Execution of Prolog Programs

- **Unification:** (variable binding)
  Specializes general rules to apply to a specific problem.

- **Backward Chaining**
  - **Top-Down Reasoning**
    - Goal Directed Reasoning:
      Reduces a goal to one or more subgoals.

- **Backtracking:**
  Systematically searches for all possible solutions that can be obtained via unification and backchaining.

### Prolog Execution Trace

```
Call: (7) tripe(austin, chicago) ? creep
Call: (8) flight(austin) ? creep
Fal: (8) flight(austin) ? creep
Ref: (7) tripe(austin, chicago) ? creep
Call: (8) flight(chicago, austin) ? creep
Fal: (8) flight(chicago, austin) ? creep
Ref: (7) tripe(austin, chicago) ? creep
Call: (8) flight(austin, chicago) ? creep
Fal: (8) flight(austin, chicago) ? creep
Ref: (7) tripe(austin, chicago) ? creep
Exit: (8) tripe(austin, chicago) ? creep
Exit: (7) tripe(austin, chicago) ? creep
```

### Prolog Execution Example
```
Prolog Execution Example

flight(austin, chicago), flight(chicago, austin), flight(miami, miami),
trip(austin, miami), trip(miami, austin), trip(chicago, austin),
trip(austin, chicago),

Example 2 Database:

harry(austin, austin),
wednesday(austin),
married(austin, austin),
married(miami, miami),
wednesday(miami),
weekend(miami),
weekend(austin),
pinecone(austin),
pinecone(miami),

Query:
pinecone(austin).
Three answers are generated:

Name: austin,
when = wednesday,
```

### Prolog Search Trees

**Prolog Search Trees**
- Encapsulates unification, backward chaining, and backtracking.
- Internal nodes are ordered list of subgoals.
- Leaves are success nodes or failures, where computation can proceed no further.
- Edges are labeled with variable bindings that occur by unification.

**Prolog Search Tree**

Describe all possible computation paths:
- There can be many success nodes.
- There can be infinite branches.

### Problem with DFS
Can get stuck on infinitely recursive paths, even when a goal is provable.

E.g.,
``` prolog
married(Y, X) :- married(X, Y),
marrird(john, sue).
The query:
marrird(sue, john)?
The problem is with left recursion.
```

Solution:
souse(X, Y) :- married(Y, X),
souse(Y, X) :- married(Y, X),

Another example, as we saw last day:
``` prolog
above(X, Y) :- above(Y, Z), on(Y, X),
above(X, Y) :- on(X, Y),
```

Prolog uses Depth-First Search (DFS),
Controlling Prolog's Reasoning with Cut

The goal "r", pronounced "cut", always succeeds immediately.

It has an important side effect: once it is satisfied, it disallows either:
- backtracking back over the cut, or
- backtracking and applying a different clause of the same predicate to satisfy the present goal.

You can think of satisfying Cut as making a commitment both:
- to the variable bindings we've made during the application of this rule, and
- to this particular rule itself.

Describing Cut (1)
The cut goal succeeds whenever it is the current goal, and the deduction tree is trimmed of all other choices on the way back to an including the point in the deduc-
tion tree where the cut was introduced into the sequence of goals.

Cut tells us: Don't ever backtrack through this point when looking for alternative solutions. It acts as a barrier beyond which Prolog will not go. All the choices made prior to the cut cease to exist and are treated as though they were the only possible choices.

You can think of Cut as telling the interpreter: "You're telling me you've got this far in the process; there's no need to backtrack and try another path for solving this goal, or to try another way of satisfying any of the subgoals that were already proved for this goal."

How to Trace Cut

When a "r" goal is satisfied:
1. Find the rule that has that cut.
2. Put an oval around the tree branch from the node where the first goal matches the head of that rule down to the node where the first goal is that cut.

For every node circled, no further branches will be expanded from that node.

1. Cut Can Reduce Your Search Space

Cut can be used to improve the efficiency of search by reducing Prolog's search space. E.g.,

When two predicates are mutually exclusive,

q(x) :- mom(x), s(x),
q(x) :- sire(x), s(x).

With cut

q(x) :- mom(x), s(x),
q(x) :- sire(x), s(x).

2. Cut Can Implement Exceptions to Rules

I.e., "To get the right answer."

Cut can be used to encode exceptions to rules. This is used in AI default reasoning:

head(mother),
head(ancestor),
head(penguin),
fig(penguin) = 1, fail,
fig(s) = head(s).

3. Cut Can Implement NAF

Cut can be used to implement negation as failure:

not(x) :- x, fail,
not(x).

Note that not is a meta-level predicate. It takes a predicate as an argument. E.g.,

not(mother).

4. Cut Can Remove Multiple Answers

Fixing our Mamma Example, Recall,

1) param(chart) :- fathal(x), param(s),
2) param(abv) :- male(s), param(t).
3) param(x) :- male(s), param(t), 1.
4) top(x) :- s(x).
5) orp(l) :- male(s), s(x),
6) top(x) :- param(x), 1.
7) top(x) :- male(s), orp(x),
8) param(x) :- l, param(x).
9) param(male,s),
10) param(s),
11) param(fathal, male),
12) param(fathal, female),
13) param(fathal, s),
14) param(s),
15) male(f),
16) female(s),
17) male(male),
18) female(f).
Cut Summary

Cuts are:

+ very powerful,

+ can help:

- improve efficiency (reduce search space)
- get the right answer (treat exceptions to rules)
- implement NAF
- remove multiple answers
- difficult to use safely,
  
- make for difficult to understand programs,