491 Knowledge Representation

Tutorial Exercise

Extended Logic Programs

Question 1

Check out the examples discussed in the lecture notes under the heading *Priorities and exceptions*.

In each case formulate representative test cases to check how the suggested formalisations deal with various combinations of facts.

You will need to calculate the answer sets. You can do this by hand, using 'splitting sets', or using the ASP solver `clasp`. `clasp` is installed in the directory `/vol/lab/CLASP`. There are very brief instructions in the `README` file.

You can find files containing all the examples on this sheet in the sub-directory `TutorialExamples`. They are all read-only so you will have to make your own copies if you want to play around with modifications.

You could also try out the various 'civilised football-supporter' examples earlier in the lecture notes to see how they behave.

Question 2

Here is an example that was formalised on an earlier exercise sheet as a normal logic program. Formulate it now as an extended logic program. The only difference will be in the treatment of 'not punished' which was implicit in the normal program representation.

(r1) Except as provided for by r2, all thieves should be punished.

(r2) Except as provided for by r3, thieves who are minors should be rehabilitated and not punished.

(r3) Any thief who is violent should be punished.

Check your formulation by computing answer sets for a thief, a thief who is a minor, a thief who is violent, a thief who is a minor and violent.

How do you express 'Closed World Assumptions' to the effect that, by default, a person is not a thief, not violent, not a minor?

What difference does it make to add these assumptions to the program?

Why do the clauses in this example look a bit different from those in Question 1 examples? For instance, why does the clause for (r1) not require a condition `not ¬punish(X)`?

Question 3

Here's a variation of a previous example, but formulated now as an extended logic program, and with `¬strong(X)` instead of `weak(X)`.

```
strong(X) ← big(X), not ¬strong(X)
strong(X) ← small(X), muscular(X), not ¬strong(X)
¬strong(X) ← small(X), not strong(X)
big(mary)
small(frank)     muscular(frank)
```

Compute the answer sets for this program.

(*Hint:* consider separately `mary` and `frank`, and use splitting sets.)

Question 4

Modify the above so that the small-muscular default has priority over (is an exception to) the general default about small people.

Compute the answer sets again. This time add also:

```
big(bill)        ¬strong(bill)
muscular(alice)  % Nothing about her size
```

Also, let's add another level of exceptions:

- Old people are typically not strong.

Let's suppose this takes priority over any other default.

Try it out on a representative set of example cases.

contd.

Question 5

Here is a slightly modified version of the rules in Question 3. This one separates `weak(X)` from `¬strong(X)` which arguably gives a nicer, more natural representation.

```
strong(X) ← big(X), not ¬strong(X)
strong(X) ← small(X), muscular(X), not ¬strong(X)
weak(X) ← small(X), not ¬weak(X)
¬weak(X) ← strong(X)
¬strong(X) ← weak(X)
big(mary)
small(frank)     muscular(frank)
```

For this version, add

- Old people are typically weak.

Why does the simple method sketched in the lecture notes for dealing with priorities and exceptions not work so well with this version? You should nevertheless be able to adapt the method to cope (in this example, if not in general).

Question 6 ***from 2002 Exam***

```
h(x) ← l(x), not ¬h(x)
¬h(x) ← t(x,y), ¬h(y)
s(x) ← l(x), not d(x)
d(x) ← t(x,y), not h(x)
l(a)
l(b)
t(a,b)
t(a,c)
¬h(a)
```

This extended logic program has exactly one answer set. Why?

Hence or otherwise, compute the answer set.

So: what are the answers to the following three queries on this extended logic program?

`h(a)` ?, `h(b)`?, `h(c)`?

Question 7

Formulate the following example as an extended logic program.

- Students are typically lazy and poor.
- Postgraduate students (PG students for short) are a type of student, and are typically not lazy.
- Business students (BS students for short) are typically PG students and are typically not poor.

You will have to make some assumptions about which defaults take priority over (are exceptions to) which.

You should also add some 'Closed World Assumptions':

- Students are not PG unless you know otherwise.
- Students are not BS unless you know otherwise.

Demonstrate how your formulation works by computing the answer sets for test cases such as the following (and others):

- Alan, a student;
- Bill, a PG student;
- Colin, a BS student but not a PG student.

Suppose Derek is either a PG student, or a BS student who is not a PG student. (We don't know which?) What conclusion does your program reach in this case?

Let's add some further levels of exceptions:

- American BS students are typically not PG, and also typically not lazy.

Try out some more test cases.

491 Knowledge Representation

Tutorial Exercise

Extended Logic Programs
SOLUTIONS

The `clasp/clingo` source files for the examples in this sheet can be found in `/vol/lab/CLASP/TutorialExamples`.

Question 1 The examples are:

- birds, normal defaults and with exceptions
- examinations
- football supporters
- football supporters, modified

You can easily compute answer sets for representative combinations of test cases using `clasp/clingo`. There is no point repeating all the outputs here. Computations using 'splitting sets' are also easy. Here are some examples. (I am only showing them to show they are all routine.)

birds + ostriches

```
can_fly(frank) ← bird(frank), not ¬can_fly(frank)
¬can_fly(frank) ← ostrich(frank), not can_fly(frank)
```

```
bird(frank) ← ostrich(frank)        { ostrich(frank), bird(frank) }
ostrich(frank)
```

The answer set of the bottom part is obvious. Simplifying the top-part leaves

```
can_fly(frank) ← not ¬can_fly(frank)
¬can_fly(frank) ← not can_fly(frank)
```

which has two answer sets.

For the version with priorites/exceptions:

```
can_fly(frank) ← bird(frank), not applies(d2,frank), not ¬can_fly(frank)
¬can_fly(frank) ← ostrich(frank), not can_fly(frank)
applies(d2,frank) ← ostrich(frank)
```

```
bird(frank) ← ostrich(frank)        { ostrich(frank), bird(frank) }
ostrich(frank)
```

(I could also have put `applies(d2,frank)` in the splitting set. It would be quicker.)

The answer set of the bottom part is obvious. Simplifying the top-part leaves


```
can_fly(frank) ← not applies(d2,frank)
¬can_fly(frank) ← not can_fly(frank)
```

```
applies(d2,frank)        { applies(d2,frank) }
```

which gives one answer set: { `applies(d2,frank)`, `¬can_fly(frank)` }.

Question 2

```
punish(X) ← thief(X), not except_r2(X)
except_r2(X) ← thief(X), minor(X), not except_r3(X)
¬punish(X) ← thief(X), minor(X), not except_r3(X)
rehab(X) ← thief(X), minor(X), not except_r3(X)
except_r3(X) ← thief(X), violent(X)
punish(X) ← thief(X), violent(X)
```

Here I have used predicates `except_r2(X)` and `except_r3(X)` because that's what I used in the earlier exercise sheet on normal logic programs. Obviously I could have used `except(r2,X)` and `except(r3,X)` instead, or `applies(r2,X)` and `applies(r3,X)`.

PS: I asked why the clauses in this example look a bit different from those in Question 1 examples. Why does the clause for (r1) not require a condition `not ¬punish(X)`?

It is because rules (r1)–(r3) are not *defaults*. They don't say 'typically'. There aren't any implicit exceptions besides those mentioned explicitly in the rule statments.

We can compute answer sets for various combinations of data (thief, minor, violent) either by using splitting sets or, in this case, by stratifying the program and using the ABW construction. Or use `clasp/clingp`. The source file is in the `TutorialExamples` directory.

Here it is stratified, with some data:

```
punish(X) ← thief(X), not except_r2(X)                 P3
punish(X) ← thief(X), violent(X)
.....................................................
except_r2(X) ← thief(X), minor(X), not except_r3(X)    P2
¬punish(X) ← thief(X), minor(X), not except_r3(X)
rehab(X) ← thief(X), minor(X), not except_r3(X)
.....................................................
except_r3(X) ← thief(X), violent(X)                    P1
.....................................................
thief(a)                                               P0
thief(b)     minor(b)
thief(c)     violent(c)
thief(d)     minor(d)     violent(d)
```

(There are other possible stratifications.)

If you are uncomfortable about stratifying an extended logic program, translate it to a normal logic program by replacing ¬punish(x) by punish*(x) and then stratify that.

$$
\begin{array}{lcll}
M_0 & = & T'_{P_0} \uparrow^{\omega} (\emptyset) & \\
 & = & \{\texttt{thief(a)}, \texttt{thief(b)}, \texttt{thief(c)}, \texttt{thief(d)}, & \\
 & & \quad \texttt{minor(b)}, \texttt{minor(d)}, \texttt{violent(c)}, \texttt{violent(d)}\} & \\
M_1 & = & T'_{P_1} \uparrow^{\omega} (M_0) & = \quad M_0 \cup \{\texttt{except\_r3(c)}, \texttt{except\_r3(d)}\} \\
M_2 & = & T'_{P_2} \uparrow^{\omega} (M_1) & = \quad M_1 \cup \{\texttt{except\_r2(b)}, \neg\texttt{punish(b)}, \texttt{rehab(b)}\} \\
M_3 & = & T'_{P_3} \uparrow^{\omega} (M_2) & = \quad M_2 \cup \{\texttt{punish(a)}, \texttt{punish(c)}, \texttt{punish(d)}\}
\end{array}
$$

Here it is using splitting sets (and each of a,b,c,d separately).

```
punish(a) ← thief(a), not except_r2(a)
punish(a) ← thief(a), violent(a)
except_r2(a) ← thief(a), minor(a), not except_r3(a)
¬punish(a) ← thief(a), minor(a), not except_r3(a)
rehab(a) ← thief(a), minor(a), not except_r3(a)
except_r3(a) ← thief(a), violent(a)
-------------------------------------------------------
thief(a)          { thief(a), minor(a), violent(a) }
```

The answer set of the bottom part is obviously {thief(a)}.
Simplifying the top part gives:

```
punish(a) ← not except_r2(a)
```

The answer set of this is {punish(a)}.

For the instance X = b:

```
punish(b) ← thief(b), not except_r2(b)
punish(b) ← thief(b), violent(b)
except_r2(b) ← thief(b), minor(b), not except_r3(b)
¬punish(b) ← thief(b), minor(b), not except_r3(b)
rehab(b) ← thief(b), minor(b), not except_r3(b)
except_r3(b) ← thief(b), violent(b)
-------------------------------------------------------
thief(b)          { thief(b), minor(b), violent(b) }
minor(b)
```

The answer set of the bottom part is obviously {thief(b), minor(b)}.
Simplifying the top part and splitting again gives:

```
punish(b) ← not except_r2(b)
except_r2(b) ← not except_r3(b)
-------------------------------------------------------
¬punish(b) ← not except_r3(b)      { except_r3(b), ¬punish(b), rehab(b) }
rehab(b) ← not except_r3(b)
```

The answer set of this bottom part is {¬punish(b), rehab(b)}.

Simplifying the top part:

```
punish(b) ← not except_r2(b)
except_r2(b)
```

This has one answer set: {except_r2(b)}.

For the instance X = c:

```
punish(c) ← thief(c), not except_r2(c)
punish(c) ← thief(c), violent(c)
except_r2(c) ← thief(c), minor(c), not except_r3(c)
¬punish(c) ← thief(c), minor(c), not except_r3(c)
rehab(c) ← thief(c), minor(c), not except_r3(c)
except_r3(c) ← thief(c), violent(c)
-------------------------------------------------------
thief(c)          { thief(c), minor(c), violent(c) }
violent(c)
```

The answer set of the bottom part is obviously {thief(c), violent(c)}.
Simplifying the top part:

```
punish(c) ← not except_r2(c)
punish(c)
except_r3(c)
```

This has one answer set: {punish(c), except_r3(c)}.

For the instance X = d:

```
punish(d) ← thief(d), not except_r2(d)
punish(d) ← thief(d), violent(d)
except_r2(d) ← thief(d), minor(d), not except_r3(d)
¬punish(d) ← thief(d), minor(d), not except_r3(d)
rehab(d) ← thief(d), minor(d), not except_r3(d)
except_r3(d) ← thief(d), violent(d)
-------------------------------------------------------
thief(d)          { thief(d), minor(d), violent(d) }
minor(d)
violent(d)
```

The answer set of the bottom part is obviously {thief(d), minor(d), violent(d)}.
Simplifying the top part and splitting again gives:

```
punish(d) ← not except_r2(d)
punish(d) ←
except_r2(d) ← not except_r3(d)
¬punish(d) ← not except_r3(d)
rehab(d) ← not except_r3(d)
-------------------------------------------------------
except_r3(d) ←          { except_r3(d) }
```

The answer set of this bottom part is obviously {except_r3(d)}.

Simplifying the top part:

```
punish(d) ← not except_r2(d)
punish(d) ←
```

This has one answer set: {punish(d)}.

'Closed World Assumptions':

```
¬thief(X) ← not thief(X)
¬minor(X) ← not minor(X)
¬violent(X) ← not violent(X)
```

These add to the answer set the literals: ¬minor(a), ¬violent(a), ¬violent(b), ¬minor(c).
They don't affect punish, ¬punish, rehab ...etc., because the literals ¬thief(X), ¬minor(X), ¬violent(X) don't appear in the body of any of the clauses above.

Question 3

```
strong(mary) ← big(mary), not ¬strong(mary)
strong(mary) ← small(mary), muscular(mary), not ¬strong(mary)
¬strong(mary) ← small(mary), not strong(mary)
-------------------------------------------------------------
big(mary)            { big(mary), small(mary), muscular(mary) }
```

The answer set of the bottom part is {big(mary)}.
Simplifying the top part:

```
strong(mary) ← not ¬strong(mary)
```

This has just one answer set: {strong(mary)}.

For the case of frank:

```
strong(frank) ← big(frank), not ¬strong(frank)
strong(frank) ← small(frank), muscular(frank), not ¬strong(frank)
¬strong(frank) ← small(frank), not strong(frank)
-------------------------------------------------------------
small(frank)           { big(frank), small(frank), muscular(frank) }
muscular(frank)
```

The answer set of the bottom part is {small(frank),muscular(frank)}.
Simplifying the top part:

```
strong(frank) ← not ¬strong(frank)
¬strong(frank) ← not strong(frank)
```

This has two answer sets: {strong(frank)} and {¬strong(frank)}.

So there are two answer sets altogether:
{ big(mary), small(frank), muscular(frank), strong(mary), strong(frank) }
{ big(mary), small(frank), muscular(frank), strong(mary), ¬strong(frank) }

Question 4

```
%* ***********
% First, write them all as normal defaults

d1.  strong(X) :- big(X), not -strong(X).
d2.  strong(X) :- small(X), muscular(X), not -strong(X).

d3.  -strong(X) :- small(X), not strong(X).

% Intended priorities: d2 > d3
% It does not matter about ordering d3 and d1 since we assume
% no-one is ever big and small at the same time.
% (We can add that as a constraint)

% Then we will extend with

d4. -strong(X) :- old(X), not strong(X).

% That will add a priority d4 > [d1,d2]. We still have d2 > d3

% We might as well incorporate d4 straight away.
************** *%

% rule d1.    d4 > d1
strong(X) :- applies(d1,X), not -strong(X).
applies(d1,X) :- big(X), not applies(d4,X).

% rule d2.    d4 > d2
strong(X) :- applies(d2,X), not -strong(X).
applies(d2,X) :- small(X), muscular(X), not applies(d4,X).

% rule d3.    d2 > d3
-strong(X) :- applies(d3,X), not strong(X).
applies(d3,X) :- small(X), not applies(d2,X).

% rule d4.
-strong(X) :- applies(d4,X), not strong(X).
applies(d4,X) :- old(X).

% --------data

% constraint for safety
:- big(X), small(X).

big(mary).
small(frank).    muscular(frank).
big(bill).      -strong(bill).
muscular(alice).
```

`clasp/clingo` computes one answer set, as expected:

```
big(mary).                          applies(d1,mary)     strong(mary).
small(frank).    muscular(frank).   applies(d2,frank).   strong(frank).
big(bill).                          applies(d1,bill).   -strong(bill).
muscular(alice)
```

There is nothing about `strong(alice)` because she is neither big nor small.

I tried some other combinations of test cases to check the answers were as expected.

{`big(mary), old(mary)`} produces one answer set:

```
big(mary).  old(mary).  applies(d4,mary).  -strong(mary).
```

Adding `strong(mary)` gives

```
big(mary).  old(mary).  applies(d4,mary).  strong(mary).
```

{`small(frank), muscular(frank), old(frank)`} produces one answer set:

```
small(frank).  muscular(frank).  old(frank).
applies(d3,frank).  applies(d4,frank).  -strong(frank).
```

Adding `strong(frank)` gives

```
small(frank).  muscular(frank).  old(frank).
applies(d3,frank). applies(d4,frank).  strong(frank).
```

{`muscular(alice), old(alice)`} produces one answer set:

```
muscular(alice).  old(alice).  applies(d4,alice).  -strong(alice).
```

Adding `strong(alice)` gives

```
muscular(alice).  old(alice).  applies(d4,alice).  strong(alice).
```

All this as expected.

Question 5 Now we have some additional background knowledge

```
-weak(X) :- strong(X).
-strong(X) :- weak(X).
```

This does not mean that everyone is either weak or strong. It only says that strong implies not weak, and that weak implies not strong (or in other words, that no-one is both strong and weak at the same time). It could be that for any given `X`. `X` is neither strong nor weak.

The problem is that now we can't tell which default rules (potentially) conflict just by looking for clauses with complementary heads. We have to know — or be able to infer — that `strong(X)` and `weak(X)` conflict, since `strong(X)` implies ¬`weak(X)` and `weak(X)` implies ¬`strong(X)`.

With that adjustment the required modification is very easy. (It is not so easy in more complicated examples.)

```
% rule d1.   d4 > d1
strong(X) :- applies(d1,X), not -strong(X).
applies(d1,X) :- big(X), not applies(d4,X).

% rule d2.   d4 > d2
strong(X) :- applies(d2,X), not -strong(X).
applies(d2,X) :- small(X), muscular(X), not applies(d4,X).

% rule d3.   d2 > d3
weak(X) :- applies(d3,X), not -weak(X).
applies(d3,X) :- small(X), not applies(d2,X).

% rule d4.
weak(X) :- applies(d4,X), not -weak(X).
applies(d4,X) :- old(X).
```

We still want

```
% constraint for safety
:- big(X), small(X).
```

I tried it out on various combinations of test data using `clasp/clingo`. As expected I got the same answers as in the previous question (except that in addition we get `-weak(mary)` as well as `strong(mary)`, `-weak(frank)` as well as `strong(frank)`, `-strong(bill)` as well as `weak(bill)`, and so on).

Question 6 Why exactly one answer set? Because the program can be *stratified*:

```
s(x) ← l(x), not d(x)      P4
.....................
d(x) ← t(x,y), not h(x)    P3
.....................
h(x) ← l(x), not ¬h(x)     P2
.....................
¬h(x) ← t(x,y), ¬h(y)      P1
¬h(a)
.....................
l(a)                       P0
l(b)
t(a,b)
t(a,c)
```

(We could also combine P_0 and P_1 above into one stratum.)

If you are uncomfortable about stratifying an extended logic program, translate it to a normal logic program by replacing `¬h(x)` by `h*(x)` and then stratify that.

Standard result: For a stratified logic program, the unique stable model (answer set) coincides with the ABW fixpoint model, which is unique.

Here's the calculation:

$$\begin{array}{rcccl} M_0 & = & T'_{P_0}\uparrow^{\omega}(\emptyset) & = & \{\texttt{l(a)},\texttt{l(b)},\texttt{t(a,b)},\texttt{t(a,c)}\} \\ M_1 & = & T'_{P_1}\uparrow^{\omega}(M_0) & = & M_0 \cup \{\neg\texttt{h(a)}\} \\ M_2 & = & T'_{P_2}\uparrow^{\omega}(M_1) & = & M_1 \cup \{\texttt{h(b)}\} \\ M_3 & = & T'_{P_3}\uparrow^{\omega}(M_2) & = & M_2 \cup \{\texttt{d(a)}\} \\ M_4 & = & T'_{P_4}\uparrow^{\omega}(M_3) & = & M_3 \cup \{\texttt{s(b)}\} \end{array}$$

Answers to queries:

- `h(a)?` No (because $\neg\texttt{h(a)} \in M_4$)
- `h(b)?` Yes (because $\texttt{h(b)} \in M_4$)
- `h(c)?` Don't know (because $\texttt{h(c)} \notin M_4$ and $\neg\texttt{h(c)} \notin M_4$)

(Of course you could also use 'splitting sets'.)

Have a look at the `gringo -t` or `clingo -t` output for this example.

Question 7

```
%* ***********
% First, write them all as normal defaults

d1.  lazy(X) :- student(X), not -lazy(X).

d2.  poor(X) :- student(X), not -poor(X).

d3.  -lazy(X) :- pg_student(X), not lazy(X).

d4.  pg_student(X) :- bs_student(X), not -pg_student(X).

d5.  -poor(X) :- bs_student(X), not poor(X).

% intended priorities: d5 > d2 ; d3 > d1

% Now the CWA assumptions

cwa1.  -pg_student(X) :- student(X), not pg_student(X).

cwa2.  -bs_student(X) :- student(X), not bs_student(X).

% cwa1 and d4 conflict: d4 > cwa1

********** *%


% rule d1.  d3 > d1
lazy(X) :- applies(d1,X), not -lazy(X).
applies(d1,X) :- student(X), not applies(d3,X).

% rule d2.  d5 > d2
poor(X) :- applies(d2,X), not -poor(X).
applies(d2,X) :- student(X), not applies(d5,X).

% rule d3.  d3 > d1
-lazy(X) :-applies(d3,X), not lazy(X).
applies(d3,X) :- pg_student(X).

% rule d4.  d4 > cwa1
pg_student(X) :- applies(d4,X), not -pg_student(X).
applies(d4,X) :- bs_student(X).

% rule d5.  d5 > d2
-poor(X) :- applies(d5,X), not poor(X).
applies(d5,X) :- bs_student(X).
```

```
% rule cwa1.  d4 > cwa1
-pg_student(X) :- applies(cwa1,X), not pg_student(X).
applies(cwa1,X) :- student(X), not applies(d4,X).

% rule cwa2.
-bs_student(X) :- applies(cwa2,X), not bs_student(X).
applies(cwa2,X) :- student(X).


% --- background knowledge

student(X) :- pg_student(X).
student(X) :- bs_student(X).

% --------data

student(alan).

pg_student(bill).

bs_student(colin).  -pg_student(colin).
```

(Thanks to Sim Virdi, MSc 2017, for spotting a typo.)

`clasp/clingo` computes one answer set (`applies` facts not shown):

```
student(alan). -pg_student(alan). -bs_student(alan). lazy(alan). poor(alan).
pg_student(bill). student(bill). -bs_student(bill).  -lazy(bill). poor(bill).
bs_student(colin). student(colin). -pg_student(colin).  lazy(colin). -poor(colin).
```

You can also try adding other facts: `-poor(alan)`, `lazy(bill)`, `bs_student(bill)`, etc, etc, in various combinations.

What about Derek? Derek is either a PG student, or a BS student who is not a PG student.

First, note: if it were 'Derek is either a PG student or a BS student' that would be easy. We can represent '`pg` or `bs`' as the pair of rules {`pg:- not bs,  bs:- not pg`}. So we could add

```
pg_student(derek) :- not bs_student(derek).
bs_student(derek) :- not pg_student(derek).
```

Let's try it. `clasp/clingo` computes *one* answer set (`applies` facts not shown):

```
pg_student(derek). student(derek). -bs_student(derek). -lazy(derek). poor(derek).
```


Surprised? What happened to the case where Derek is a BS student?

Perhaps the above is not so surprising given how we formulated it. But here is something a bit more surprising. (At least, I found it surprising.)

What we really wanted to say was 'Derek is either a PG student, or a BS student who is not a PG student', that is, something of the form:

$$\mathtt{pg} \vee (\,\mathtt{bs} \wedge \neg \mathtt{pg}\,)$$

Note that classically:

$$\begin{aligned}\mathtt{pg} \vee (\,\mathtt{bs} \wedge \neg \mathtt{pg}\,) &\leftrightarrow (\,\mathtt{pg} \vee \mathtt{bs}\,) \wedge (\,\mathtt{pg} \vee \neg \mathtt{pg}\,)\\ &\leftrightarrow (\,\mathtt{pg} \vee \mathtt{bs}\,)\end{aligned}$$

But (`pg` ∨ `bs`) is what we just tried, and it gave only one answer set!

However, we know that 'or' represented as a pair of rules using negation-as-failure is not necessarily going to behave exactly like classical disjunction. So let's try representing 'Derek is either a PG student, or a BS student who is not a PG student' like this

```
pg :- not (bs & -pg).
(bs & -pg) :- not pg.
```

This is not legal syntax though. So let's rewrite them. The first rule becomes:

```
pg :- not bs_and_not_pg.
bs_and_not_pg :- bs, -pg.
```

The second rule becomes:

```
bs :- not pg.
-pg :- not pg,
```

So finally, Derek could be represented like this:

```
pg_student(derek) :- not bs_but_not_pg(derek).

bs_but_not_pg(derek) :- bs_student(derek), -pg_student(derek).

bs_student(derek) :- not pg_student(derek).

-pg_student(derek) :- not pg_student(derek).
```

Now what do we get? `clasp/clingo` computes *two* answer sets (`applies` facts not shown):

```
bs_student(derek). student(derek). -pg_student(derek). bs_but_not_pg(derek).
lazy(derek). -poor(derek).

pg_student(derek). student(derek). -bs_student(derek).
-lazy(derek). poor(derek).
```

That's what I expected to get.

American BS students

```
%* ***********
% First, write them all as normal defaults

d1.  lazy(X) :- student(X), not -lazy(X).

d2.  poor(X) :- student(X), not -poor(X).

d3.  -lazy(X) :- pg_student(X), not lazy(X).

d4.  pg_student(X) :- bs_student(X), not -pg_student(X).

d5.  -poor(X) :- bs_student(X), not poor(X).

% intended priorities: d5 > d2 ; d3 > d1

% Now the CWA assumptions

cwa1.  -pg_student(X) :- student(X), not pg_student(X).

cwa2.  -bs_student(X) :- student(X), not bs_student(X).

% cwa1 and d4 conflict: d4 > cwa1

% Now we add the American BS students

am1. -pg_student(X) :- bs_student(X), american(X), not pg_student(X).

am2. -lazy(X) :- bs_student(X), american(X), not lazy(X).

% am1 and d4 conflict: am1 > d4

% am2 and d1 conflict: am2 > d1

********** *%


% rule d1.  d3 > d1, am2 > d1
lazy(X) :- applies(d1,X), not -lazy(X).
applies(d1,X) :- student(X), not applies(d3,X), not applies(am2,X).

% rule d2.  d5 > d2
poor(X) :- applies(d2,X), not -poor(X).
applies(d2,X) :- student(X), not applies(d5,X).
```



```
% rule d3.  d3 > d1
-lazy(X) :-applies(d3,X), not lazy(X).
applies(d3,X) :- pg_student(X).

% rule d4.  d4 > cwa1  but am1 > d4
pg_student(X) :- applies(d4,X), not -pg_student(X).
applies(d4,X) :- bs_student(X), not applies(am1,X).

% rule d5.  d5 > d2
-poor(X) :- applies(d5,X), not poor(X).
applies(d5,X) :- bs_student(X).

% rule cwa1.  d4 > cwa1
-pg_student(X) :- applies(cwa1,X), not pg_student(X).
applies(cwa1,X) :- student(X), not applies(d4,X).

% rule cwa2.
-bs_student(X) :- applies(cwa2,X), not bs_student(X).
applies(cwa2,X) :- student(X).

% rule am1. am1 > d4
-pg_student(X) :- applies(am1,X), not pg_student(X).
applies(am1,X) :- bs_student(X), american(X).

% rule am2. am2 > d1
-lazy(X) :- applies(am2,X),  not lazy(X).
applies(am2,X) :- bs_student(X), american(X).

% --- background knowledge

student(X) :- pg_student(X).
student(X) :- bs_student(X).

% --------data

american(amy).
bs_student(amy).
```

This behaved exactly as expected. (Amy is not a PG, not lazy, not poor.)