**Syntax - 1**

happy(yolanda).

listens2music(mia).

listens2music(yolanda):- happy(yolanda).

playsAirGuitar(mia):- listens2music(mia).

playsAirGuitar(yolanda):- listens2music(yolanda).

=> 5 clauses

=> 2 factsf

=> 3 rules

=> predicates = listens2music, happy, playsAirGuitar

=> , (and)

=> ; (or)

=> :- (if)

=> . (end of clause)

Terms

/ \

Complex Terms Simple Terms

/ \

Constants Variables

/ \

Atoms Numbers

Atoms:

- start with lowercase

Variables:

- start with uppercase

**Arity - 1**

=> happy(yolanda). Happy is arity 1 (happy/1)

**Unification - 2**

1. If T1 and T2 are constants, then

T1 and T2 unify if they are the same atom, or the same number.

2. If T1 is a variable and T2 is any type of term, then T1 and T2 unify, and T1 is instantiated to T2. (and vice versa)

3. If T1 and T2 are complex terms then they unify if:

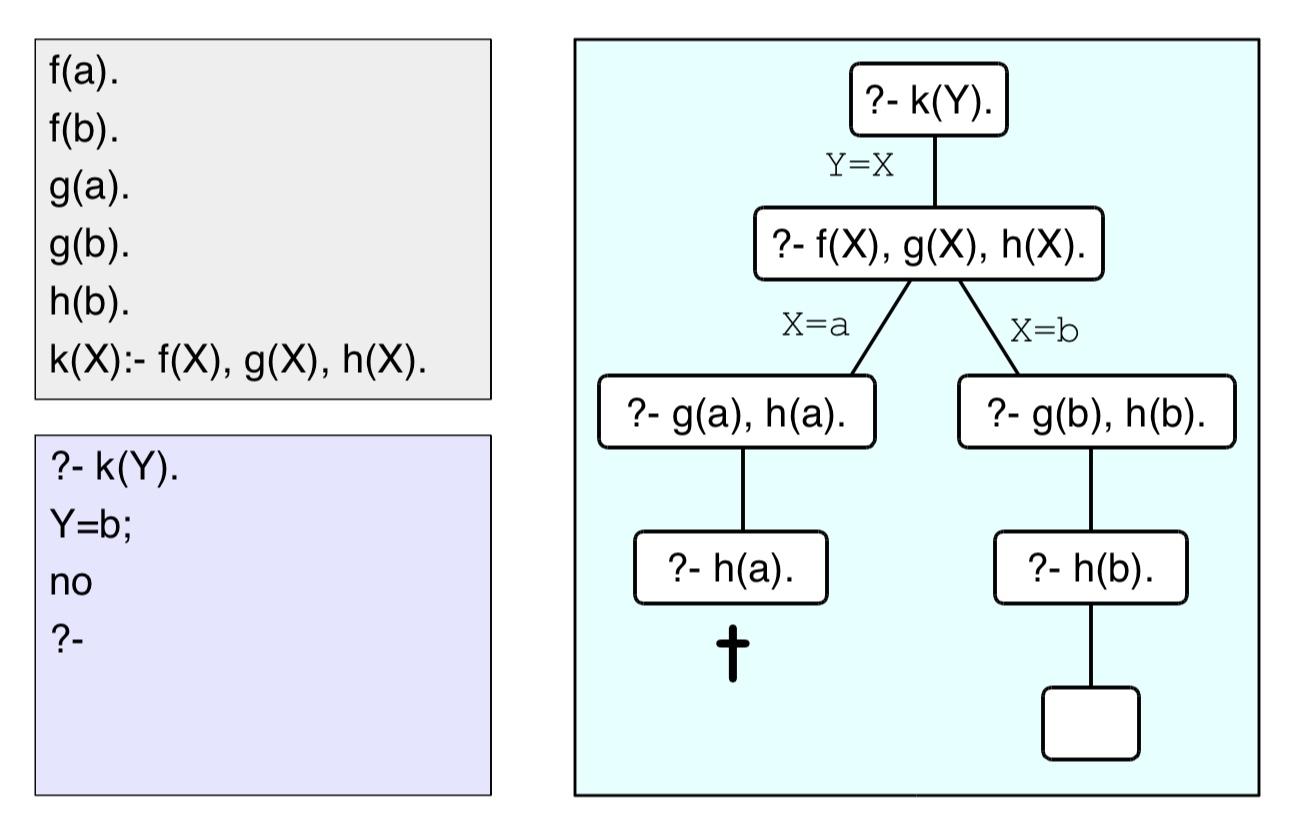
* a) They have the same functor and arity, and
* b) all their corresponding arguments unify, and
* c) the variable instantiations are compatible.

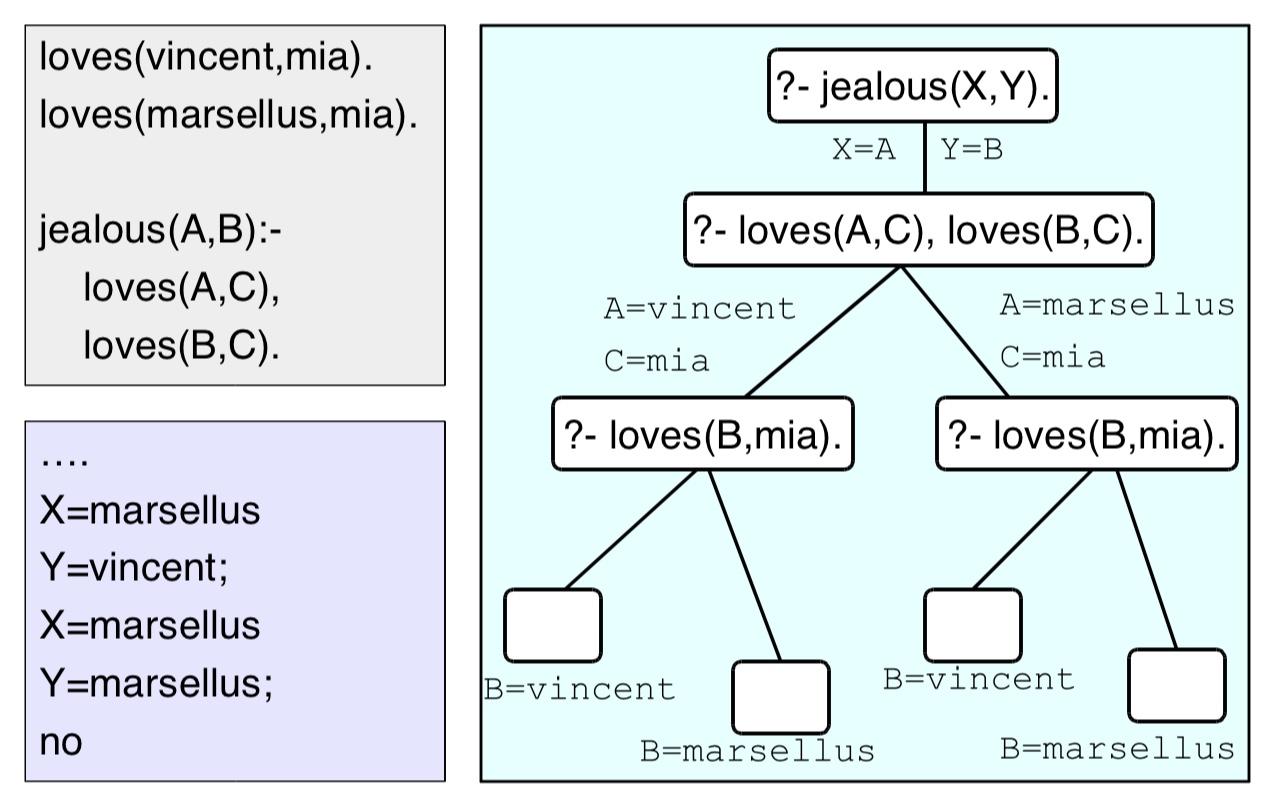
=> =/2 used to test unification

=> X=mia, X=vincent

=> Returns no as X already instantiated to mia

**Search Trees - 2**

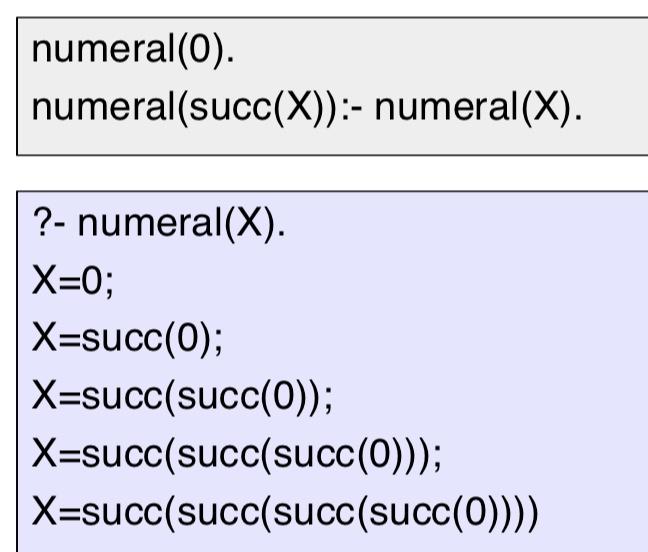
****

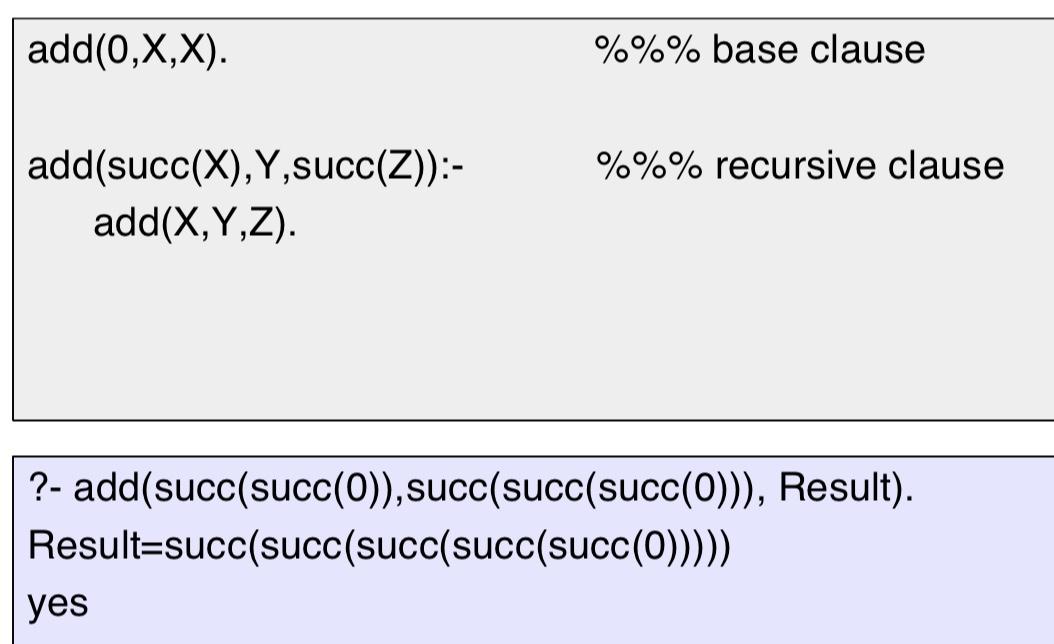
****

**Recursion - 3**

=> Predicates can be defined recursively

isDigesting(X,Y):- justAte(X,Y). isDigesting(X,Y):- justAte(X,Z), isDigesting(Z,Y).





=> Prolog searches knowledge bases from top to bottom, processes clauses from left to right

**Lists - 4**

=> Can contain anything

=> Lists have head (1st item) and tail (everything else)

=> Tail is always a list (may be empty list [ ])

=> Empty list has neither head nor tail

=> | operator splits list into head/tail

?- [Head|Tail] = [mia, vincent, jules, yolanda].

Head = mia

Tail = [vincent,jules,yolanda]

Yes

?- [X1,X2,X3,X4|Tail] = [mia, vincent, marsellus, jody, yolanda].

X1 = mia

X2 = vincent

X3 = marsellus

X4 = jody

Tail = [yolanda] yes

**Anonymous Variable - 4**

=> \_ not interested in what value it would be instantiated to

=> It is independent, not bound to anything

?- [ \_,X2, \_,X4|\_ ] = [mia, vincent, marsellus, jody, yolanda].

X2 = vincent

X4 = jody

yes

**Member predicate - 4**

=> Approach defining predicates by looking at the base case

=> Usually something to do with empty list

member(X,[X|T]).

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

member/2

?- member(vincent,[yolanda,trudy,vincent,jules]).

yes

**Arithmetic - 5**

=> +, -, \*, / are ordinary prolog terms, which don’t carry out arithmetic

?- X = 3 + 2.

X = 3+2

yes=0

=> The is/2 predicate forces arithmetic evaluation

?- 4 is 2+3.

no

?- X is 3 ∗ 4. X=12

yes

=> Some restrictions

=> Variables must be on the RHS

=> 3+2 is the same as +(3,2)

?- 3 + 2 is X.

ERROR: is/2: Arguments are not sufficiently instantiated

**Accumulators** **- 5**

=> Finding the length of a list as an example

Without Accumulator

=> Length of the list defined as empty list being length 0, non empty list length 1 + length of tail

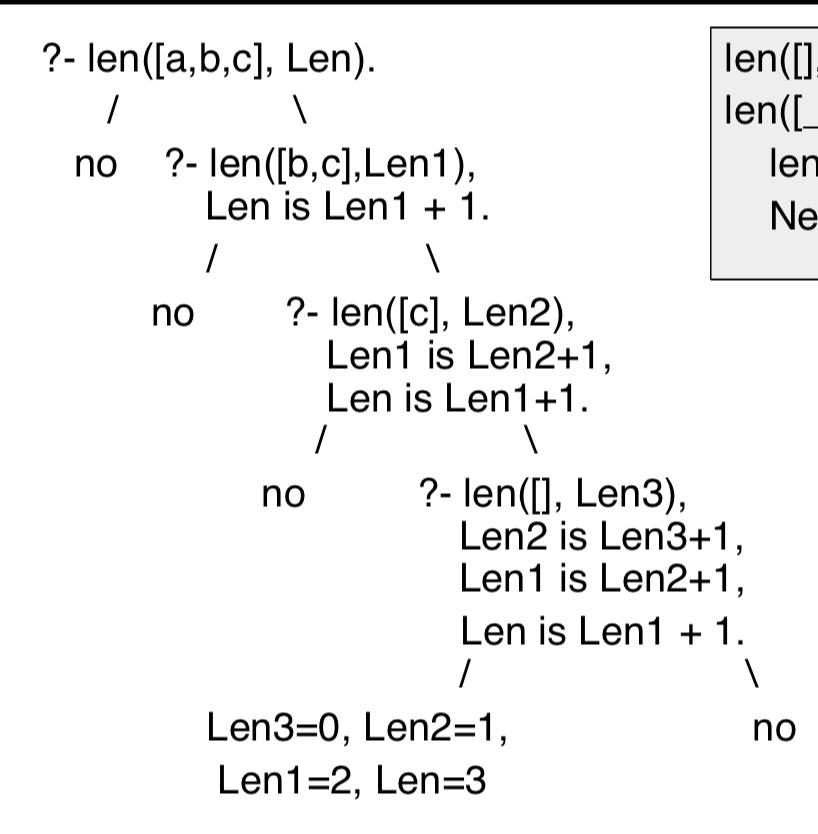
len([],0).

len([\_|L],NewLength):-

len(L,Length),

NewLength is Length + 1

=> Search tree



With Accumulator

=> Said to be tail recursive

=> Means the result is fully calculated once we get to the base case

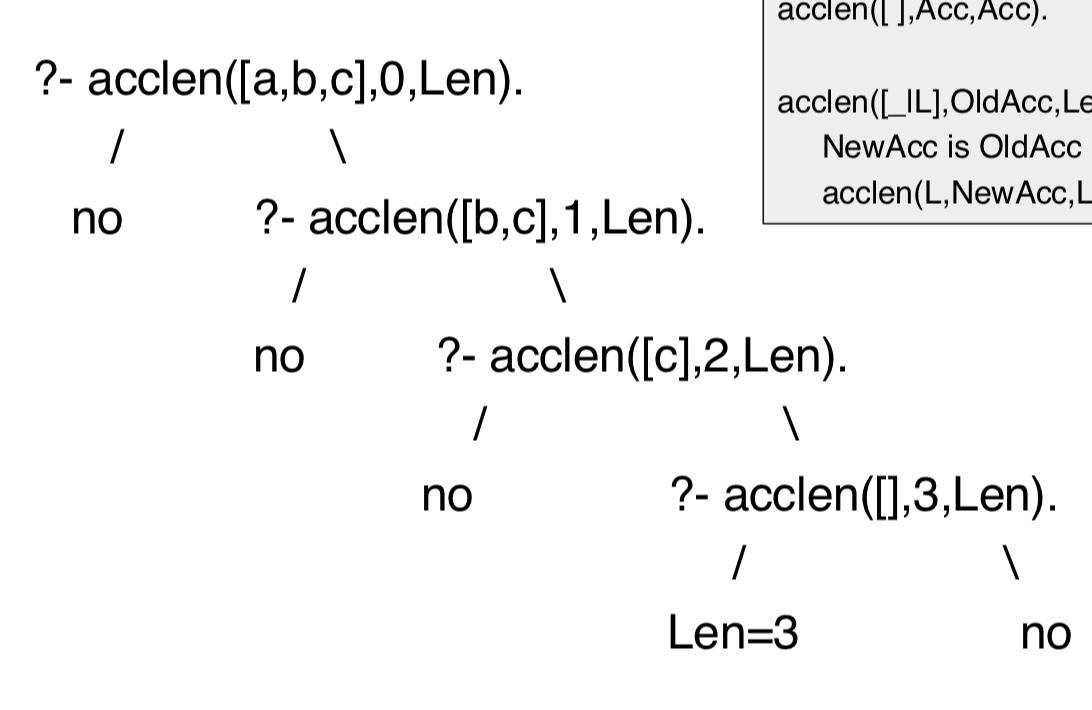
=> Without tail recursion, there are still goals on the stack when we reach the base case

acclen([],Acc,Acc).

acclen([\_|L],OldAcc,Length):-

NewAcc is OldAcc + 1,

acclen(L,NewAcc,Length).



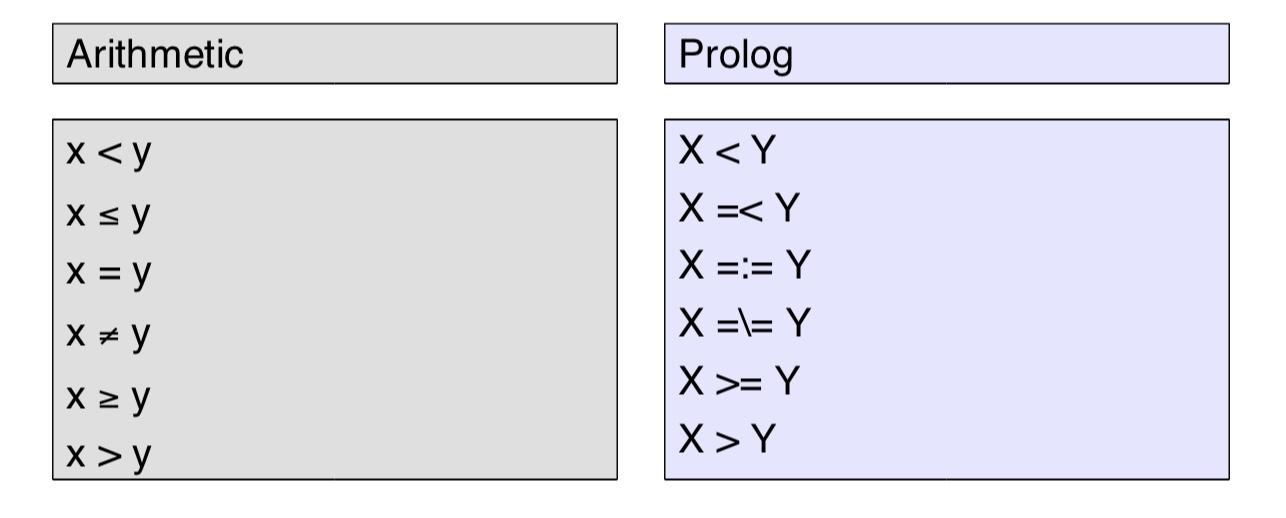
=> Wrapper predicate can be used:

length(List,Length):-

acclen(List,0,Length).

**Comparing Numbers - 5**

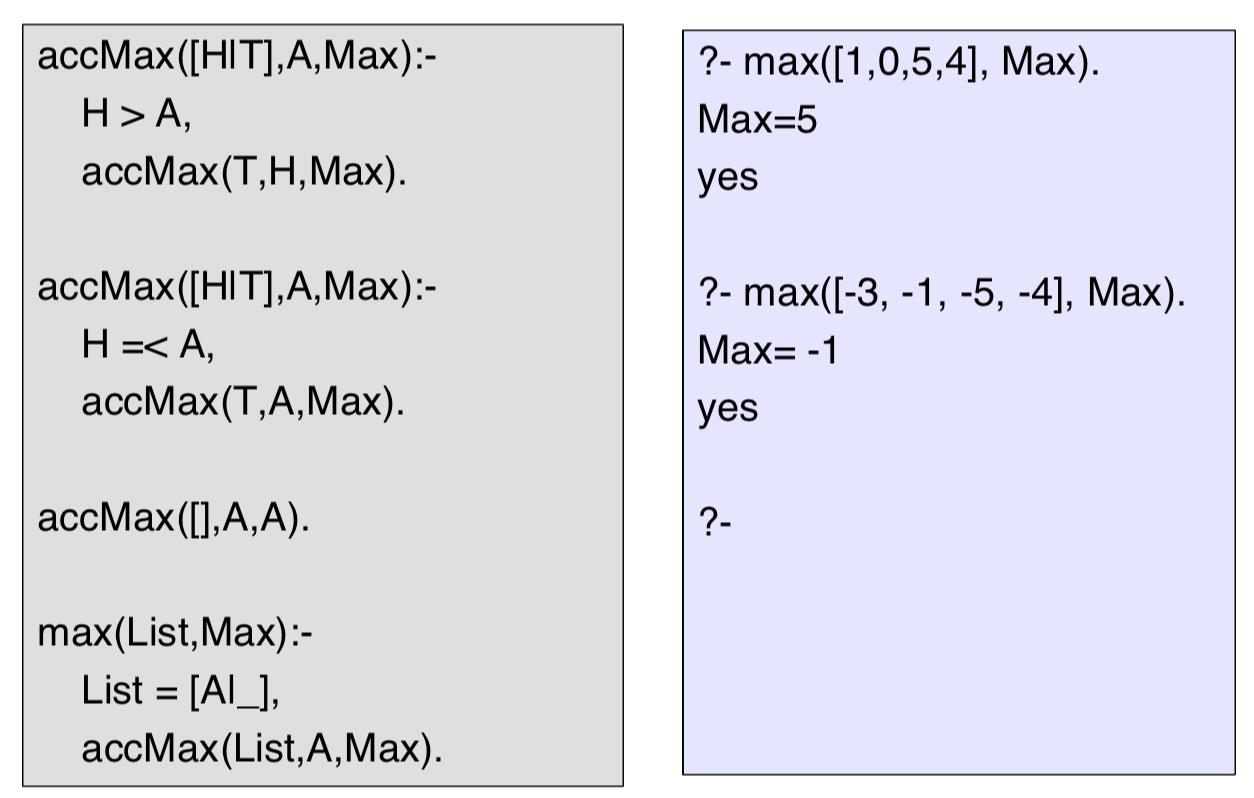
=> Force both RHS and LHS args to be evaluated



=> Example is max element in a list

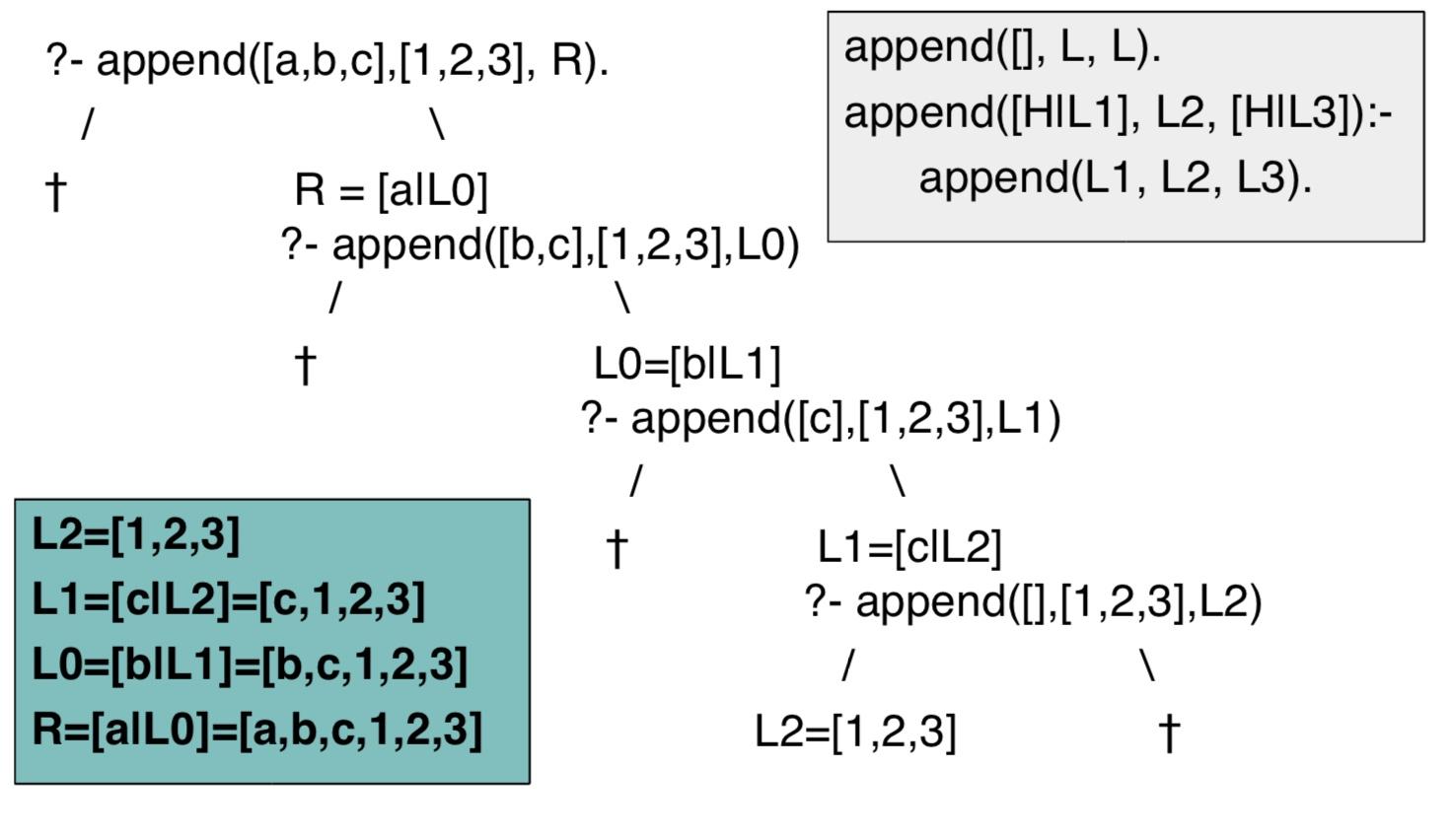
=> Idea is to use an accumulator, which keeps track of highest value

=> If a higher value is found, acc updated



**Append - 6**

=> Definition of append, appending to empty list returns the same list, concatenating a non empty list [H|T] with list L is a list with head H and the result of concatenating T with L



=> Since append recurses the 1st argument, it is more efficient to call it with the 1st argument as shortest list

**Reversing List - 6**

Naive Reverse

=> example list = [a,b,c,d]

1. If we reverse the empty list, we obtain the empty list

2. If we reverse the list [H|T], we end up with the list obtained by reversing T and concatenating it with [H]

=> If we reverse the tail of this list we get [d,c,b].

=> Concatenating this with [a] yields [d,c,b,a]

naiveReverse([],[]).

naiveReverse([H|T],R):-

naiveReverse(T,RT),

append(RT,[H],R).

Reverse using accumulator

accReverse([],L,L). // Unify rev and acc

accReverse([H|T],Acc,Rev):-

accReverse(T,[H|Acc],Rev).

reverse(L1,L2):-

accReverse(L1,[],L2).

**DCGs - 7**

=> Non terminals correspond to grammatical categories (ie noun)

=> Terminals correspond to lexical items (words)

=> Parse trees represent the syntactic structure of a string

=> A string is grammatical(with respect to a grammar) if a parse tree can be constructed from the grammar, otherwise ungrammatical(a recogniser is a program which does this)

=> A parser does the same but also produces a parse tree

=> Context-free language is generated by a context free grammar

CFG Recogniser

=> Difference lists

s(A-C):- np(A-B), vp(B-C).

np(A-C):- det(A-B), n(B-C). // noun phrase

vp(A-C):- v(A-B), np(B-C). // verb phrase

vp(A-C):- v(A-C).

det([the|W]-W).

det([a|W]-W).

n([man|W]-W).

n([woman|W]-W).

v([shoots|W]-W).

?- s([the,man,shoots,a,man]-[ ]).

yes

?- s(X-[ ]).

S = [the,man,shoots,the,man];

S = [the,man,shoots,a,man];

....

DCG Example

s --> np, vp.

np --> det, n.

vp --> v, np.

vp --> v.

det --> [the].

det --> [a].

n --> [man].

n --> [woman].

v --> [shoots].

?- s([a,man,shoots,a,woman],[ ]).

yes

?- s(X,[ ]).

S = [the,man,shoots,the,man];

S = [the,man,shoots,a,man];

....

=> Rules can be defined recursively, but should be without left-recursion

✘ s --> s, conj, s. ✔s --> simple\_s, conj, s.

✘ s --> np, vp. ✔s --> simple\_s.

✔simple\_s --> np, vp.

Formal DCG Example

s --> [].

s --> l,s,r.

l --> [a].

r --> [b].

?- s([a,a,a,b,b,b],[ ]).

yes

?- s([a,a,a,a,b,b,b],[ ]).

No

?- s(X,[ ]).

X = [ ];

X = [a,b];

X = [a,a,b,b];

X = [a,a,a,b,b,b] ....

**DCG Extra Arguments - 8**

=> For example: he/him, she/her are both pronouns, the string ‘her shoots she’ is accepted.

=> Need to differentiate between subject/object pronouns

=> Can use extra arguments in the rules to prevent this

s --> np(subject), vp.

np(\_) --> det, n.

np(X) --> pro(X).

vp --> v, np(object).

vp --> v.

det --> [the].

det --> [a].

n --> [woman].

n --> [man].

v --> [shoots].

pro(subject) --> [he].

pro(subject) --> [she].

pro(object) --> [him].

pro(object) --> [her].

=>

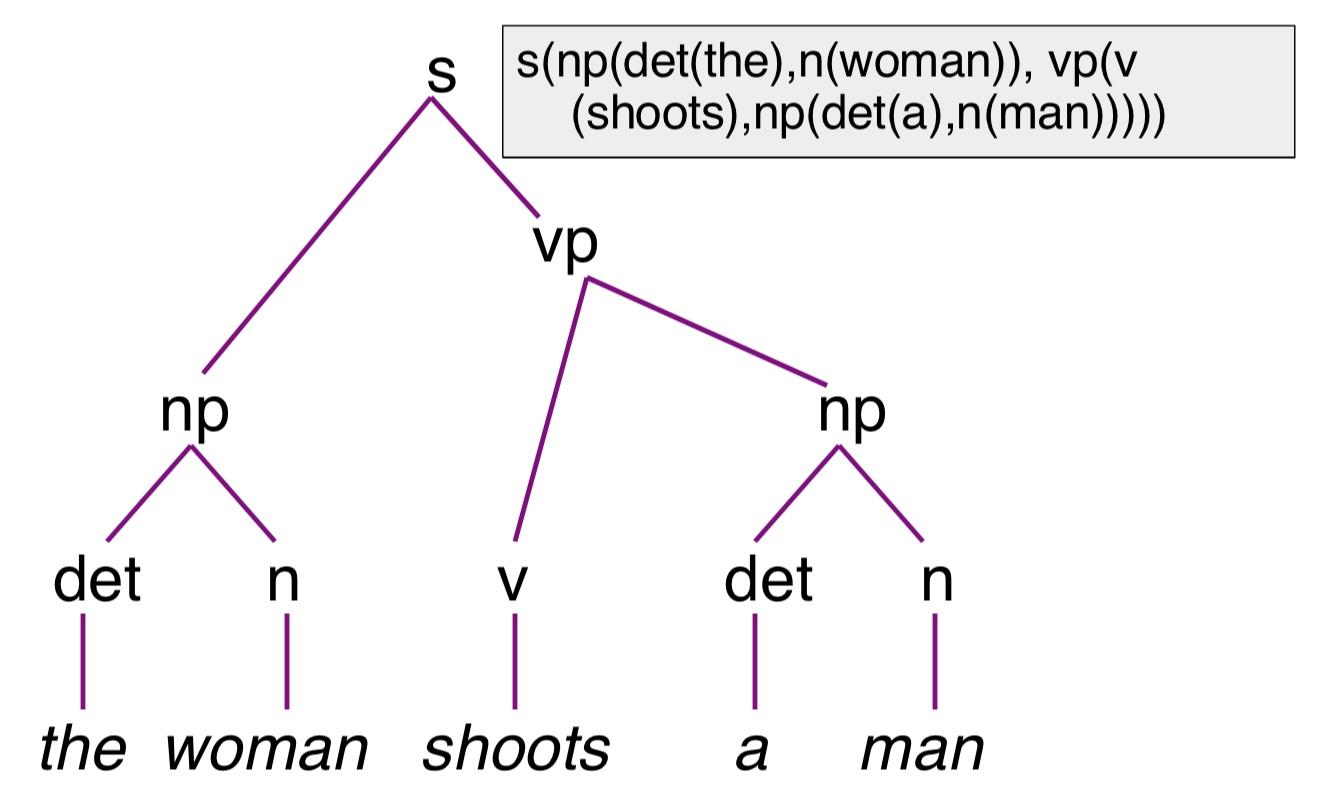
s --> np(subject),vp.

is the same as

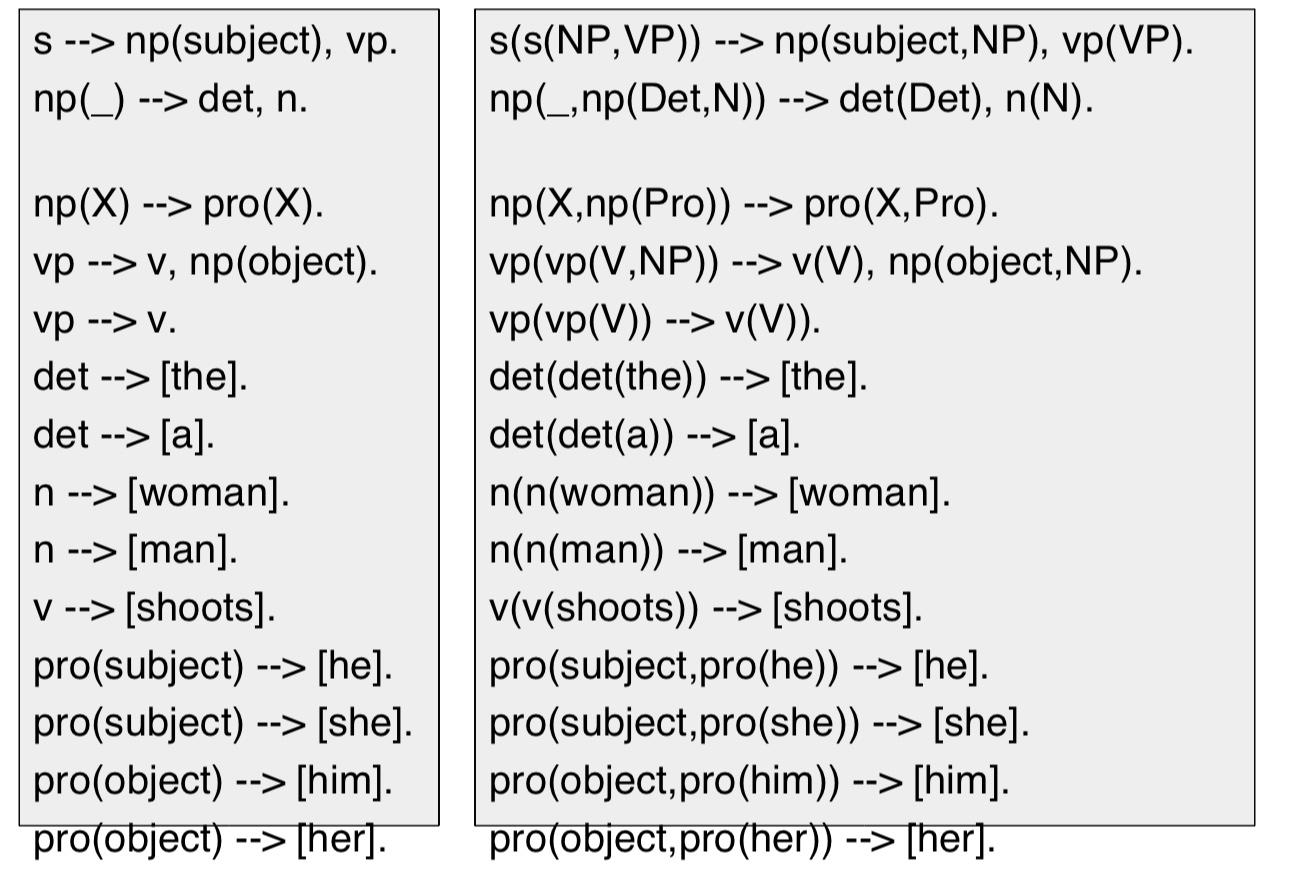
s(A,B):- np(subject,A,C), vp(C,B).

**Building Parse Trees - 8**

=> Result we want



=> Modification to grammar, return the rule as an extra argument, ignoring other extra arguments at top level rule (ie subject/object ignored in ‘s’)



**Using Extra Arguments With Formal Languages - 8**

=> Using anbncn \ {ε} as an example

=> Not a context free language as we need to know the number (n) of a’s,b’s,c’s to produce

s(Count) --> as(Count), bc(Count), cs(Count). // Count unifies

as(0) --> [].

as(succ(Count)) --> [a], as(Count).

bs(0) --> [].

bs(succ(Count)) --> [b], bs(Count).

cs(0) --> [].

cs(succ(Count)) --> [c], cs(Count).

=> Can also use any prolog predicate in { }

s(Count) --> as(Count), bc(Count), cs(Count).

as(0) --> [].

as(NewCnt) --> [a], as(Cnt), {NewCnt is Cnt + 1}.

bs(0) --> [].

bs(NewCnt) --> [b], bs(Cnt), {NewCnt is Cnt + 1}.

cs(0) --> [].

cs(NewCnt) --> [c], cs(Cnt), {NewCnt is Cnt + 1}.

**Comparing Terms - 9**

=> Using the ==/2 predicate

=> 2 different uninstantiated variables returns false

=> 2 same uninstantiated returns true

=> Variables instantiated with term T are identical to T

?- a=U, a==U.

true.

?- X==Y.

false.

=> Prolog regards a and ‘a’ to be the same

?- a==’a’

true.

=> Using the \==/2 predicate

=> Succeeds in all cases where ==/2 fails

**Comparing lists - 9**

?- [a,b,c,d] == [a,b,c|[d]]

true.

?- [a,b,c,d] == [a,b,c,d|[]]

True.

=> The internal list structure:

* Empty list []
* Non empty list .(term, list),

=> Examples:

?- .(a, []) == [a]. true

?- .(f(d,e)) == [f(d,e]. true

?- .(a, .(b, .(c, []))) == [a,b,c]. true

=> Can be read as a tree example:

?- .(a, .(.(b, .(c, [])), .(d, []))) == [a, [b, c], d]. true

.

/ \

A .

/ \

. .

/ \ / \

b . d []

/ \

c []

**Checking Type of a Term - 9**

atom/1:

?- atom(a). true.

?- atom(7). false.

?- atom(X). False.

integer/1, float/1

number/1 (either float or integer)

atomic/1 (a constant)

?- X=a, atom(X). true.

?- atom(X), X=a. false.

var/1 (uninstantiated variable)

nonvar/1 (instantiated variable)

?- nonvar(X). false.

?- nonvar(mia). true.

?- nonvar(23). true.

**Structure of Terms - 9**

=> Want to know functor, arity, arguments

=> User functor/3 predicate

?- functor(friends(lou, andy), F, A).

F = friends.

A = 2.

true.

?- functor([a,b,c], F, A).

F = .

A = 2

true.

?- functor(mia, F, A).

F = mia.

A = 0.

true.

=> arg/3 returns the argument at position n (starting at 1)

?- arg(2, likes(lou, andy), A).

A = andy.

true.

**Cuts - 10**

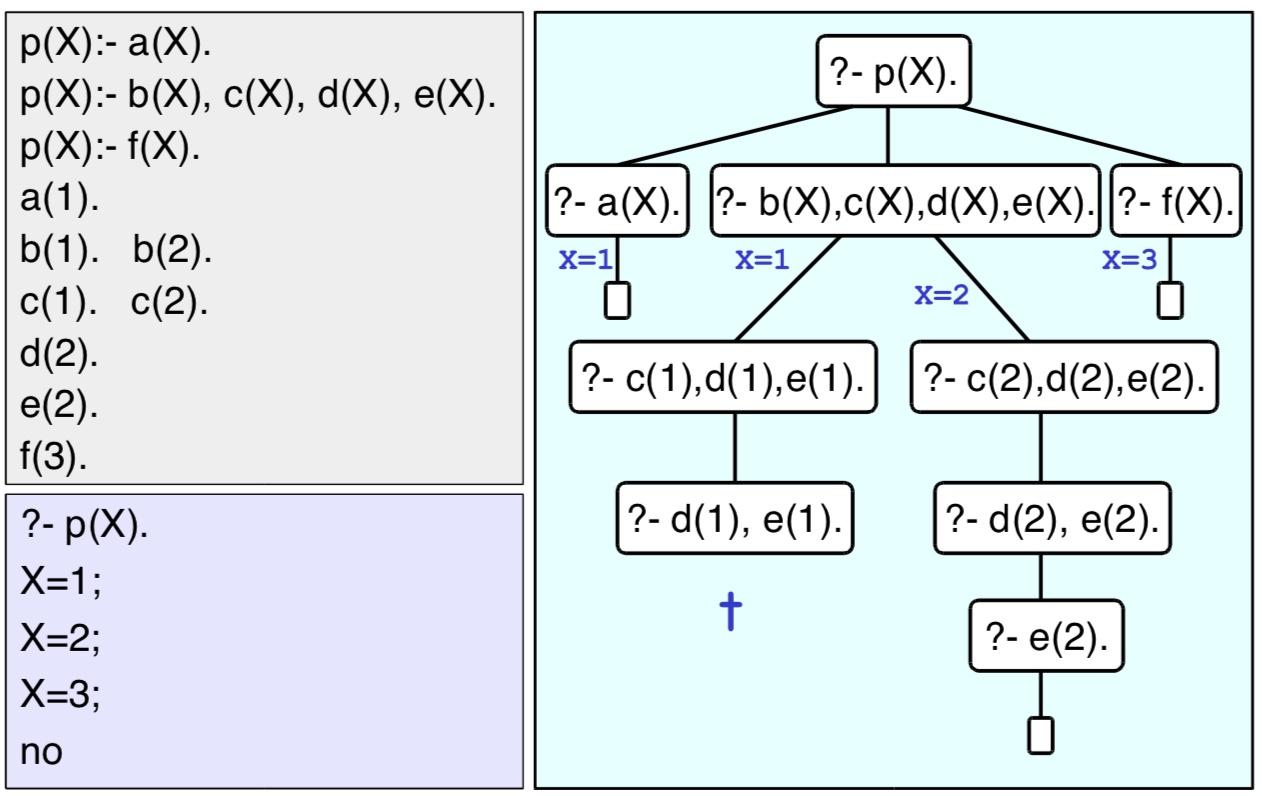
=> Cut predicate !/0

=> Controls backtracking, useful as prolog wastes time exploring all possibilities

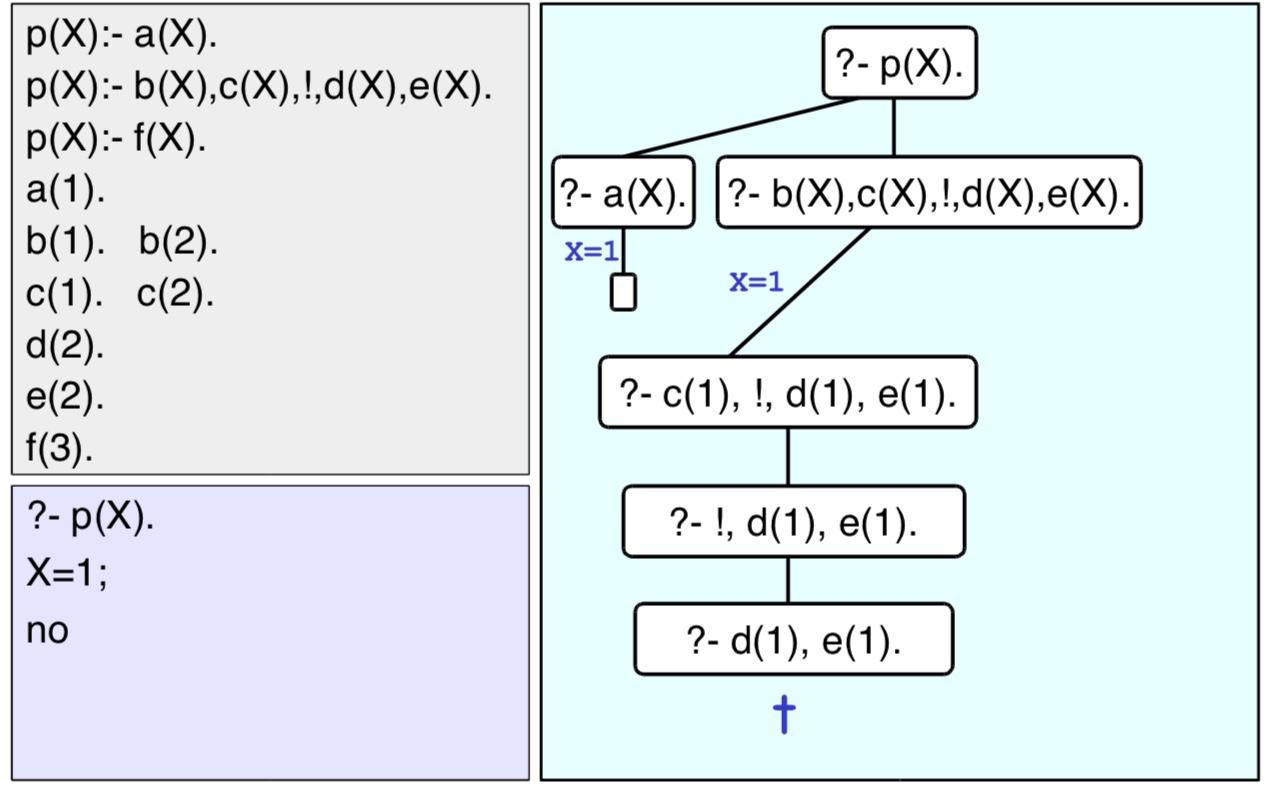
=> Cuts always succeed

=> Cuts commit the choices made on the left hand side of the cut

Cut Free

`

With Cuts



**Green Cuts - 10**

=> Example: max/3 succeeds if arg 3 is the max of arg 1, 2

max(X,Y,Y):-

X =< Y.

max(X,Y,X):-

X>Y.

?- max(2,3,3).

yes

?- max(7,3,7).

yes

?- max(2,3,2).

no

?- max(2,3,5).

no

* Inefficient, if ?- max(3,4,Y). is called, it will correctly unify with Y=4, but will look for more solutions if asked.
* Pointless
* Modified version

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

max(X,Y,X):- X>Y.

* If first clause succeeds, the cut commits this choice, second clause not considered
* If first clause fails, second clause is considered
* Green cut: the meaning of the predicate does not change

**Red Cuts - 10**

=> Same predicate

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

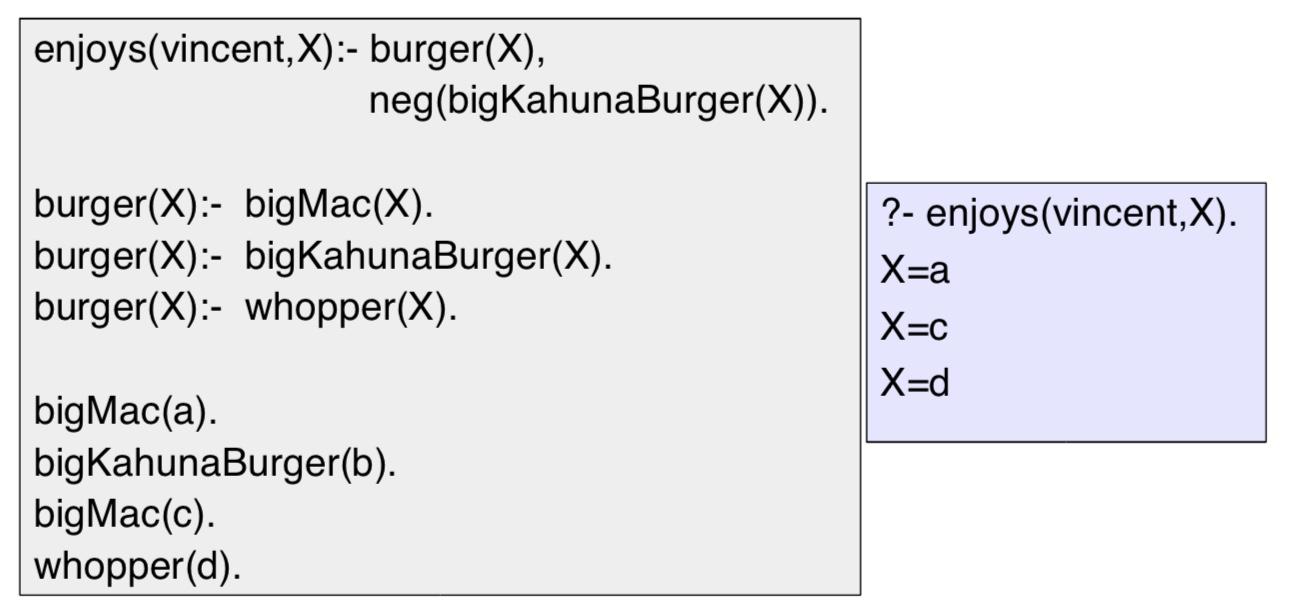
max(X,Y,X).

* Red cut, as if we take the cut out, we do not get an equivalent program

**Negation As Failure - 10**

neg(Goal):- Goal, !, fail.

neg(Goal).



=> \+ predicate can also be used

enjoys(vincent,X):- burger(X),

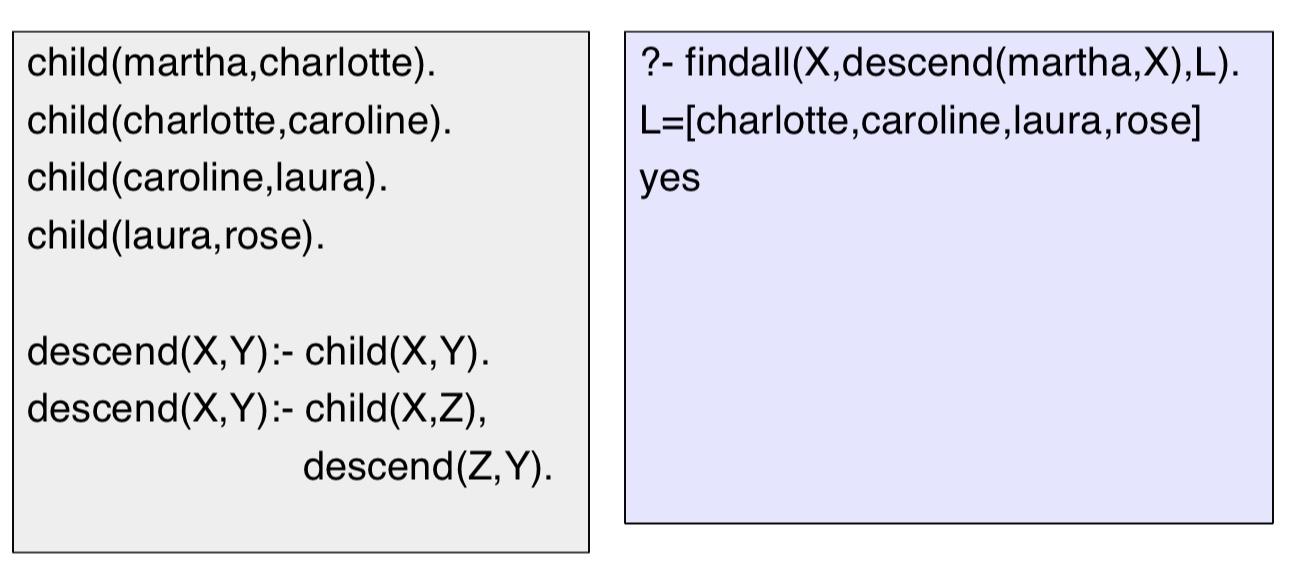
\+ bigKahunaBurger(X).

**Built in predicates**

findall/3

=> ?- findall(O,G,L).

=> Produces a list L of all the objects O that satisfy the goal G



bagof/3

=> ?- bagof(O,G,L).

=>

setof/3