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).

snowy(C)

\_C = \_X

AND

snowy(X)

success

cold(seattle)
fails;
backtrack.

X = seattle

OR

rainy(X)

cold(X)

X = rochester

cold(rochester)

cold(rochester)
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

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

![Diagram](image_url)

-c = _x

x = seattle

AND

OR

rainy(seattle) rainy(rochester)

cold(seattle)
cold(rochester)

cold(seattle) fails; no backtracking to rainy(X).

GOAL FAILS.
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).

C = troy FAILS
snowy(X) is committed to bindings (X = seattle).
GOAL FAILS.

C = troy

\(_C = \_X\)

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

\(\text{AND}\)
\(\text{OR}\)
\(\text{OR}\)

\(\text{rainy}(X)\)
\(\text{cold}(X)\)
\(\text{cold}(\text{rochester})\)

\(\text{rainy}(\text{seattle})\)
\(\text{rainy}(\text{rochester})\)

\(\text{X = seattle}\)
Cut (!) Example 3

```prolog
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).

C = troy SUCCEEDS
Only rainy(X) is committed to bindings (X = seattle).

C = _C = _X

snowy(X)

C = troy

snowy(troy)

AND

rainy(X)

OR

cold(X)

OR

cold(rochester)

X = seattle

rainy(seattle)

rainy(rochester)

C. Varela
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).

snowy(C)

\_C = \_X

success

cold(seattle) fails;
backtrack.

cold(X),

cold(rochester)

X = rochester

OR

AND

rainy(X),

X = seattle

OR

rainy(seattle)

rainy(rochester)
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), !.

_C = _X

X = seattle

X = rochester

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

success

AND

OR

X = rochester

cold(rochester)
# First-Class Terms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>call(P)</code></td>
<td>Invoke predicate as a goal.</td>
</tr>
<tr>
<td><code>assert(P)</code></td>
<td>Adds predicate to database.</td>
</tr>
<tr>
<td><code>retract(P)</code></td>
<td>Removes predicate from database.</td>
</tr>
<tr>
<td><code>functor(T, F, A)</code></td>
<td>Succeeds if <code>T</code> is a <em>term</em> with ( F ) functor and ( A ) arity.</td>
</tr>
</tbody>
</table>
not $P$ is not $\neg P$

• 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 $P$ (or $\backslash + P$ in some Prolog implementations) can succeed simply because our current knowledge is insufficient to prove $P$. 
More not vs \( \neg \)

\[
\begin{align*}
? &- \text{ snowy}(X). \\
X &- \text{ rochester} \\
? &- \text{ not}\left(\text{ snowy}(X)\right). \\
\text{no}
\end{align*}
\]

Prolog does not reply: \( X = \text{ seattle}. \)

The meaning of \( \text{ not}\left(\text{ snowy}(X)\right) \) is:

\[ \neg \exists X \left[ \text{ snowy}(X) \right] \]

rather than:

\[ \exists X \left[ \neg \text{ snowy}(X) \right] \]
## Fail, true, repeat

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fail</td>
<td>Fails current goal.</td>
</tr>
<tr>
<td>true</td>
<td>Always succeeds.</td>
</tr>
<tr>
<td>repeat</td>
<td>Always succeeds, provides infinite choice points.</td>
</tr>
</tbody>
</table>

repeat.

repeat :- repeat.
not Semantics

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

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

\[
?\leftarrow \text{not}(\text{not(}\text{snowy}(X)))). \\
X=_{G147}
\]
Conditionals and Loops

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

natural(1).
natural(N) :- natural(M), N is M+1.
my_loop(N) :- N>0,
            natural(I), I<=N,
            write(I), nl,
            I=N,
            !, fail.

Also called *generate-and-test*. 
Prolog lists

• \([a, b, c]\) is syntactic sugar for:

\[
\text{.(a, .(b, .(c, []))))
\]

where \([],\) is the empty list, and \(\text{.}\) is a built-in cons-like functor.

• \([a, b, c]\) can also be expressed as:

\[
[a | [b, c]] , \text{or} \\
[a, b | [c]] , \text{or} \\
[a, b, c | []]
\]
Prolog lists append example

append([], L, L).
append([H|T], A, [H|L]) :- append(T, A, L).
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 \texttt{not(not(snowy(City)))}. When are variables bound/unbound in the search/backtracking process?

10. PLP Exercise 11.6 (pg 571).

11. PLP Exercise 11.7 (pg 571).