1

Knowledge Based Systems

(CM0377)

Lecture 8

(Last modified 5th March 2001)

2

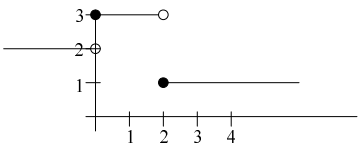
Controlling the search

- We will look at two ways of controlling the search in Prolog:
  - the *cut,* which prunes the search tree; and
  - *not,* which is implemented as *negation-as-failure*

3

The cut

- Quite often, want to 'prune' the search tree for efficiency. Consider implementation of:

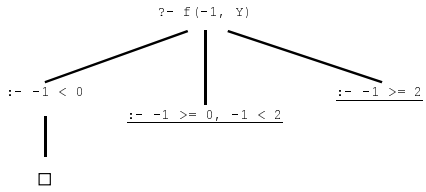


- Could write

```
f(X, 2):-X<0.
f(X, 3):-X>=0, X<2.
f(X, 1):-X>=2.
```

4

SLD tree for `?- f(-1, Y)`



5

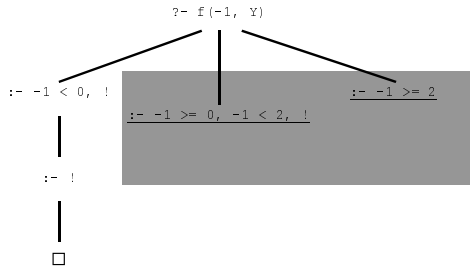
Motivation for cut

- We'll correctly get one solution in this example, but if we ask Prolog for more solutions it searches the failure nodes first.
- If we *know* that at a particular 'choice point' at most one of the branches of the subtree rooted at this node will ever be a success branch, then can use the 'cut' to cut off futile search:

```
f(X, 2):-X<0, !.
f(X, 3):-X>=0, X<2, !.
f(X, 1):-X>=2.
```

6

Revised SLD tree



7

The cut ...

- Definitions:
  - "Once you've reached me, stick with all variable substitutions you've found after you entered my clause"

  or, put another way,
  - "Don't try to find alternative solutions to literals to the left of the cut; don't try alternative clauses for the one in which the cut was encountered either"
- Note:
  - The cut, !, is **true** by definition; it succeeds, but prunes the choice point containing the head of the clause in which the cut is found.
  - Also, cut doesn't affect backtracking to the *right* of the cut.

8

Procedural/Declarative interpretation

- We could now use our superior knowledge of how Prolog will search, in order to omit bits of the above program. So the folowing program will give the same results for goals of the form `?- f(<some integer>, Y):`

```
f(X, 2):-X<0, !.
f(X, 3):-X<2, !.
f(X, 1).
```

9

Procedural/Declarative interpretation (ctd.)

- But the *procedural interpretation* is now different from the *declarative interpretation,* making the latter program very difficult to read.
- Original example was of using 'green' cuts.
- The above example employs 'red' cuts.
- Generally use red cuts sparingly!

10

Negation-as-failure

- Suppose we wished to determine whether a second-hand car was suitable for us:

```
suitable_car(X) :-
        price_of(X, P),
        affordable(P),
        manufacturer_of(X, M),
        acceptable(M),
        not(stolen(X)).
```

11

Negation (ctd.)

- In the nature of things, it will be possible in some cases to establish the car *has* been stolen, but can never be *certain* a car *hasn't* been stolen (e.g. perhaps it hasn't been reported yet).
- So logically we should be able to say 'I don't know!'.
- In practice, in Prolog, `not(stolen(X))` *succeeds* if, and only if, `stolen(X)` is *unprovable* - 'negation as failure'

12

Negation (ctd.)

- `not` is a built-in predicate defined thus:

```
not(Goal):- Goal, !, fail.
not(Goal).
```

- Note that Prolog allows us to use a term as an atom, and 'call' it. In fact, `Goal` in the above program is shorthand for `call(Goal)`.
- `call`, and `not`, are *meta-predicates:* predicates that take formulas from the same logical language in which they are written as arguments.

13

Another example

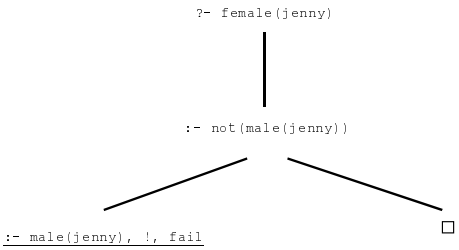
```
male(jim).
male(fred).
female(X):-not(male(X)).
```

- Consider the query:

```
?-female(jenny).
```

14

SLD tree



15

A couple of points ...

- Although *not* isn't true logical negation, we can often use it with far less damage to the meaning of a program than the cut - it's 'higher level'
- In SICSTUS Prolog, use '\+' instead of 'not'
- Recommended reading: Flach, pp. 41-58 (much more detail + examples)
- Next time - meta-programs and their role in reasoning