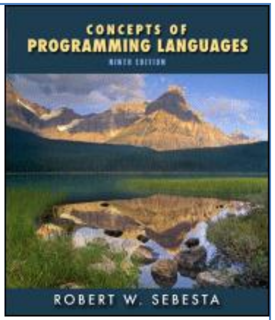
Prolog

1 Ch16 – Logic Programming Languages

2 Download SWI-Prolog and SWI-Prolog-Editor

3 Reference

- Prolog Programming A First Course, Paul Brna

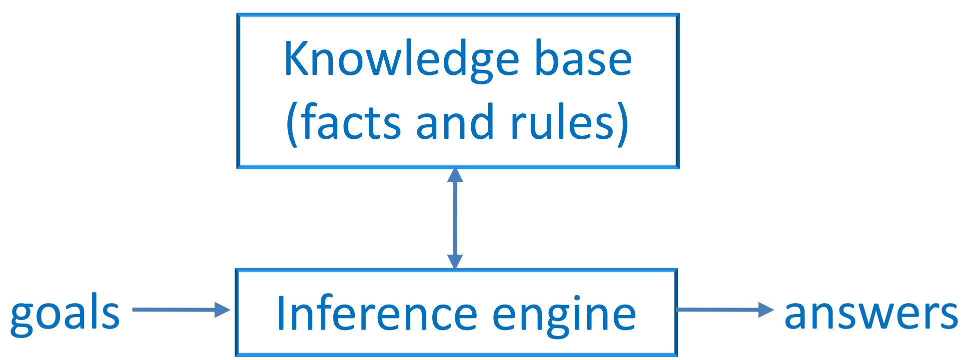


Logic languages

- Characteristics
  - Logic languages are based on symbolic logic.
  - Logic languages are declarative languages.

    One needs only declare the what is knowledge.

    The how to knowledge is built in the inference engine.



Propositional logic

- Proposition
  - A proposition is a statement that can be assigned a truth value.
- Propositional logic (Boolean algebra)
  - Use symbols to represent atomic propositions
  - Use logical operators to express compound propositions
  - Example

    Snoopy likes C++. (p)

    Snoopy likes PL. (q)

    If Snoopy likes C++ and PL, $\overbrace{\text{he is crazy.}}^{(r)}$ $(p \land q \rightarrow r)$

Propositional logic

- Prolog program in propositional logic
  - Let demo.pl be the file containing the Prolog program:

```
p.            % fact
q.            % fact
r :- p, q.    % rule
```

  - Type pl in command line
  - ?- ['demo.pl'] . % or, consult('demo.pl').

```
% demo.pl compiled 0.00 sec, 788 bytes
?- p.         % goal; Does Snoopy like C++?
true.
?- r.         % goal; Is Snoopy crazy?
true.
```

First-order logic

- Predicate calculus (First order logic)
  - Use constants, variables, and function symbols with arguments to represent individuals.
  - Use predicate symbols with or without arguments to express atomic propositions.
    - A predicate symbol without arguments is just a propositional symbol.
    - Propositional logic is a subset of predicate calculus.
  - Use logical operators and quantifiers ($\forall$, $\exists$) to express compound propositions

First-order logic

- Predicate calculus (First order logic)
  - Example

    likes(snoopy,c++) % likes is a predicate symbol

    likes(snoopy,pl)

    likes(father(snoopy),c++) % father is a function symbol

    $\forall X(likes(X,c{+}{+}) \wedge likes(X,pl) \rightarrow crazy(X))$
- Prolog program in first-order logic
  - likes(snoopy,'c++').

    likes(snoopy,pl).

    likes(father(snoopy),'c++').

    crazy(X) :- likes(X,'c++'), likes(X,pl). % rule, ∀X

First-order logic

- Prolog program in first-order logic
  - Variables begin with upper case letters or _
    Other symbols must begin with lower case letters, unless they are quoted, e.g. 'Snoopy'.
  - ?- crazy(X). % goal, ∃X, Is there anybody crazy?
    X = snoopy . % . – stop
    ?- crazy(X).
    X = snoopy ; % ; (or space) – redo the goal
    false. % no more answer
    ?- crazy(snoopy).
    true . % had better prompt immediately

Horn clauses

- Terms
  1. Constants (i.e. individuals) and variables are terms.
  2. If f is an n-ary functor (i.e. predicate or function symbol) and t_1, ..., t_n are terms, f(t_1,...,t_n) is a (compound) term.
  - If f is a function symbol, the term denotes an individual. If f is a predicate symbol, the term denotes a proposition.
- Clauses
  - A clause is a universally quantified formula of the form

    H_1; ...; H_m :- B_1, ..., B_n ; or , and :- imply

    where H_i's and B_j's are compound terms whose principal functors are predicate symbols.

Horn clauses

- Horn clauses
  - A clause with $m \leq 1$ is called a Horn clause, i.e.

    H :- $B_1, \ldots, B_n$

    :- $B_1, \ldots, B_n$

    where H is called the head, and B_i's the body.
  - A Horn clause has no negative and disjunctive information.
  - Prolog is based on the Horn clause subset of first order logic.
  - There are 4 kinds of Horn clauses:

    Head yes/no

    Body yes/no

Horn clauses

- Horn clauses
  - Case 1: Rule, $m = 1, n > 0$

    crazy(X) :- likes(X,'c++'), likes(X,pl)
  - Case 2: Fact, $m = 1, n = 0$

    likes(snoopy,pl) :-   % unconditioned conclusion (fact)

    It is as if there is a true in the body, since

    likes(snoopy,pl) ← true

    ⇔ ~true ∨ likes(snoopy,pl)

    ⇔ likes(snoopy,pl)

Horn clauses

- Horn clauses
  - Case 3: Negated goal, $m = 0, n > 0$

    :- crazy(X)   % unconcluded condition (goal)

    It is as if there is a false in the head, since

    $\forall X\ (\text{false} \leftarrow \text{crazy}(X))$

    $\Leftrightarrow \forall X\ (\sim\text{crazy}(X) \lor \text{false})$

    $\Leftrightarrow \forall X \sim\text{crazy}(X)$

    $\Leftrightarrow \sim\exists X\ \text{crazy}(X)$
  - Case 4: The empty clause, $m = 0, n = 0$

    :- means

    false $\leftarrow$ true $\Leftrightarrow$ false, i.e. it represents a contradiction.

Refutation proof

- Refutation proof
  - Let KB be the universally quantified knowledge base, and G be an existentially quantified goal
  - Refutation proof of KB → G (≡ KB ∧ ~G → false)
    1. Negate the goal, i.e. make the clause :- G.
    2. Start with KB ∧ ~G and try to deduce the empty clause :-
- The resolution principle (RP) for propositional logic
  - H :- B_1, …, B_n % facts or rules

    :- H, G_2, …, G_m % negated goals

    :- B_1, …, B_n, G_2, …, G_m

    ∵ H, G_2, …, G_m :- B_1, …, B_n , G_2, …, G_m

Refutation proof

- Refutation proof in propositional logic
  - Example

1	p :-	% fact
2	q :-	% fact
3	r :- p, q	% rule
4	:- r	% negated goal
5	:- p, q	% 3, 4, RP
6	:- q	% 1, 5, RP
7	:-	% 2, 6, RP

```
p.
q.
r :- p, q.

?- r.
```

Pragmatically, Prolog answers goals by backward chaining

i.e. starting with the goals, it tries to see whether the required conditions are all known facts.

Refutation proof

- Unification

  Two terms s and t are unifiable if there is a unifier (i.e. substitution) Θ such that $s\Theta = t\Theta$

s	t	Θ
likes(snoopy,pl)	likes(snoopy, 'c++')	Not unifiable
likes(snoopy,pl)	likes(snoopy,pl)	{}
likes(snoopy,pl)	likes(snoopy,X)	{ X=pl }
crazy(X)	crazy(Y)	{ X=Y } { X=Y=snoopy }, …

  The last two terms have infinitely many unifiers.

  Among them, { X=Y } is called the most general unifier (mgu).

Refutation proof

- The resolution principle (RP) for first-order logic
  - If H and G_1 are unifiable with the mgu θ (i.e. $H\Theta = G_1\Theta$), then

$$\frac{\begin{array}{l} H \text{ :- } B_1, \ldots, B_n \\ \quad \text{:- } G_1, G_2, \ldots, G_m \end{array}}{\quad \text{:- } B_1\Theta, \ldots, B_n\Theta, G_2\Theta, \ldots, G_m\Theta}$$

$$\Rightarrow \quad H\theta \text{ :- } B_1\Theta, \ldots, B_n\Theta$$
$$\Rightarrow \quad \text{:- } G_1\Theta, G_2\Theta, \ldots, G_m\Theta$$

∵ Horn clauses are universally quantified

  - Observe that logical variables represent individuals, rather than memory locations.

Refutation proof

- Refutation proof in first-order logic (1)
  - Example

```
1  likes(snoopy,'c++') :-                   % fact
2  likes(snoopy,pl) :-                      % fact
3  crazy(X) :- likes(X,'c++'), likes(X,pl)  % rule
4  :- crazy(snoopy)                         % negated goal
5  :- likes(snoopy,'c++'),likes(snoopy,pl)  % 3, 4, { X=snoopy }
6  :- likes(snoopy,pl)                      % 1, 5, {}
7  :-                                       % 2, 6, {}
```

Refutation proof

- More on unification (1)
  - Variables in facts and rules must be renamed in order for unification to work properly.
  - Example

    likes(X,pl). % fact, Everybody likes pl.

    ?- likes(snoopy,X). % goal, What does snoopy like?

    Without renaming, the two terms are not unifiable.

    But, the two X's are different.

    (The scope of a variable is the clause in which it appears.)

    likes(X1,pl). % rename X as X1

    ?- likes(snoopy,X). % { X1=snoopy, X=pl }

Refutation proof

- Refutation proof in first-order logic (2)
  - Example

1	likes(snoopy,'c++') :-	% fact
2	likes(snoopy,pl) :-	% fact
3	crazy(X) :- likes(X,'c++'), likes(X,pl)	% rule
4	:- crazy(X)	% negated goal
5	:- likes(X1,'c++'), likes(X1,pl)	% 3, 4, { X=X1 }
6	:- likes(snoopy,pl)	% 1, 5, { X1=snoopy }
7	:-	% 2, 6, {}

It follows from the three unifiers that X=snoopy.

Refutation proof

- More on unification (2)
    - Unification is two-way pattern matching.
    - Unification subsumes parameter-passing.
    - In Prolog, the functionality of a parameter may be in or out, depending on the goal.

      But, it cannot be inout for lack of assignment.
    - Example

```
crazy(X) :- likes(X,'c++'), likes(X,pl).
?- crazy(snoopy).        % in
?- crazy(X).             % out
```

Refutation proof

- More on unification (3)
  - If one term contains a variable that occurs in another term, the two terms aren't unifiable.

    e.g. X and s(X) aren't unifiable, since it results in an infinite subsitution:

    X = s(X) = s(s(X)) = s(s(s(X))) = …
  - Occurs check – Check if a variable occurs in a term
  - For efficiency reason, most Prolog implementations omit the occurs check.
  - Omitting the occurs check makes Prolog unsound.
  - Soundness – Every deducible result is correct.

    Completeness – Every correct result is deducible.

Refutation proof

- More on unification (3)
  - Example

```
eq(X,X).                  % Every number equals itself.
eow :- eq(Y,s(Y)).        % Let s be the successor function
?- eow.                   % If y = y+1 then eow
true.
```

But, the correct answer is no, since eq(Y,s(Y)) and eq(X,X) aren't unifiable, due to infinite substitution

$Y = X = s(Y) \Rightarrow Y = s(Y)$

N.B. $\forall Y(\text{eow :- eq}(Y,s(Y))) \equiv \forall Y(\text{eow} \vee \sim\text{eq}(Y,s(Y)))$

$\equiv \text{eow} \vee \forall Y \sim\text{eq}(Y,s(Y)))$

$\text{eow :- } \exists Y\, \text{eq}(Y,s(Y))) \equiv \text{eow} \vee \sim\exists Y\, \text{eq}(Y,s(Y)))$

Refutation proof

- Refutation proof search strategies
  - A refutation proof requires two choices:
    Goal order: choose the goal to reduce
    Clause order: choose the clause to effect the reduction
- Prolog's search strategies
  - Goal order: left to right
  - Clause order: top to bottom with backtracking
- Example
  - parent(a,b).
    parent(b,c).
    ancestor(X,Y) :- parent(X,Y).
    ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).

Refutation proof

- Example (Cont'd)
  - $\forall X \forall Y \forall Z$ (ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y))

    $\equiv \forall X \forall Y$ (ancestor(X,Y) :- $\exists Z$(parent(X,Z), ancestor(Z,Y)))
- Declarative vs procedural readings
  - Declarative readings

    Read the clauses as logical formulas
  - Procedural readings

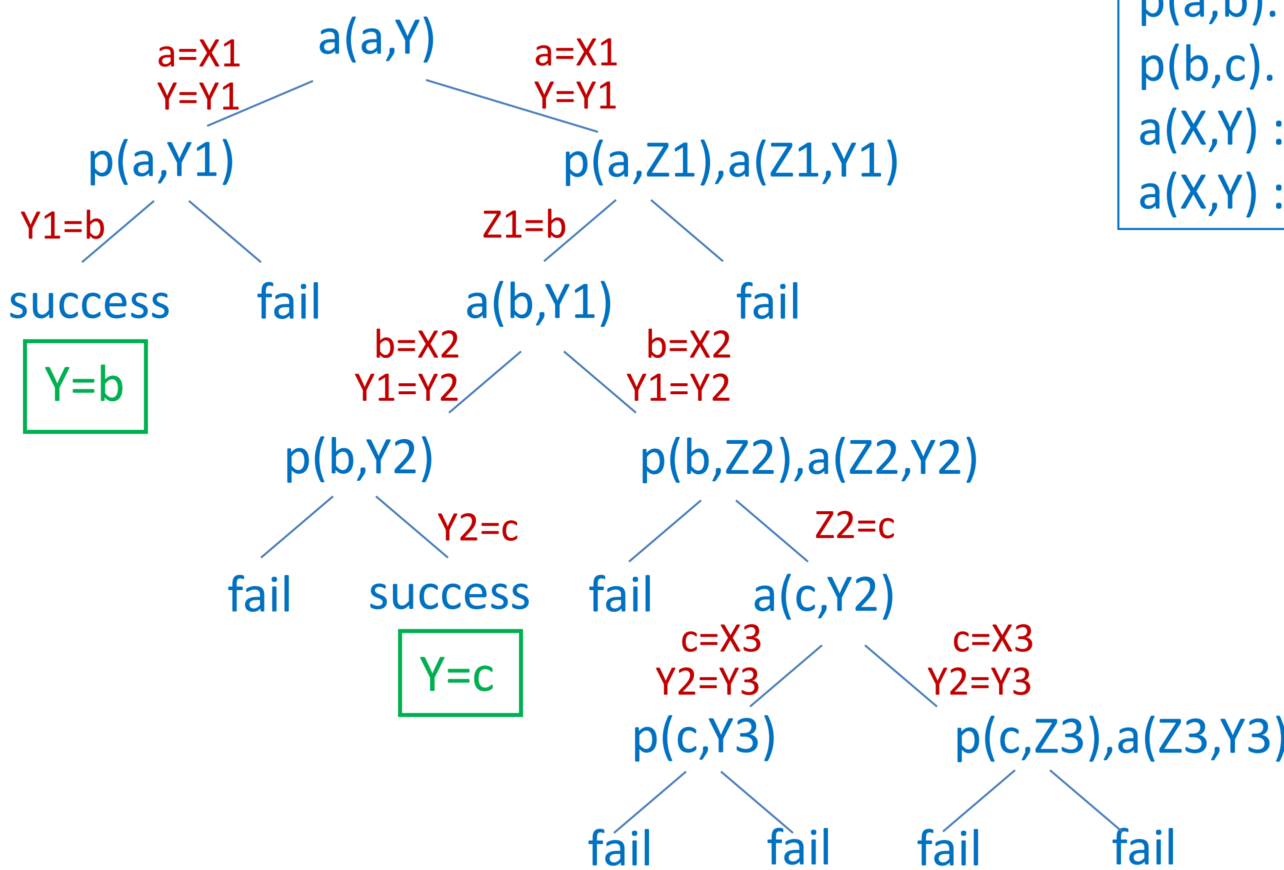
    Read the clauses as procedures

    e.g. ancestor is a procedure name; X and Y are parameters

Prolog's search strategies

- Search tree for the goal a(a,Y)

```
p(a,b).
p(b,c).
a(X,Y) :- p(X,Y).
a(X,Y) :- p(X,Z), a(Z,Y).
```

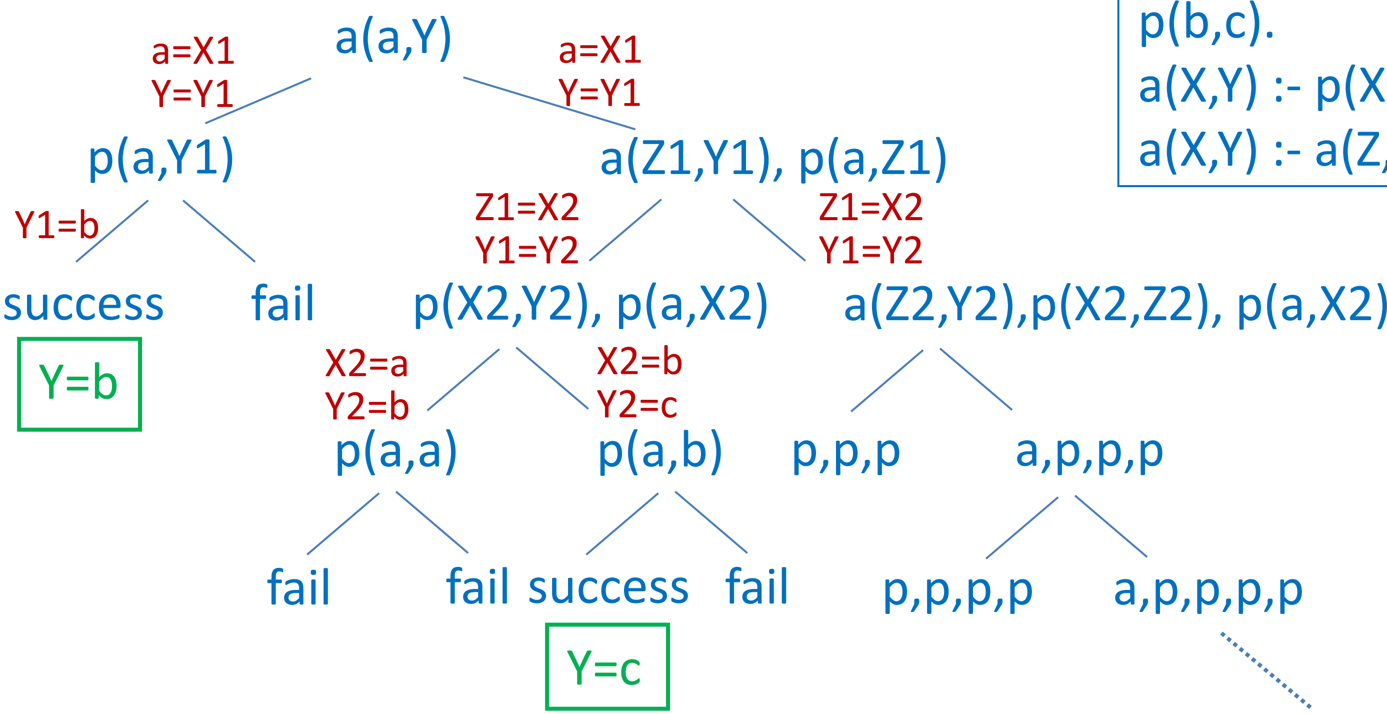


N.B. Clause order determines the order of solutions found.

Prolog’s search strategies

- Search tree for the goal a(a,Y)

```
p(a,b).
p(b,c).
a(X,Y) :- p(X,Y).
a(X,Y) :- a(Z,Y), p(X,Z).
```



N.B. Goal order determines the search tree.

Prolog's search strategies

- Incompleteness
  - Prolog's deduction system is incomplete.
  - Example

```
p(a,b).
p(c,b).
p(X,Y) :- p(Y,X).     % symmetric
p(X,Y) :- p(X,Z), p(Z,Y).     % transitive
```

No matter how the clause order and goal order are arranged, Prolog's depth-first search cannot deduce p(a,c).

N.B. Breadth-first search can deduce p(a,c).

List processing

- Lists

```
[a] = [a|[]]
[a,b] = [a|[b]] = [a|[b|[]]]

?- [X|Xs] = [a,b,c].     % = is the unification operator.
                         % Are the two lists unifiable?
X = a,                   % Yes, they are.
Xs = [b, c].
?- [X,b,c] = [a|Xs].
X = a,
Xs = [b, c].
```

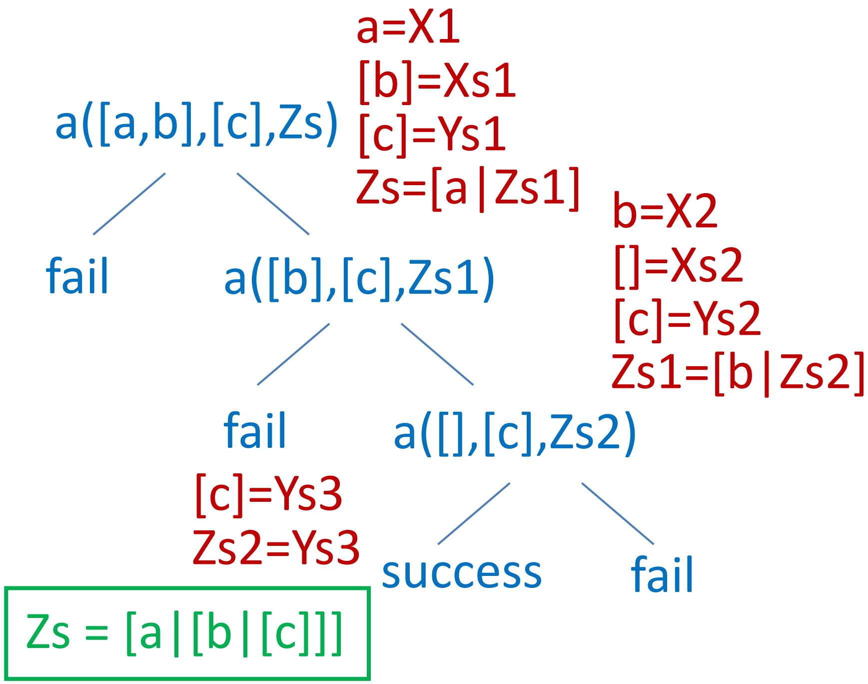
N.B. Unification is two-way pattern matching.

List processing

- Append **relation**

```
% append(Xs,Ys,Zs) ≡ Zs is the result of concatenating Xs and Ys
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).

?- append([a,b],[c],[a,b,c]).
true.
?- append([a,b],[c],Zs).
Zs = [a, b, c].
?- append([a,b],Ys,[a,b,c]).
Ys = [c].
```



List processing

- Append relation

```
?- append(Xs,Ys,[a,b,c]).
Xs = [],
Ys = [a, b, c] ;
Xs = [a],
Ys = [b, c] ;
Xs = [a, b],
Ys = [c] ;
Xs = [a, b, c],
Ys = [] ;
false.
```

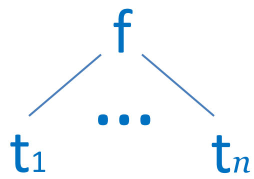
```
?- append(Xs,Ys,Zs).
Xs = [],
Ys = Zs ;
Xs = [_G390],
Zs = [_G390|Ys] ;
Xs = [_G390, _G396]
Zs = [_G390, _G396|Ys]
```

N.B. Data structures may contain logical variables.

N.B. A logic program represents several procedural programs.

Term structure

- Term structure
  - A term $f(t_1,\ldots,t_n)$ is represented as a directed acyclic graph (DAG)



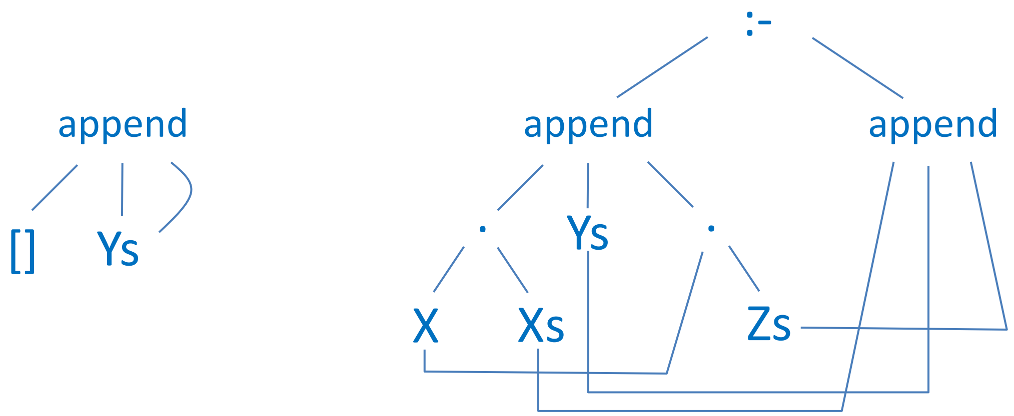
  - The logic operators :- and , are all functors.
  - A list [X|Xs] is a shorthand of the term .(X,Xs), where . is called the dot functor.
  - All occurrences of a variable in a term share the same node in the DAG.

Term structure

- Term structure

  append([],Ys,Ys).

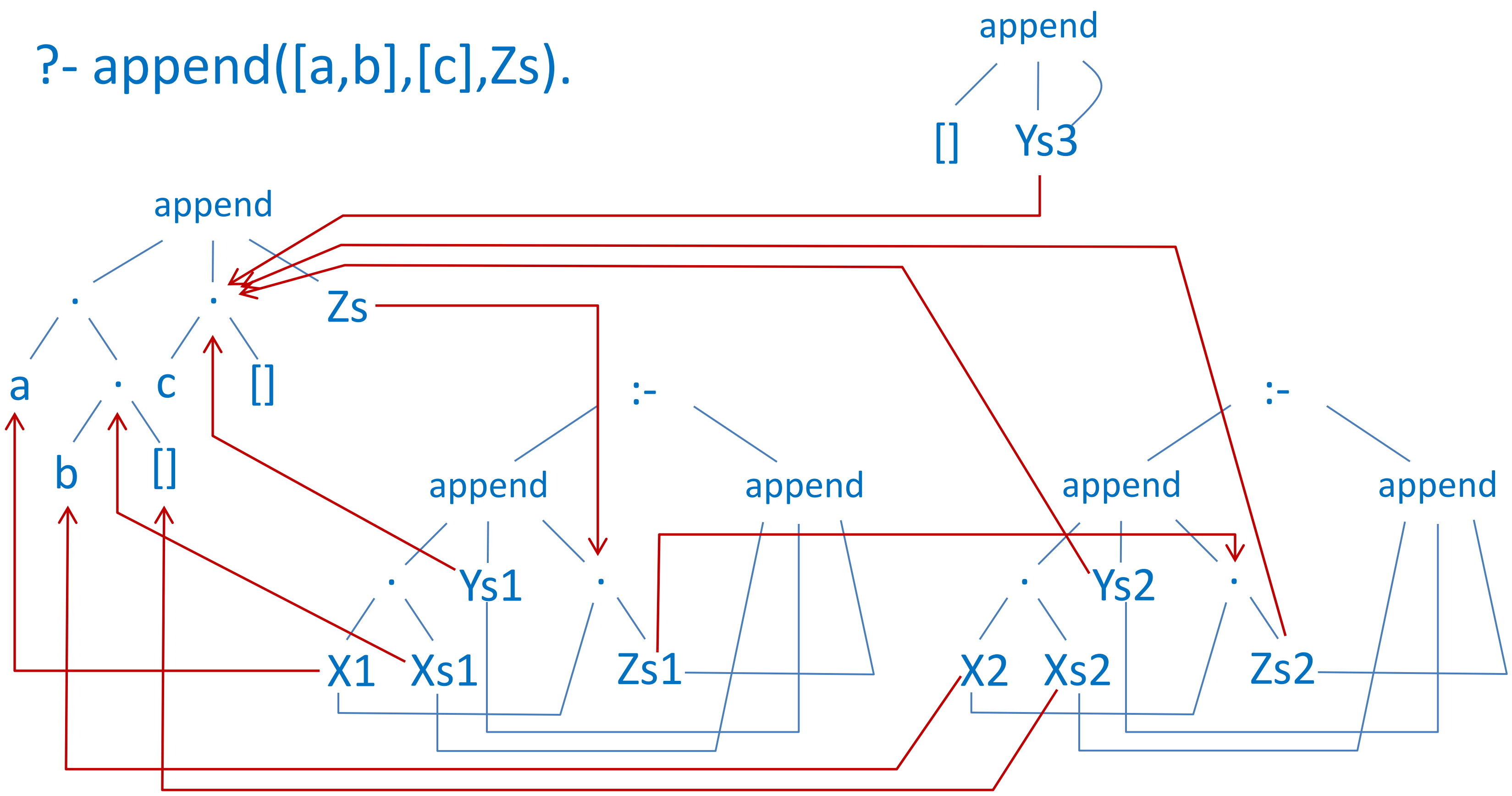
  append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).



Term structure

- Term structure

?- append([a,b],[c],Zs).



Arithmetic

- Arithmetic by reduction
  - The natural numbers are built from the constant 0 and the successor function s. That is, they are represented by
    0, s(0), s(s(0)), s(s(s(0))), …
  - % add(X,Y,Z) ≡ Z is the sum of X and Y

```
add(0,Y,Y).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
?- add(s(s(0)),s(0),Z).          % Z = s(s(s(0)))
?- add(s(s(0)),Y,s(s(s(0)))).    % Y = s(0)
?- add(X,Y,s(s(0))).             % X=0,Y=s(s(0));
                                 % X=s(0), Y=s(0); X=s(s(0)), Y=0
```

Arithmetic

- Arithmetic by evaluation
  - For efficiency reason, Prolog does arithmetic via evaluation

    i.e. using operators such as

    comparison operators < > =< >= =:= =\=

    arithmetic operators + - * / mod
- The is operator
  - X is Y % Unify X with the value of arithmetic expression Y

    It isn't the assignment operator.

    ?- N is 5, N is N-1.
    false.

    ?- N is 5, M is N-1.
    N = 5,
    M = 4.

Arithmetic

- Example

% is	% unification	% equality
?- X is 2+3.	?- X = 2+3.	?- X =:= 2+3.
X = 5.	X = 2+3.	error: X isn't instantiated
?- 5 is 2+3.	?- 5 = 2+3.	?- 5 =:= 2+3.
true.	false.	true.
?- 2+3 is 2+3.	?- 2+3 = 2+3.	?- 2+3 =:= 2+3.
false.	true.	true.

N.B. System predicates will not be retried on backtracking

Arithmetic

- Factorial relation

If (n==0) return 1;
else if (n>0) return n*f(n-1);

```
% f(N,F) ≡ F = N!
f(0,1).
f(N,F) :- N>0, N1 is N-1, f(N1,F1), F is N*F1.
```

?- f(3,F).	?- f(N,6).
F = 6 ;	Error: N isn't instantiated.
false.	

The condition N>0 is necessary for preventing goals with non-positive N from being tried. In particular, it prevents the goal f(0,1) or f(0,2) from being tried on backtracking.

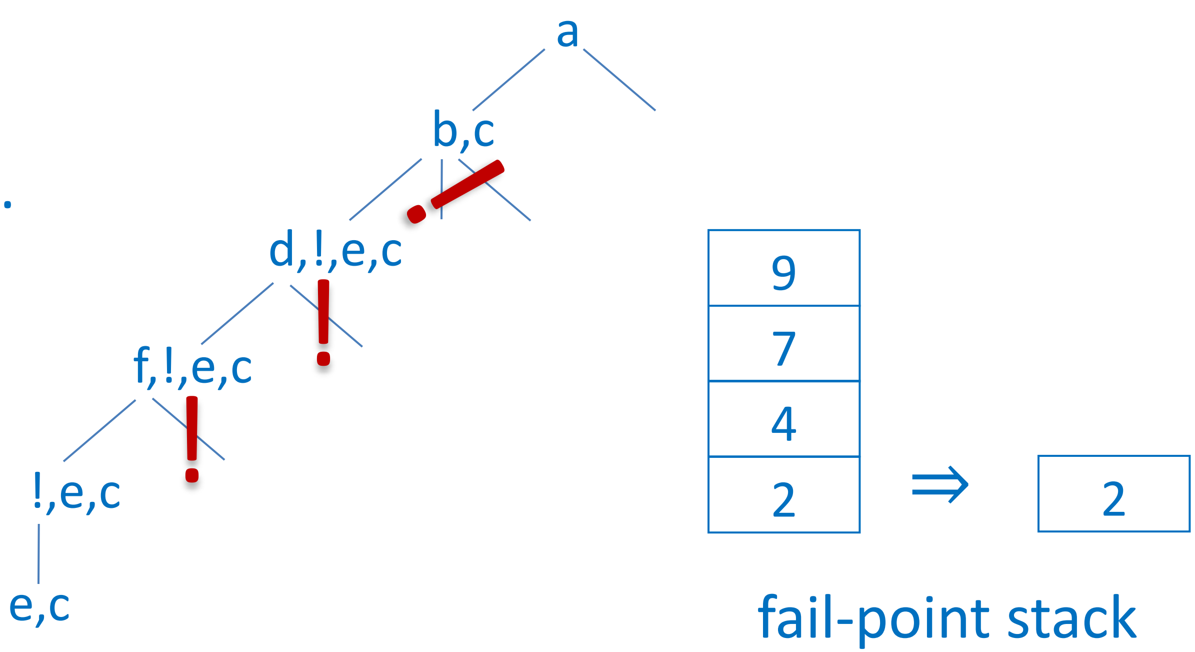
Cut

- The cut operator !

  $\underbrace{H :- B_1, \ldots,}\ !, \ldots, B_n$

  All these goals, including their subgoals, won’t be backtracked.

  1 a :- b, c.
  2 a :- …
  3 b :- d, !, e.
  4 b :- …
  5 b :- …
  6 d :- f.
  7 d :- …
  8 f.
  9 f : - …



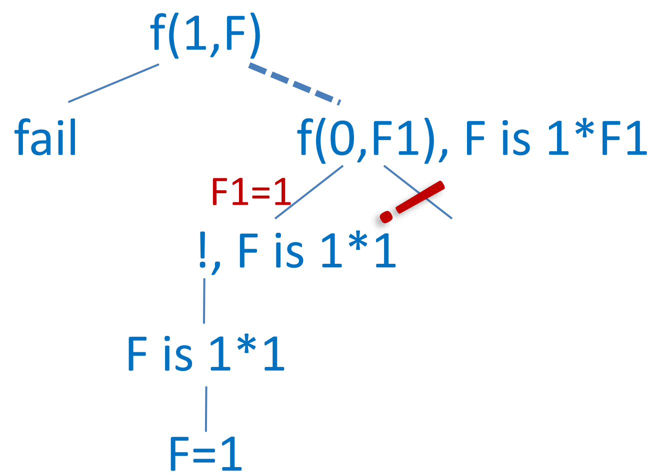
Cut

- Example

  % f(N,F) ≡ F = N!

  f(0,1) :- !.   % confirm the choice

  f(N,F) :- N>0, N1 is N-1, f(N1,F1), F is N*F1.

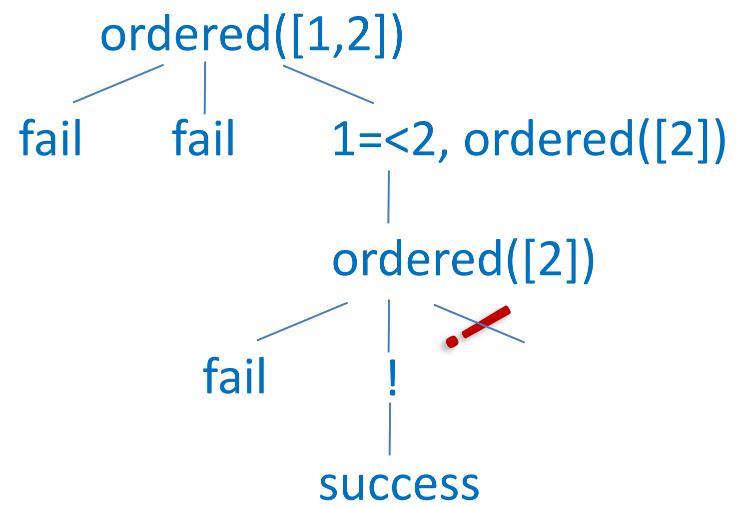


  The condition N>0 is still needed to terminate goals like
  ?- f(0,2).

Cut

- Example

```
% ordered(Xs) ≡ Xs is an ordered list
ordered([]) :- !.                % confirm the choice
ordered([_]) :- !.               % confirm the choice
ordered([X,Y|Xs]) :- X =< Y, ordered([Y|Xs]).
```



Cut

- Example – Bubble sort

```
% bsort(Xs,Ys) ≡ Ys is an ordered permutation of Xs
bsort(Xs,Xs) :- ordered(Xs), !.
bsort(Xs,Ys) :- append(As,[X,Y|Bs],Xs),      % generate
                X > Y,                        % test
                !,           % terminate generate-and-test
                append(As,[Y,X|Bs],Xs1),
                bsort(Xs1,Ys).
```

bsort([1,4,3,2],Ys) → bsort([1,3,4,2],Ys) → bsort([1,3,2,4],Ys) → bsort([1,2,3,4],Ys)