Planning in prolog

- **Blocks on a table**
  - Initial state:

```
on(a, table).
on(b, c).
on(c, table).
```

  - Our program will modify on/2, so make it dynamic:

```
:- dynamic on/2.
```

  - Put one block on top of another (or the table):

```
put_on(A,B) :-
    A \== table,                % Check that everything is ok.
    A \== B,
    on(A,X),
    clear(A),
    clear(B),
    retract(on(A,X)),           % Modify the state.
    assert(on(A,B)),
    assert(move(A,X,B)).        % Keep a record.
```

  - What does it mean for a block or the table to be clear?

```
clear(table).
clear(B) :-
    not(on(_X,B)).
```

  - We can now move blocks around:

```
?- put_on(a, b).
Yes
?- listing(on).
on(b, c).
on(c, table).
on(a, b).
?- put_on(a, c).
No
?- put_on(b, table).
No
?- put_on(a, table).
Yes
?- put_on(b, a).
Yes
```

- **Planning more advanced movements**
  - Recursive movement of blocks:

```
r_put_on(A,B) :-
    on(A,B).                % Base case: Already done.
r_put_on(A,B) :-
    not(on(A,B)),
    A \== table,
    A \== B,
    clear_off(A),           % Fails if we cannot clear off A.
```

```
        clear_off(B),
        on(A,X),
        retract(on(A,X)),
        assert(on(A,B)),
        assert(move(A,X,B)).
```

- Clearing off to make room:

```
clear_off(table).          % There is always room on the table.
clear_off(A) :-
    not(on(_X,A)).         % Already clear.
clear_off(A) :-
    A \== table,
    on(X,A),
    clear_off(X),          % Recursively clear off the block above.
    retract(on(X,A)),
    assert(on(X,table)),
    assert(move(X,A,table)).
```

- We can now move anything:

```
?- listing(on).
on(a, table).
on(b, c).
on(c, table).
Yes
?- r_put_on(c, a).
Yes
?- listing(on).
on(a, table).
on(b, table).
on(c, a).
```

How does it work?

- **Example program: A binary relation and its closure (G&J 8.2.2)**

```
rel(a, b).
rel(a, c).
rel(b, f).
rel(f, g).
clos(X, Y) :- rel(X, Y).
clos(X, Y) :- rel(X, Z), clos(Z, Y)
```

- **Example query: ?- clos(a, f).**
  - Find facts and rules that "fit" the goal clos(a, f).
  - Does clos(X, Y) "fit"?
    - Yes, because we can substitute the constant a for the variable X and the constant f for the variable Y.
  - Can replace clos(a, f) with rel(a, Z), clos(Z, f).
  - Then what?
  - rel(a, Z) matches rel(a, b).
  - We now have rel(a, b), clos(b, f).
  - The first clause, rel(a, b). is a fact.

    - The second clause matches rel(b, f).
    - rel(b, f) is a fact.
    - So the answer is yes.
- **Several other computations are possible also**
    - Some fail, some don't terminate.
    - See figure 8.6 in G&J (see slides)
- **Also possible with several succeeding computations.**
    - clos(a, X) (try it!)
    - Several succeeding computations.
- **Search tree**
    - See figure 8.6 in G&J (see slides)
    - Backtracking: When a computation fails (or when the used does not accept an answer).
    - How do we search the tree—depth first or breadth first?
        - Depth first may enter nonterminating computation.
            - Sound but not complete.
        - Breadth first is complete and sound.
        - Prolog uses depth first; some other languages use breadth first.
    - Can we get a "no" answer if the tree is infinite? No.

Unification

- **Motivation**
    - We informally talked about a clause "matching" or "fitting" another clause.
    - Will now look a the matching in more detail.
- **Relationships to other things we've seen**
    - Generalization of a function call
        - Function call: Formal parameters are unbound, but become bound to the values of the actual parameters when the function is called.
        - Unification: Unbound names in both clauses can become bound.
    - Combined pattern matching and binding of variables.
        - Similar to pattern matching in Haskell.
- **Define t1 and t2**
    - t1: goal to prove
    - t2: fact or rule's head with which match is tried
- **"t1 is more general than t2"**
    - There exists a substitution μ s.t. t2 = μ(t1)
    - There does not exist a substitution π s.t. t1 = π(t2)
    - Example: f(X, Y) is more general than f(a, Y)
        - μ = {<X, a>}
        - "Substitute a for X."
        - Can only substitute for variables, not for constants.
- **"t1 and t2 are variants"**
    - There exist substitutions μ and π s.t. μ(t1) = t2 and π(t2) = t1

- Example: f(X, Y) and f(Z, W) are variants.
  - μ = {<X, Z>, <Y, W>}
  - π = {<Z, X>, <W, Y>}
- **"t1 and t2 unify"**
  - Two terms unify if a substitution can be found that makes them equal.
- **"μ is a unifier"**
  - A substitution that makes two terms unify is called a unifier.
  - Example:
    - s1 = {<X, a>, <Y, c>, <Z, b>, <W, c>}
      is a unifier for f(X, b, Y) and f(a, Z, W)
      - s1(f(X, b, Y)) = f(a, b, c)
      - s1(f(a, Z, W)) = f(a, b, c)
- **"μ is the most general unifier (MGU)"**
  - μ is the most general unifier (MGU) of t1 and t2 if μ(t1) = μ(t2) is the most general instance of both t1 and t2.
    - "most general instance" means that no other instance is more general
  - Example:
    - s2 = {<X, a>, <Y, T>, <Z, b>, <W, T>}
      is the MGU for f(X, b, Y) and f(a, Z, W).
- **Unification algorithm**
  - Figure 8.3 in G&J (or see slides)