Proof steps - Control

- Order of rules
- Backtracking
- Control predicates
  - cut
  - fail

Order of rules is important!

remember:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

versus:

```
member(X,[_|T]) :- member(X,T).
member(X,[X|_]).
```

other order of enumeration!

```
?- member(X,[a,b,c]).
```

Order

- behavior correct/incorrect
- complete/incomplete
- terminating/cyclic
- efficient/not efficient

Order

- terminating/cyclic Example:

```
neighbor(a,b).
neighbor(b,c).
neighbor(c,d).
...
path(St,Zi) :- neighbor(St,Zi).
(a)
path(St,Zi) :- path(St,Zw), neighbor(Zw,Zi).
or
(b)
path(St,Zi) :- neighbor(St,Zw), path(Zw,Zi).
```

Order

- efficient/not efficient Example:

Familiy Data:

```
siblings([heinz,inge,olaf,eva,theo]).
siblings([peter,maria,johann]).
siblings([ute]).
```

```
female([inge,eva,maria,ute]).
male([heinz,olaf,theo,peter,johann]).
```

```
married(ute,heinz).
married(peter,inge).
married(maria,olaf).
married(eva,johann).
```

```
children(ute,heinz,[rosa,django,nicole
children(inge,peter,[petra,uli,karl).
children(maria,olaf,[josef,paula]).
```

Order

- efficient/not efficient Example:

Family data: Alternatives Pros and Cons??

```
niece(Pers,Niece) :- siblingof(Pers,Sibl),
          childof(Sibl,Niece), female(Niece).
niece(Pers, Niece) :- married(Pers,spouse), siblingof(spouse,Sibl),
          childof(Sibl,Niece), female(Niece).
```

or

```
niece(Pers, Niece) :- female(Niece), childof(Sibl,Niece), siblingof(Pers,Sibl).
niece(Pers,Niece) :- female(Niece), childof(Sibl, Niece), married(Pers,Spouse),
          siblingof(Spouse,Sibl),
```

efficient/not efficient

- Task:
  - as few steps as possible!
  - search space as small as possible!

Search space as small as possible!

- Example:

```
fondofchildren(X) :- aunt(X), friendly(X).

aunt(X) :- niece(X,_).
aunt(X) :- nephew(X,_).

friendly(ute).

?- fondofchildren(maria).
```

Search space as small as possible!

- Unnecessary to compute all proofs of *aunt* by backtracking!
- Can be avoided by Cut (marked by: !) :
- Effect of Cut:
  - ! predicate that is always true.
  - ! prevents backtracking into the domain marked by the Cut:
- Position of the Cut:
  - within a rule: p(X) :- a(X), b(X), !, c(X), d(X).
  - at the end of a rule: p(X) :- a(X), b(X), c(X), d(X), !.
- If Cut is reached by Redo (i.e. from ‚backwards'), then the entire predicate 'fails'

therefore…

```
aunt(X) :- niece(X,_), !.
aunt(X) :- nephew(X,_), !.
```

or…

```
fondofchildren(X) :- aunt(X), !, friendly(X).
```

What is the difference?

Difference

```
aunt(X) :- niece(X,_), !.
aunt(X) :- niece(X,_), !.
```

or…

```
fondofchildren(X) :- aunt(X), !, friendly(X).
fondofchildren(X) :- italian(X).      ←not reached
```

Cut can be efficient only or change behavior completely!

green cut (efficient only) and
red cut (changing the logic)

```
aunt(X) :- niece(X,_), !.
aunt(X) :- nephew(X,_), !.
```

or…

```
fondofchildren(X) :- aunt(X), !, friendly(X).
fondofchildren(X) :- italian(X).  ←cannot be reached in the
                                    presence of unfriendly aunts
```

green cut (efficient only) and red cut (changing the logic)

another example

```
max(X,Y,Y) :- X =< Y.
max(X,Y,X) :- X>Y.
```

```
max(X,Y,Y) :- X =< Y,!.
max(X,Y,X) :- X>Y.
```

```
max(X,Y,Y) :- X =< Y,!.
max(X,Y,X).
```

```
?- max(2,3,2).
```

… completeness of definition depends on how the predicate shall be used ...

```
max0(X,Y,Z) :- var(Z), max(X,Y,Z).
```

Use cases…

- check with instantiated values

```
?- sum(2,3,5).
```

- compute (exactly one) value:

```
?- sum(2,3,X).
```

➢ return different values according to use case

Example…

```
member_a(A,[A|L]) :- atom_concat(a,_,A).
member_a(A,[_|L]) :- member_a(A,L).
```

Return several values by backtraching

```
?- member_a(A,[anna,ute,alfred,jochen,arne]).
anna ;
alfred ;
arne ;
no.
```

alternatively… with control predicate fail

return several values (without manual backtracking):

```
member_a(A,[A|L]) :- atom_concat(a,_,A), write(A), nl, fail.
member_a(A,[_|L]) :- member_a(A,L).
```

```
anna
alfred
arne
no.
```

The predicate *fail*

- always fails:

  i.e. always triggers backtracking from the position where it is placed!

- often used as negative test together with Cut:

  ```
  test(X,Y) :- excludingcondition(X,Y), !, fail.
  test(X,Y) :- positivecondition1(X,Y),…
  …
  ```

Normally backtracking does not keep track about intermediate solutions…

- Disadvantage:

  intermediate knowledge normally gets lost

Alternative …

- Store the values computed:

accumulator

A definition with accumulator …

```
member_a(L,M) :- member_a(L,[],M).

member_a([A|L],M,MO) :- atom_concat(a,_,A),
      member_a(L,[A|M],MO).
member_a([_|L],M,MO) :- member_a(L,M,MO).
member_a([],M,M).
```

Another definition with accumulator …

more intelligently… with one list only

```
member_a([A|L],[A|M]) :- atom_concat(a,_,A),
      member_a(L,M).
member_a([_|L],M) :- member_a(L,M).
member_a([],[]).
```

Accumulators can contribute to efficiency!

Example: reverse

```
naiverev([],[]).
naiverev([H|T],R) :- naiverev(T,RevT),
      append(RevT,[H],R).
```

versus

```
rev(L,R) :- accRev(L,[],R).

accRev([H|T],A,R) :- accRev(T,[H|A],R).
accRev([],A,A).
```