### 10.3 Negation as failure

One of Prolog's most useful features is the simple way it lets us state generalizations. To say that Vincent enjoys burgers we just write:

```
enjoys(vincent,X) :- burger(X).
```

But in real life rules have exceptions. Perhaps Vincent doesn't like Big Kahuna burgers. That is, perhaps the correct rule is really: Vincent enjoys burgers, *except* Big Kahuna burgers. Fine. But how do we state this in Prolog?

As a first step, let's introduce another built in predicate `fail/0`. As its name suggests, `fail` is a special symbol that will immediately fail when Prolog encounters it as a goal. That may not sound too useful, but remember: *when Prolog fails, it tries to backtrack*. Thus `fail` can be viewed as an instruction to force backtracking. And when used in combination with cut, which *blocks* backtracking, `fail` enables us to write some interesting programs, and in particular, it lets us define exceptions to general rules.

Consider the following code:

```
enjoys(vincent,X) :- big_kahuna_burger(X),!,fail.
enjoys(vincent,X) :- burger(X).

burger(X) :- big_mac(X).
burger(X) :- big_kahuna_burger(X).
burger(X) :- whopper(X).

big_mac(a).
big_kahuna_burger(b).
big_mac(c).
whopper(d).
```

The first two lines describe Vincent's preferences. The last six lines describe a world containing four burgers, `a`, `b`, `c`, and `d`. We're also given information about what kinds of burgers they are. Given that the first two lines really do describe Vincent's preferences (that is, that he likes all burgers except Big Kahuna burgers) then he should enjoy burgers `a`, `c` and `d`, but not `b`. And indeed, this is what happens:

```
?- enjoys(vincent,a).
yes

?- enjoys(vincent,b).
no

?- enjoys(vincent,c).
yes

?- enjoys(vincent,d).
yes
```

How does this work? The key is the combination of `!` and `fail` in the first line (this even has a name: its called the *cut-fail combination*). When we pose the query `enjoys(vincent,b)`, the first rule applies, and we reach the cut. This commits us to the choices we have made, and in particular, blocks access to the second rule. But then we hit `fail`. This tries to force backtracking, but the cut blocks it, and so our query fails.

This is interesting, but it's not ideal. For a start, note that the ordering of the rules is crucial: if we reverse the first two lines, we *don't* get the behavior we want. Similarly, the cut is crucial: if we remove it, the program doesn't behave in the same way (so this is a *red* cut). In short, we've got two mutually dependent clauses that make intrinsic use of the procedural aspects of Prolog. Something useful is clearly going on here, but it would be better if we could extract the useful part and package it in a more robust way.

And we can. The crucial observation is that the first clause is essentially a way of saying that Vincent does *not* enjoy X if X is a Big Kahuna burger. That is, the cut-fail combination seems to be offering us some form of negation. And indeed, this is the crucial generalization: the cut-fail combination lets us define a form of negation called *negation as failure*. Here's how:

```
neg(Goal) :- Goal,!,fail.
neg(Goal).
```

For any Prolog goal, `neg(Goal)` will succeed precisely if `Goal` does *not* succeed.

Using our new `neg` predicate, we can describe Vincent's preferences in a much clearer way:

```
enjoys(vincent,X) :- burger(X), neg(big_kahuna_burger(X)).
```

That is, Vincent enjoys X if X is a burger and X is not a Big Kahuna burger. This is quite close to our original statement: Vincent enjoys burgers, except Big Kahuna burgers.

Negation as failure is an important tool. Not only does it offer useful expressivity (notably, the ability to describe exceptions) it also offers it in a relatively safe form. By working with negation as failure (instead of with the lower level cut-fail combination) we have a better chance of avoiding the programming errors that often accompany the use of red cuts. In fact, negation as failure is so useful, that it comes built in Standard Prolog, we don't have to define it at all. In Standard Prolog the operator `\+` means negation as failure, so we could define Vincent's preferences as follows:

```
enjoys(vincent,X) :- burger(X), \+ big_kahuna_burger(X).
```

Nonetheless, a couple of words of warning are in order: *don't* make the mistake of thinking that negation as failure works just like logical negation. It doesn't. Consider again our burger world:

```
burger(X) :- big_mac(X).
burger(X) :- big_kahuna_burger(X).
burger(X) :- whopper(X).

big_mac(c).
big_kahuna_burger(b).
big_mac(c).
whopper(d).
```

If we pose the query `enjoys(vincent,X)` we get the correct sequence of responses:

```
X = a ;

X = c ;

X = d ;

no
```

But now suppose we rewrite the first line as follows:

```
enjoys(vincent,X) :- \+ big_kahuna_burger(X), burger(X).
```

Note that from a declarative point of view, this should make no difference: after all, *burger(x) and not big kahuna burger(x)* is logically equivalent to *not big kahuna burger(x) and burger(x)*. That is, no matter what the variable *x* denotes, it impossible for one of these expressions to be true, and the other expression to be false. Nonetheless, here's what happens when we pose the same query:

```
enjoys(vincent,X)

no
```

What's going on? Well, in the modified database, the first thing that Prolog has to check is whether `\+ big_kahuna_burger(X)` holds, which means that it must check whether `big_kahuna_burger(X)` fails. But this succeeds. After all, the database contains the information `big_kahuna_burger(b)`. So the query `\+ big_kahuna_burger(X)` fails, and hence the original query does too. In a nutshell, the crucial difference between the two programs is that in the original version (the one that works right) we use `\+` only *after* we have instantiated the variable `X`. In the new version (which goes wrong) we use `\+` before we have done this. The difference is crucial.

Summing up, we have seen that negation as failure is not logical negation, and that it has a procedural dimension that must be mastered. Nonetheless, it is an important programming construct: it is generally a better idea to try use negation as failure than to write code containing heavy use of red cuts. Nonetheless, ``generally" does not mean ``always". There *are* times when it is better to use red cuts.

For example, suppose that we need to write code to capture the following condition: *p holds if a and b hold, or if a does not hold and c holds too.* This can be captured with the help of negation as failure very directly:

```
p :- a,b.

p :- \+ a, c.
```

But suppose that `a` is a very complicated goal, a goal that takes a lot of time to compute. Programming it this way means we may have to compute `a` twice, and this may mean that we have unacceptably slow performance. If so, it would be better to use the following program:

```
p :- a,!,b.

p :- c.
```

Note that this is a red cut: removing it changes the meaning of the program. Do you see why?

When all's said and done, there are no universal guidelines that will cover all the situations you are likely to run across. Programming is as much an art as a science: that's what makes it so interesting. You need to know as much as possible about the language you are working with (whether it's Prolog, Java, Perl, or whatever) understand the problem you are trying to solve, and know what counts as an acceptable solution. And then: go ahead and try your best!