Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 3, Part 3: Additional Prolog Topics (cut, backtracking, not)

Multiple Solutions and Backtracking

- Many times there exist multiple problem solutions or alternative paths to be searched for a solution.
- Prolog marks potential alternatives for possible subsequent investigation.
- This leads to *backtracking*.
- Typing the ’;’ is equivalent to asking Prolog to display an alternative solution, determined by backtracking to the most recent[a] point marked for backtracking.
- Otherwise, only the *first* one found by Prolog is shown.

[a] In terms of the unification search strategy.

```
/* smpl-unify1.pro */

goal1(X,Y) :- first(X), second(Y).

goal2(X) :- first(X), second(X).

first(1).
first(2).
first(3).
second(2).
second(4).
second(6).
```

The cut

The cut (!) is used as a predicate[a] which always succeeds, but with a significant side effect: *all backtracking points (if they exist) up to the cut are erased.* Thus the cut, in a sense, forces commitment to a solution found prior to the occurrence of the cut.

[a] with a strange notation.

not, fail, true, call and the cut

Prolog provides the **not** predicate, which must be used with care.

> The **not** predicate, as would logically be expected, succeeds if unification of its argument fails[a].

[a] Note this is different from generation of an exception for an undefined predicate.

Examples using not

Now let's look at some help results and examples:

```
?- help(not).
not(+Goal)
    Succeeds  when Goal cannot  be proven.

?- true.

Yes
?- fail.

No
?- not(true).

No
?- not(fail).

Yes
```

```
?- help(call).
call(+Goal)
    Invoke Goal as a goal. Note that clauses may have variables as
    subclauses, which is identical to call/1, except when the argument
    is bound to the cut. See !/0.

Yes
?- help(!).
!
    Cut. Discard choice points of parent frame and frames created
    after the parent frame.
```

Defining `not`

Although it is built in, using the preceding predicates we may now define **`not`** as follows:

```
not(P) :- call (P), !, fail.
not (P).
```

Observe how the use of **`call`**, the cut (!), **`fail`** and Prolog's backtracking mechanism (note the order of the clauses above) allow implementation of the **`not`** predicate.

Three Faces of Prolog

Consider the Prolog clause (rule):

```
a :- b1, b2 ,..., bn
```

or the equivalent logical expression

$$b1 \cap b2 \cap \ldots bn \rightarrow a \tag{1}$$

There are multiple viewpoints of this clause:

1. Declarative viewpoint: a is true if b1 and b2 and ... bn are true
2. Procedural viewpoint: to find if a is true, determine if b1 is true and then determine if b2 is true and so on
3. Behavioral[a] viewpoint: process or goal a may be replaced by a set of processes/goals {b1,b2, ... bn}.

[a] This is key to considering parallel implementation.