved: the difference between the workloads of
%           referees is at most 1.

% maximize the number of "yes" cases.
------------------------------------------------------------

#show review/2.
```

```
p(1..2).
-p(3..4).
```

is

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

but the program

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

has no stable models. In response to the program

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

CLINGO displays two answers,

```
-p(2) -p(3)
```

and

```
-p(2) -p(3) p(1)
```

—including `p(2)` is not an option, because it is complementary to `-p(2)`.

Exercise 6.14. What stable models do you think will be produced by CLINGO for the following programs?

(a)

```
{p}.
q :- not p.
r :- -p.
```

(b)

```
{p}.
q.
-q :- not p.
```

If an encoding uses a pair of propositions with opposite meanings then it is convenient to denote one of them by a negated atom. For example, the program from Exercise 2.17 (page 24) uses the symbol `coprime/1` for the set of numbers between 1 and n that are coprime with k, and the symbol `noncoprime/1` for the set of numbers between 1 to n that do not have this property. Listing 6.11 shows the modification of that program in which `noncoprime/1` is replaced by `-coprime/1`. Listing 6.12 shows the modification of the program from Exercise 2.18 in which `more_than_three/1` is replaced by `-three/1`.

In the last example, the rule in Line 10 expresses that a number between 1 and n cannot be represented as the sum of three squares if there is no evidence that it can. In other

Listing 6.11: Coprime numbers encoded using classical negation

```
% Numbers from 1 to n that are coprime with k.

% input: positive integer n, k.

-coprime(N) :- N=1..n, I=2..N, N\I=0, k\I=0.
% achieved: -coprime/1 is the set of numbers from {1,...,n}
%           that are not coprime with k.

coprime(N) :- N=1..n, not -coprime(N).
% achieved: coprime/1 is the set of numbers from {1,...,n}
%           that are coprime with k.

#show coprime/1.
```

Listing 6.12: Integers requiring four squares encoded using classical negation

```
% Numbers from 1 to n that cannot be represented as the sum
% of 3 complete squares.

% input: positive integer n.

three(N) :- N=1..n, I=0..n, J=0..n, K=0..n, N=I**2+J**2+K**2.
% achieved: three/1 is the set of numbers from {1,...,n} that
%           can be represented as the sum of 3 squares.

-three(N) :- N=1..n, not three(N).
% achieved: -three/1 is the set of numbers from {1,...,n} that
%           can't be represented as the sum of 3 squares.

#show -three/1.
```

words, this rule represents the *closed world assumption* for the property `three/1` in the domain $\{1, \dots, n\}$. The general form of closed world assumption rules is

```
-p(X) :- ··· , not p(X).
```

where the part of the body denoted by dots describes the domain. The rule in Line 9 of Listing 6.11 expresses the opposite assumption about the property `coprime/1`: a number between 1 and n is coprime with k if there is no evidence that it is not.

Classical negation is useful for distinguishing between what is known to be false and what is unknown. For example, the facts

```
candidate(a; b; c; d; e).
elected(a; b).
-elected(c; d).
```

describe the status of five candidates in an election: a and b have won, c and d have lost, and the status of e is still undecided.

6.6 Symbolic Functions

In the programs that we have seen so far, all terms are formed from constants and variables using arithmetic operations. The syntax of CLINGO allows us to form terms also in another way—using a symbolic constant as the name of a function. A symbolic constant followed by a list of terms (separated by commas) in parentheses is a term. For instance, the expressions `f(X,Y)` and `f(g(a),10)` are terms that use `f` as a function of two variables and `g` as a function of one variable. The expression `p(f(X,Y),f(g(a),10))` is an atom, and `f(X,Y) != Z` is a comparison.

Terms containing a symbolic function are similar to records in imperative programming languages.

In this more general setting, an extra clause needs to be added to the recursive definition of the set of values of a ground term given in Section 4.6:

5. If t is $f(t_1, \dots, t_k)$ then the values of t are terms of the form $f(v_1, \dots, v_k)$, where $v_1, \dots, v_k$ are values of $t_1, \dots, t_k$ respectively.

For instance, the only value of `f(a,g(2+2))` is `f(a,g(4))`.

Exercise 6.15. Find all values of the term `f(1..2,g(1..2))`.

Exercise 6.16. What stable models do you think will be produced by CLINGO for the following programs?

(a)

```
p :- f(1,2) = 7.
```

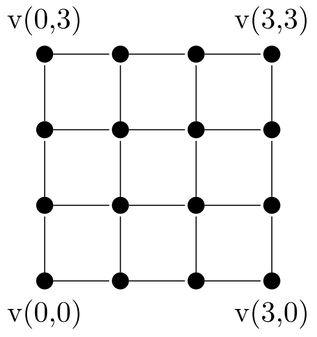

Figure 6.2: Graph with 16 vertices and 24 edges.

(b)

```
p(f(1..2,g(1..2))).
q(X):- p(f(_,X)).
```

Symbolic functions can be used, for instance, to describe points in a coordinate plane. The graph shown in Figure 6.2 can be represented by the rules

```
vertex(v(0..3,0..3)).
edge(v(R,C),v(R,C+1)) :- R = 0..3, C = 0..2.
edge(v(R,C),v(R+1,C)) :- R = 0..2, C = 0..3.
```

If we give these rules as input to the program in Listing 6.7 then CLINGO will determine that the chromatic number of the graph is 2.

We will see other programs that use symbolic functions in Chapter 7.

6.7 Bibliographical and Historical Remarks

The `#count` and `#sum` aggregates are inherited by CLINGO from two older answer set solvers, SMODELS [99] and DLV [75]. Propositional images of programs with aggregates [43] are syntactically more general than propositional rules in the sense of Section 4.1—they may contain implications in the body of an implication, as in the formula $p \leftarrow (q \leftarrow r)$. The definition of a stable model was extended to arbitrary propositional formulas in 2005 [36], and it has a simple characterization in terms of equilibrium logic [101].

Other definitions of stable models for programs with aggregates proposed in the literature [34, 54, 104, 112] are not equivalent to the semantics adopted by the designers of CLINGO, but differences can be seen mostly in artificially constructed counterexamples [62].

`#maximize` and `#minimize` directives appeared originally in SMODELS. In DLV, optimization is achieved using "weak constraints"; that mechanism is available in the language of CLINGO as well.

The approach to computing winners in an election presented in Section 6.2 is known as the *Borda rule*. Many other rules of this kind have been proposed. The *k-approval rule*, for instance, where k is a positive integer, assigns one point to each candidate who is ranked among the first k. (The case of $k = 1$ is called the *plurality rule*). According to the *veto rule*, the winner is the candidate who is ranked last by the smallest number of voters. To give one more example, the *Copeland rule* compares candidates pairwise; a candidate who is preferred by more voters than another gets one point. The DEMOCRATIX system [22] encodes a large number of voting rules in the language of CLINGO, and it is used as a computational tool by researchers in the area of voting theory. The encodings are similar to the one shown in Listing 6.4. DEMOCRATIX is available as a web application, along with its source code and the encodings of the voting rules, at `http://democratix.dbai.tuwien.ac.at/`.

Classical negation was added to the syntax of logic programs in 1990 [51, 102]. With this feature included in the language, the correspondence between rules and defaults mentioned in Section 5.12 is applicable to an arbitrary default (5.70) such that each of the formulas F_i, G_j, H in it is a literal (that is, an atom possibly prefixed by $\neg$). For example, the default

$$\frac{p_1, \neg p_2 \ : \ q_1, \neg q_2}{\neg r}$$

corresponds to the rule

```
-r :- p1, -p2, not -q1, not q2.
```

Informally, the meaning of logic programs with two kinds of negation can be described by three principles: satisfy the rules of the program; do not believe in contradictions; believe in nothing you are not forced to believe [49]. The first and the last of these principles correspond to clauses (i) and (ii) of the characterization of stable models on page 86; not believing in contradictions corresponds to eliminating the stable models containing complementary pairs.

The term "closed world assumption" was originally introduced in the context of deductive databases [106].

Symbolic functions are inherited by answer set solvers from Prolog. They were covered by the original definition of a stable model [50].

Several useful constructs available in CLINGO, including conditional literals, external functions, and multi-shot solving, go beyond the scope of this book. They are described in Potassco User Guide, which can be downloaded from the website of the Potassco project, `https://potassco.org`. Furthermore, we do not discuss here rules with ordered disjunction [16], ASP with sorts [4], ASP with consistency-restoring rules [7, 5], constraint ASP [8, 9, 46, 67], ASP with preferences [17], and probabilistic ASP [23].

One other interesting extension of answer set programming is motivated by the fact that ASP definitions of functions are somewhat cumbersome. Compare, for instance, the definition of the predicate `fac/2` in lines 5 and 6 of Listing 2.6 (page 26) with the definition of factorial in Haskell (page 10). This observation has led several researchers to the idea of merging ASP with functional programming [6, 11, 19].