Logic Programming (PLP 11.3)

Prolog Imperative Control Flow: Backtracking Cut, Fail, Not

Carlos Varela

Rennselaer Polytechnic Institute

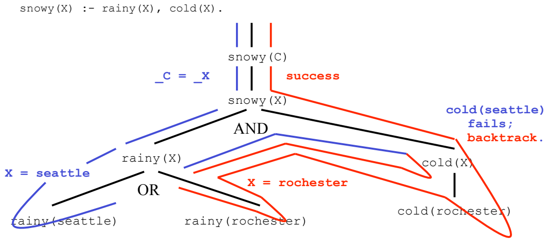
September 6, 2007

Backtracking

- *Forward chaining* goes from axioms forward into goals.
- *Backward chaining* starts from goals and works backwards to prove them with existing axioms.

Backtracking example

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```



Imperative Control Flow

- Programmer has *explicit control* on backtracking process.

Cut (!)

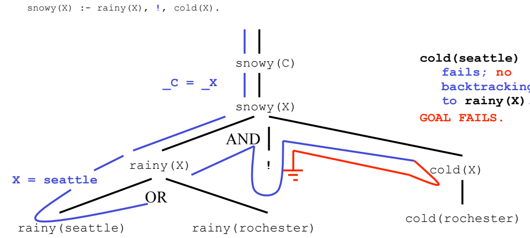
- As a goal it succeeds, but with a side effect:
  - Commits interpreter to choices made since unifying parent goal with left-hand side of current rule.

Cut (!) Example

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).
```


Cut (!) Example

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).
```

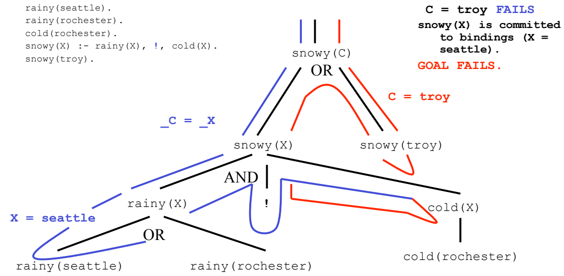


Cut (!) Example 2

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).
snowy(troy).
```


Cut (!) Example 2

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).
snowy(troy).
```

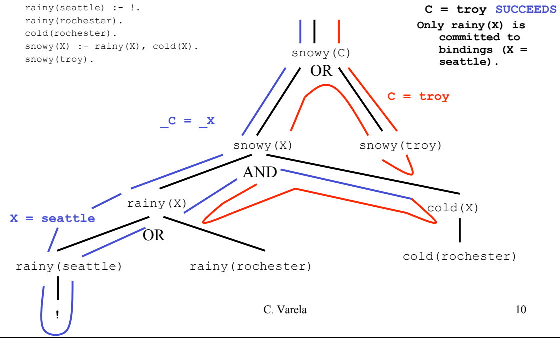


Cut (!) Example 3

```
rainy(seattle) :- !.
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
snowy(troy).
```


Cut (!) Example 3

```
rainy(seattle) :- !.
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
snowy(troy).
```

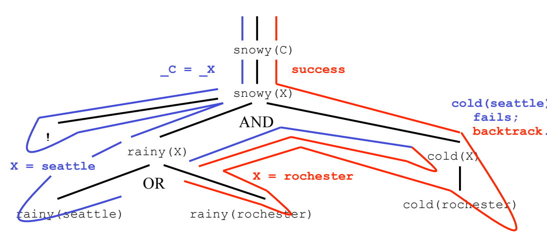


Cut (!) Example 4

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- !, rainy(X), cold(X).
```


Cut (!) Example 4

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- !, rainy(X), cold(X).
```

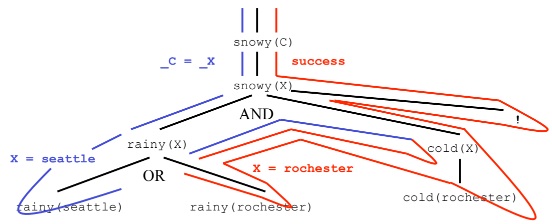


Cut (!) Example 5

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X), !.
```


Cut (!) Example 5

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X), !.
```



First-Class Terms

`call(P)`	Invoke predicate as a goal.
`assert(P)`	Adds predicate to database.
`retract(P)`	Removes predicate from database.
`functor(T,F,A)`	Succeeds if `T` is a *term* with *functor* `F` and *arity* `A`.

`not P` is not `¬P`

- In Prolog, the database of facts and rules includes a list of things assumed to be **true**.
- It does not include anything assumed to be **false**.
- Unless our database contains everything that is **true** (the *closed-world assumption*), the goal `not P` can succeed simply because our current knowledge is insufficient to prove `P`.

More `not` vs ¬

```
?- snowy(X).
X = rochester
?- not(snowy(X)).
no
```

Prolog does not reply: **`X = seattle.`**

The meaning of `not(snowy(X))` is:

$\neg\exists X\ [snowy(X)]$

rather than:

$\exists X\ [\neg snowy(X)]$

Fail, true, repeat

`fail`	Fails current goal.
`true`	Always succeeds.
`repeat`	Always succeeds, provides infinite choice points.

```
repeat.
repeat :- repeat.
```

not Semantics

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

Definition of `not` in terms of failure (`fail`) means that variable bindings are lost whenever `not` succeeds, e.g.:

```
?- not(not(snowy(X))).
X=_G147
```


Conditionals and Loops

```
statement :- condition, !, then.
statement :- else.

natural(1).
natural(N) :- natural(M), N is M+1.
my_loop(N) :- natural(I), I<=N,
              write(I), nl,
              I=N,
              !, fail.
```

Also called *generate-and-test*.

Prolog lists

- `[a,b,c ]` is syntactic sugar for:

  `.(a,.(b,.(c, [])))`

  where `[]` is the empty list, and `.` is a built-in cons-like functor.

- `[a,b,c ]` can also be expressed as:

  `[a | [b,c ]]` , or
  `[a, b | [c ]]` , or
  `[a,b,c | []]`

Prolog lists append example

```
append([],L,L).
append([H|T ],A, [H,L ]) :- append(T,A,L).
```


Exercises

8. What do the following Prolog queries do?

```
?- repeat.
?- repeat, true.
?- repeat, fail.
```

   Corroborate your thinking with a Prolog interpreter.
9. Draw the search tree for the query "`not(not(snowy(City)))`". When are variables bound/unbound in the search/backtracking process?
10. PLP Exercise 11.24 (pg 655).
11. *PLP Exercise 11.34 (pg 656).