BACKTRACKING

- Forward chaining goes from axioms forward into goals.
- Backward chaining goes from goals and works backward to prove them with existing axioms.
BACKTRACKING

rainy(seattle).

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

\[
\begin{align*}
\text{snowy}(c) & \quad \text{\textbf{AND}} \\
-x = X & \quad \text{\textbf{OR}} \\
\text{snowy}(X) & \quad \text{\textbf{OR}} \\
\text{rainy}(seattle) & \\
\text{rainy(rochester)} & \\
\text{cold(rochester)} & \\
x = rochester & \\
\text{\textbf{BACKTRACK}}
\end{align*}
\]
IMPERATIVE CONTROL FLOW

Programmer has explicit control on backtracking process.

CUT (!)

As a goal it succeeds, but with side-effect:

Commits interpreter to choices made since unifying parent goal with left-hand side of current rule.
BACKTRACKING

rainy (seattle).
rainy (rochester).
cold (rochester).

snowy (X) :- rainy (X), cold (X).

| snowy (c)
| _c = _X
| snowy (X)
AND

| x = seattle
OR

rainy (seattle)      rainy (rochester)

| cold (seattle) fails; no backtracking to "rainy".
| GOAL fails.

cold (rochester)
BACKTRACKING

rainy (seattle).
rainy (rochester).
cold (rochester).
snowy (x) :- rainy (x), cold (x).
snowy (troy).

\[ \neg C = \neg X \]

\[ X = \text{seattle} \]

\[ \text{rainy (seattle)} \]

\[ \text{rainy (rochester)} \]

\[ \text{cold (rochester)} \]

\[ X = \text{troy} \]

\[ \text{succeed?} \quad \neg \neg \]

\[ \text{snowy (c)} \]

\[ \text{snowy (x)} \]

\[ \text{AND} \]

\[ \text{cold (x)} \]

\[ \text{snowy (troy)} \]
BACKTRACKING

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

\[ x = \text{seattle} \]
\[ \neg x = \neg c \]
\[ \text{succeed.} \]
\[ \text{succeed.} \]
\[ e = \text{troy} \]
\[ \text{snowy(troy)} \]
\[ \text{snowy(X)} \]
\[ \text{AND} \]
\[ \text{rainy(X)} \]
\[ \text{OR} \]
\[ \text{rainy(rochester)} \]
\[ \text{cold(rochester)} \]
\[ \text{cold(rochester)} \]
\[ \text{OR} \]
\[ \text{rainy(seattle)} \]
BACKTRACKING

rainy (seattle).

rainy (rochester).
cold (rochester).

snowy (X) :- rainy (X), cold (X).

|   ↑   |
|snowy(c) |

|   |   |
snowy(X) AND

|   |   |
rainy(X) OR

rainy (seattle) rainy (rochester)
cold (rochester)
BACKTRACKING

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

\[
\begin{align*}
& \text{snowy (c)} \\
& \text{snowy (X)} \\
& \text{AND} \\
& \text{rainy (x)} \\
& \text{OR} \\
& \text{rainy (seattle)} \\
& \text{rainy (rochester)} \\
& \text{cold (rochester)} \\
\end{align*}
\]
**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 w/ functor $F$ and arity $A$. 
not P is not 7P

- 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.
NOT SEMANTICS

\[
\text{not}(P) := \text{call}(P), !, \text{fail}, \\
\text{not}(P).
\]

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

\[
?\text{- not(not(snowy(x)))).}
\]
\[
X = _G147
\]
MORE NOT US T

?- snowy(X).
    X = rochester
?- not(snowy(X)).
    no // it does NOT reply:
    X = seattle

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

\[ \neg \exists X \text{[snowy}(X)] \]
rather than:
\[ \exists X \text{[} \neg \text{snowy}(X)] \]
\text{not(\text{not(\text{snowy}(X)))})}

\text{call(\text{not(\text{snowy}(X)))}}

\text{not(\text{snowy}(X))} ! \quad \text{fail}

\text{call(\text{snowy}(X))}

\text{snowy}(X) ! \quad \text{fail}

\text{x = rochester}

\text{true}
fail, true, ...

fail
true
repeat

fail current goal.
always succeed.
always succeed, provides infinite choice points

repeat.
repeat :- repeat.

Exercise:
what do the following queries do?
?- repeat.
?- repeat, true.
?- repeat, fail.
\[ \text{not}(P) : \text{call}(P), !, \text{fail}. \]
\[ \text{not}(P). \]
\[ \text{IF-THEN} (\rightarrow) \]
\[ \rightarrow (\text{If, Then, Else}) \]
\[ : \text{If, !, Then}. \]
\[ \rightarrow (\text{If, Then, Else}) : \text{Else}. \]
\[ \rightarrow (\text{If, Then, Else}) \]
\[ : \text{If, Then, !}. \]
**Conditionals and Loops**

statement ::= condition, !, then-part.
statement ::= else-part.

natural (1),
natural (N) ::= natural (M),
                N is M+1.

my-loop (N) ::= natural (I), I≥N,
                write(I), nl,
                I = N, // test-
                !, fail. = Next-fail.

Also called generate-and-test.