Backtracking

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

**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).

\[
\text{snowy}(X) \ :- \ \text{rainy}(X), \ !, \ \text{cold}(X).
\]
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 4

rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- !, rainy(X), cold(X).
Cut (!) Example 5

```prolog
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 ¬

- `not(snowy(X)).
  X = rochester
  no
  Prolog does not reply: X = seattle.

The meaning of `not(snowy(X)). X = seattle` is:

- `¬∃X [snowy(X)]`
- Rather than:
- `∃X [¬snowy(X)]`

Fail, true, repeat

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

```prolog
repeat.
repeat :- repeat.
```
Semantics

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

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

\[ ?- \text{not}(\text{not}(\text{snowy}(X))). \]

Conditionals and Loops

\[ \text{statement} := \text{condition}, \! , \text{then}. \]
\[ \text{statement} := \text{else}. \]

\[ \text{natural}(1). \]
\[ \text{natural}(N) :- \text{natural}(M), N \text{ is } M+1. \]
\[ \text{my_loop}(N) :- \text{natural}(I), I \leq N, \]
\[ \text{write}(I), \text{nl}, \]
\[ I = N, \]
\[ \! , \text{fail}. \]

Also called \textit{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:
  
  - \([a, b, c]\) , or
  
  - \([a, b | [c]]\) , or
  
  - \([a, b, c | []]\)

Prolog lists append example

\[ \text{append([], L, L)}. \]
\[ \text{append([H|T], A, [H,L]) :- append(T, A, L)}. \]

Exercises

8. What do the following Prolog queries do?

\[ ?- \text{repeat}. \]
\[ ?- \text{repeat}, \text{true}. \]
\[ ?- \text{repeat}, \text{fail}. \]

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 11.24 (pg 655).
11. *PLP Exercise 11.34 (pg 656).