# Extended Logic Programs

Marek Sergot
Department of Computing
Imperial College, London

February 2014 (v1.6b)

Extended logic programs (Gelfond & Lifschitz) combine negation-by-failure not with classical truth-functional ('strong', 'explicit') negation ¬. The semantics is given in terms of *answer sets* — a generalisation of the definition of *stable model*.

## Syntax

A *normal logic program* (sometimes a 'general logic program') is a set of clauses of the form:

$$A \;\leftarrow\; B_1, \ldots, B_m \qquad (m \geq 0)$$

where $A$ is an *atom* and every $B_i$ is an atom or of the form not $A_i$ where $A_i$ is an atom. not denotes negation by failure. Henceforth we'll call an expression of the form not $A_i$ a *nbf-literal.*

From now on a *literal* is an expression of the form $A$ or $\neg A$ where $A$ is an atom.
An *extended logic program* is a set of clauses of the form

$$L \;\leftarrow\; L_1, \ldots, L_m \qquad (m \geq 0)$$

where $L$ is a *literal* and every $L_i$ is either a *literal* or a nbf-literal of the form not $L_i'$ where $L_i'$ is a literal.

The literal $L$ is the *head* of the clause; $L_1, \ldots, L_m$ is the *body* of the clause. When the body is empty ($n = 0$ above) the arrow $\leftarrow$ is usually omitted.
As usual we'll write $head(r)$ for the head of a clause $r$ (except now it can be a literal not just an atom), $body^-(r)$ for the set of literals $L_i$ where not $L_i$ is in the body of $r$, and $body^+(r)$ for the literals in the body of $r$ that are not nbf-literals. ($body^+(r)$ and $body^-(r)$ might not be disjoint.)

For example, if $r$ is the clause:

$$\neg p \leftarrow p,\ \text{not}\, \neg s,\ \text{not}\, q, \neg t$$

then $head(r) = \neg p$, $body^+(r) = \{p, \neg t\}$ and $body^-(r) = \{\neg s, q\}$.

Clearly extended logic programs contain normal logic programs (and definite logic programs) as a special case.

## Simple story (basic idea)

Treat literals $\neg A$ as if they were *atoms.* Reducts, answer sets, splitting sets, etc, then work exactly as usual, except that an answer set will be a set of *literals* not a set of atoms as in the case of normal logic programs. There will have to be some refinements to deal with the case that an answer set contains both $A$ and $\neg A$ but we will leave that aside for a minute.

**Example** (Basic idea: treat all literals $\neg A$ as if they were atoms)

$$\begin{aligned} p(X) &\leftarrow\; q(X),\ \text{not}\ \neg p(X) \\ \neg p(X) &\leftarrow\; r(X),\ \text{not}\ p(X) \end{aligned}$$

(Cf. the example of the 'Nixon diamond': $p$ is 'pacifist', $q$ is 'Quaker', $r$ is 'Republican'.)

Suppose first we add $\{q(a) \leftarrow\}$.
Take splitting set $\{q(a), \neg q(a), r(a), \neg r(a)\}$. Simplifying leaves one clause

$$\{p(a) \leftarrow\; \text{not}\ \neg p(a)\}$$

Answer set is $\{p(a)\} \cup \{q(a)\}$.

Now suppose we add $\{q(a) \leftarrow\ ;\ \neg p(a) \leftarrow\}$.
Take splitting set $\{q(a), \neg q(a), r(a), \neg r(a)\}$. Simplifying leaves two clauses

$$\{p(a) \leftarrow\; \text{not}\ \neg p(a)\ ;\ \neg p(a) \leftarrow\}$$

Answer set is $\{\neg p(a)\} \cup \{q(a)\}$.

Suppose we add $\{r(b) \leftarrow\}$.
Take splitting set $\{q(b), \neg q(b), r(b), \neg r(b)\}$. Simplifying leaves one clause

$$\{\neg p(b) \leftarrow\; \text{not}\ p(b)\}$$

Answer set is $\{\neg p(b)\} \cup \{r(b)\}$.

Now suppose we add $\{q(c) \leftarrow\ ;\ r(c) \leftarrow\}$.
Take splitting set $\{q(c), \neg q(c), r(c), \neg r(c)\}$. Simplifying leaves two clauses

$$\{p(c) \leftarrow\; \text{not}\ \neg p(c)\ ;\ \neg p(c) \leftarrow \text{not}\ p(c)\}$$

So *two* answer sets: $\{p(c)\} \cup \{q(c), r(c)\}$ and $\{\neg p(c)\} \cup \{q(c), r(c)\}$
(and no way to choose between them).

## Semantics — Answer sets

For an extended logic program $P$, the set of literals in the language of $P$ is denoted $Lit(P)$ (or just $Lit$ when $P$ is obvious from context).

The literals $A$ and $\neg A$ (any atom $A$) are said to be *complementary*.

For shorthand, when $L$ is a literal (i.e., of the form $A$ or $\neg A$), $\overline{L}$ denotes the literal complementary to $L$: when $L = A$ then $\overline{L} = \neg A$; when $L = \neg A$ then $\overline{L} = A$.

For a normal logic program, a stable model is a set of *atoms*.
For extended logic programs, an *answer set* is a set of *literals*.
As in the case of stable models for normal logic programs, answer sets for an extended logic program $P$ will be defined in terms of a reduct.

For an answer set $S$ (a set of literals):

- the atom $A$ is true in $S$ if $A \in S$;
- the nbf-literal not $A$ is true in $S$ if $A \notin S$;
- the literal $\neg A$ is true in $S$ if $\neg A \in S$;
- the nbf-literal not $\neg A$ is true in $S$ if $\neg A \notin S$.

$T_P$ is defined as usual except now it maps sets of *literals* to sets of *literals*.

$$T_p(X) \stackrel{\text{def}}{=} \{\, head(r) \mid r \text{ is a ground instance of a clause in } P, \; body^+(r) \subseteq X,\ body^-(r) \cap X = \emptyset \,\}$$

## Why 'answer set' and not 'stable model'?

Because, for an extended logic program $P$, an answer set of $P$ is not necessarily a model of $P$.

$p(a) \leftarrow$ `not` $\neg q(a)$
$\neg q(b)$

$\{p(a), \neg q(b)\}$ is the answer set. But it's not a model. Why? Because it's not an *interpretation* – the truth values of $p(b)$, $q(a)$ are not specified.

(You could think of $\{p(a), \neg q(b)\}$ as representing a *three-valued* interpretation

$$\{p(a) \mapsto \mathtt{t},\ p(b) \mapsto \mathtt{u},\ q(a) \mapsto \mathtt{u},\ q(b) \mapsto \mathtt{f}\}$$

but I won't pursue three-valued interpretations/models in these notes.)

## Answer sets of an extended logic program $P$

First, consider programs $P$ **without any occurrences** of negation-by-failure not.

The *answer set* of a program $P$ without any occurrences of negation-by-failure not is the smallest (set inclusion) subset $S$ of $Lit(P)$ such that:

- $S$ is *closed* under the rules of $P$: for any clause $L \leftarrow L_1, \ldots, L_n$ of $P$, if $\{L_1, \ldots, L_n\} \subseteq S$ then $L \in S$, i.e. $T_P(S) \subseteq S$;
- if $S$ contains a pair of complementary literals $A$ and $\neg A$, then $S = Lit(P)$.

How do we know such a set $S$ exists and/or is unique? Knaster-Tarski theorem (remember?). We will check this claim later in the course.

Notice:

- There is only one answer set containing complementary literals: it is (by definition) $Lit(P)$. (But see comments later on 'irritating point of detail'. We are really only interested in consistent answer sets.)
- Every program $P$ without any negation-by-failure not has a *unique answer set* which will be denoted $\mathrm{M}(P)$. (Though it may be inconsistent, i.e. $Lit(P)$.)
- For a program $P$ without negation-by-failure not and without any 'explicit' negation $\neg$ (i.e., a definite clause program), the complementary literal condition does not apply, and $\mathrm{M}(P)$ is the *least Herbrand model* of $P$.

Now the general case.

Let $P$ be an extended logic program without variables. (i.e., form all ground instances of a program with variables).
For any set $S$ of literals, $P^S$ (the *reduct*) is the logic program obtained from $P$ by deleting:

- each clause that has a condition not $L$ in its body where $L \in S$;
- each condition of the form not $L$ in the bodies of the remaining clauses.

$P^S$ is obviously a logic program with no occurrences of negation-by-failure not.

Clearly this is just the generalisation of reduct as in the definition of stable models, but now with literals and not just atoms.

The *answer sets* of $P$ are those satisfying the equation:

$$S = \mathrm{M}(P^S)$$

So this is exactly as for stable models for normal logic programs, but generalising to deal with literals in clauses and answer sets and not just atoms.
(And remember: $P^S$ has no occurrences of negation-as-failure not; but it may have occurrences of $\neg$. $\mathrm{M}(P^S)$ does not denote the least Herbrand model of $P^S$. It is the least Herbrand model of $P^S$ when $P^S$ contains no occurrences of $\neg$.)

**Example** One version of the birds-ostriches can fly example. (We'll consider some other versions in a minute.)

- Typically, a bird can fly.
- Except that ostriches, which are birds, cannot fly.
- And wounded birds cannot fly.

Here is one possible formulation as an extended logic program.

```
can_fly(X) ← bird(X), not abnormal_bird(X)
bird(X) ← ostrich(X)
abnormal_bird(X) ← ostrich(X)
¬can_fly(X) ← ostrich(X)
abnormal_bird(X) ← bird(X), wounded(X)
¬can_fly(X) ← wounded(X)
```

And some facts:

```
bird(arthur).       ¬wounded(arthur).
ostrich(bill).
bird(colin).        wounded(colin).
¬ostrich(dave).     wounded(dave).
```

If we construct the answer sets we get just one: the facts above plus {`bird(bill)`} plus

```
{ can_fly(arthur), abnormal_bird(bill), abnormal_bird(colin),
    ¬can_fly(bill), ¬can_fly(colin), ¬can_fly(dave) }
```

In summary we can interpret the answer set like this:

| | bird | ostrich | wounded | abnormal_bird | can_fly |
|---|---|---|---|---|---|
| arthur | yes | unknown | no | unknown | yes |
| bill | yes | yes | unknown | yes | no |
| colin | yes | unknown | yes | yes | no |
| dave | unknown | no | yes | unknown | no |

So in this version:

| | | |
|---|---|---|
| `abnormal_bird(X)` | *blocks* | `can_fly(X)` |
| `ostrich(X)` | *implies* | `¬can_fly(X)` |
| `wounded(X)` | *implies* | `¬can_fly(X)` |

**Note:** we need to say that an ostrich is an abnormal bird, because otherwise we get inconsistent answer set for any ostrich. (And likewise for wounded birds.)

(Thanks to Jinyi Shan and Nuri Cingillioglu, MEng4 2016-17, for pointing out some minor but confusing omissions in earlier versions of these notes.)

**Example** Here is a formulation in the style of previous examples:

```
can_fly(X) ← bird(X), not abnormal_bird(X)
bird(X) ← ostrich(X)
abnormal_bird(X) ← ostrich(X)
abnormal_bird(X) ← bird(X), wounded(X)
```

And some facts:

```
bird(arthur).
ostrich(bill).
bird(colin).      wounded(colin).
                  wounded(dave).
```

Viewing the above as a *normal logic program* with *stable model semantics* we get the following stable model:

```
      {  bird(arthur), can_fly(arthur),
   bird(bill), ostrich(bill), abnormal_bird(bill),
 bird(colin), wounded(colin), abnormal_bird(colin),
             wounded(dave)   }
```

This is a stable model — a set of *atoms*. Since it is a Herbrand model, we interpret it like this:

| | bird | ostrich | wounded | abnormal_bird | can_fly |
|---|---|---|---|---|---|
| arthur | yes | no | no | no | yes |
| bill | yes | yes | no | yes | no |
| colin | yes | no | yes | yes | no |
| dave | no | no | yes | no | no |

*Syntactically* a normal logic program (like the one above) is also a special case of an extended logic program. Viewing the same set of rules as an *extended logic program* with *answer set semantics*, we read

```
      {  bird(arthur), can_fly(arthur),
   bird(bill), ostrich(bill), abnormal_bird(bill),
 bird(colin), wounded(colin), abnormal_bird(colin),
             wounded(dave)   }
```

as an *answer set* — a set of literals — not a model. This answer set says:

| | bird | ostrich | wounded | abnormal_bird | can_fly |
|---|---|---|---|---|---|
| arthur | yes | unknown | unknown | unknown | yes |
| bill | yes | yes | unknown | yes | unknown |
| colin | yes | unknown | yes | yes | unknown |
| dave | unknown | unknown | yes | unknown | unknown |

To get the same answers as given by the normal logic program we need to add extra rules to express the 'Closed World Assumption' implicit in the normal logic program.

```
¬can_fly(X) ← not can_fly(X)
¬bird(X) ← not bird(X)
¬ostrich(X) ← not ostrich(X)
¬wounded(X) ← not wounded(X)
```

An extended logic program can provide much more precision. For example, here is another possible formulation as an extended logic program:

```
can_fly(X) ← bird(X), not abnormal_bird(X)
bird(X) ← ostrich(X)
abnormal_bird(X) ← ostrich(X)
¬can_fly(X) ← ostrich(X)
abnormal_bird(X) ← bird(X), wounded(X)
¬ostrich(X) ← not ostrich(X)
¬wounded(X) ← not wounded(X)
```

Facts:

```
bird(arthur).
ostrich(bill).
bird(colin).     wounded(colin).
                 wounded(dave).
```

| | bird | ostrich | wounded | abnormal_bird | can_fly |
|---|---|---|---|---|---|
| arthur | yes | no | no | unknown | yes |
| bill | yes | yes | no | yes | no |
| colin | yes | no | yes | yes | unknown |
| dave | unknown | no | yes | unknown | unknown |

The above is just one possible formulation out of many. For example, we can adjust the 'Closed World' specifications. As another illustration, here is a more cautious formulation. Remove

```
¬ostrich(X) ← not ostrich(X)
```

and replace

```
abnormal_bird(X) ← ostrich(X)
```

with

```
abnormal_bird(X) ← not ¬ostrich(X)
```

Now we can't conclude, e.g.

```
can_fly(arthur)
```

until we add explicitly

```
¬ostrich(arthur)
```

Obviously there are other possible formulations. And as the examples get more complex so the number of possible variations increases. We can say precisely what we want to say (even if we don't always know what it is we want to say).

**Example: further refinement** *Typically* ostriches do not fly. But there are exceptions ....
One possible formulation:

```
can_fly(X) ← bird(X), not abnormal_bird(X)
bird(X) ← ostrich(X)
abnormal_bird(X) ← ostrich(X), not abnormal_ostrich(X)
¬can_fly(X) ← ostrich(X), not abnormal_ostrich(X)
          ⋮
```

As written above, an abnormal ostrich can fly because an abnormal ostrich is not an abnormal bird.
Compare:

```
can_fly(X) ← bird(X), not abnormal_bird(X)
bird(X) ← ostrich(X)
abnormal_bird(X) ← ostrich(X)
¬can_fly(X) ← ostrich(X), not abnormal_ostrich(X)
can_fly(X) ← abnormal_ostrich(X)
          ⋮
```

Here, an abnormal ostrich is still an abnormal bird. It can fly, not because it is a bird, but because it is an abnormal ostrich.

Which of these formulations is better? Take your pick. (It depends on what you want to say.)

More suggestions later.

**Example**

- Europeans are typically civilised, unless they are not civilised.
- Football supporters are typically not civilised, unless they are educated.
- Even educated football supporters, on the other hand, are not civilised if they are drunk.

Here is one possible formulation as an extended logic program (out of many):

```
civil(X) ← european(X), not ¬civil(X)
¬civil(X) ← football_supporter(X), not ab_football_supporter(X)
ab_football_supporter(X) ← football_supporter(X), educated(X)
¬civil(X) ← football_supporter(X), drunk(X)
```

There are *many* other possible formulations, depending on what it is we want to say exactly. Does the one shown above work as intended? Maybe. We have to check. (Tutorial exercise.)

Here is another possible formulation (out of many):

```
civil(X) ← european(X), not ab_european_civil(X)
¬civil(X) ← football_supporter(X), ¬educated(X), not civil(X)
ab_european_civil(X) ← european(X), football_supporter(X), ¬educated(X)
¬civil(X) ← football_supporter(X), drunk(X)
```

Does *this one* work as intended? Maybe. Maybe not. (Tutorial exercise.)

What about an uneducated football supporter, not European, who we can see is civilised?

```
¬educated(pablo).      football_supporter(pablo).
¬european(pablo).      civil(pablo).
```

We should be able to add exceptions like this (it says 'typically').

That aside, what about football supporters who are not known to be educated, and not known to be not educated? We could add

```
¬educated(X) ← not educated(X)
```

or

```
educated(X) ← not ¬educated(X)
```

depending on which way round we wanted the default to work.

And similarly for drunk.

Again we see the precision available here — even if it is not always clear from the imprecise English formulation what exactly the rules are trying to say.

We will look at some more systematic ways of approaching these examples in a minute.

## Exception structures

If we have 'explicit' negation ¬, and negation by failure, why do we need all these 'abnormality predicates' as well? Because without, we don't express the exception structure properly.

d1. Typically, a bird can fly. Except that ...

d2. Typically ostriches, which are birds, cannot fly.

Consider this formulation:

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

Suppose `frank` is an ostrich. Can he fly or not?

Unknown, according to this formulation — There are two answer sets (Cf. 'Nixon diamond').

The above does not capture the exception structure properly. Rule d2 is an exception to rule d1, not the other way round.

Solution?

(1) Use some 'abnormality predicates', or something similar.

```
can_fly(X) ← bird(X), not ab_bird_fly(X)
¬can_fly(X) ← ostrich(X), not can_fly(X)
ab_bird_fly(X) ← ostrich(X)
bird(X) ← ostrich(X)
```

(2) Devise some way of specifying *priorities* — saying which default rule takes priority in case of conflicting conclusions.

Some suggestions for doing (2) by converting to (1) will come in a minute.

## Irritating point of detail – Inconsistent answer sets?

What are we going to do about programs like the following?

$$\begin{aligned} P_1 &= \{\, \neg p \leftarrow q\,; \quad p \leftarrow q\,; \quad q\,\} \\ P_2 &= \{\, \neg p \leftarrow \mathrm{not}\, q\,; \quad p \leftarrow \mathrm{not}\, q\,\} \end{aligned}$$

*In practice* we don't have them. In practice, answer set solvers automatically build in an extra constraint that ensures answer sets are consistent (don't have complementary literals).

*However*: for technical reasons it is convenient to allow inconsistent answer sets *in very special circumstances*. Here are some remarks, for the sake of completeness.
Recall that, by definition, the answer set of a program $P$ without negation-by-failure (not) is the smallest (set inclusion) subset $S$ of $\mathit{Lit}(P)$ such that:

- $S$ is *closed* under the rules of $P$: $T_P(S) \subseteq S$;
- if $S$ contains a pair of complementary literals $A$ and $\neg A$, then $S = \mathit{Lit}(P)$.

So for any logic program $P$ there are only two possibilities:

- all answer sets of $P$ are consistent (no complementary literals); or
- $X = \mathit{Lit}(P)$ is the only answer set of $P$.

This isn't *quite* as obvious as it seems.

**Example**

$$P_1 = \{\neg p \leftarrow q\,; \quad p \leftarrow q\,; \quad q\}$$

$P_1$ has no occurrences of negation-as-failure. Its answer set is $\mathrm{M}(P_1)$. Since there are complementary literals $p$ and $\neg p$ in any set $S$ closed under $T_{P_1}$, $\mathrm{M}(P_1)$ by definition is $\mathit{Lit}(P_1)$, i.e., $\{p,\ \neg p,\ q,\ \neg q\}$.

**Example**

$$P_2 = \{\neg p \leftarrow \mathrm{not}\, q\,; \quad p \leftarrow \mathrm{not}\, q\}$$

$P_2$ is different: it has occurrences of negation-by-failure. Does it have any answer sets?
Suppose that $S$ is an answer set and consider separately the two cases $q \in S$ and $q \notin S$.
(1) If $q \in S$, then the reduct $P_2^S = \emptyset$. $\mathrm{M}(\emptyset) = \emptyset$. But $q \notin \emptyset$. Contradiction.
(2) If $q \notin S$, then the reduct $P_2^S = \{p,\ \neg p\}$, and $\mathrm{M}(P_2^S) = \mathit{Lit}(P_2)$ by definition. But $q \in \mathit{Lit}(P_2)$, so $q \in \mathrm{M}(P_2^S)$, and hence $q \in S$ if $S$ is an answer set. Contradiction.
Conclusion: $P_2$ has no answer sets.

## Reduction to normal logic programs

Let $P$ be an extended logic program.

For any predicate $p$ occurring in $P$, let $p^*$ be a predicate of the same arity. $P^*$ stands for the normal logic program obtained from $P$ by replacing every occurrence of a negative literal $\neg p(X_1, \ldots, X_n)$ by the new positive literal $p^*(X_1, \ldots, X_n)$.

For any subset $S \subseteq \mathit{Lit}(P)$, $S^*$ stands for the set of atoms obtained from $S$ by replacing every negative literal $\neg p(X_1, \ldots, X_n)$ in $S$ by the positive literal $p^*(X_1, \ldots, X_n)$.

**Proposition** A consistent set $S \subset \mathit{Lit}(P)$ ($S \neq \mathit{Lit}(P)$) is an answer set of an extended logic program $P$ if and only if $S^*$ is an answer set (stable model) of the normal logic program $P^*$.

So . . . An answer set will be a set of *literals* but apart from that reducts, stratification results, splitting sets, . . . etc, work in exactly the same way but with (ground) literals instead of (ground) atoms.

**Notice:** This proposition holds when $S \subset \mathit{Lit}(P)$ is consistent (contains no complementary literals). What if $S$ does contain complementary literals? The proposition does not say. Some people find the following account helpful. (If you don't, just ignore it.) It is from notes *Answer Set Programming* by Torsten Schaub, Martin Gebser, Marius Schneider (University of Potsdam):
A set $X$ of literals is an answer set of a logic program $P$ if $X$ is an answer set of

$$P \cup \{\, B \leftarrow A, \neg A \mid A \in \mathit{Atoms}(P),\ B \in \mathit{Lit}(P)\,\}$$

If you don't find this observation helpful, just ignore it. Either way you look at it, consider the following example.

**Example** (from the notes by Torsten Schaub et al)

- $P_1 = \{\, \texttt{cross} \leftarrow \mathrm{not}\ \texttt{train}\,\}$
  - Answer set: $\{\, \texttt{cross}\,\}$
- $P_2 = \{\, \texttt{cross} \leftarrow \neg\texttt{train}\,\}$
  - Answer set: $\emptyset$
- $P_3 = \{\, \texttt{cross} \leftarrow \neg\texttt{train},\ \neg\texttt{train} \leftarrow\,\}$
  - Answer set: $\{\, \texttt{cross},\ \neg\texttt{train}\,\}$
- $P_4 = \{\, \texttt{cross} \leftarrow \neg\texttt{train},\ \neg\texttt{train} \leftarrow,\ \neg\texttt{cross} \leftarrow\,\}$
  - Answer set: $\{\, \texttt{cross},\ \neg\texttt{cross},\ \texttt{train},\ \neg\texttt{train}\,\}$
- $P_5 = \{\, \texttt{cross} \leftarrow \neg\texttt{train},\ \neg\texttt{train} \leftarrow \mathrm{not}\,\texttt{train},\ \neg\texttt{cross} \leftarrow\,\}$
  - No answer set (try $\texttt{train} \in X$ and $\texttt{train} \notin X$)

## Priorities and exceptions: sketch

Here is a general method for dealing with *exceptions*, treating them as *prioritized* defaults. The most general case is a bit more complicated than this, but here is a simple method that works well for many examples.

There is a *lot* more that could be said about this. I am afraid I don't have time.

**'Normal defaults'** For reasons that will be clear later when we look at Default Logic, I am going to use the term **normal default** for rules of the following special forms:

$$A \leftarrow L_1, \ldots, L_m, \text{ not } \neg A$$
$$\neg A \leftarrow L_1, \ldots, L_n, \text{ not } A$$

$A$ is an atom, $L_i$ are literals, and there are no other occurrences of negation-by-failure not. The terminology is from Reiter Default Logic.

## Example

$d_1$. Typically, a bird can fly.

```
can_fly(X) ← bird(X), not ¬can_fly(X)
```

$d_2$. Except that, typically, ostriches cannot fly.

```
¬can_fly(X) ← ostrich(X), not can_fly(X)
```

Background knowledge:

```
bird(X) ← ostrich(X)
```

The problem is that ostriches are birds, so for them we get conflicting defaults, and multiple answer sets.
The intention is that $d_2$ takes priority over $d_1$: $d_2 > d_1$.

Solution:

- Name the conditions under which a default applies.
- Look for conflicts (rules with conflicting heads).
- Add explicit conditions to express the priorities.

```
d1.  can_fly(X) ← applies(d1,X), not ¬can_fly(X)   % NOTE!
     applies(d1,X) ← bird(X), not applies(d2,X).

d2.  ¬can_fly(X) ← applies(d2,X), not can_fly(X)
     applies(d2,X) ← ostrich(X)

     bird(X) ← ostrich(X)
```

We could also add, say

```
¬can_fly(X) ← dead(X)     % no exceptions
```

## Example (exceptions to exceptions)

$d_1$. Examinations are typically pleasant.

```
pleasant(X) ← exam(X), not ¬pleasant(X)
```

$d_2$. Except that examinations at Imperial College are typically not pleasant.

```
¬pleasant(X) ← ic_exam(X), not pleasant(X)
```

$d_3$. Except that examinations in DoC are typically pleasant.

```
pleasant(X) ← doc_exam(X), not ¬pleasant(X)
```

Exceptions/priorities: $d_3 > d_2 > d_1$.

Add `applies`, look for conflicts, add conditions to deal with priorities:

```
d1.  pleasant(X) ← applies(d1,X), not ¬pleasant(X)
     applies(d1,X) ← exam(X), not applies(d2,X)

d2.  ¬pleasant(X) ← applies(d2,X), not pleasant(X)
     applies(d2,X) ← ic_exam(X), not applies(d3,X)

d3.  pleasant(X) ← applies(d3,X), not ¬pleasant(X)
     applies(d3,X) ← doc_exam(X)

     exam(X) ← ic_exam(X)
     ic_exam(X) ← doc_exam(X)
```

Note how the normal defaults are encoded! (It is not obvious why in this example, but it will be in more complex ones.)

Note:

- The single fact `exam(491)` gives `applies(d1,491)` and hence `pleasant(491)`.
- The single fact `ic_exam(491)` gives `applies(d2,491)` and hence `¬pleasant(491)`.
- The single fact `doc_exam(491)` gives both `applies(d1,491)` *and* `applies(d3,491)`, and hence `pleasant(491)` for two reasons. This is deliberate. You might not like it: maybe you think `d1` does not apply in this case. But that is much more complicated to encode— and as it turns out, often not what we want anyway.

## Example (football supporters)

First, express as normal defaults:

$d_1$. Europeans are typically civilised.

`civil(X) ← european(X), not ¬civil(X)`

$d_2$. Football supporters are typically not civilised, unless they are educated.

`¬civil(X) ← football_supporter(X), ¬educated(X), not civil(X)`

Side remark: 'unless' is *very* ambiguous. Here I have chosen to represent '$P$ if $Q$ unless $R$' as $P \leftarrow Q \wedge \neg R$. There are other possibilities, depending on what we think 'unless' means in this example. However, that is tangential to the point at issue (defaults and exceptions) so I will not pursue it. (See modified example later.)

$d_3$. Even educated football supporters, on the other hand, are not civilised if they are drunk. (Typically)

`¬civil(X) ← football_supporter(X), educated(X), drunk(X), not civil(X)`

Side remark: the condition `educated(X)` could be deleted but it is probably clearer to leave it.

Exceptions/priorities: $\{d_2, d_3\} > d_1$.

Encode, look for conflicts, add conditions to deal with priorities:

$d_1$. `civil(X) ← applies(d1,X), not ¬civil(X)`
`applies(d1,X) ← european(X), not applies(d2,X), not applies(d3,X)`

$d_2$. `¬civil(X) ← applies(d2,X), not civil(X)`
`applies(d2,X) ← football_supporter(X), ¬educated(X)`

$d_3$. `¬civil(X) ← applies(d3,X), not civil(X)`
`applies(d3,X) ← football_supporter(X), educated(X), drunk(X)`

Try it out on different combinations of facts. (See tutorial exercises.)

## Example (football supporters, modified)

Here is an example of a different kind of 'unless'.

$d_1$. Europeans are typically civilised.

`civil(X) ← european(X), not ¬civil(X)`

$d_2$. Football supporters are typically not civilised . . .

`¬civil(X) ← football_supporter(X), not civil(X)`

$d_2'$. . . . unless they are educated.

`civil(X) ← football_supporter(X), educated(X), not ¬civil(X)`

$d_3$. Even educated football supporters, on the other hand, are not civilised if they are drunk. (Typically)

`¬civil(X) ← football_supporter(X), educated(X), drunk(X), not civil(X)`

Exceptions/priorities: $d_3 > d_2' > d_2 > d_1$.

Encode, look for conflicts, add conditions to deal with priorities:

$d_1$. `civil(X) ← applies(d1,X), not ¬civil(X)`
`applies(d1,X) ← european(X), not applies(d2,X)`

$d_2$. `¬civil(X) ← applies(d2,X), not civil(X)`
`applies(d2,X) ← football_supporter(X), not applies(d2dash,X)`

$d_2'$. `civil(X) ← applies(d2dash,X), not ¬civil(X)`
`applies(d2dash,X) ← football_supporter(X), educated(X), not applies(d3,X)`

$d_3$. `¬civil(X) ← applies(d3,X), not civil(X)`
`applies(d3,X) ← football_supporter(X), educated(X), drunk(X)`