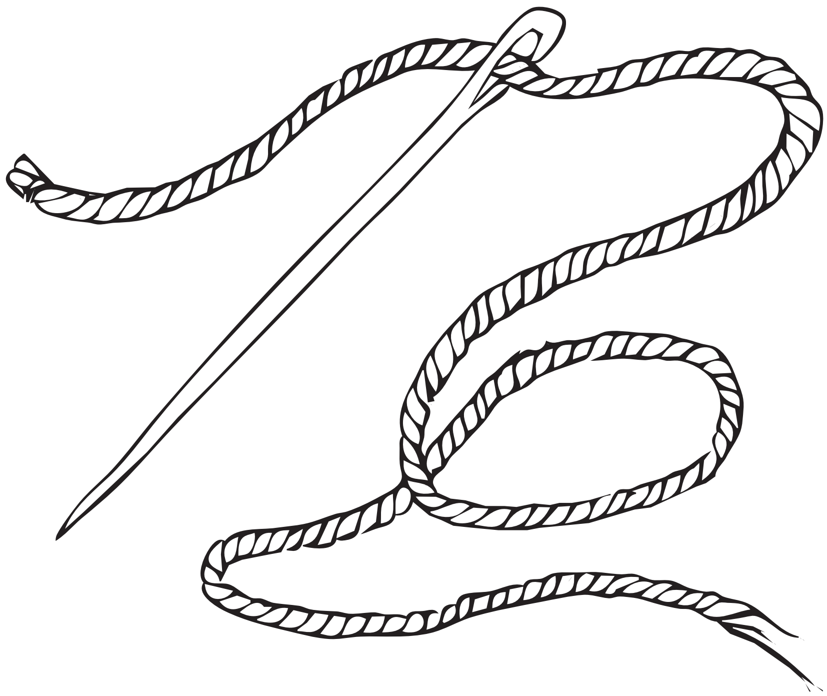
# Lparse 1.0
# User's Manual



**Tommi Syrjänen**

# Contents

## 0.1 Licence

## 0.2 Recent Changes

This section gives quick overview on recent changes that are made in the accepted language. The more detailed version notes are contained in file NEWS in the distribution directory.

`Lparse-`**1.0.3**
: contains several changes. The most important is that the definition of domain predicates extended to cover stratified rules. See Section 4.4 for details. Second, there's now an alternative syntax for declarations where the keywords are preceded by the character `#`. Two new declarations, `#option` and `#domain` were added (see Section 5.5).

`Lparse-`**1.0.1**
: added a new internal function. The call '`weight(a(X))`' returns the weight of the literal $a(X)$.

`Lparse-`**0.99.61**
: adjusted the weight declarations a little bit so that now negative literals are defaulted to positive ones if there are no explicit declarations for them. This behavior can be turned off with the command line argument '`--separate-weight-definitions`'.

`Lparse-`**0.99.60**
: changed the behavior of weight declarations so that now positive and negative literals are differentiated. For example,

```
weight a = 5.
weight not a = 10.
```

assigns different weights for $a$ and not $a$.

The default behavior of quoted strings also changed and now quotes are always retained and $a$ and "$a$" are different atoms. The old behavior can be accessed with the '`--drop-quotes`' command line argument.

`Lparse-`**0.99.58**
: allows expressions to be used in a more intuitive way inside constraint and weight literals. For example, in a rule

```
a(X,Y) :- 1  X + Y == 0, Y < X  1,
      foo(X,Y).
```

the atom a(X,Y) is true when exactly one of the expressions evaluate to true.

Additionally, an empty conditional literal is now treated as an unsatisfied literal in all cases.

## 0.3 Notation

The parts of this manual that are changed in the current manual version are denoted by putting a continous black line in the left margin, as in this paragraph.

Similarily, changes that have happened recently are denoted by a dashed line in the left margin.Similarily, changes that have happened recently are denoted by a dashed line in the left margin.

# Chapter 1

# Introduction

SMODELS is a system for answer set programming. It consists of `smodels`, an efficient implementation of the stable model semantics for normal logic programs, and `lparse`, a front-end that transforms user programs into form that `smodels` understands.

Answer set programming [6, 10] is a programming paradigm completely different from traditional procedural programming. Instead of writing algorithms to solve a problem in hand, the programmer describes the problem using a formal language and an underlying engine finds a solution to the problem.

SMODELS programs are written using standard (though extended) logic programming notation. That is, the programs are composed of atoms and inference rules. An atom represents a claim about the problem universe and it may be true or false. Inference rules are used to encode relationships between atoms. An answer to a problem is a set of atoms, called a stable model, that tell which atoms are true.

A SMODELS program may have one, none, or many stable models. The stable models of a program may be seen as a set of rational beliefs about the program. That is, if we think that a program is a knowledge base encoding the relationships between objects and a stable model is a set of those things in our universe that we believe to be true, then our beliefs are consistent and well-founded. Consistency means that we don't believe in two contradictionary things and well-foundedness means that we have some reason for our belief. We don't want to believe that the Moon is made of green cheese unless somebody gives a coherent theory that explains why Moon is actually a big dairy product. For formal definition of the stable model semantics see Section 4.2.

SMODELS has two parts, `smodels` and `lparse`[1]. The first part, `smodels`, is the actual logic programming engine doing all the hard work and `lparse` just adds a layer of syntactic sugar on top of it. The `smodels` has been developed in the Laboratory for Theoretical Computer Science in Helsinki University of Technology by Patrik Simons [13, 8, 11, 14, 12, 15, 9, 10] and the `lparse` has

[1] Actually, in Section 1.4 we see that there are also other front-ends in addition to `lparse`.

been developed by Tommi Syrjänen [19].

The newest versions of `smodels`, `lparse`, and this document are available at

`http://www.tcs.hut.fi/Software/smodels`.

## 1.1 Short Primer on Logic Programming

This section has a really short primer on the subject of logic programming in general. It is aimed for readers who have no prior experience on logic programming. For an extensive treatment on the traditional logic programming techniques, you should consult *The Art of Prolog* by Sterling and Shapiro [18].

There are basically four kinds of things in logic programming languages: atoms, constants, variables, and rules. They are presented here rather informally. For formal stuff, see Chapter 4.

**Constant**

Constants are the individual things that exists in the universe of the problem domain. Constants are either numbers or symbolic constants. In most logic programming languages the initial letter of a constant is written in lower case.

Few examples of constants: `a, 10, foo, bar`.

**Variable**

Variables are used to generalize things. Unlike traditional programming language, you don't usually assign a value to a variable directly. Instead, the underlying engine finds the correct values (or *substitutes* constants in place of the variables) for them.

Variables start with a capital letter, like in: `X, Foo, Bar`.

**Atom**

An atom consists of a predicate symbol that is followed by a parenthesized list of constants or variables. Atoms are used to express relationships between constants. For example, an atom $parent(john, jill)$ might tell us that John is Jill's parent. An atom has two possible truth values, true and false.

**Rule**

Rules allow us to make inferences based on the predicates. For example, a rule:

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
```

would tell us that if $X$ and $Y$ both have the same parent $Z$, then they are siblings. Rules are composed of two parts: the head (part to the left of '`:-`') and the body (right to '`:-`').

> The idea here is that if every atom in the body (or tail as it is also often called) is true, the head must also be true. So if there is some way to substitute constants for the variables $X$, $Y$, and $Z$ such that both $parent(Z, X)$ and $parent(Z, Y)$ are true, we can infer that $sibling(X, Y)$ is also true.

In classic logic programming languages (like Prolog), the inferences are usually made top-down. That is, we give some atom as a query string, and the system tries to find a way to make it true.

For example, if we were interested in finding out whether Jack is Jill's sibling, we would issue a query (called a *goal*) $sibling(jack, jill)$? The system would then scan through the rules until it finds some rule which has the predicate `sibling` as its head. After finding the rule presented above, the system would substitute $jack$ for the variable $X$ (denoted: $X/jack$) and $jill$ for $Y$ ($Y/jill$).

Now the system has established that Jack and Jill are siblings if there exists some $Z$ that is parent of both of them. Next, the system would issue a new query (called a *subgoal*) $parent(Z, jack)$ in order to find the parent of Jack. Suppose that Joan is Jack's mother. Then the subgoal succeeds and the systems finds the substitution $Z/joan$.

The system will then check whether Joan is also Jill's mother by issuing the query $parent(joan, jill)$. If the query succeeds, the system answers yes to our original question ($sibling(jack, jill)$?). If Joan is not Jill's mother, the query fails and the logic programming engine backtracks and tries to find another substitution for $Z$. If no such substitutions can be found, the systems answers no to our question. If there are more than one rule for a predicate, the rules are tried in order. If one fails, the next one is checked.

Quite often the logic programs can be divided into two parts: a set of inference rules and a database of facts for the rules to make inferences with.

For example, the following program encodes a simple family database.

**Program 1.1**

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
mother(X,Y) :- parent(X,Y), female(X).
uncle(X,Y) :- parent(Z, Y), sibling(Z,X), male(X).
female(joan).  female(jill).  male(jack).
parent(joan, jack).  parent(joan, jill).
```

## 1.2 Stable Model Basics

Traditional logic programming systems are query-driven. That is, you enter a question and the system then tries to find an answer to it. At any point only those variables that are somehow involved in the query have values binded to them.

SMODELS works in a different way. In the first phase all variables are removed from the program by substituting all possible values for them in all rules. This phase is called grounding (see Section 4.3) and it's the `lparse`'s job. In the next phase `smodels` computes the stable models of the program.

A model is a set of atoms that satisfies every rule in the program. A rule is satisfied if either its head is true in the model or some literal in the rule body is false. A model is stable when it meets some other requirements, that are formalized in Section 4.2. Informally, a model is stable if every atom in it has some "reason" to be there: for each atom in the model there has to be some rule that has the atom as a head such that the rule body is true in the model.

As a simple example of what stable models are about, consider the following program segment that could represent a part of a PC configuration system:

**Program 1.2**

```
ide_drive :- hard_drive, not scsi_drive.
scsi_drive :- hard_drive, not ide_drive.
scsi_controller :- scsi_drive.
hard_drive.
```

Here the first rule says that if we have a *hard_drive* in our computer and we don't have a *scsi_drive*, we must have an *ide_drive* in it. The next rule says the same thing about *scsi_drive*: if we have a *hard_drive* that is not an *ide_drive*, it has to be a *scsi_drive*. The third rule states that if we have a *scsi_drive* in the computer, we must also include a *scsi_controller* in it. The last rule is a fact that tells that our computer has a *hard_drive* in it.

This program has two stable models. The first one is:

$$M_1 = \{hard\_drive, ide\_drive\}$$

and the second one is:

$$M_2 = \{hard\_drive, scsi\_drive, scsi\_controller\}.$$

The first two rules of the program represent a choice point: if we have a hard drive in the computer, we must choose between an ide-drive and a scsi-drive.

If we add *scsi_controller* to the first stable model the resulting set of atoms is still a model of the program in the propositional sense but it is no longer a stable model. That is because *scsi_controller* is needed only when *scsi_drive* is present, and when it is missing there is no reason to add *scsi_controller* in the model.

The semantics of ordinary rules matches that of logical implication: if the body is true, the head must also be true. In the program above, we used recursive not-atoms to model a choice. The following program segment illustrates further this practice:

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

If $b$ is not true, then $a$ must be true and vice versa. However, this construction is not exclusive or of the atoms, since it is possible that some other part of the program forces both $a$ and $b$ to be true. To get XOR one should add a rule of the form:

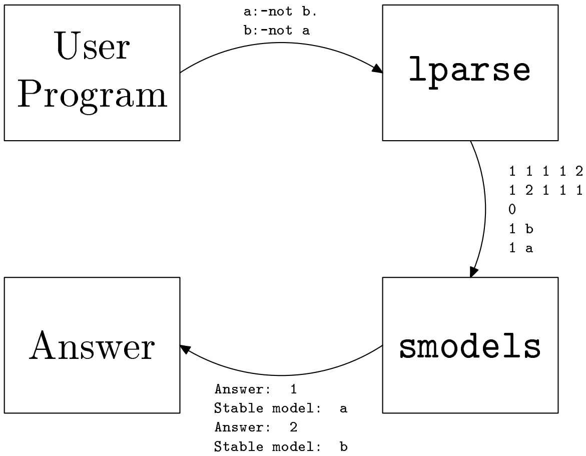



Figure 1.1: The way of a logic program

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

to the program. Rules without heads act as integrity constraints; if a body of such a rule is satisfied, the model candidate is rejected.

## 1.3 A Practical Example

The basic concepts are introduced here by using the NODE COLORING problem as an example. In a NODE COLORING instance we are given a set of nodes and a set of edges that connect the nodes. The problem is to use some fixed number of colors to color each node so that two adjacent nodes don't have same color. In this example we use three colors: red, blue, and yellow.

As this example uses only basic rules, it is quite long. In Chapter 6 we see how the extended rules can be used to encode this problem using only two rules.

The NODE COLORING can be implemented with the following program.

**Program 1.3**

```
color(red).  color(blue).  color(yellow).
col(X,red) :- node(X), not col(X, blue), not col(X,yellow).
col(X,blue) :- node(X), not col(X, red), not col(X,yellow).
col(X,yellow) :- node(X), not col(X, blue), not col(X,red).
fail :- edge(X,Y), color(C), col(X,C), col(Y,C).
node(a).  node(b).
node(c).  node(d).
edge(a,b).  edge(b,c).
edge(c,d).  edge(d,a).
compute 1 { not fail }.
```

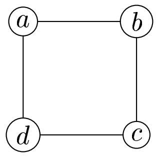



Figure 1.2: The graph of Program 1.3

The first line:

```
color(red).  color(blue).  color(yellow).
```

defines the colors we are allowed to use. The predicate *color* is defined in such way that it can be used as a domain predicate (see Section 4.4) later in the program.

The rule,

```
col(X,red) :- node(X), not col(X, blue), not col(X,yellow).
```

states that if a node is not *blue* and it is not *yellow*, then the node has to be *red*. The next two rules are otherwise identical but they are for colors *blue* and *yellow*. Here the predicate *node*$(X)$ acts as a domain predicate that enumerates the possible values of the variable $X$.

The rule

```
fail :- edge(X,Y), color(C), col(X,C), col(Y,C).
```

ensures that neighbouring nodes have different colors. The atom *fail* is true exactly when two connected nodes have the same color. Later we force `smodels` to search for only models where *fail* is not true. Here the predicate *node* is not needed to give domain for node variables $X$ and $Y$ because *edge* is also a domain predicate.

The next part of the program:

```
node(a).  node(b).
node(c).  node(d).
edge(a,b).  edge(b,c).
edge(c,d).  edge(d,a).
```

defines a simple graph with four nodes and four edges. This graph is shown in Figure 1.2. Usually, the graph is stored in another file so that we don't have to duplicate the inference rules for each graph.

The last line of the program:

```
compute 1 { not fail }.
```

tells `smodels` that we want only one model and that the atom *fail* may not be in the model. This rules out all models where two adjacent nodes have the same color.

The atom *fail* is used to signal that something is wrong with the model, namely that two adjacent nodes have the same color. Later in the program we demand that *fail* may not be true in any stable model of the program.

The same effect could be achieved by using construction:

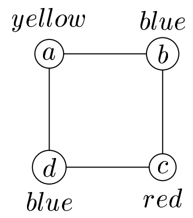



Figure 1.3: An answer of Program 1.3

```
:- edge(X,Y), color(C), col(X,C), col(Y,C).
```

Rules with empty heads work as constraints on the models of the program. If a variable binding makes the the body of the rule true, the binding is discarded as there's no way to satisfy its head.

The code of the NODE COLORING example is stored in directory `examples` as `color1.lp` of the `lparse` distriburion. The example graph is in the same directory as `graph1`.

The process of running this program through smodels would look like:

```
% lparse color1.lp graph1 | smodels
smodels version 2.10.  Reading...done
Answer:  1
Stable Model:  edge(d,a) edge(c,d) edge(a,b) edge(b,c)
node(a) node(d) node(c) node(b) col(a,yellow) col(c,red)
col(d,blue) col(b,blue) color(yellow) color(red) color(blue)
True
Duration:  0.030
Number of choice points:  3
Number of wrong choices:  0
Number of atoms:  24
Number of rules:  35
Number of picked atoms:  45
Number of forced atoms:  0
Number of truth assignments:  152
Size of searchspace (removed):  12 (0)
```

As the first line of output, `smodels` prints its version information. The next lines give the first model found. In the model nodes $d$ and $b$ are colored blue, node $a$ is yellow and node $c$ is red. The answer is also shown in Figure 1.3.

The word `true` below the model tells that there may be also other models, but `smodels` didn't compute them[2]. If there are no models left, the line reads `false`. The same message is also displayed when the program has no models at all.

The rest of the lines give some statistics about your program. The `duration`

[2]It's possible that there are no more models, but smodels reports `true` always when the whole search space is not explored.

tells how long the search took in seconds.

The number of *choice points* tell how many times `smodels` had to guess a truth value for a ground atom. This time `smodels` guessed the correct value for each atom (number of *wrong choices* is zero) and thus it didn't have to backtrack.

The next lines tell that there were a total of 24 *atoms* and 35 *rules* in the grounded program (the original had four non-ground rules, 11 facts, and 12 atoms). In general, the number of choice points is more important than the number of rules or atoms when we want to compare complexities of problems.

The rest of the lines show how the `smodels` heuristics worked for this program. The number of *picked atoms* tell how many times smodels lookahead heuristics managed to pick a truth value to an atom. The number of *forced atoms* tell how many atoms were added to the model because their negation would have caused a contradiction. The number of *truth assignments* tells how many times `smodels` assigned a truth value to an atom.

The *size of searchspace* tells the maximum number of choices we may have to do before we can be certain whether a model exists or not. In this example the size of the search space is half the number of atoms, since the eleven domain predicates are always true and the truth value of *fail* depends directly on the values of the color predicates.

It is often useful to be able to see what exactly is output from `lparse`. This can be accomplished by using the `-t` command line argument:

```
% lparse -t color1.lp graph1
edge(d,a).
edge(c,d).
edge(a,b).
edge(b,c).
node(a).
node(d).
node(c).
node(b).
fail :- col(d,yellow), col(a,yellow).
fail :- col(c,yellow), col(d,yellow).
fail :- col(a,yellow), col(b,yellow).
[24 further output lines snipped]
compute 1 { not fail }
```

At the beginning of this program the domain predicates are output just like they were entered in the input program. The next three lines define three of the cases where the constraints of the problem instance are broken because two neighboring nodes have the same color.

## 1.4 Different Smodels Front-Ends

Currently, there is a host of different front-ends to smodels:

1. `lparse` is the most feature-rich of the different parsers and front ends and it is the default one you should use when you are writing SMODELS programs.

2. `smodels` API is a library interface that allows you to call the `smodels` procedure from any C++ program. Currently, there is no single document that explains the API but the example directory of the `smodels` distribution contains four examples on using it.

3. *parse* is the original parser of smodels. It produces only `smodels` 1.x output, and is now outdated.

4. *pparse* ("primitive parser") is a simple parser that produces `smodels` 2.x output but it accepts only ground programs. Its syntax for extended rules is different from `lparse`'s syntax.

5. *mcsmodels* ("model checking smodels") is a deadlock and reachability checker that can be used to verify 1-safe Petri nets [4]. It is written by Keijo Heljanko and it is available at

   `http://www.tcs.hut.fi/ kepa/tools/`

6. *dlsmodels* is an older version of mcsmodels which only detects deadlocks. It is available at same place as mcsmodels.

# Chapter 2

# Installation

Lparse comes now with an installation script `configure` that should make the installation easier. The procedure to follow is:

1. `cd` to the directory containing `lparse` source code and type `./configure` to configure lparse for your system.

2. Type `make` to compile the binaries.

3. Type `make install` to install lparse.

4. If you want to remove the object files from the source code directory you may type `make clean` to do it.

By default, `lparse` is installed to the directory `/usr/local/bin`. You may change the directory by giving `configure` the option `--prefix=`*path*.

There is a suite of SMODELS programs that can be used to check that `lparse` is functioning correctly. Currently, it is not yet complete, but I'll expand its functionality in forthcoming lparse releases.

The test programs are stored in directory `tests` and the expected results are in directory `tests/results`. There is a perl script `test_all` that performs all tests and reports any errors. The tests can be also run with `make check` command.

As a security issue, you should *never* run `lparse` with setuid bit set. It is possible to call user-defined C/C++-functions from SMODELS programs (see Section 5.7.3) and if the setuid bit is on, a malicious adversary can run basically anything with the permissions of the owner of `lparse`.

## 2.1 Installation on Windows systems

Lparse has been succesfully compiled on the Microsoft Windows 95/98 systems with Borland C++ version 5.5. It may compile also with other compilers and other Windows versions but that hasn't been tested. If you have the GNU programming tools installed on the system, you can follow the directions in the previous section, otherwise ensure that your compiler is correctly installed and configured and issue the following commands to the command prompt:

1. `setup.bat`
2. `cd src`
3. `make`

The command `setup.bat` copies the Windows configuration file to the correct place and creates the makefile. The command `make` creates the actual executable `lparse.exe` in the `src` directory of the distribution and it may then be copied to the desired place. A precompiled Windows 98 binary is available at the `lparse` homepage `http://www.tcs.hut.fi/Software/smodels/lparse`.

# Chapter 3

# Invoking Lparse

The command line synopsis of `lparse` is as follows:

```
Usage:  lparse  [-1] [-c const=number]
                [-d all | facts | positive | none]
                [-D] [-g file] [-i] [-n number]
                [-r [1 | 2 | 3]] [-t] [-v] [-w n]
                [-W warning] [--dlp] [--atom-file file]
                [--allow-inconsistent-answers]
                [--drop-quotes] [--partial]
                [--separate-weight-definitions]
                [--true-negation] [--version]
                file1 file2 ...
```

The meanings of the options will be detailed in Section 3.1. Meanwhile, the rest of this section shows how `lparse` is most often used in practise.

`Lparse` is used as a front end to `smodels`. The usual way to do this is to pipe the output of `lparse` directly to `smodels`:

`% lparse` *`input_file`*$_1$ *`input_file`*$_2$ `| smodels`

It is also possible to output the ground program in plain text, using the option `-t`:

`% lparse-t` *`input_file`*

If the logic program has some varying integer parameter, its value can be entered from command line with the option `-c`:

`% lparse -c` *`parameter=value input_file`*

By default the output file format is for `smodels` 2.x, but it is also possible to output `smodels` 1.x format by using the option `-1`. Note that there are only few (if any) reasons to use `smodels` version 1.x as `smodels` 2.x can do anything that version 1.x can do and it's much better doing it.

## 3.1 Lparse Options

The available options are:

`-1`

Use `smodels` 1.x output format.

`-c` ***const=n***

Define the identifier *const* to be a numeric constant with the value $n$. This definition overrides any `const` statements for $n$ in the program.

`-d all | facts | positive | none`

Control which domain predicates are emitted. The default is `facts`.

- `all`: All domain literals are emitted. Rules with unsatisfiable negative literals in their bodies are not removed.
- `facts`: All domain predicates in the rule bodies are dropped from output.
- `none`: All domain literals are dropped.
- `positive`: Negative domain literals in the rule bodies are dropped.

`-D`

Debug `lparse` data structures. With this option `lparse` creates really small internal storage tables so that their behavior can be tested more easily. Not recommended for normal use.

`-g` ***file***

Read a previously grounded *file* to memory before grounding the program. This is useful when you don't want to ground the full program each time. See Section 7.4 for more details.

`-i`

Disable the internal functions.

`-n` ***n***

Set the desired number of models to $n$. This overrides any `compute`-statements in the program.

`-r [1|2|3]`

Enable the regular model extension (see Section 4.8.2 for details). The optional numeric parameter is used to control what integrity constraints are emitted.

`-t`

Print output as text.

`-v`

Print `lparse` version information

`-w` *n*

Set the default weight of literals to $n$. If this option is not given, the default weight will be 1.

`-W` ***warning***

Control the warnings emitted by `lparse`. Possible values are: `all`, `arity`, `extended`, `library`, `unsat`, `weight`, `syntax`, `typo`, and `error`. For a detailed explanation, see Section 7.3.

`--atom-file` *file*

Output the symbol table of the grounded program to *file*. See Section 7.4.4 for more details.

`--allow-inconsistent-answers`

Enables the classical negation extension and additionally reports also inconsistent answer sets. See Section 4.7 for details.

`--dlp`

Use disjunctive logic programming semantics with choice rules. For details, see Section 4.8.1.

`--drop-quotes`

Remove quotation marks from strings when possible. For example "$a$" becomes $a$ but "123" doesn't change.

`--partial`

Enables the partial model extension. This switch is identical in behavior to `-r 2`.

`--separate-weight-definitions`

Do not default the weight of a negative literal to the weight of a positive one.

`--true-negation`

Enables the classical negation extension. See Section 4.7 for details.

## 3.2 Smodels Options

These options are current for smodels version 2.25.

Usage: `smodels [number] [-w] [-nolookahead] [-backjump] [-randomize] [-internal] [-tries` *number*`] [-conflicts` *number*`] [-seed` *number*`]`

The first *number* determines the number of stable models to be computed. A zero indicates all.

`-backjump`

Allow backtrack to jump over several choice points.

`-nolookahead`
: Do not use lookahead at all.

`-sloppy_heuristic`
: Use reduced lookahead heuristics. This may speed `smodels` if the program has an "easy" structure.

`-randomize`
: Make a randomized but complete search.

`-internal`
: Simplify the program and print it out in an internal form.

`-w`
: Compute only the well-founded model of the program.

`-tries` ***number***
: Do a stochastic random search for *number* times. This method is not complete.

`-conflicts` ***number***
: When doing a stochastic search, stop when *number* conflicts are found.

`-seed` ***number***
: Use *number* as the seed for random parts of computation.

# Chapter 4

# Theoretical Stuff

This chapter discusses the theoretical foundation of SMODELS so it shouldn't surprise anybody that the material in this chapter is quite theoretical. The only things that are really necessary to know while using `lparse` are the informal parts of Section 4.4 (pages 25–27).

If the presentation seems to be too heavy on first reading, feel free to skip to the start of next chapter on page 38[1].

## 4.1 Basic Terminology

An *atom* is of the form $p(a_1, \ldots, a_n)$ where $p$ is a $n$-ary predicate symbol and $a_1, \ldots, a_n$ $(n \geq 0)$ are terms. A *term* may be either a variable, a constant, or a function $f(t_1, \ldots, t_m)$ where $t_1, \ldots, t_m$ are terms. An atom (or a function) with no variables is called a *ground* atom (function).

A *literal* is either an atom $a$ or its negation not $a$. Literals of the form not $a$ are called *not-atoms* or *negative literals*.

A *rule* is of the form

$$h \leftarrow l_1, \ldots, l_n \ . \tag{4.1}$$

where the rule *head* $h$ is an atom and literals $l_1, \ldots, l_n$ $(n \geq 0)$ form the rule *body*[2] If the rule body is empty $(n = 0)$, the rule is called a *fact*. A rule is called a *Horn rule* if it doesn't have any negative literals. A normal *logic program* is a set of rules.

The rules of the form (4.1) are called *basic rules*. `Lparse` also supports a number of extended rule types. However, for a while we consider only basic rules and the extensions are presented in Section 4.6.

The *Herbrand universe* of a logic program is the set of all ground terms that can be constructed using function symbols and constants in that program. The *Herbrand base* **HB** of a logic program is the set of all ground atoms that can

[1] Of course, after reading the informal parts.

[2] also called the rule *tail*.

be constructed using the predicates in the program and terms in the Herbrand universe.

For example, in a program

**Program 4.1**

```
d(a).  d(b).
foo(X) :- not bar(X).
bar(X) :- not foo(X).
```

the Herbrand universe is $\{a, b\}$ (these two constants are the only ground terms of the program), and the Herbrand base is $\{foo(a), foo(b), bar(a), bar(b)\}$.

An *instance* of an atom, a literal, a rule, or a function is constructed by replacing all variables in it by ground terms. The *Herbrand instantiation* of a program is the set of all ground instances of the rules of the program that may be constructed using terms in the Herbrand universe of the program.

For example, the Herbrand instantiation of the Program 4.1 is:

**Program 4.2**

```
d(a).  d(b).
foo(a) :- not bar(a).
foo(b) :- not bar(b).
bar(a) :- not foo(a).
bar(b) :- not foo(b).
```

An *interpretation* $I$ of a logic program is a subset of its Herbrand base. An interpretation assigns a *truth valuation* $\mathcal{V}(a)$ to each element of the base. If an atom $a$ is in the interpretation, its valuation $\mathcal{V}(a) = \mathsf{true}$ in the interpretation, otherwise $\mathcal{V}(a) = \mathsf{false}$. An *extension* $E_p$ of a predicate $p$ is the set

$$E_p = \{a \mid \mathcal{V}(a) = \mathsf{true} \text{ and the predicate symbol of } a \text{ is } p\} \ . \qquad (4.2)$$

An atom $a$ is *satisfied* by an interpretation if its valuation is $\mathsf{true}$. A not-atom not $a$ is satisfied if the valuation of $a$ is $\mathsf{false}$. A basic rule is satisfied if either at least one of its body literals is not satisfied by the interpretation or if its head is satisfied.

If an interpretation satisfies every rule in the Herbrand instantiation of a program, it is a *(Herbrand) model* of the program.

**Example 4.1**

An interpretation $I = \{d(a), d(b), foo(a), bar(b)\}$ is a model of Program 4.1 because all rules of Program 4.2 are satisfied:

Rules `d(a); d(b); foo(a) :- not bar(a);` and `bar(b) :- not foo(b)` are satisfied because their heads are in the model. The rule `foo(b) :- not bar(b)` is satisfied because `not bar(b)` is not satisfied. $\Diamond$

A model is *minimal* if no proper subset of it is also a model. An old theorem by M.H. van Emden and R.A. Kowalski [20] states that a logic program without negative literals has a unique minimal Herbrand model that is called the *meaning* of the program. The minimal model $M$ of a program $P$ can be found by using the Knaster-Tarski operator $T_P$ where

$$T_P(M) = \{h \mid h \leftarrow l_1, \ldots, l_n \text{ is a rule in the program and } l_1, \ldots, l_n \in M\} \tag{4.3}$$

We can construct the minimal model by starting with an empty set and applying the $T_P$ operator until a fixed point is found.

**Example 4.2**

Suppose that the Herbrand instantiation of a program is

```
a.  b.
c :- a.
d :- c, b.
e :- f.
```

Then the minimal model is found in following steps:

1. Initially, $M = \emptyset$. Now $c$, $d$, and $e$ cannot be added to the model since their bodies are not in the partial model. However, since the bodies of first two rules are empty, they may be added to it to get a partial model $M = \{a, b\}$.
2. Now the body of the rule for $c$ is in the partial model, so $c$ is added to the partial model and the resulting partial model $M = \{a, b, c\}$.
3. Because both $c$ and $b$ are in the partial model, $d$ must be added to it also, and $M = \{a, b, c, d\}$.
4. There are no more rules with true bodies that can be added to the model, so the meaning of this program is $M = \{a, b, c, d\}$.

$\Diamond$

## 4.2 Stable Model Semantics

The concept of meaning that was defined above doesn't generalize nicely to logic programs with negations and something else is needed. The *stable model semantics* [2] for a ground logic program $P$ is defined in the following way:

Let $M$ be any set of atoms from $P$. The *reduct* $P^M$ of the program with respect to $M$ is obtained by deleting from $P$

1. each rule that has a negative literal not $a$ in its body when $a$ belongs to $M$; and
2. all negative literals in the bodies of remaining rules.

As $P^M$ is negation-free, it has a unique minimal Herbrand model. If this model coincides with $M$, then $M$ is a *stable model* of the program $P$.

The intuitive explanation of the reduct is that if we believe that $M$ is the set of all true ground atoms of the program, then any rule that depends on a literal not $b$ when $b$ is true cannot be used in deduction and may be discarded. Also, every literal not $b$ is trivially satisfied if $b$ is false and can be discarded. If $M$ is the set of atoms that follow logically from $P^M$, we can say that our belief was consistent and "rational".

**Example 4.3**

Consider the program $P$:

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

The program has two stable models, $M_1 = \{a\}$ and $M_2 = \{b\}$. We can see that $M_1$ is a stable model by first taking the reduct to get the program $P^M$:

```
a.
```

As the only remaining rule has an empty body, the minimal model of the program is $\{a\}$ that is the same set that we started with. The other stable model comes similarly.

In propositional logic this program has also third model, $\{a, b\}$ but this model is not stable, as its reduct is empty so the minimal model of the reduct is also empty, not $\{a, b\}$.

◊

All stable models are *justified* in the sense that every atom in a model has to have some *reason* to be in there; if an atom is in a model, there has to be a rule such that the atom occurs at the head of it and all body literals are satisfied. The third model in the above example was not justified because both rules had unsatisfied bodies.

The set of stable models for a non-ground logic program is defined to be the set of stable models of the Herbrand instantiation of the program.

## 4.3 Grounding

A *grounding* transforms a normal logic program into an equivalent ground logic program where the equivalence is defined as having the same set of stable models. A grounding is *local* if it is possible to do the grounding one rule at a time. A *variable binding* is a substitution that maps a subset of variables of a rule into ground terms.

In the previous section we defined the set of stable models of a general program to be the set of stable models of its Herbrand instantiation. Why not use it directly? The answer is that in practice it is not possible to generate all Herbrand instances of a given logic program because the size of the Herbrand instantiation is practically always exponential with respect to the size of the original problem.

Usually most of the rules in the Herbrand instantiation have unsatisfiable bodies and they may be discarded without affecting the set of stable models.

Consider the following program:

**Program 4.3**

```
d(a).  e(b).  e(c).
foo(X) :- d(X), not bar(X).
bar(X) :- d(X), not foo(X).
```

In the Herbrand instantiation of the program, the last two rules are both instantiated for each of the three constants of the program. The complete instantiation is:

**Program 4.4**

```
d(a).  e(b).  e(c).
foo(a) :- d(a), not bar(a).
foo(b) :- d(b), not bar(b).
foo(c) :- d(c), not bar(c).
bar(a) :- d(a), not foo(a).
bar(b) :- d(b), not foo(b).
bar(c) :- d(c), not foo(c).
```

It is not possible to deduce atoms $d(b)$ and $d(c)$ in the Program 4.4 and any rule that depends on either of them cannot have its body true. Those rules can be discarded without affecting stable models. In effect, the program can be shortened to:

**Program 4.5**

```
d(a).  e(b).  e(c).
foo(a) :- d(a), not bar(a).
bar(a) :- d(a), not foo(a).
```

The crucial question now is, how do we know which rules can be dropped out? `Lparse` does the job by dividing the predicates into two classes, domain and non-domain predicates. The intuition is such that the non-domain predicates are the ones that we are interested in and domain predicates just give all possible variable bindings.

Informally domain predicates are the predicates that are not defined using negative recursion.

In Program 4.3 predicates $d$ and $e$ are domain predicates, while *foo* and *bar* are not since they depend recursively on each other. The recursion is negative since there are the two `not`s in the rules. There is a longer discussion on domain predicates in the next section.

## 4.4 Domain Predicates

In beginning of this section the notion of *domain predicates* is introduced quite informally and the formal definitions are in Section 4.4.4. There are also some tips on how to construct domains using relational algebra in Section 4.4.3.

### 4.4.1 What's New?

**NOTE THAT THE DEFINITION OF DOMAIN PREDICATES HAS CHANGED IN LPARSE-1.0.3. I HAVEN'T HAD TIME TO DOCUMENT THE NEW FORMAL DEFINITION BUT IT WILL COME SOON. MEANWHILE, HERE IS A BRIEF AND HIGHLY INFORMAL DESCRIPTION ABOUT THE NEW DOMAIN PREDICATES.**

In `lparse` the grounding is done using domain predicates. Previously, a predicate was a domain predicate exactly when it didn't have recursion in its definition. Now a domain predicate can be defined via positive recursion.

The main idea is that a hierarchy is created from the predicate symbols where a predicate $P$ is on higher level than the predicate $Q$ if $P$ depends on $Q$. The actual definition is a little quirkier since we want that two predicates that depend positively on each other may be in the same level. If two predicates depend negatively on each other, they will be assigned to the highest level of the hierarchy, as well as all predicates that depend on them.

A predicate that is not on the highest layer (called $\omega$-layer is a domain predicate). The extended domain restriction condition states that all variables that occur in a rule have to occur in some positive literal that is on lower level than the rule head.

For example, consider the program:

**Program 4.6**

```
number(0..n).
odd(X+1) :- number(X), even(X).
even(X+1) :- number(X), odd(X).
even(0).
two_divides(X) :- odd(X).
interesting(X) :- number(X), not dull(X).
dull(X) :- number(X), not interesting(X).
interesting_odd(X) :- odd(X), interesting(X).
```

In this program all predicates except *dull*, *interesting*, and *interesting_odd* are domain predicates. The predicates *dull* and *interesting* depend on each other negatively and *interesting_odd* depends on *interesting*. The dependency graph of the program is shown in Figure 4.1 and its stratification in Figure 4.2.

The predicate hierarchy is constructed using the following rules:

1. If $P$ depends on $Q$ and $Q$ doesn't depend on $P$, then $P > Q$;
2. If $P$ depends positively on $Q$ and $Q$ depends positively on $P$, then $P = Q$;

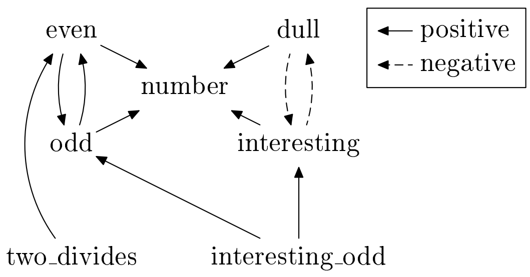



Figure 4.1: The dependency graph of the program (4.6).

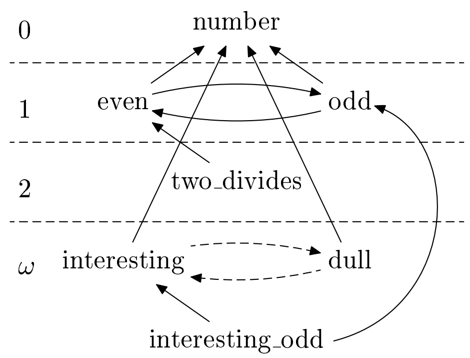



Figure 4.2: The predicate hierarchy of Program 4.6

3. If $P$ and $Q$ depend on each other and there is a negative edge in their dependency cycle, then $P = Q = \omega$.

A more formal definition will come in this manual pretty soon.

### 4.4.2 Informal Domains

`Lparse` divides the predicate symbols of a logic program into two classes, domain and non-domain predicates. The domain predicates are used to find all possible variable bindings of rules during the grounding.

For example, in the program

**Program 4.7**

```
d(a,b).  d(b,c).  d(c,a).
foo(X,Y,Z) :- d(X,Y), d(Y,Z), not bar(X,Y,Z).
bar(X,Y,Z) :- d(X,Y), d(Y,Z), not foo(X,Y,Z).
```

the predicate $d/2$ is the only domain predicate. When grounding the rules for *foo* and *bar*, `lparse` gets the possible variable bindings from literals $d(X, Y)$ and $d(Y, Z)$.

In this program there are three possible variable bindings for the two non-ground rules:

1. $\{X/a, Y/b, Z/c\}$, from $d(a,b)$ and $d(b,c)$.
2. $\{X/b, Y/c, Z/a\}$, from $d(b,c)$ and $d(c,a)$.
3. $\{X/c, Y/a, Z/b\}$, from $d(c,a)$ and $d(a,b)$.

Thus, the resulting ground program will be:

**Program 4.8**

```
d(a,b).  d(b,c).  d(c,a).
foo(a,b,c) :- d(a,b), d(b,c), not bar(a,b,c).
foo(b,c,a) :- d(b,c), d(c,a), not bar(b,c,a).
foo(b,c,a) :- d(b,c), d(c,a), not bar(c,a,b).
bar(a,b,c) :- d(a,b), d(b,c), not foo(a,b,c).
bar(b,c,a) :- d(b,c), d(c,a), not foo(b,c,a).
bar(b,c,a) :- d(b,c), d(c,a), not foo(c,a,b).
```

In practice, the non-domain predicates are the "interesting" predicates, and the domain predicates are just programmer's way to tell precisely what instances of the "interesting" predicates we want to consider.

After the grounding is done it is often possible to drop all domain literals from the program. After all, we already know that they will be true in every stable model of the program. `Lparse` command-line argument `-d none` (see Section 3.1) drops all domain literals from the program.

We can use as a domain predicate (nearly) any predicate that has a fixed extension in all stable models of the program. That is, if we know all true ground instances of the predicate even before we ask smodels to compute them, the predicate is a domain predicate.

For example, in the program:

**Program 4.9**

```
p(X,Y) :- d1(X,Y), not q(X,Y).
q(X,Y) :- d1(X,Y), not p(X,Y).
d1(X,Y) :- d2(X), d3(Y), not d4(X).
d2(a).  d2(b).  d3(c).  d4(a).
```

predicates $d_1$, $d_2$, $d_3$, and $d_4$ are domain predicates. All rules with $d_2$, $d_3$, or $d_4$ as heads are facts so the extensions of those predicates are given directly in the program ($\{d_2(a), d_2(b)\}$ for $d_2$ and $\{d_3(c)\}$ and $\{d_4(a)\}$ for $d_3$ and $d_4$).

As the only rule for $d_1$ has only domain literals in its body, we also can compute its extension with relational algebra by taking a natural join over the extensions of $d_2$, $d_3$, and $d_4$. The resulting extension is simply $\{d_1(b,c)\}$ ($d_1(a,c)$ is not in the extension because $d_4(a)$ is always true and thus the variable binding doesn't satisfy the literal not $d_4(X)$.

The extensions of predicates $p$ and $q$ can't be computed beforehand, since they depend on each other recursively. The program has one choice point where either $p(b, c)$ is added to the model or $q(b, c)$ is added to model, but not both. Thus the extension of $p$ will be either $\{p(b, c)\}$ or it will be empty, depending on which atom got included in the model. Similarily for $q$.

*Here should be an example that details the transitive closure in domain predicates.*

There is one more complication about domain predicates: If a predicate occurs as a head in an extended rule (see Section 5.4) it may not be a domain predicate. The reason why choice rules are excluded is clear as they implicitly add a choice point to the program. The reason why other rules are also excluded is implementation specific, as extended rules are processed after the domain generation. This will probably change in a future release, but for the time being you cannot use a predicate as a domain predicate if it occurs as a head in an extended rule.

### 4.4.3 Constructing Domains

All basic relational algebra operations can be done on the extensions of domain predicates and the result will still be a domain predicate. The operations are:

1. Union $P \cup R$ of the extensions of $P$ and $R$:

```
U(X) :- P(X).
U(X) :- R(X).
```

2. Intersection $P \cap R$ of the extensions:

```
I(X) :- P(X), R(X).
```

3. Set difference $P \setminus R$ of the extensions:

```
D(X) :- P(X), not R(X).
```

4. Cartesian product $P \times R$:

```
C(X,Y) :- P(X), R(Y).
```

5. Natural join $P \bowtie R$:

```
J(X,Y,Z) :- P(X,Y), R(Y,Z).
```

6. Symmetric difference $P \triangle R$:

```
S(X) :- P(X), not R(X).
S(X) :- R(X), not P(X).
```

### 4.4.4 Formal Domains

The *dependency graph* $G_P = (V_P, E_P)$ of a logic program $P$ is constructed as follows:

1. $V_P = \{p \mid p \text{ is a predicate symbol in } P\}$.

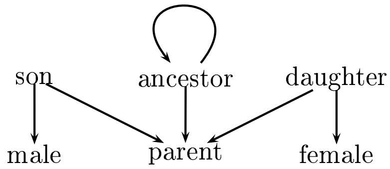



Figure 4.3: The dependency graph of family relationships

2. $(a, b) \in E_P$ if and only if there exists a rule in $P$ where $a$ is the predicate symbol in the head and $b$ is a predicate symbol in the rule body.

A predicate $a$ *depends* on a predicate $b$ if and only if there exists a path from $a$ to $b$ in the dependency graph. A predicate $p$ of a logic program $P$ is a *domain predicate* if and only if it holds that every path starting from node $p$ in the dependency graph is cycle free.

**Example 4.4**

Consider the following program:

**Program 4.10**

```
ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
ancestor(X,Y) :- parent(X,Y).
son(X,Y) :- parent(Y,X), male(X).
daughter(X,Y) :- parent(Y,X), female(X).
```

The dependency graph of this program is shown in Figure 4.3. We can see from the graph that *ancestor* depends on itself and on *parent*. Likewise, *daughter* depends on *parent* and *female*. Here all predicates but *ancestor* are domain predicates. ◊

## 4.5 Domain-Restricted Programs

A rule is *domain-restricted* if it holds that if a variable appears in the rule it also appears in a positive domain literal in the body.

A logic program is *domain-restricted program* if and only if every rule in it is domain-restricted.

**Example 4.5**

Suppose that $d$ is a domain predicate and there are two rules:

```
p(X,Y) :- d(X,Y), not q(X,Y).
q(X,Y) :- not d(X,Y), not q(X,Y).
```

Now the first rule is domain-restricted since both $X$ and $Y$ occur in the domain literal $d(X, Y)$. The second rule is not domain-restricted, since the domain literal is negative, and not positive. ◊

The reason for having domain restriction is that during grounding we have to know exactly what ground instances of rules are needed.

For example, if we tried to use rules of the following form:

```
a(X) :- not b(X,Y).
```

we would have to generate a ground instance for each imaginable binding of $Y$. In effect, we would have to use every single constant that appears somewhere in the program as a possible binding for $Y$ and there may be thousands of them. So the domain predicates are used to "restrict the domain" of variables.

**Example 4.6**

The Program 4.10 is otherwise domain-restricted but the variable $X$ in the first rule doesn't occur in a domain predicate. We can fix the situation by defining a new predicate, *person*, and rules

```
ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y), person(X).
person(X) :- female(X).
person(X) :- male(X).
```

$\diamondsuit$

The earlier `smodels` front-end, *parse*, allowed rules of the form:

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

where $q$ was any predicate symbol. *Parse* computed the needed ground instances by dropping all negative literals and then computing a deductive closure of the rules. This method had the weakness that the whole program had to be kept in memory during grounding, which severely affected the performance of the system.

## 4.6 Weight Constraints

The SMODELS versions 2.0 and later have three extended rule types in addition to basic rules that were defined in Section 4.2: choice, constraint, and weight rules. The formal semantics of all three can be defined through use of *weight constraints* and *weight constraint rules*. In `lparse` the weight constraints are implemented as special literal types.

Basically, a weight constraint is something of the form

$$L \leq \{a_1 = w_1, \ldots, a_n = w_n, \text{not } b_1 = w_{n+1}, \ldots, \text{not } b_m = w_{m+n}\} \leq U \quad (4.4)$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms, $L$ and $U$ are the integral *lower* and *upper* bounds, and $w_1, \ldots, w_{m+n}$ are *weights* of the literals. Later on, we denote the weight of a literal $l$ with $w(l)$. Here we consider only positive weights as the negative weights can be removed from a program as by negating the weight and inverting the literal.

The intuitional semantics of a weight constraint is that it is satisfied exactly when the sum of weights of satisfied literals $l_1, \ldots, l_n$ is between $L$ and $U$, inclusive. The stable model semantics for weight constraint rules was first defined in [12].

**Example 4.7**

Let $M = \{a, b\}$ and

$$\begin{array}{l} C_1 = 2 \leq \{a = 1, \text{not } b = 1, \text{not } c = 1\} \leq 3 \\ C_2 = 1 \leq \{a = 1, b = 1, c = 1\} \leq 1 \end{array}$$

Now $M$ satisfies $C_1$ because literals $a$ and not $c$ are satisfied and their total weight is greater than the lower bound and lower than the upper bound. However, $C_2$ is not satisfied as the sum of the weights of $a$ and $b$ is greater than the upper bound. $\diamondsuit$

A *weight constraint rule* is of the form

$$C_0 \leftarrow C_1, \ldots C_n \tag{4.5}$$

where $C_0$, ..., $C_n$ are weight constraints. We also have to restrict $C_0$ so that may not have any negative literals in it.

A weight constraint rule is satisfied if $C_0$ is satisfied whenever $C_1$, ..., $C_n$ are satisfied.. Analogously to the definition of normal logic programs, a *weight constraint program* is a set of weight rules.

A *reduct* $C^M$ of a weight constraint $C$ with respect to a set $M$ of atoms is obtained by removing the upper bound and all negative literals from it, and subtracting the weights of satisfied not-atoms from the lower bound:

$$C^M = L' \leq \{a_i = w_i \mid a_i \in M\} \tag{4.6}$$

where

$$L' = L - \sum_{b_i \notin M} w(b_i) \tag{4.7}$$

The reduct of a weight rule $R^M$ of a weight rule $R$ with respect to a set $M$ of atoms is the set of rules

$$R^M = \begin{cases} \emptyset & \text{, if } \exists C_{i \geq 1} : M \not\models C_i \\ \{h \leftarrow C_1^M, \ldots C_n^M \mid \\ \quad h \in M \text{ and } h \text{ is an atom in } C_0\} & \text{, otherwise} \end{cases} \tag{4.8}$$

where $M \not\models C$ denotes that $M$ doesn't satisfy the weight constraint $C$. The above definition looks quite ugly but basically it just says that a rule with an unsatisfied body is dropped out of the reduct and the negative literals are dropped from the remaining rules.

I guess that most of the readers can guess by now how the reduct is defined for weight constraint programs. It is formed by taking union over reducts of individual rules.

$$P^M = \{R^M \mid R \in P\} \tag{4.9}$$

The rules in the reduct have all same form: they have a single atom as head and there are no negative literals in the weight constraints of the rule body. We call these rules *Horn weight rules* analogously to the definition for basic rules.

We can define the Knaster-Tarski operator $T_P$ for Horn weight rules in a similar way that it was defined for basic Horn rules.

$$
\begin{aligned}
T_P(M) = \{h \mid h \leftarrow C_1, \ldots C_n &\text{ is a rule in the program} \\
\text{and } C_1, \ldots, C_n &\text{ are all satisfied by } M\}
\end{aligned}
\tag{4.10}
$$

**Example 4.8**

Consider the following program[3] $P$:

**Program 4.11**

```
a :- 1 [ a = 1 ].
b :- 0 [ b = 100 ].
c :- 6 [ b = 5, d = 1], 2 [ b = 2, a = 2].
d :- 1 [ a = 1, b = 1, c = 1 ].
```

We start iterating the $T_P$ from the empty set.

1. $T_P(\emptyset) = \{b\}$, because the empty set satisfies the body of the second rule.
2. $T_P(\{b\}) = \{b, d\}$, because $b = 1$ is enough to satisfy the constraint of the last rule.
3. $T_P(\{b, d\}) = \{b, c, d\}$, as $w(\{b, d\}) = 6 \geq 6$ in the first constraint of the rule for $c$ and $w(\{b\}) = 2 \geq 2$ in the second constraint.
4. $T_P(\{b, c, d\}) = \{b, c, d\}$ and the fixed point is found.

$\Diamond$

Unfortunately we can't define the stable models of a weight rule program using only reducts. The problem is that we threw out all upper bounds of weight constraints while computing the reduct and we have to ensure that the model satisfies also the upper bounds.

A set $M$ of atoms is a *stable model* of a weight constraint program $P$ if and only if the following two conditions are met:

1. $M$ satisfies all rules in $P$; and
2. $M$ is the least fixpoint of $T_{P^M}(\emptyset)$.

**Example 4.9**

Let a program $P$ be simply:

[3]Here we use the `lparse` syntax. Namely, the $\leq$ symbol is left out and the square brackets denote weight constraints.

**Program 4.12**

```
1 [ a = 1, b = 1 ] 1.
```

Now $M_1 = \{a\}$ is a stable model of $P$ as $M_1$ satisfies the only rule and the reduct $P^M$ of the program is

```
a.
```

Similarily for $M_2 = \{b\}$. On the other hand, $\{a, b\}$ is not a stable model even though the reduct of the program is

```
a.
b.
```

since the combined weights of $a$ and $b$ are more than the upper bound of the rule. $\Diamond$

**Example 4.10**

Let's now look at a slightly more complex example program $P$:

**Program 4.13**

```
1 [ a=1, b=1, c=1 ] 2 :- 2 [ d=1, not b=1, not e=3 ] 4.
1 [ d=3, e=2 ] 5.
```

Now $M_1 = \{a, d, e\}$ is one stable model of the program. The reduct $P^{M_1}$ is now

```
a :- 1 [ d =1 ].
d.
e.
```

and the least fixpoint of $T_{P^{M_1}} = M_1$ and the model satisfies all rules. However, $M_2 = \{d, a\}$ is not a stable model because the constraint in the first rule body is not satisfied ($w(M) = 5 \geq 4$) so it is dropped out from the reduct and there is no longer any way to deduce $a$. $\Diamond$

In the beginning of this section I claimed that negative weights can be removed. This is proven in [12] and I only present here the translation. A weight constraint

$$L \leq \{a = -w_a, \text{not } b = -w_b\} \leq U \tag{4.11}$$

can be transformed to an equal form

$$L + w_a + w_b \leq \{\text{not } a = w_a, b = w_b\} \leq U + w_a + w_b \tag{4.12}$$

The idea here is that instead of subtracting $w_a$ from the total weight when $a$ is `true`, we add $w_a$ if $a$ is not true and we raise the bounds with the same amount, the net result being the same.

## 4.7 Classical Negation

The basic version of the stable model semantics has only *negation as failure*. That is, we conclude not $a$ if we can't prove that $a$ is true. Sometimes this is not desirable. For example, suppose that we want to check whether it is safe to cross railroad tracks. This could be expressed with a rule:

$$safe \leftarrow \text{not } train \ .$$

The problem here is that we consider the crossing to be safe if we can't prove that a train is coming. A more safe approach would be to declare the crossing safe only if we can prove that train is, indeed, not coming:

$$safe \leftarrow \neg train \ .$$

This stronger negation is called the *classical negation* and programs utilizing it are usually called *extended logic programs*. The extended logic programs were proposed in [3].

The semantics of the extended logic programs differs a bit from the stable model semantics of oridinary programs. The *answer set* $S$ of an extended program $P$ is a minimal subset of the Herbrand base $\mathbf{HB}(P)$ of the program such that:

1. for any rule $l_o \leftarrow l_1, \ldots, l_n$ in $P$, if $l_1, \ldots, l_n \in S$, then $l_o \in S$; and
2. if $S$ contains a pair of complementary literals, then $S = \mathbf{HB}(P)$.

There are two different ways how an extended program can fail to have an useful answer set. Either there is no answer at all or the only answer is inconsistent.

**Example 4.11**

Consider the following program $P$:

$$\begin{array}{rl} a \leftarrow \text{not } b & b \leftarrow \text{not } a \\ d \leftarrow \text{not } c & c \leftarrow \text{not } d \\ \neg e \leftarrow \text{not } a & e \leftarrow \text{not } c \\ f \leftarrow \text{not } f, b, d & \end{array}$$

This program has three answer sets:

$$\begin{aligned} S_1 &= \{a, d, \neg e\} \\ S_2 &= \{b, c, e\} \\ S_3 &= \{a, b, c, d, e, \neg e, f\} \end{aligned}$$

The first two answer sets, $S_1$ and $S_2$ are consistent and $S_3$ is inconsistent. It is not possible to have an answer set where both $b$ and $d$ are true because it will cause the last rule to fire causing a contradiction. ◊

`Lparse` handles extended programs by transforming the programs into normal logic programs. The rules are otherwise unaffected but for each negative atom $\neg a$ that occurs in the program, the rule:

$$\leftarrow a, \neg a \tag{4.13}$$

is added. Note that this approach differs from the semantics described above in that the inconsistent answer sets are automatically rejected. As this information is sometimes useful, `lparse` has a variant behavior that can be initialized with the command line option `--allow-inconsistent-answers`. With it, a special atom `INCONSISTENT` is added to the program that is true whenever the stable model contains complementary literals:

$$\text{INCONSISTENT} \leftarrow a, \neg a \ . \tag{4.14}$$

## 4.8 Partial Models and Disjunctive Programs

SMODELS and `lparse` now include facilities to compute stable and partial models for disjunctive logic programs. However, this functionality is still quite primitive. Disjunctive logic programs are discussed in detail in [1] and the theoretical basis for the partial model expansion is introduced in [5].

The disjunctive model semantics is enabled by selecting the command line option `--dlp` and the partial model expansion by the option `--partial`. The exact output of the partial model expansion can be altered with the `-r` command line option.

### 4.8.1 Disjunctive Programs

A *disjuntive rule* is of the form

$$a_1 \mid a_2 \mid \cdots \mid a_n \leftarrow \mathit{body} \tag{4.15}$$

If the body of a disjunctive rule is true, at least one of the head atoms $a_1$, ..., $a_n$ has to be true. A *disjunctive logic program* is a set of disjunctive rules.

The *reduct* $P_D^M$ of a disjunctive program $P_D$ is obtained using the procedure that was used with normal logic programs. That is, all rules with unsatisfiable negative literals in the body are dropped as well as all remaining negative literals. A set $M$ of atoms is a *disjunctive stable model* of $P_D$ iff $M$ is a minimal model of $P_D^M$. Note that $P_D^M$ may have more than one minimal model.

It was easy to find the minimal model of the reduct of a normal logic program. However, this is not the case with disjunctive programs. In fact, after we have found a model for the reduct it is still a NP-complete problem to find out whether it is minimal or not.

Because of this complexity, pure `smodels` doesn't handle disjunctive programs correctly. However, you can still solve disjunctive queries by using two interleaved `smodels` processes. The file `example4.cc` in the `smodels/examples` directory shows how it is done.

**Example 4.12**

Consider a disjunctive program $P_D$:

**Program 4.14**

```
a | b :- c.
c :- not d.
d :- not c.
```

This program has three disjunctive stable models. Either $c$ or $d$ has to be in a model and if $c$ is chosen, we have to add $a$ or $b$ in the model. Thus, the models are: $M_1 = \{d\}$, $M_2 = \{c, a\}$, and $M_3 = \{b, c\}$. The fourth possibility, $M' = \{c, a, b\}$ is not a disjunctive model since $M_2$ is a model and $M_2 \subset M'$ so that $M'$ is not minimal. $\Diamond$

A disjunctive rule of the form (4.15) is very similar to a weight constraint rule of the form

$$1 \leq \{a_1, a_2, \ldots, a_n\} \leftarrow \mathit{body} \tag{4.16}$$

but there is one subtle difference: disjunctive stable models are always minimal while there may be non-minimal stable models of weight constraint programs. For example, if we replace the first rule of Program 4.14 with

```
1 { a, b } :- c.
```

then $\{a, b, c\}$ is a stable model of the program.

### 4.8.2 Partial Models

As we saw in Section 4.1, an interpretation of a logic program assigns a truth value to atoms in the Herbrand base of the program. Each atom is either true or false in the interpretation. A *partial interpretation* is an interpretation that assigns a definite truth value only to some atoms of the base and the rest of the atoms are said to have unknown truth value. Formally, a partial interpretation $I$ of a program $P$ is a pair $\langle T, F \rangle$ of subsets of the Herbrand base $\mathbf{HB}(P)$ of the program such that $T \cap F = \emptyset$. The atoms in sets $I^t = T$, $I^f = F$, and $I^u = \mathbf{HB}(P) - (T \cup F)$ are considered to be true, false, and unknown in the interpretation.

A *partial model* is a partial interpretation that satisfies all rules of the program. However, the concept of satisfaction has to be refined to take into accord the fact that there are now three different truth values. We impose an order $<$ on truth values so that

$$\mathsf{false} < \mathsf{unknown} < \mathsf{true}$$

The truth value $I(B)$ of the body of a rule is obtained by taking minimum of the truth values of the literals in the body and the truth value $I(H)$ of the head is the maximum of the truth values of the head atoms. A rule is satisfied when $I(H) \geq I(B)$.

**Example 4.13**

Suppose that we have a partial interpretation

$$I(H) = \langle \{b\}, \{d\} \rangle$$

Suppose further that we have two rules:

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

Now the first rule is satisfied since $I(a) = \mathsf{unknown} \geq I(c) = \mathsf{unknown}$. The second rule is not satisfied as $I(d) = \mathsf{false} < I(a) = \mathsf{unknown}$. $\Diamond$

A partial model $I$ that assigns a definite truth value to all atoms ($I^u = \emptyset$) is called a *total model* or alternatively a *regular model*. A *total stable model* $M$ of $P$ is a total model that is additionally a minimal model of the reduct $P^M$. A *partial stable model* $M$ of $P$ is a minimal partial model of $P^M$.

We can compute the partial stable models of a disjunctive or normal logic program $P$ by using a translation that maps it into another program $\mathrm{Tr}(P)$ that has the property that $M$ is a stable model model of $\mathrm{Tr}(P)$ if and only if $M$ is a partial stable model of $P$.

We start the translation by adding a new atom $a'$ for each atom $a \in \mathbf{HB}(P)$. The intuitive meaning of $a'$ is that it is *potentially true*. A rule of the form:

$$a_1 \mid \cdots \mid a_n \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_l \tag{4.17}$$

is replaced with the rules:

$$\begin{aligned} a_1 \mid \cdots \mid a_n &\leftarrow b_1, \ldots, b_m, \text{not } c'_1, \ldots, \text{not } c'_l \\ a'_1 \mid \cdots \mid a'_n &\leftarrow b'_1, \ldots, b'_m, \text{not } c_1, \ldots, \text{not } c_l \ . \end{aligned} \tag{4.18}$$

Additionally, $\mathrm{Tr}(P)$ includes the rule:

$$a' \leftarrow a \tag{4.19}$$

for all $a \in \mathbf{HB}(P)$ so an atom is always potentially true if we know that it is true.

Let $M$ be a stable model of $\mathrm{Tr}(P)$. Then, there exists a partial stable interpretation $N$ of $P$ that corresponds to $M$. The truth value of an atom $a$ in $N$ is obtained by the following three rules:

1. If both $a, a' \in M$, then $I(a) = \mathsf{true}$;
2. If $a' \in M$ and $a \notin M$, then $I(a) = \mathsf{unknown}$; and
3. Otherwise, $I(a) = \mathsf{false}$.

The formal proof that this translation works is presented in [5].

**Example 4.14**

Consider a disjunctive program $P$:

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

The translated program $\mathrm{Tr}(P)$ is:

```
a | b :- not c'.
a' | b' :- not c.
b :- not b'.
b' :- not b.
c :- not c'.
c' :- not c.
a' :- a.
b' :- b.
c' :- c.
```

The only stable model of $\mathrm{Tr}(P)$ is $\{b', c'\}$ and it correspond to the partial interpretation of $P$ where $b$ and $c$ are both unknown and $a$ is false. $\Diamond$

Because the partial model translation introduces a dependency loop for each predicate, it is not done for domain predicates. Instead, the rule

```
d'(X) :- d(X).
```

is added for each domain predicate $d$.

The behavior of the partial model translation can be altered with the `-r` command line switch. In the first alternative (`-r 1`) the rules of the form (4.19) are left out. This option is useful when one wants to find all possible fixpoints of the program.

The second alternative (`-r 3`) adds constraints of the form

$$a \leftarrow a' \tag{4.20}$$

for all $a \in \mathbf{HB}(P)$ to the program. These ensure that an atom is true always when it is possibly true.

## 4.9 Computational Complexity

*This section will appear here when I have time to write it.*

# Chapter 5

# Language

This chapter describes the SMODELS language. Each different language feature has its own section here.

## 5.1 Comments

You can have comments in SMODELS programs. The comment character is '%'. A comment then lasts util the end of the row.

## 5.2 Terms

There are four different types of terms: constants, variables, functions, and ranges.

**Constant**

A constant is either a symbolic constant or a numeric constant. A symbolic constant is a string of letters and numbers which may also contain underscores (_) starting with a lower case letter, or a sequence of characters that is enclosed within double quotes (").

The quote characters are retained in the quoted strings by default, even if unnecessary, and `a` is different from `"a"`. This behavior can be altered with the `--drop-quotes` command line argument.

A numeric constant is an integer. Currently the allowed range for numbers goes only from $-2^{30}$ to $2^{30}$. This is due to seriously limited implementation of constants and will be removed in some later release (see Section 8).

Sample constants: `0, 1020, cons_tant, "quoted constant"`.

It is also possible to define a symbolic constant to act as an numeric constant by adding a constant declaration (see Section 5.5). If you add the line `const` *foo* `=` *number* to your program, from that point on every occurrence of the constant *foo* will be substituted by *number*. Alternatively, you can use the `-c` command line option.

In general, you may use an expression that evaluates to a constant value everywhere where you can use constants, that is, it is legal to use constructs like:

```
const double = 2 * foo.
```

**Variable**

A variable is a string of letters and numbers that may also contain underscores _ starting with an upper case letter

Sample variables: `X, Time_1`.

**Function**

A function is either a function symbol followed by a parenthesised argument list or an builtin arithmetical expression. A function may be either a numerical function that is actually used to compute something or it is a symbolic function of the form `foo(a)`, which basically just defines a new constant that gets the name `foo(a)`.

A numerical function may be either built-in internal function or a user specified C or C++ function that is linked to lparse dynamically (see Section 5.7.3).

The internal comparison functions (`eq`, `neq`, `lt`, `le`, `gt`, and `ge`) can be used with both numeric and symbolic constants but the rest of internal functions allow only numeric arguments.

The use of functions is explained more fully in Section 5.7.

Sample functions: `X+1, times(X, 5, plus(Y, 1))`.

**Range**

A range is of the form:

*start* `..` *end*

where *start* and *end* are constant valued arithmetic expressions. A range is a notational shortcut that is mainly used to define numerical domains in a compact way. A range is expanded by defining a new domain predicate and adding all elements of the range to its extension. The range is then replaced by a variable that gets its domain from the new domain predicate.

For example, a fact `a(1..3).` is a shortcut for

```
a(X) :- int1(X).
int1(1).
int1(2).
int1(3).
```

Ranges can also be used in rule bodies with the same semantics.

```
b :- a(1..3).
```

expands to

```
b :- a(X), int1(X).
int1(1).
int1(2).
int1(3).
```

Now `b` is true if any of `a(1)`, `a(2)`, or `a(3)` is true.

## 5.3 Atoms and Literals

Starting from version 2.0 `smodels` has offered support for three extended rule types: choice, constraint, and weight. `Lparse` further enhances this by allowing a rule have an arbitary number of constraint and weight literals in a rule body. Constraint and weight literals are both called extended literals. In addition to basic and extended literals, `lparse` has one further literal type, namely, conditional literal.

**Atom**

An atom is a predicate symbol that is optionally followed by a parenthesized list of terms.

Sample atoms: `foo(X), a, foo`.

It is possible to give multiple argument lists to an atom. These constructs are of the form:

a(*arguments*$_1$; *arguments*$_2$; $\cdots$; *arguments*$_n$)

When multiple argument lists appear in a rule body or in a choice rule head (see Section 5.4 on page 43), a new literal is constructed for each list. That is, a rule

```
{ a(X;Y+1) } :- d(X;Y).
```

is expanded to

```
{ a(X), a(Y+1) } :- d(X), d(Y).
```

In a basic rule head a new rule is constructed for each argument list so that

```
foo(a ; b; c).
```

becomes

```
foo(a).
foo(b).
foo(c).
```

**Basic literal**

A basic literal is either an atom $a$ or its negation not $a$.

Beginning with lparse-0.99.57 you have been able to use classical negations in your programs. The classical negation $\neg a$ of an atom $a$ is denoted by a minus sign that is immedietely before the atom name: `-a`. The classical negation extension is enabled by the command line option `--true-negation`.

Sample literals: `a(X), not b(X)`.

**Constraint literal**

A constraint literal is of the form

```
lower { l_1, l_2, ... , l_n } upper
```

where *lower* and *upper* are arithmetic expressions and $l_1, \ldots, l_n$ are basic or conditional literals.

A constraint literal is satisfied if the number of satisfied literals in the body of the constraint is between *lower* and *upper* (inclusive). If the lower bound is missing, zero is substituted in its place and if the upper bound is missing, any number of literals may be true.

It is possible to have variables in the rule bounds. For example, the following rule could be used to count the length of a path in a graph:

```
length(N) :- N { in_path(X,Y) : edge(X,Y) } N,
             possible_length(I).
```

During grounding a constraint literal is replaced by two basic literals and two new rules are added to the program. For example, a rule:

```
h(a) :- 2 { d_1(a), d_2(a), d_3(a), d_4(a) } 3
```

is transformed to

```
h(a) :- int1(a), not int2(a).
int1(a) :- 2 { d1(a), d2(a), d3(a), d4(a) }.
int2(a) :- 4 { d1(a), d2(a), d3(a), d4(a) }.
```

Here $int_1$ and $int_2$ are new internal predicates. The transformation is done because smodels allows only one constraint per rule and only lower bounds are examined.

**Weight literal**

A weight literal is of the form

```
lower [ l_1=w_1, ..., l_n=w_n ] upper
```

A weight literal behaves otherwise just like a constraint literal but each literal may be given a different integral weight.

The weight literal is satisfied if the sum of the weights of satisfied literals in the rule body is between *lower* and *upper* (inclusive). If the lower bound is missing, $-\infty$ is substituted.

The weights $w_1, \ldots, w_n$ may be any expressions. If there are variables in weight expressions, they must be domain restricted.

The weights may be given in the weight literal body, or they may be defined earlier by global weight declarations. The local values override the global values. If there is neither a global or a local value, the default value is used. The default is normally 1 but you can change it with the command line argument `-w` (see Section 3.1).

Negative weights are handled by inverting the literal ($a$ becomes not $a$ and not $a$ becomes $a$), changing the sign of the weight, and adding the absolute value of the weight to the bounds.

For example,

```
lower [ a_1=-w_1, not a_2=-w_2 ]
```

becomes

```
lower+w_1+w_2 [ not a_1=w_1, a_2=w_2 ].
```

**Conditional literal**

A conditional literal is of the form:

```
p(X) : q(X)
```

where $p(X)$ is any basic literal and $q(X)$ is a domain predicate. If the extension of $q$ is $\{q(a_1), q(a_2), \ldots, q(a_n)\}$, the above condition is semantically equivalent to writing $p(a_1)$, $p(a_2)$, $\ldots$, $p(a_n)$ in the place of condition.

For example,

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

will be grounded to give

```
q(1).  q(2).
a :- 1 { p(1), p(2) }.
```

Semantically the expansion of conditions takes place after the variables that occur also in another part of the rule are instantiated. For example in the program

```
d(1..2).
a(X) :- 1 { p(X,Y) : d(Y) }, d(X).
```

the variable $X$ will be first instantiated to give the program

```
d(1).  d(2).
a(1) :- 1 { p(1, Y) : d(Y) }, d(1).
a(2) :- 1 { p(2, Y) : d(Y) }, d(2).
```

In the next step the conditions are expanded to give

```
d(1).  d(2).
a(1) :- 1 { p(1, 1), p(1, 2) }, d(1).
a(2) :- 1 { p(2, 1), p(2, 2) }, d(2).
```

In practice, the conditions are expanded as early as possible and if none of the variables in a condition occur in other parts of a rule, the condition will be expanded before anything else is done.

It is also possible add many conditions to one literal. For example in the conditional literal:

```
p(X,Y) : q_1(X) : q_2(Y)
```

domain predicate $q_1$ gives values to $X$ and $q_2$ gives values to $Y$.

## 5.4 Rule Types

The version 2.0 of `smodels` added support for three new rule types: choice, constraint, and weight rules. In `lparse` the constraint and weight rules are handled by more general constraint and weight literals that were introduced in the section above.

**Basic rule**

A basic rule is of the form:

```
h(X) :- a_1(X), ..., a_n(X), not b_1(X), ..., not b_m(X).
```

If $a_1(X), \ldots, a_n(X)$ are in a stable model and $b_1(X), \ldots, b_n(X)$ are not, the head atom $h(X)$ is put also in the model.

If the rule has no head, all model candidates that satisfy the rule body are discarded.

**Choice rule**

A choice rule has one of the following two forms:

```
lower { h_1, ..., h_n } upper :- body.
lower [ h_1=w_1, ..., h_n=w_n ] upper :- body.
```

If the *body* of a choice rule is satisfied, the number (or total weight) of $h_1, \ldots, h_n$ that are `true` in the model will be between *lower* and *upper*, inclusive.

If both bounds are missing, any number of head literals may be included in a model. It is then said that the *body* gives the head atoms a reason to be in a model, but it doesn't force them to be in it.

For example, the program

```
1 { a, b } .
```

has three stable models, $M_1 = \{a\}$, $M_2 = \{b\}$, and $M_3 = \{a, b\}$.

It is also possible to use a weight literal as a head. In that case, the total weight of the satisfied literals in it has to be between the bounds.

There exists also a special notation

```
h_1 | ... | h_n :- body.
```

that is a shorthand for

```
1 { h_1, ..., h_n } 1 :- body.
```

That is, using |-notation you ge an exclusive or of the head atoms. Also, the notation is used with disjunctive semantics that are explained in Section 4.8.1.

Internally, a choice rule with non-zero bounds will be translated into three rules:

```
{ h_1, ..., h_n } :- body.
:- upper + 1 { h_1, ..., h_n }, body.
:- n − lower + 1 { not h_1, ..., not h_n }, body.
```

This transformation is done because `smodels` doesn't allow bounds in choice rules.

## 5.5 Declarations

There are eight kinds of declarations in the language: constant, domain, external, hide, function, option, show and weight declarations. The declarations may occur anywhere in the source code. The only restriction is that you have to declare a constant or a function before you use it.

All declarations start with a '#' symbol. The old practice of writing them without it is still supported but it is recommended to exclusively use the new convention since some future version will drop the support.

**Constant declaration**

A constant declaration is of the form

```
#const bar = expr.
```

This declares the identifier *bar* as being a numeric constant with the value *expr* that may be any constant valued expression. It is also possible to define constants from the command line by using the option `-c`. There are more about numeric constants in Section 5.2.

**Domain declaration**

A domain declaration is of the form

```
#domain a(X).
```

The domain declaration above asserts that the variable `X` should always get its domain from the literal `a(X)`. In practice, this is implemented by adding `a(X)` into the tails of all rules where `X` occurs. The literal is added to all such rules, no matter whether `X` would be otherwise restricted or not.

**Example 5.1**

Consider the following program:

```
#domain a(X, Y), b(Z).
foo(X, Y) :- not bar(X, Y).
bar(X, Z) :- not foo(X, Z).
```

Before the grounding actually takes place, this program is transformed in the form:

```
foo(X, Y) :- a(X,Y), not bar(X, Y).
bar(X, Z) :- a(X,Y), b(Z), not foo(X, Z).
```

Note that both `a(X,Y)` and `b(Z)` are added to the body of the second rule. ◊

**External declaration**

An external declaration has the form

```
#external p(X).
```

where `p` is a domain predicate. External predicates are used during grounding to find out possible instantiations of rules depending on them but they are left out of the actual output.

**Example 5.2**

The program $P$:

```
#external a(X).
a(1..2).
b(X) :- a(X).
```

is grounded to the following program:

```
b(1) :- a(1).
b(2) :- a(2).
```

◊

External predicates are useful when the program is large and it has to be grounded many times with slightly different extensions for domain predicates. The solution is to ground the program once with all possible values for the domain predicates using external declarations and reading the grounded program in with `-g` command line argument when the correct extension is known.

**Example 5.3**

Suppose that the program in Example 5.2 is stored in `foo.lp` and `bar.lp` contains only the fact:

```
a(1).
```

Then, you could compute the stable models of `foo.lp` using `bar.lp` as the source for extension of `a` with the following command lines:

```
% lparse foo.lp > full_grounding
% lparse -g full_grounding bar.lp | smodels
Answer:  1
Stable Model:  b(1) a(1)
True
```

◊

For a more thorough explanation on `external` and its possible uses, see Section 7.4 on page 68.

**Function declaration**

A function declaration is of the form:

```
#function foo .
```

This declares the identifier *foo* to be used as a numeric function throughout the program. The user defined functions are described in more detail in Section 5.7.3.

**Hide declaration**

A hide declaration has two possible forms:

```
#hide.
#hide p(X,Y).
```

The `smodels` 2.x offers a feature where some atoms may be hidden from the output. The atoms still affect the computation in the usual way but they are not printed. The internal predicates generated by `lparse` are automatically hidden. A hide declaration without arguments marks all predicates as hidden by default and the `show` declaration can then be used to tell what atoms are included in the output.

The second hide declaration above marks all ground instances of the 2-ary predicate $p$ to be hidden. The arguments of $p$ are used only to distinct predicates with different arities from each other.

For example, if the program

```
#hide p(X).
p(1..2).
p(X,Y) :- p(X), p(Y).
```

is given to `smodels` as input, the output will be:

```
smodels version 2.16.  Reading...done
Answer:  1
Stable Model:  p(1,1) p(2,1) p(1,2) p(2,2)
True
```

**Option declaration**

An option declaration is of the form

```
#option command-line-option.
```

This declaration can be used to set `lparse` command line arguments in the source code. This is mainly useful if your program uses some of the supported translations, for example `--true-negation` or `--partial`.

**Show declaration**

A show declaration has the form

```
#show p(X,Y).
```

This declaration is the opposite of a hide declaration. It tells `lparse` that the 2-ary predicate $p$ should be shown in the model. This is only useful when all predicates are hidden using an empty `hide` declaration.

**Weight declaration**

A weight declaration has two possible forms:

```
#weight literal  = expr.
#weight literal1 = literal2.
```

The first one declares the default weight of a literal *literal* to be *expr*. Any variables occuring in *expr* must also occur in *literal*. The weights will be instantiated during grounding. The second one declares the weight of $literal_1$ to be the same as the weight of $literal_2$ that may be defined in other part of the program.

When a weight is needed for a literal $a$ that doesn't have an explicit weight assigned to it, `lparse` starts looking for weight definitions that it has seen before the line in question. `Lparse` tries to unify $a$ with each definition and when a match is found, it uses that weight. If no weight definition matches $a$, the global default weight (1, if not set with the `-w` option) is used.

For example, in a program

```
#weight p(X,Y) = 1.
#weight p(a,Y) = 2.
#weight p(a,b) = 3.
a :- 2 [ p(a,b)=1, p(a,a), p(b,b) ].
```

the weight of $p(a, b)$ is 1 because the explicit definition overrides all global definitions. `Lparse` then starts looking for weight of $p(a, a)$. It first tries to unify $p(a, a)$ and $p(a, b)$, but fails because $a \neq b$. Unifying with $p(a, Y)$ succeeds when the variable $Y$ is binded to the value $a$. Thus, the weight of $p(a, a)$ is 2. Using a similar process, `lparse` determines that the weight of $p(b, b)$ is 1.

If the weight of a literal is defined more than once, only the latest definition is used. For example:

```
#weight p(X) = X.
#weight p(2) = 10.
a :- 2 [ p(1), p(2) ].  % weights:  p(1) = 1, p(2) = 10
#weight p(Y) = Y+5.
b :- 2 [ p(1), p(2) ].  % weights:  p(1) = 6, p(2) = 7
```

The weights may be defined separately for positive and negative literals. If a negative literal doesn't have a matching negative weight declaration, it uses a corresponding positive one by default. This behavior can be changed with the option '`--separate-weight-definitions`'. For example, after:

```
#weight a = 2.
#weight not a = 3.
#weight b = 5
```

the weight of $a$ is 2, of not $a$ is 3, and both $b$ and not $b$ have weights 5. However, with the above command line argument the weight of not $b$ is the default 1.

## 5.6 Statements

The statements are used to specify desired properties of the models. There are two kinds of statements, compute and optimize statements.

**Compute statement**

A compute statement is of the form:

```
compute number { a_1, ..., a_n, not b_1, ..., not b_m }.
```

Only stable models containing $a_1, \ldots, a_n$ and not containing $b_1, \ldots, 1b_m$ are computed. The number of generated models is controlled by *number*. If *number* is 0 or the identifier *all*, all models are computed. The default number of models is 1.

**Optimize statement**

An optimize statement has four possible forms:

```
maximize { a_1, ..., a_n, not b_1, ..., not b_m }.
maximize [ l_1 = w_1, ..., l_n =w_n ].
minimize { a_1, ..., a_n, not b_1, ..., not b_m }.
minimize [ l_1 = w_1, ..., l_n = w_n ].
```

When an optimize statement is given, `smodels` tries to find models with as many (or as few) of the given literals as possible. You may also use weights with these literals and then the model with maximal (or minimal) weight is returned. The optimize statements use braces analogously to constraint and weight literals; with curly braces the number of true literals is maximized (or minimized) and with square brackets the weight of true literals is maximized.

However, the behavior of `smodels` is not the one that would come to mind first. Namely, `smodels` first searches a single model and prints it. After that, `smodels` prints only "better" models. For example, if in the first model includes three optimized atoms, only those with four or more are returned afterwards.

If there are many optimize statements, they are considered in fixed order, the last one being the strongest. When comparing two models $M_1$ and $M_2$ the last optimize statement is considered first. If it gives different values for both models, the rest of the statements are not evaluated at all. Only if $M_1$ and $M_2$ tie with respect to the last optimize statement the next one before it is used, and so on.

## 5.7 Functions

There are two kinds of functions in `lparse`, numerical functions and symbolic functions. The difference is that a numerical function is used to compute some concrete numeric value but a symbolic function basically just defines a new constant that is the value of the function.

For example, in a program:

```
d(1..2).  e(a ; b).
q(X+1) :- d(X).
p(f(X)) :- e(X).
```

there are two functions, $X + 1$ and $f(X)$. Here $X + 1$ is a numerical function and its value is computed during grounding. On the other hand, $f(X)$ is a symbolic function and only thing that grounding does to it is to instantiate $X$. The grounded program is:

```
d(1).  d(2).  e(a).  e(b).
q(2) :- d(1).
q(3) :- d(2).
p(f(a)) :- e(a).
p(f(b)) :- e(b).
```

The functions $f(a)$ and $f(b)$ are treated just like constants by `smodels`.

A numerical function has to be declared with a function declaration before they can be used.

### 5.7.1 Numerical Functions

Numerical functions can occur in two different roles in rules, either as a term or as a test in a rule body.

As a term, a numerical function gives a value to an argument of a predicate. During grounding the rule the function is called and the return value is used as the argument. Internally the function is replaced by a new variable and an assign function in the rule body.

For example,

Table 5.1: `Lparse` internal functions

<table>
<tr><th>Function</th><th>Operator</th><th>Function</th><th>Operator</th></tr>
<tr><td><code>plus(X,Y)</code></td><td><code>X + Y</code></td><td><code>lt(X,Y)</code></td><td><code>X < Y</code></td></tr>
<tr><td><code>minus(X,Y)</code></td><td><code>X - Y</code></td><td><code>gt(X,Y)</code></td><td><code>X > Y</code></td></tr>
<tr><td><code>times(X,Y)</code></td><td><code>X * Y</code></td><td><code>le(X,Y)</code></td><td><code>X <= Y</code></td></tr>
<tr><td><code>div(X,Y)</code></td><td><code>X / Y</code></td><td><code>ge(X,Y)</code></td><td><code>X >= Y</code></td></tr>
<tr><td><code>mod(X,Y)</code></td><td><code>X mod Y</code></td><td><code>eq(X,Y)</code></td><td><code>X == Y</code></td></tr>
<tr><td><code>assign(X,Y)</code></td><td><code>X = Y</code></td><td><code>neq(X,Y)</code></td><td><code>X != Y</code></td></tr>
<tr><td><code>and(X,Y)</code></td><td><code>X & Y</code></td><td><code>xor(X,Y)</code></td><td><code>X ^ Y</code></td></tr>
<tr><td><code>or(X,Y)</code></td><td><code>X | Y</code></td><td><code>bnot(X)</code></td><td><code>~X</code></td></tr>
<tr><td><code>abs(X)</code></td><td><code>| X |</code></td><td><code>weight(a(X))</code></td><td></td></tr>
</table>

```
p(X, Y, X+Y) :- d(X,Y).
```

becomes

```
p(X, Y, Z) :- d(X,Y), Z = X+Y.
```

If we ground the above rule with a variable binding $\{X/1, Y/2\}$, the ground rule becomes

```
p(1,2,3) :- d(1,2).
```

As a test, a function works as a constraint for possible variable values. If the function returns false, i.e. 0, when called with a given variable binding, the binding is discarded. If it returns true, i.e. any value other than 0, the ground rule corresponding to the binding is printed. After the test has been performed succesfully, the function is removed before printing the ground rule.

For example, the program:

```
d(1). d(2).
q(X,Y) :- d(X), d(Y), X < Y.
```

is grounded to give

```
d(1). d(2).
q(1,2) :- d(1), d(2).
```

A numerical function has to be declared (see Section 5.5) before it can be used.

### 5.7.2 Internal Functions

There are 17 different arithmetical functions built in `lparse`. The internal functions are automatically declared unless the `-i` command line option is given. The `lparse` interal functions are shown in Table 5.1.

There is an infix operator defined for each internal function[1]. The notations $f(X,Y)$ and $X \circ_f Y$ are interchangeable. In fact, the latter is converted internally to the former. All internal functions accept many arguments, so you can call for example $plus(X, Y, Z, 2)$. The operator precedence is shown in Table 5.2.

[1] With the exception of `weight(a(X))`.

Table 5.2: Lparse operator precedence

| |
|---|
| $-$ (unary), ~ |
| $*$, /, mod |
| $+$, $-$ |
| ==, !=, <, >, <=, >= |
| &, \|, ^ |
| = |

Only comparison functions allow symbolic constants as their arguments and the others work only for numbers. The symbolic constants are compared using lexicographic, that is phonebook, ordering.

The functions `and`, `or`, `xor`, and `b_not` implement the bitwise logical operations.

The `assign`-function returns always 1 when the assignment succeeds, so the constructs like `X = Y = 5` don't work like they do in some other languages.

The `weight` function is a special case that takes a basic literal as its argument and returns its weight.

**Example 5.4**

Consider the program:

```
weight a(X) = X.
b(1..4).
c(X) :- b(X), weight(a(X)) <= 2.
d(X,Y) :- b(X), b(Y), weight(a(X)) + weigh(a(Y)) > 4.
```

The extension of $c$ will be $\{c(1), c(2)\}$ and the extension of $d$ will be $\{d(1,4), d(2,3), d(3,2), d(4,1)\}$. ◊

### 5.7.3 User-Defined Functions

It is possible to add a user-defined C or C++ function to `lparse`. In earlier versions of `lparse` you had to use the perl script `register` to link the functions statically to `lparse` binaries. The current method allows dynamic linking of shared library files to `lparse`.

Just about every C/C++-function can be linked with lparse, but those functions that meant to be directly called from logic programs should have the prototype:

```
long foo(int num_args, long *args)
```

That is, `lparse` passes the arguments in an array of long integers with an integer parameter telling the lenght of the array. The symbolic constants are encoded as indices to a symbol table where the actual strings are stored. The numeric constants can be handled using normal functions and operators but the only way to handle symbolic constants is to use the `lparse` API that is introduced in the next section.

`Lparse` uses the normal C convention of treating value 0 as false and every other value as true.

Before you can use your own function, you have to add it to a shared library and tell `lparse` where the library can be found. `Lparse` searches for libraries from the following directories:

1. A path stored in an environmental variable `LPARSE_LIBRARY_PATH`.
2. Directories specified in the file `~/.lparserc`.
3. A path stored in an environmental variable `LD_LIBRARY_PATH`.

`Lparse` library definitions may be placed either in the environmental variable `LPARSE_LIBRARIES` or in the file `.lparserc`.

An example `.lparserc` is:

```
LPARSE_LIBRARY_PATH = /home/tss/lparse-libs
LPARSE_LIBRARIES = libfoobar.so
```

If you have files `foo.c` and `bar.c` that you want to use with lparse, you can create the shared library with following steps. (These steps work with `gcc` on a Linux system; if you use some other system they may or may not work.)

1. Compile the files into object files using the option`-fPIC`:

   ```
   % gcc -fPIC -Wall -c foo.c
   ```

2. Create the shared library. If your functions call some library functions, it will be safest to link these to the library. Here we suppose that `foo.c` uses some functions defined in standard math library:

   ```
   % gcc -shared -Wl,-soname,libfoobar.so -o libfoobar.so
     foo.o bar.o -lm
   ```

3. Move `libfoobar.so` to a suitable place and put a pointer to it into `.lparserc`.

There is an example makefile in the `lib` directory of the `lparse` distribution that can be used as a model for generating your own libraries.

The final step is to tell `lparse` that you want to use your own functions. This is done by adding a function declaration

```
function foo.
```

to the program. When a function is declared, `lparse` goes on and tries to find the function symbol from the libraries. It will pick the first match that it finds and display an error if no matches are found. If `lparse` founds some other external symbol with the same name, such as a character array, it will die horribly trying to jump at the symbol. If you get a lot of segmentation faults while using your own functions, this may be one reason.

```
lparse_constant_t lparse_constant_type(long constant)
int lparse_is_numeric(long constant)
int lparse_is_symbolic(long constant)
char *lparse_get_symbolic_constant_name(long symbolic_constant)
long lparse_get_symbolic_constant_index(char *symbolic_constant)
int lparse_symbol_exists(char *symbolic_constant)
long lparse_create_new_symbolic_constant(char *arg)
```

Figure 5.1: `Lparse` API functions

### 5.7.4 Lparse API

The `lparse` version 0.99.47 added a programming API that allows user-defined functions to manipulate symbolic constants. The API functions are defined in the file `lparse.h` that is located in the `lib` directory of `lparse` distribution.

The `lparse.h` contains declarations of seven functions that provide the basic constant handling capabilities. The functions are listed in Figure 5.1 and their descriptions are below. The current version of the API doesn't differentiate between symbolic constants and symbolic functions. That is, a symbolic function is internally treated as a oridinary constant that just happens to have a specific form.

The header file defines an enumeration `lparse_constant_t` to hold the possible types of lparse constants:

```
typedef enum { LP_NUMERIC, LP_SYMBOLIC } lparse_constant_t;
```

The following functions are available:

**lparse_constant_t lparse_constant_type(long *constant*)**
: The function `lparse_constant_type` returns the type (`LP_NUMERIC` or `LP_SYMBOLIC`) of the argument *constant*.

**int lparse_is_numeric(long *constant*)**
: Returns *true* if *constant* is a numeric constant, *false* otherwise.

**int lparse_is_symbolic(long *constant*)**
: Returns *true* if *constant* is a symbolic constant, *false* otherwise.

**char *lparse_get_symbolic_constant_name(long *symbolic_constant*)**
: Returns a pointer to the symbol table entry of *symbolic_constant* or `NULL` if it is not defined. As the pointer is to the actual symbol table, don't mess with it.

**long lparse_get_symbolic_constant_index(char **symbolic_constant*)**
: Returns the `lparse` symbolic constant that corresponds to the character string *symbolic_constant* or a special value `LP_INVALID_CONSTANT` if the symbolic constant is not defined.

**int lparse_symbol_exists(char *symbolic_constant)**

Returns *true* if *symbolic_constant* is defined as a symbolic constant and *false* otherwise.

**long lparse_create_new_symbolic_constant(char *new_string)**

The function stores its argument into the symbol table and returns its symbolic constant index value. It is safe to add same constant many times to the table and all calls will return the same value.

**Example 5.5**

The following code can be used to identify whether a constant is a symbolic function (this example can also be found in the `lib` directory of `lparse` distribution):

```
/* apitest.c -- a small example on
   how the lparse API can be used */

#include "lparse.h"
#include <string.h>

/* This function returns the constant 'true' if
   its first argument is a symbolic function,
   and 'false' otherwise */
long is_symbolic_function(int nargs, long *args)
{
  char *st = 0;
  if (lparse_is_symbolic(args[0])) {
    st = lparse_get_symbolic_constant_name(args[0]);

    /* supposes that there is a '(' in a constant only
       if it actually is a symbolic function */
    if (strstr(st, "(")) {
      return lparse_create_new_symbolic_constant("true");
    }
  }
  return lparse_create_new_symbolic_constant("false");
}
```

Supposing that the above function was compiled and linked to `liblparse.so`. Then, the following program

**Program 5.1**

```
function is_symbolic_function.
a(is_symbolic_function(1)).
b(is_symbolic_function(foo(1))).
c(is_symbolic_function(bar)).
```

Table 5.3: `Lparse` keywords

| | |
|---|---|
| `compute` | `const` |
| `external` | `function` |
| `hide` | `maximize` |
| `minimize` | `mod` |
| `not` | `show` |
| `weight` | |

gives the following output:

```
smodels version 2.23.  Reading...done
Answer:  1
Stable Model:  c(false) b(true) a(false)
True
```

◊

## 5.8 Keywords

`Lparse` has a set of keywords that may not be used for other purposes. The keywords are shown in Table 5.3. In addition to keywords `lparse` uses internal atoms, predicates, and variables. The names of the internal predicates and atoms start with an underscore (_). The names of internal variables start with `I_`. You should avoid using the internal symbols in your programs, or strange behavior may result.

# Chapter 6

# Examples

The source code of all examples in this chapter is included in the `examples` directory of lparse tarball. More examples can be found from `lparse-demo.tgz` which is available at

`http://www.tcs.hut.fi/pub/smodels/lparse/`.

## 6.1 Node Coloring

**Program 6.1**

```
% Node coloring problem by Tommi Syrjänen.
% Given a graph given as a set of nodes and edges
% find a way to color the nodes with 'n' colors such that
% two adjacent nodes are not colored with the same color.
const n=3.  % this can be changed with the
            %command line argument '-c'.
color(1..n).
% Each node should have exactly one color:
1 { node_color(N, C) : color(C) } 1 :- node(N).
% Two adjacent nodes have to have different colors:
 :- node_color(X, C), node_color(Y,C), edge(X,Y), color(C).
% Typical command line
% lparse -d none -c n=3 color2.lp graph | smodels
```

## 6.2 Logical Puzzles

The `example/puzzle` directory contains a couple SMODELS programs that solve logical puzzles that are taken from Raymond Smullyan's excellent *Forever Undecided* [17]. All following programs should be run with the command line

```
% lparse file | smodels 0
```

so that all possible answers are generated.

**Knights and Knaves**

The Island of Knights and Knaves has two types of inhabitants: knights, who always tell the truth, and knaves, who always lie.

One day, three inhabitants ($A$, $B$, and $C$) of the island met a foreign tourist and gave the following information about themselves:

1. A said that B and C are both knights.
2. B said that A is a knave and C is a knight.

What types are A, B, and C?

This logical puzzle can be solved with the following program:

**Program 6.2**

```
% Each person is either a knight or a knave
1 { knight(P), knave(P) } 1 :- person(P).

% There are three persons in this puzzle:
person(a ; b ; c).

% Rest of this program models the two hints.
% Hint 1:
% If A tells the truth, B and C are both knights
2 { knight(b), knight(c) } 2 :- knight(a).

% If A lies, both cannot be knights.
 :- knave(a), knight(b), knight(c).

% Hint 2:
% If B tells the truth, A is a knave and B is a knight
2 { knave(a), knight(c) } 2 :- knight(b).

% If B lies, one of the claims has to be false
 :- knave(b), knave(a), knight(c).
```

**Martian-Venusian Club, part 1**

On Ganymede — a satellite of Jupiter — there is a club known as the Martian-Venusian Club. All members are either from Mars or from Venus, although visitors are sometimes allowed. An earthling is unable to distinguish Martians from Venetians by their appearance. Also, earthlings cannot distinguish either Martian or Venusian males from females, since they dress alike. Logicians, however, have an advantage, since the Venusian women always tell the truth and the Venusian men always lie. The martians are the opposite; the Martian men tell the truth and the Martian women always lie.

One day a visitor met two Club members, Ork and Bog, who made the following statements:

1. Ork: Bog is from Venus.
2. Bog: Ork is from Mars.
3. Ork: Bog is male.
4. Bog: Ork is female.

Where are Ork and Bog from, and are they male or female?

**Program 6.3**

```
% All persons are from Mars or Venus
1 { martian(P), venetian(P) } 1 :- person(P).

% All persons are male or female
1 { female(P), male(P) } 1 :- person(P).

% All persons either lie or tell the truth depending on
% their origins and sex.
lies(P) :- person(P), martian(P), female(P).
lies(P) :- person(P), venetian(P), male(P).
truthful(P) :- person(P), martian(P), male(P).
truthful(P) :- person(P), venetian(P), female(P).

% A person may not tell the truth and lie at the
% same time
 :- person(P), lies(P), truthful(P).

% Persons:
person( ork; bog ).

% Hints
% 1.
venetian(bog) :- truthful(ork).
 :- lies(ork), venetian(bog).

% 2.
martian(ork) :- truthful(bog).
 :- lies(bog), martian(ork).

% 3.
male(bog) :- truthful(ork).
 :- lies(ork), male(bog).

% 4.
female(ork) :- truthful(bog).
 :- lies(bog), female(ork).
```

**Martian-Venusian Club, part 2**

The Martians and the Venetians often intermarry, and there are several mixed couples in the club. One couple approached the visitor and the following conversation ensued:

1. Visitor: Where are you from?
2. A: From Mars.
3. B: That's not true!

Was the couple mixed or not?

The program to solve this one uses the same basic foundations as the puzzle above and below only the changed parts are shown.

**Program 6.4**

```
% Persons:
person( a; b ).

% The persons in this puzzle are married so they can't
% have the same sex.
 :- male(a), male(b).
 :- female(a), female(b).

% The hints.
% 1.
martian(a) :- truthful(a).
 :- martian(a), lies(a).

% 2.
lies(a) :- truthful(b).
 :- lies(a), lies(b).
```

## 6.3 Planning

The largest example program in the `example` directory is `logistics.lp` which shows a way to encode planning problems as SMODELS programs. The program is too big to be included in this manual entirely but here are some selected bits.

In a planning problem, we are given a descriptions of the initial state of the world and the desired goal state. In addition, we are given a set of actions that can be used to change the state of world.

In logistics domain, we have an action `load_truck(Object, Truck)` that is used, naturally enough, to load packages into trucks. The precondition for this operation is that the object and truck are at the same place and the effect is that the package will be inside the truck.

The first step in converting the action to smodels rules is to add a third argument to it, namely time, to it. Thus we'll use predicate `load_truck(Object, Truck, Time)` to model the loading.

A natural way to encode actions is to use choice rules, in form:

```
{ action } :- preconditions.
```

If the body of a choice rule is true in a model, the head may be true in it, but it doesn't have to. Thus, when the preconditions are fulfilled we can either perform the action or decide to do some other action.

The effects of an action are implied by the action:

```
effects :- action.
```

The blocking of conflicting actions is done by adding a constraint that says that if an action doesn't change its own precondition, then the precondition has to hold also at the next instant:

```
precondition(I+1) :- action, precondition(I).
```

Using these guidelines we can encode `load_truck` with following rules:

**Program 6.5**

```
{ load_truck(Obj, Tr, I) } :-
   at(Obj, Loc, I),
   at(Tr, Loc, I),
   object(Obj),
   truck(Tr),
   location(Loc),
   time(I).
%Effects:
in(Obj, Tr, I+1) :-
   load_truck(Obj, Tr, I),
   truck(Tr),
   object(Obj),
   time(I).
changes(Obj, I) :-
   load_truck(Obj, Tr, I),
   truck(Tr),
   object(Obj),
   time(I).
% As one of the preconditions for load_truck(Obj, Tr, I) is
% at(Tr, Loc, I) and the operator doesn't change it, the
% truck has to be at the same place at the next instant.
at(Tr, Loc, I+1) :-
   load_truck(Obj, Tr, I),
   at(Tr, Loc, I),
   location(Loc),
   truck(Tr),
   object(Obj),
   time(I).
```

In addition to the operators we also need a set of frame axioms that take care of those parts of the world that doesn't change at an time step and that keep the system in consistent state:

**Program 6.6**

```
% FRAME AXIOMS
% Everything stays at the same place where it is unless
% some action moves it.
at(Obj, Loc, I+1) :-
   at(Obj, Loc, I),
   not changes(Obj, I),
   object(Obj),
   location(Loc),
   time(I).
% An object may not be in two places at the same time
 :- 2 { at(Obj, Loc, I) : location(Loc),
   in(Obj, Cont, I) : container(Cont) },
   object(Obj),
   time(I).
```

# Chapter 7

# Writing Smodels Programs

This chapter contains some miscellaneous topics on writing programs for SMODELS. The first section detains the operation of an Emacs major-mode for writing SMODELS code. The two following sections contain some tips for debugging logic programs and using parser warnings.

## 7.1 Editing Smodels Programs with Emacs

There is a major-mode for writing SMODELS programs with Emacs. It is defined in file `smodels-mode.el` which is located int the `lib` directory of `lparse` distribution.

To use the mode you have to copy `smodels-mode.el` into some directory that is mentioned in your `load-path` Emacs variable and add a command to load the mode when needed.

For example, suppose that you want to use the directory `~/.elisp` for all your Emacs Lisp files. Then you could copy `smodels-mode.el` to that directory and add the following commands to your `.emacs` startup file:

```
;; First set the load path
(setq load-path (cons "~/.elisp" load-path))

;; Load smodels-mode automatically when needed.
(autoload 'smodels-mode "smodels-mode" "Smodels Editing Mode" t)

;; Use smodels-mode for all files that end with `.lp'.
(setq auto-mode-alist (cons '("\\.lp$" . smodels-mode)
                              auto-mode-alist))
;; Turn syntax highlighting on automatically
(add-hook 'smodels-mode-hook 'turn-on-font-lock)
```

`Smodels-mode` knows how to indent SMODELS programs and performs some syntax highlighting. All keywords are printed with `font-lock-keyword-face` font, built-in functions with `font-lock-builtin-face`, and variables with `font-lock-variable-face`.

You can also run `lparse` and `smodels` processes under Emacs with `smodels-mode`. You can choose what parser you want to use with the command `M-x smodels-set-parser`. By default `lparse` is used. The actual `smodels` version that is used to compute the models can be set with `M-x smodels-set-program`. By default, `smodels` is used. You can set command line arguments with `M-x smodels-set-parser-arguments` and `M-x smodels-set-program-arguments`.

There are four different commands that can be used to start `smodels` processes:

- `M-x smodels-compute-buffer` grounds the current buffer with chosen parser, sends the results to a `smodels` process, and prints the stable models in buffer `*smodels*`.

- `M-x smodels-compute-files` works like `smodels-compute-buffer` but it allows you to process programs that are stored in multiple files. When you use this function the first time it asks what files you want to include with the run. If you want to change the file list later, you can do it with command `M-x smodels-set-file-list`.

- `M-x smodels-parse-buffer` sends the current buffer to the parser and displays the output in the buffer `*smodels*`. By default, the command displays its output in plain text format but if you include an argument (i.e. if you invoke it with `C-u M-x smodels-parse-buffer`), the output is in smodels internal format.

- `M-x smodels-parse-files` grounds many files.

The `smodels-mode` keymap binds above functions to following keys:

- `C-c C-b` is binded to `smodels-parse-buffer`

- `C-c C-f` is binded to `smodels-parse-files`

- `C-t C-b` is binded to `smodels-compute-buffer`

- `C-t C-f` is binded to `smodels-compute-files`

## 7.2 Debugging Smodels Programs

So now you have written a large logic program, but only thing that smodels wants to answer is `False`. Finding problems within logic programs may be tedious and frustrating since a small typo may ruin the whole program. It is not possible to give a debugging procedure that works every time, but here are some things that I have found helpful.

- Make sure that all constants begin with a lower case letter and that all variables begin with an upper case letter. I once spent a couple of hours searching for a mysterious bug in a planning model that seemed to come and go. The cause for the bug was that in one rule I had written `at(tr, Loc, I)` instead of `at(Tr, Loc, I)`. When there was only one truck (namely, `tr`) in the model, everything worked well, but when there were more trucks a plan couldn't be found because only one of the trucks could move.

- Comment rules out one at a time to see which rule causes the contradiction. Of course, if the problem is that some necessary rule is missing, you cannot find it this way. It is also possible that the rule commented out works correctly, but some other incorrect rule conflicts with it.

- Make the domains as small as possible. If your program works well when there is only one item of some type and it fails when there are more of them, it is quite likely that some rule demands that if something is true for one item, it is true for all of them. For example, I once tried to split the predicate `drive_truck(Tr, From, To, I)` into three smaller predicates: `drives(Tr, I)`, `moves_from(Tr, From, I)`, and `moves_to(Tr, To, I)`. The rule that I wrote for `moves_to` was of the form:

  ```
  moves_to(Tr, To, I) :-
     truck(Tr),
     location(To),
     time(I),
     drives(Tr, I).
  ```

  Grounding transformed this rule into form:

  ```
  moves_to(tr, a, 1) :- drives(tr, 1).
  moves_to(tr, b, 1) :- drives(tr, 1).
  ```

  That is, the rule demanded that if a truck drives somewhere at all, it must at the same time drive to each possible location.

- Use `compute` statements to test what combination of atoms cause contradictions. Start with an empty `compute` statement and if `smodels` finds a model, continue by adding some atoms that should be in the model to see where the things go wrong.

  For example, when debugging my planning encoding, I used `compute` statements of the form

  ```
  compute 1 { load_truck(pct,tr,1), drive_truck(tr,a,b,2),
              unload_truck(pct,tr,3), at(pct,b,4) }.
  ```

  after adding each operator to check that the program could find at least some legal plan.

- Use the `-t` option to see what exactly `lparse` does to your program. Because the ground programs are often quite big you should comment out everything that in your opinion is not related to the bug.

- Use the parser warnings. The warnings are detailed in the next section.
- You can also try to compute the partial models of the program using the opition `--partial`. This may sometimes give some hints on where the problem is located.

  For example, a program

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

  doesn't have any models since not $a$ implies $a$, which causes a contradiction. The problem here is that a `not` was forgotten from the first rule. Using the option `--partial` we get the following result:

  ```
  % lparse --partial foo | smodels
  smodels version 2.25.  Reading...done
  Answer:  1
  Stable Model:  a' b'
  True
  ```

  The atoms $a'$ and $b'$ are true in the partial model but $a$ and $b$ are not, suggesting that the problem is somehow related to $a$ and $b$ atoms, since they are only possibly true.

## 7.3 Parser Warnings

`Lparse` can detect some possible errors from its input. The checks are mostly aimed to detect constructs that have caused trouble earlier[1]. Most of the constructs are sometimes useful, but in wrong places they have caused a lot of debugging. If you want to use the warnings effectively, you should know what causes warnings and why.

`-W arity`

`Arity` detects cases when a predicate symbol has two different arities in same program. This warning is mainly intended for catching bugs where you forgot an argument from a predicate. For example

```
p(a).
p(a,a).
```

causes the warning

```
2: Warning: predicate 'p' is used with 2 arguments at line 2,
   while it is also used with 1 argument at line 1.
```

There are cases when when it is useful to have many different arities for same predicate. For example, you could want to say something like

[1] If you are sometimes bitten badly by something you think `lparse` should detect automatically, please send me an email about it and I'll try to incorporate a new warning for it in later releases.

```
goal :- goal(I), time(I).
```

to say that it doesn't matter when the goal is found. Of course, in many of these cases it would be better to use different predicate symbols.

`-W extended`

`Extended` detects general problems with extended rules. This is a warning that should be enabled nearly always. At least I haven't found any legitimate use for the constructs that trigger this warning.

Currently, `extended` warns about cases where you have explicitly defined weights in places where they don't have any effect. Most often this happens because you have used curly braces instead of brackets.

For example,

```
a :- 2 { b = 3, not c = 2 }.
```

gives the warning

```
1: Warning: weight defined for literal 'b' in a constraint
            rule.
1: Warning: weight defined for literal 'not c' in a constraint
            rule.
```

Here the fix is to replace '{' and '}' with '`[`' and '`]` '.

`-W library`

`Library` causes a warning in two cases: if you have defined a library file that doesn't exist in `.lparserc` or if you declare a function two times.

`-W similar`

`Similar` detects cases where you might have misspelled the initial letter of a constant or variable. For example, in the program

```
a(1..5).
b(3..7).
c(I) :- a(I), b(i).
```

the variable `I` was misspelled in the literal `b(i)` so the domain of the predicate `c(X)` is empty instead of `c(3..5)` as intended. If the warning `similar` is enabled, `lparse` prints:

```
3: Warning: constant 'i' is similar to variable 'I'
   (other occurrences of 'i' are not checked)
```

`Lparse` warns only the first possible typo that it sees.

This warning is quite often a false alarm, since there are many reasons why a program would have similar names for constants and variables. However, enabling this warning cathces some of the most annoying bugs in logic programs.

-W unsat
: Unsat is another option that can catch hard-to-find typos. Given this option lparse warns if there is some predicate symbol that can't be satisfied because it doesn't occur in a head of any rule. This option is mainly intended to find out cases where you have misspelled a predicate. For example, in the program

```
const max_time = 10.
time(1 ..  max_time).
{ action(I) } :- preconition(I), time(I).
precondition(I) :- time(I).
```

the first non-trivial rule is probably trying to say that an `action` is possible only if its `precondition` is true, but the predicate was misspelled. Sometimes these errors are a pain to find and correct. Using warning `unsat` `lparse` prints:

```
3: Warning: predicate 'preconition/1' doesn't occur in any rule
            head.
```

This option is similar to the one above in that there are many cases where it is not an error to have some unsatisfiable predicate in the program. For example, in the program,

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

we want that `a` is not in any model unless we specifically allow it by adding `enable_a` as a fact into the program.

-W weight
: Weight prints a warning if you use the default weight of a literal in some weight rule. Default weights are very often useful but if you want to define weight explicitly for each literal, you can set this option to catch typos in global weight definitions. For example, you might have program

```
e(1..3).
weight c(1) = 10.
weight c(2) = 15.
weight d(3) = 20.
a :- 30 [ c(X) : e(X) ].
```

Here the idea is, that `a` should be true if the total weight of true `c(X)` atoms is more than 30. However, the last weight definition was misspelled so that the weight of `c(3)` is the default 1 instead of 20.

Given the option `-W weight` `lparse` warns:

```
5: Warning: default weight used for literal 'c(3)'
```

-W error
: Error causes `lparse` to treat all warnings as errors.

There are three options that can be used to set more than one warning flag at a time:

- `-W all` enables all warnings.
- `-W syntax` sets `arity`, `extended`, and `weight`.
- `-W typo` sets `similar` and `unsat`.

## 7.4 Handling BIG programs

You sometimes meet problems where the `smodels` finds the stable models in few seconds but `lparse` takes 20–30 seconds (or even more) to ground the problem. If these are isolated cases, the problem is not severe. On the other hand, if you want to alter the `compute` statements just a bit or tweak the domain predicates, having to ground the whole program every time can be quite a nuisance.

This problem can be partially solved using `external` declarations and the command line option `-g`. The option allows you to read in a previously grounded progam and add new rules to it.

In this section we consider three cases that occur in practice:

1. we don't have to change the actual grounded program but we want to change the `compute` statement;
2. there's a domain predicate whose extension is not available during grounding but we know what is the largest possible extension; and
3. we have an existing program and we want to enlarge the extensions of domain predicates to get more rules.

The second case occurs for example in the configuration management problem; we know that there's a set of components that the user can choose but we don't know what the actual choices will be for each configuration task beforehand. An example of the third case occurs in planning problems where we may want to increase the number of time steps if a plan can't be found.

For the rest of this section we will examine the program `test.lp`:

**Program 7.1**

```
a(1 .. max_a).
b(X) :- a(X), not c(X).
c(X) :- a(X), not b(X).
```

### 7.4.1 Altering the `Compute` Statement

So, you have now written `test.lp` and you want to test it out with several different `compute` statements without having to ground the program again each time.

First step is to ground the program the first time:

```
% lparse -c max_a=2 test.lp > test\_output
```

This command grounds `test.lp` and stores the output in the file `test_output`. The argument `-c max_a=2` sets the extension of $a$ to $\{a(1), a(2)\}$.

Next step is to write a source file `compute.lp` that contains the `compute` statement:

`compute { b(1) }.`

Now you can find all models of Program 7.1 with the command line:

`$ lparse -g test_output compute.lp | smodels 0`

The output of the command is:

```
smodels version 2.25. Reading...done
Answer: 1
Stable Model: b(1) c(2) a(1) a(2)
Answer: 2
Stable Model: b(1) b(2) a(1) a(2)
False
```

Changing the `compute.lp` to

`compute { b(1), b(2) }.`

you will get only one model, as expected:

```
smodels version 2.25. Reading...done
Answer: 1
Stable Model: b(1) b(2) a(1) a(2)
False
```

Now, with all this success you might want to simplify the `compute` statement a bit and change it to the form:

`compute { b(X) : a(X) }.`

However, quite surprisingly the answer is not the same as it was to the earlier query! Instead of having only one model containing $b(1)$ and $b(2)$, you will get all four models of Program 7.1:

```
smodels version 2.25. Reading...done
Answer: 1
Stable Model: b(1) c(2) a(1) a(2)
Answer: 2
Stable Model: b(1) b(2) a(1) a(2)
Answer: 3
Stable Model: c(1) b(2) a(1) a(2)
Answer: 4
Stable Model: c(1) c(2) a(1) a(2)
False
```

What happened here? The explanation lies in the fact that once a rule is grounded its structure is lost. There is no connection from the grounded rule to the original non-ground rule that generated it. Similarily, the knowledge of domain predicates and their extensions is lost.

`Lparse` first read the ground program in, recognizing and storing the atom names in the process. Then it simply passed the rules, unmodified, to `smodels` and started to read in `compute.lp` and process. As it didn't give any definitions to the predicate $a/1$, the condition in the `compute` statement was expanded to the empty set of literals that didn't constraint the models in any way.

In theory it would be possible to construct the domains from the grounded program but this would cause other problems:

1. it is possible that the program was originally grounded with `-d none` and there are actually no domain predicates left at all; and
2. computing domains would make it impossible to incrementally ground a program as all rules would need to be grounded for the old domains as well as the new ones.

### 7.4.2 Restricting the Extensions of Domain Predicates

In the second scenario you want to alter the domains of domain predicates and you have the advantage of knowing the maximal extensions for each predicate. In terms of Program 7.1, you would know that there were at most `max_a` instances of $a(X)$ but you didn't know which ones are necessary.

These situations can be handled with the `external` declarations. We start by modifying `test.lp` a bit:

**Program 7.2**

```
external a(X).
a(1 .. max_a).
b(X) :- a(X), not c(X).
c(X) :- a(X), not b(X).
```

We can now check with the `-t` option what happens to the program:

```
% lparse -t -c max_a=2 test.lp
c(1) :- a(1), not b(1).
c(2) :- a(2), not b(2).
b(1) :- a(1), not c(1).
b(2) :- a(2), not c(2).
```

As we see, `lparse` created a rule for each possible binding of $a(X)$ didn't include them as facts in the program. Now we can specify the extension of $a$ in a separate program `a_def.lp`:

```
a(1).
```

We can now combine `a_def.lp` with the `test_output` generated as in the previous section to get:

```
% lparse -g test_output a_def.lp | smodels 0
smodels version 2.25. Reading...done
Answer: 1
Stable Model: b(1) a(1)
Answer: 2
Stable Model: c(1) a(1)
False
```

Note that you cannot use a `hide` declaration on a predicate that is declared to be `external`. The reason for this is that once you have thrown the name away, only thing that is left is the numerical index of the atom and it isn't possible to associate it to its original representation.

If you really wish, you can read more than one grounded program in with `-g` switches. However, this will probably cause problems as different programs may have used the same numerical index for different atoms. `Lparse` notices and gives an error message for many of these cases but if there are hidden or internal atoms, it is very likely that the programs will mix together incorrectly. So use multiple `-g` options only if you are absolutely sure that the atom lists are identical for both programs.

### 7.4.3 Enlarging the Extensions of the Domain Predicates

In the third scenario we have grounded the program but we want to add more instances of domain predicates to it. This can be done most conveniently when using numeric domains but the same principles hold also for symbolic domains. Again, we start by modifying `test.lp`:

**Program 7.3**

```
a(min_a .. max_a).
b(X) :- a(X), not c(X).
c(X) :- a(X), not b(X).
```

First we ground the program with the initial values of domain predicates:

```
% lparse -c min_a=1 -c max_a=2 test.lp > test_output
```

Suppose that we now want to increase the maximum value for $a$ to 3. We can do that by reading in the grounded program, setting the minimum and maximum values to 3 and grounding the original program again:

```
% lparse -g test_output -c min_a=3 -c max_a=3 test.lp > new_output
% smodels 0 < new_output
```

### 7.4.4 Hacking Bits and Pieces Together by Hand

If the grounded program is sufficiently big, just running it through `lparse` can take too much time as each atom has to be individually processed. In those cases you may have to roll up your sleeves and do some manipulation directly to the grounded program. When doing this, you may find it helpful to read Section B where the internal `smodels` 2.x format is explained.

There is one command line option, `--atom-file` that can be used to divide the `lparse` output into two parts: the rules and the rest stuff. Given the command line

```
% lparse --atom-file atoms program.lp
```

`lparse` prints the ground rules of `program.lp` to standard output and sends the symbol table and `compute` statement to the file `atoms`.

For example, we can extract the symbol table of Program 7.1 with the command line:

```
% lparse --atom-file atoms -c max_a=2 test.lp
1 1 1 1 2
1 3 1 1 4
1 2 1 1 1
1 4 1 1 3
1 5 0 0
1 6 0 0
```

The rules are in `smodels` internal format. The most interesting rules for hacking around are the last two rules. Rules of the form:

```
1 n 0 0
```

denote that the atom number $n$ is a fact in the program (i.e., there's a basic rule with $n$ as the head and an empty body in the program).

After the preceeding command line the `atoms` file looks like this:

```
1 c(1)
2 b(1)
3 c(2)
4 b(2)
5 a(1)
6 a(2)
0
B+
0
B-
0
1
```

The first part is the symbol table giving the representation for each atom. The part after `B+` contains the atoms that have to be in a model (the zero designates the end of this part), part after `B-` contains the negative `compute` statement and the final line gives the desired number of models.

The easiest way to modify grounded programs by hands is to ground it with `external` declarations, read in the symbol table and `compute` statement, and add some of the external atoms as facts to the program.

## 7.5 Miscellanous Tips

This section contains a few miscellanous tips that can help in writing SMODELS programs. Most of this stuff is also mentioned elsewhere in the manual, but I thought that it would be nice to collect them to one place. This section will probably increase in size in the future revisions when more things come to my mind.

**The `-t` option**
: The `-t` is probably the most important `lparse` command line option as you can use it to see what actually happened to your program when it went through `lparse`.

**The `-d` option**
: The second most important option is `-d none` that simplifies the output by leaving out the domain predicates. It also speeds `smodels` a little.

**Hide declarations**
: Another way how you can get simpler models is to use `hide` declarations liberally. Postprocessing the answer is much easier when you don't have to worry about some hundreds of uninteresting atoms.

**Using a predicate as its own condition**
: You may use a domain predicate as its own condition in constraint and weight literals. For example, you might want to say that a graph is big if there are more than 100 edges:

```
big_graph :- 101 { edge(X,Y) : edge(X,Y) }.
```

# Chapter 8

# Future Development

I have a couple of ideas that will probably be included in `lparse` some day. They include:

- Extended support for data types. I vision system like one that is used in the programming language Scheme, that is, the user may freely mix normal integers, floating point numbers, and bignums and the system takes care of rest. However, as this requires rewriting of quite large parts of `lparse` code this will not happen very soon.
- Some sort of unified method for setting attributes for predicates, like marking them hidden, setting weights, and like. This feature would prevent "keyword-pollution" that has been threatening `lparse` lately. The feature will be implemented in such way that it will be easy to convert current programs into new format automatically.
- Possibility of using predicates that occur as head of a special rule as domain predicates.

If you have some suggestions or if you found a bug in `lparse`, please send email it for me (*tommi. syrjanen@hut.fi*).

# Appendix A

# Smodels API

This section contains a brief overview of the `smodels` programming API. The API is a library interface that allows a C++ program to construct logic programs and compute their stable models. Unfortunately, the current interface is quite unintuitive and there's no easy way of using the functionality of `lparse` with it, so in practice only ground programs can be handled conveniently. However, the integration of these functionalities will be added in a future version and the library interface will be cleaned.

## A.1 Installing and Using the API

Before you can use `smodels` as a library you have to compile it as such. This can be achieved by typing:

```
% make lib
```

in the `smodels` source directory. The library build process uses GNU libtool that should be installed on the system. The command:

```
% make libinstall
```

will then install the libraries to a path that is specified in the `Makefile`. The default path is `/usr/local/lib`. The command doesn't install the header files anywhere, so they should either be copied to a suitable location by hand or the programs should be compiled with the '`-I`' compiler option. The most important header files are listed in Figure A.1.

If the compiler and linker can find the header files and the `libsmodels.la` file, respectively, you can link the library to your programs in the usual way, using the '`-lsmodels`' argument to instruct the linker:

```
% gcc -o foo -I/smodels/header/path foo.cc -lsmodels
```

The `smodels` example directory contains several examples on API use.

## A.2 Header Files

This section goes through the five most important header files and presents important classes and methods that are defined in them. However, this is not intended to be a complete reference but it is more like a cookbook of useful stuff.

### A.2.1 defines.h

The most important thing in the header file `defines.h` is the definition of different rule types:

```
typedef enum {
  ENDRULE,
  BASICRULE,
  CONSTRAINTRULE,
  CHOICERULE,
  GENERATERULE,
  WEIGHTRULE,
  OPTIMIZERULE
} RuleType;
```

The meaning of different rule types can be best explained by showing the corresponding logic program segments.

**ENDRULE**
An ENDRULE is not a concrete rule but it is used as a placeholder before the actual rule type is decided. You don't have to worry about it.

**BASICRULE**
A basic rule is a normal logic programming rule that doesn't contain any constraint or weight literals. For example, a rule

```
a :- b1, b2, b3, not c1, not c2, not c3.
```

is a basic rule.

**CONSTRAINTRULE**
A constraint rule has a basic literal as its head and its tail is one constraint literal with only a lower bound. For example, the rule

| | |
|---|---|
| `api.h` | Functions for creating logic programs |
| `atomrule.h` | Definitions of atoms and rules |
| `defines.h` | General definitions |
| `stable.h` | Functions for reading programs from files |
| `smodels.h` | Functions for computing stable models |

Figure A.1: Important `smodels` header files

```
a :- 2 { b1, b2, b3, not c1, not c2, not c3 }.
```

is a constraint rule.

**CHOICERULE**
A choice rule has a constraint literal with no bounds as its head and the body has only basic literals:

```
{ a1, a2, a3 } :- b1, b2, b3, not c1, not c2, not c3.
```

**GENERATERULE**
Generate rules are not used anymore since their semantics couldn't be defined in a nice way.

**WEIGHTRULE**
A weight rule is otherwise similar to a constraint rule but every literal in the tail should have an explicit weight defined for it. For example,

```
a :- 2 [ b1=1,b2=2,b3=3,not c1=1,not c2=2,not c3=3 ].
```

**OPTIMIZERULE**
An optimize rule handles minimize and maximize statements. However, only minimization is explicitly modeled and maximization has to be done by negating all literals in the rule body. Also, every literal in the rule body needs a weight definition for it.

### A.2.2 api.h

The header file `api.h` contains the class Api that can be used to create and manipulate logic programs. The class has the following methods:

**Constructor**
The constructor of the Api class is:

```
Api (Program *pr);
```

The class Program is defined in the header `program.h` and it is not necessary to know about its internal details to be able to use the API. The only thing that you need to ensure is that an Api has a valid pointer to use. In practice, most often you want to use a program that is tied to an instance of the Smodels class that will be presented below. So an Api is usually created using the following commands:

```
Smodels smodels;
Api api(&smodels.program);
```

You'll get a segmentation fault if the pointer is not valid, so it is not one of those hard-to-detect bugs.

**Destructor**
The destructor of the class is simply:

```
virtual ~Api();
```

and it doesn't do anything special.

**Creating Rules**

The rules are created using the following three methods:

```
void begin_rule (RuleType type);
void add_head (Atom *a);
void add_body (Atom *a, bool pos);
void add_body (Atom *a, bool pos, Weight w);
void end_rule ();
```

The possible rule types are described in the above section. When using all other rule types but CHOICERULE you may add only one head to a rule. The `pos` argument to an `add_body` call decides whether the literal is positive or negative.

**Example A.1**

The rule

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

could be created with

```
api.begin_rule(BASICRULE);
api.add_head(a);
api.add_body(b, true);
api.add_body(c, false);
api.end_rule();
```

supposing that the Atoms `a`, `b`, and `c` have been defined accordingly.
◊

**Setting bounds**

The bounds for constraint and weight rules can be defined using the following two functions:

```
void set_atleast_body (long);
void set_atleast_weight (Weight);
```

Currently the type `Weight` is defined with a typedef:

```
typedef unsigned long Weight;
```

Note that both of these functions have to be called before `end_rule()`.

**Creating and Manipulating Atoms**

There are five functions that can be used to create and manipulate atoms.

```
virtual Atom *new_atom ();
void set_name (Atom *a, const char *name);
Atom *get_atom (const char *name);
void set_compute (Atom *a, bool in_model);
void reset_compute (Atom *a, bool in_model);
```

The `new_atom()` function creates a new atom and returns a pointer to it. The atom doesn't have a name assigned to it and it has to be set with the `set_name()` function. Note that you have to call the `remember()` function before you can define names. If you know a textual representation of an atom, you can get a pointer to it with the `get_atom` function. Note that `get_atom` returns NULL if no such atom is defined.

The two last functions are used to define the compute statement. A statement

```
set_compute(a, true);
```

asserts that the atom `a` has to be true in the models. Correspondingly,

```
set_compute(a, false);
```

asserts that it must be false. If neither statement is given, the atom may be true or false. If both are given, the result is an immediate contradiction.

The last function can be used to remove an atom from the compute statement. The boolean argument is used to indicate whether the atom is removed from the positive or the negative compute list.

**Example A.2**

Suppose that you want to define an atom $a$ and demand that it has to be true in all models. Then you can do it with the program snippet:

```
api.remember();
Atom *a = api.new_atom();
api.set_name(a, "a");
api.set_compute(a, true);
```

◊

**Handling Atom Names**

There are two functions that control whether atom names are allowed or not:

```
void remember ();
void forget ();
```

You can call `remember()` when you want to use names for atoms and `forget()` when you want to get rid of them.

**Miscellanous Functions**

```
void copy (Api *ap);
void done ();
```

The function `copy()` generates a new copy of the Api that is given as the argument. The function `done()` is called when all rules of the program have been constructed. This is used to signal the Api class that it can now create the internal data structures for the program. You can't add new rules or atoms to an Api after the `done()` call.

The following code example is based on the file `example.cc` that is in the `examples` directory of the `smodels` distribution.

**Example A.3**

We will create the Api representation of the simple program:

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

The resulting C++ code is:

```
#include "smodels.h"
#include "api.h"
int main ()
{
  Smodels smodels;
  Api api (&smodels.program);

  // Keep track of atom names
  api.remember ();

  // Define the atoms
  Atom *a = api.new_atom ();
  Atom *b = api.new_atom ();
  api.set_name (a, "a");
  api.set_name (b, "b");

  // define the rule a :- not b
  api.begin_rule (BASICRULE);
  api.add_head (a);
  api.add_body (b, false);
  api.end_rule ();
  // and b :- not a.
  api.begin_rule (BASICRULE);
  api.add_head (b);
  api.add_body (a, false);
  api.end_rule ();

  // compute statement
  api.set_compute(a, true);

  // signal the end
  api.done();
}
```

◊

### A.2.3 `atomrule.h`

The header file `atomrule.h` contains the definitions of the classes Atom and Rule. Both contain lots of internal stuff that is not necessary for using the API. However, the Atom class contains few useful methods and attributes:

**Constructor**

The constructor is defined as:

```
Atom (Program *p);
```

The argument `p` is a pointer to the program where the atom occurs. You shouldn't have to create atoms directly as it is better to use the `new_atom` method of the Api class.

**Name**

The name of an atom (if defined) can be obtained with the method:

```
const char *atom_name ();
```

**Truth value**

The truth value of an atom in a stable model is stored in two attributes:

```
bool Bpos : 1;
bool Bneg : 1;
```

The variable `Bpos` is true when the atom is in a stable model and `Bneg` is true when the atom is not in the model. If both are false, then the atom is neither true or false. Note that this may not occur in normal use.

**Compute statement**

The following two attributes control whether an atom is in the compute statement:

```
bool computeTrue : 1;
bool computeFalse : 1;
```

You can also use the `set_compute` method of the Api class to do this.

### A.2.4 `smodels.h`

The header file `smodels.h` contains the definition of the class Smodels that implements the actual computation of stable models. The most important methods of the class are:

**Constructor**

```
Smodels ();
```

The constructor doesn't take any arguments.

**Initialization**

After the rules of a logic program have been generated using the Api class that is tied to a Smodels instance, the computation has to be initialized with the initialization function:

```
void init ();
```

**Computation**

The stable models are computed with the `model` function.

```
int model (bool look = true, bool jump = false);
```

The function returns 1 if a model is found and 0 if there are no more models. Successive calls return all models of the program. For example, the code snippet:

```
while (smodels.model()) {
  // do something
}
```

can be used to generate all stable models.

The arguments to the function control whether lookahead heuristics is used and whether backjumping techniques are enabled. Lookahead helps in most programs so it is usually a good idea to leave it on. However, the algorithm is quadratic and with programs that have "easy" structure may be slower with it. Backjumping is sometimes worthwile but usually it costs more than it helps so it is off by default.

**Examining Models**

After a model has been computed, it can be printed with the function

```
void printAnswer ();
```

If you want to go through the atoms one at a time, you can do it by going through the atom list of the Program component using the following code snippet:

```
Node *nd = smodels.program.atoms.head();
for ( ; nd ; nd = nd->next ) {
  if (nd->atom->Bpos) { // the atom is true
    // do something
  } else if (nd->atom->Bneg) { // the atom is false
    // do something
  }
}
```

**Resetting the Computation**

The function

```
void revert ();
```

undoes all changes that have been done after calling `init()`. In particular, it will destroy all backtrack information so the next `model()` call will again find the first model. This call is useful when you want to use many compute statements.

**Miscellanous Bits**

The next functions may be useful if you want to do something special with the logic program. They are called by the above functions but sometimes it may be useful to call them independently.

```
void setup ();
void setup_with_lookahead ();
```

These two functions simplify the program by first computing a deductive closure of the rules and then dropping all unsatisfiable atoms and rules from it. The latter function also does one round of lookahead examination.

```
bool conflict ();
```

The `conflict()` function checks whether a contradiction is found. Also, if optimize statements are used, this function checks whether the model candidate is better than the currently best found. Note that this function clears the conflict flag so you'll have to be careful with it.

```
void lookahead ();
bool lookahead_no_heuristic ();
void heuristic ();
```

The two `lookahead` functions choose what literal should be added to the model candidate. The index number of the chosen atom is stored into the variable `hi_index`. If the best choice is a positive literal, the boolean variable `hi_is_positive` is set to be true, otherwise it is set to false.

```
void expand ();
```

The function `expand()` computes the deductive closure of the atoms whose truth value is known.

```
int wellfounded ();
```

This function computes and prints the well-founded model of the program.

```
void setToBTrue (Atom *a);
void setToBFalse (Atom *a);
```

These two functions can be used to set truth values of atoms directly. Note that usually it is better to set the compute statement, instead.

### A.2.5 `stable.h`

The header file `stable.h` contains the class Stable that can be used to read logic programs from files. The programs should be stored in `smodels` internal format that is explained in Section B. The important methods are:

**Constructor**

The constructor is simply

```
Stable ();
```

**Reading programs**

Logic programs can be read with the method

```
int read (istream &f);
```

The argument `f` should be bound to the file storing the program.

### A.2.6 Example

This section contains an example API function that can be used to compute partial models of logic program, not in the sense of Section 4.8, but in the sense that it will find a partial model candidate that can possibly be extended to a full stable model. This kind of function may be useful when the program is large and we want to construct the model interactively.

For conveniety, this function is added to the Smodels class, but it might be used also outside it.

```
bool Smodels::partial_model ()
{
  // continue as long as there are possible models
  while (!fail)
  {
    // compute the deductive closure of
    // the current partial model
    expand ();
    if (conflict ()) // is the situation consistent?
      backtrack (true);
    else if (covered ())
      return true; // a full model
    else if (!lookahead_no_heuristic ())
      return true; // a partial model
  }
  return false; // no models left
}
```

This function would then be used like:

```
while (1) {
  if (partial_model ()) {
    // process the partial model and possibly ask
    // the user how the model should be extended.
    // the selections can be expressed using
    // setToBTrue(atom) and setToBFalse(atom)
    // calls. Alternatively, you can use the sequence
    //      heuristic (); choose ();
    // to use smodels's heuristics.
  } else {
    // print some diagnostics
    backtrack (true)
  }
}
```

# Appendix B

# Smodels Internal Format

The language that `smodels` 2.x accepts is much simpler than the one accepted by `lparse`. During grounding `lparse` transforms the complex rules to those accepted by `smodels`. There are four different rule types: basic rules, constraint rules, choice rules, and weight rules.

Additionally, the minimize statements are internally represented by their own rule type. The maximize statements are changed into minimize statements by negating all literals in them.

Basic rules are the normal rules that don't use any extended features. Constraint rules correspond to `lparse` rules of the form

```
a :- 2 { b, c, not d }.
```

choice rules have the form

```
{ a, b, c } :- d, e, not f, not g.
```

and weight rules have the form

```
a :- 2 [ b=1, c=2, not d=3 ].
```

In all cases there may be only one special construct in one rule and there are only lower bounds for constraint and weight rules.

Internally `smodels` uses integers as atoms and the atom names are stored in a separate symbol table. The `smodels` expects to read first the actual rules of the program, next the symbol table, and finally the compute statement. The different parts are separated by a line that has only a '0'.

The different sections are best introduced by an example. Consider the following program:

**Program B.1**

```
a :- not b.
b :- not a.
:- b.
compute { a }.
```

The internal format for this program is:

```
1 2 1 1 3
1 3 1 1 2
1 1 1 0 3
0
2 a
3 b
0
B+
2
0
B-
1
0
1
```

The first part of the listing consists of the the rules of the program:

```
1 2 1 1 3
1 3 1 1 2
1 1 1 0 3
0
```

The first number denotes the rule type. All of the rules are basic rules so the number is one in all cases. The next number identifies the head of the atom. In this case, the atom `a` is represented by 2 and the atom `b` by 3. The atom number 1 is an internal atom named `_false` that is true when a model candidate should be rejected.

Next comes the body definition. The first number is the total number of literals in the body and the second one is the number of negative literals. The rest of the line contains the numbers of the literals, with negative ones being in front. The line with just 0 signals the end of the rules.

The second part is the symbol table containing the atom definitions:

```
2 a
3 b
0
```

If an atom is left out the symbol table, it is considered to be a hidden atom and it is not printed in the model. In this example, the first atom is hidden.

The third part contains the compute statement

```
B+
2
0
B-
1
0
1
```

After `B+` comes the positive compute statement, that is, a list of atoms that should be true in the model. After `B-` comes a list of atoms that shouldn't be in the model. The last 1 signifies the number of models that should be calculated.

Table B.1: The mapping of atoms that are used in the examples

| atom | number |
|---|---|
| a | 1 |
| b | 2 |
| c | 3 |
| d | 4 |
| e | 5 |

## B.1 Rule Types

**Basic Rule**

A basic rule has the form:

`1 head #literals #negative` *`negative positive`*

Where `head` is the atom that is the head of the rule, `#literals` is the total number of literals in the rule body, `#negative` is the number of negative literals in the rule body, *negative* is the list of negative literals, and *positive* is the list of positive literals.

**Example B.1**

Let the atoms be defined as in Table B.1. Then, the rule:

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

is represented as:

```
1 1 4 2 3 5 2 3
```

◊

**Constraint Rule**

A constraint rule has the form:

`2 head #literals #negative bound` *`negative positive`*

where `head`, `#literals`, `#negative`, *negative*, and *positive* are as with basic rules and `bound` is the amount of literals in body that has to be true so that the head is true.

**Example B.2**

Given the bindings in Table B.1, the rule

```
a :- 2 { b, c, not d }.
```

is represented as

```
2 1 3 1 2 4 2 3
```

◊

**Choice Rule**

A choice rule has the form

`3 #heads` *`heads`* `#literals #negative` *`negative positive`*

Most of the entries are the same as with basic rules. The entry `#heads` denotes the number of the atoms in a choice rule head and *heads* is the list of the atoms in the rule head.

**Example B.3**

Using the usual bindings for the atoms, the rule

```
{ a, b, c } :- e, not d.
```

is represented as

```
3 3 1 2 3 2 1 4 5
```

◊

**Weight Rule**

A weight rule has the form

`5 head bound #lits #negative` *`negative positive weights`*

Most of the entries are the same as in constraint rules. The entry *weights* is a list of the weights of the literals in the rule body.

**Example B.4**

The rule:

```
a :- 3 [ b=1, not c=2 ].
```

is represented as:

```
5 1 3 2 1 3 2 2 1
```

◊

**Minimize Rule**

A minimize rule is of the form:

`6 0 #lits #negative` *`negative positive weights`*

Note that each literal has to have an explicit weight assigned to it. Maximization can be achieved by negating all literals in the statement body.

**Example B.5**

The statement:

```
maximize [ a=5, not b = 10 ].
```

is represented as:

```
6 0 2 1 2 1 10 5
```

◊

In case you wonder, the missing rule type 4 was originally used for generate rules that were essentially choice rules with bounds. As they caused semantical troubles, they were removed from use.

# Bibliography


[1] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364 – 418, September 1997.

[2] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. The MIT Press.

[3] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the 7th International Conference on Logic Programming*, pages 579–597, Jerusalem, Israel, June 1990. The MIT Press.

[4] Keijo Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 240–254. Springer-Verlag, 1999.

[5] Tomi Janhunen, Ilkka Niemelä, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning*, 2000.

[6] W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, pages 375–398. Springer-Verlag.

[7] I. Niemelä and P. Simons. Evaluating an algorithm for default reasoning. In *Working Notes of the IJCAI'95 Workshop on Applications and Implementations of Nonmonotonic Reasoning Systems, Montreal, Canada*, pages 66–72, Montreal, Canada, August 1995.

[8] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, September 1996. The MIT Press.

[9] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on*

*Logic Programming and Nonmonotonic Reasoning*, pages 317–331, El Paso, Texas, USA, December 1999. Springer-Verlag.

[10] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, April 2000.

[11] Ilkka Niemelä and Patrik Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 420–429, Dagstuhl, Germany, July 1997. Springer-Verlag.

[12] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the Fifth Interational Conference on Logic Programming and Nonmonotonic Reasoning.* Springer-Verlag, December 1999.

[13] P. Simons. Efficient implementation of the stable model semantics for normal logic programs. Research Report 35, Helsinki University of Technology, Helsinki, Finland, September 1995.

[14] P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research Report 47, Helsinki University of Technology, Helsinki, Finland, August 1997.

[15] P. Simons. Extending the stable model semantics with more expressive rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 305–316, El Paso, Texas, USA, December 1999. Springer-Verlag.

[16] P. Simons. Extending and implementing the stable model semantics. Research Report 58, Helsinki University of Technology, Helsinki, Finland, 2000.

[17] Raymond Smullyan. *Forever Undecided.* Alfred A. Knopf. Inc, 1988.

[18] Leon Sterling and Ehud Shapiro. *The Art of Prolog, 2nd ed.* MIT press, 1994.

[19] T. Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical Report B 18, Helsinki University of Technology, Helsinki, Finland, October 1998.

[20] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23:733–742, 1976.

# Index