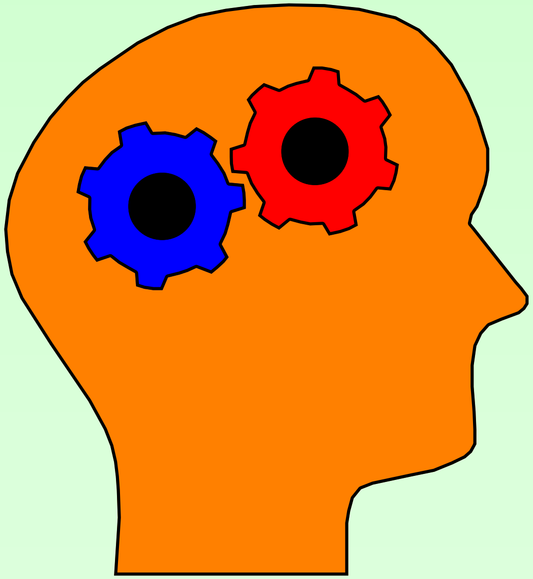
CS2104: Programming Languages Concepts

Lecture 8-9 : **Prolog & CLP**



"Logic, Relational and Comstraint Programming"

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Prolog – Some Highlights

- Atoms, Variables & Terms
- Relations and Clauses
- Unification
- List Manipulation
- Arithmetic
- Backtracking, Cuts, Negation
- (Finite) Constraint Solving

Reference --- An Introduction to Prolog Programming
http://staff.science.uva.nl/~ulle/teaching/prolog/prolog.pdf

Atoms, Terms and Variables

- Atoms are constants (starts with lower-case letter).

  ```
  cat, neil, john, 5, -1, mary, car
  ```

- Variables start with *upper-case* letter or *underscore*

  ```
  X, Y, Y2, Result, _var,  _1,
  ```

- Terms are used to form tree-like data structures:

  ```
  node(node(dog,nil),leaf(cat)),
  cons(2,nil), cons(cat,cons(1,nil))
  ```

- Can mix terms with variables.

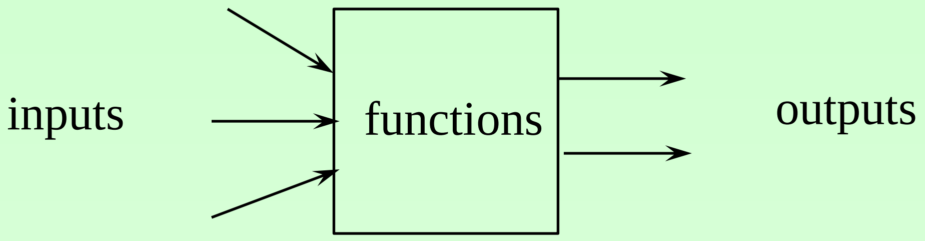
  ```
  node(X,Y), node(V,V), cons(2,T), cons(H,T)
  ```

- Untyped language.

Relations vs Functions

- Prolog allows *relations* to be specified.

- This is facilitated by unification mechanism

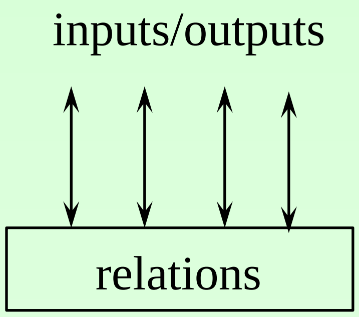


inputs

functions

outputs

C, Java, Haskell



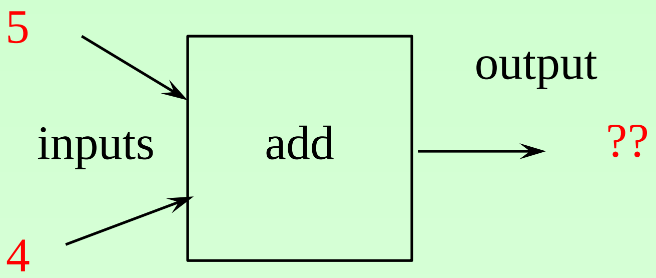
inputs/outputs

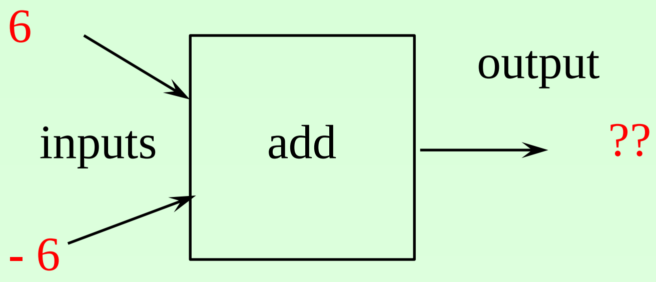
relations

Prolog

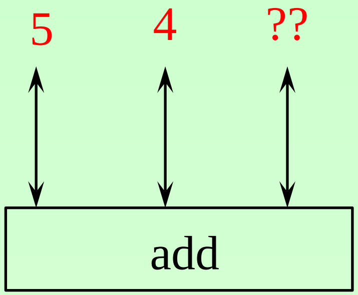
Addition as a Function

- Let us illustrate addition as a function.

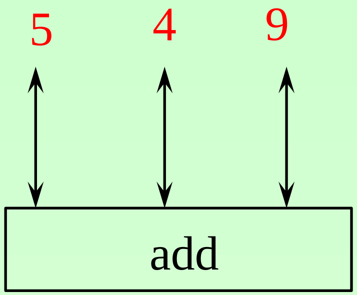




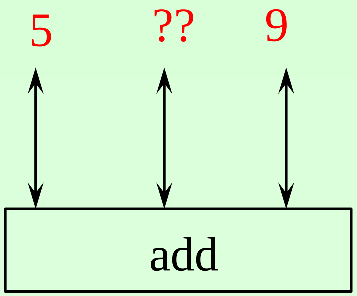
Addition as a Relation



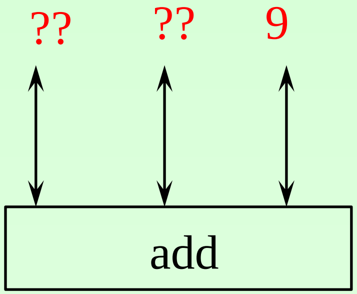
adding
`add(5,4,R)`



checking
`add(5,4,9)`



subtracting
`add(5,Y,9)`



enumerating
`add(X,Y,9)`

Facts & Clauses

Relation via Facts

- We can provide facts as relations.

```
father(john, mary).
father(john, tom).
father(kevin, john).
mother(eva, tom).
mother(eva, mary).
mother(cristina, john).
male(john).
male(kevin).
male(tom).
female(eva).
female(cristina).
female(mary).
```

Query on Facts

- Who is the father of mary?

```
father(X, mary).
```

- Who are child(ren) of eva?

```
mother(eva, C).
```

- Who are daughter(s) of eva?

```
mother(eva, C), female(C).
```

denotes conjunction ∧

Derived Facts via Horn Clauses

- Can construct Horn clause of the form:

$$\texttt{pred(}\ldots\texttt{)}\ \ \texttt{:-}\ \texttt{pred}_1\texttt{(..)},\texttt{pred}_2\texttt{(..)},\texttt{..},\texttt{pred}_n\texttt{(..)}.$$

- Logical meaning:

$$\texttt{pred}_1\texttt{(..)}\wedge\texttt{pred}_2\texttt{(..)}\wedge\ \ldots\ \wedge\texttt{pred}_n\texttt{(..)}\ \rightarrow\ \texttt{pred(}\ldots\texttt{)}$$

Derived Facts via Horn Clauses

- Parent Relation:

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

- Another way to express disjunction:

```
parent(X,Y) :- father(X,Y); mother(X,Y).
```

denotes disjunction $\vee$

Derived Facts via Horn Clauses

- Daughter:

```
daughter(X,Y) :- female(X), parent(Y,X).
```

- Sibling:

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X\==Y.
```

- Grandparent:

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

- Brother:

```
brother(X,Y) :- male(X),sibling(X,Y).
```

Recursive Horn Clauses

- Horn Clauses may be recursive

- How would you express the "ancestor" relation?

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

- Careful with left recursion : Infinite loop due to *depth-first* search procedure.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(Z,Y),parent(X,Z).
```

Unification

Unification by Example

- Unification is denoted by equality.

- Some examples:

```
a = X                      ➔ success X=a
a = b                      ➔ fail
n(a,X) = n(Y,b)            ➔ success X=b, Y=a
n(a,X) = n(X,b)            ➔ fail
n(a,X) = n(X,a)            ➔ success X=a
n(a,X) = n(X,p(a))         ➔ fail
n(a,Y) = n(X,p(a))         ➔ success X=a,Y=p(a)
n(Y,X) = n(X,p(a))         ➔ success X=p(a),Y=X
```

functor (or data constructor)

Essence of Unification

- Unification `t1=t2` requests may contain variables.

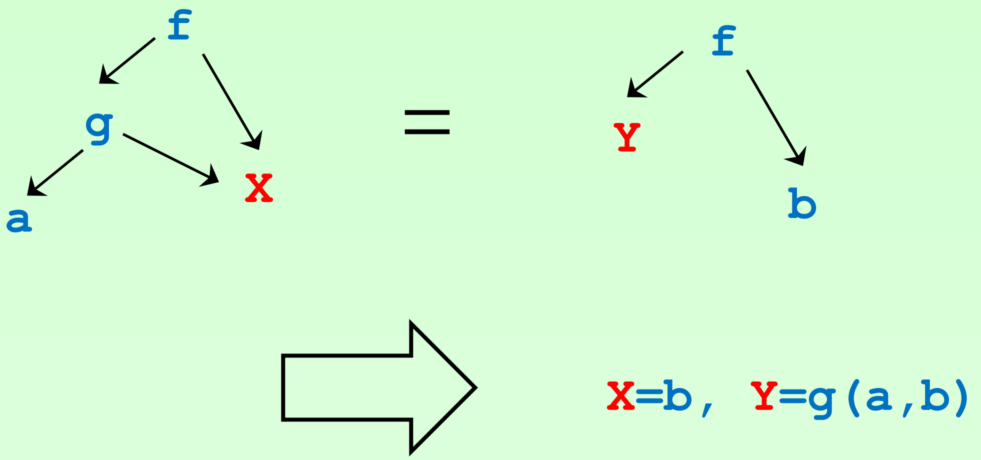
- The system computes a *substitution* for the variables, so that two terms can be made equal.

- Once a variable become bound, it cannot be changed. This is essentially a *single-assignment* property.

Tree Representation of Unification

- Example:

```
?- f(g(a,X),X) = f(Y, b)
```



```
X=b, Y=g(a,b)
```

Unification Algorithm (no variables)

1. Initial unification request: $\Sigma_1=\Pi_1$, $\Sigma_2=\Pi_2$, …
2. If `functor(`Σ_1`)≠functor(`Π_1`)` or `arity(`Σ_1`)≠arity(`Π_1`)` then exit with failure.
3. If `arity(`Σ_1`)=0`, remove $\Sigma_1=\Pi_1$ from the unification request and go to last step.
4. Denote by Σ_{11}`,` Σ_{12}`,...,` Σ_{1k}, the arguments of Σ_1 and denote by Π_{11}`,` Π_{12}`,...,` Π_{1k}, the arguments of Π_1.
5. Set the new unification request to:

   $\Sigma_{11}=\Pi_{11}$, $\Sigma_{12}=\Pi_{12}$,…, $\Sigma_{1k}=\Pi_{1k}$, $\Sigma_2=\Pi_2$, …
6. If current unification request is not empty, go to the first step. Otherwise, terminate with success

Unification Algorithm (with variables)

1. Initial unification request: $\Sigma_1=\Pi_1$, $\Sigma_2=\Pi_2$, …
   If Σ_1 or Π_1 is a variable, add $\Sigma_1=\Pi_1$ to the answer, and apply it as substitution to $\Sigma_2=\Pi_2$, … and go to last step
2. If **`functor(`**Σ_1**`)≠functor(`**Π_1**`)`** or **`arity(`**Σ_1**`)≠arity(`**Π_1**`)`** then exit with failure.
3. If **`arity(`**Σ_1**`)=0`**, remove $\Sigma_1=\Pi_1$ from the unification request and go to last step.
4. Denote by $\Sigma_{11}, \Sigma_{12}, ..., \Sigma_{1k}$, the arguments of Σ_1 and denote by $\Pi_{11}, \Pi_{12}, ..., \Pi_{1k}$, the arguments of Π_1.
5. Set the new unification request to:

   $$\Sigma_{11}=\Pi_{11}, \Sigma_{12}=\Pi_{12},\ldots, \Sigma_{1k}=\Pi_{1k}, \Sigma_2=\Pi_2, \ldots$$

6. If current unification request is not empty, go to the first step. Otherwise, terminate with success

Unification Algorithm Example

```
?- f(g(a,X),X) = f(Y, b)
```

Resolution

- *Resolution* : the process of answering a query.

- *Pattern-matching* is a special case of unification.

- Important concept : *variable renaming*.

  All variables in a rule are replaced by completely new variables

  - Example : `ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y)`
  - 1st Renaming:
    ```
    ancestor(X1,Y1) :- parent(X1,Z1),ancestor(Z1,Y1)
    ```
  - 2nd Renaming:
    ```
    ancestor(X2,Y2) :- parent(X2,Z2),ancestor(Z2,Y2)
    ```

Resolution Algorithm

1. Assume a query : A_1, A_2, … A_n

2. Pick a matching rule from the program and *rename* its variables: `H :- B`$_1$`,B`$_2$`,…,B`$_k$`.`

3. New goal: **($H=A_1$),** `B`$_1$`,B`$_2$`,…,B`$_k$, **A_2, …** A_n

4. Variable bindings may be generated by the unification request **($H=A_1$)**.
   ➔ Add them to the answer, replace bound variables by its substitution over the entire query.

5. Continue from Step 1 until query is empty, and return the answer.

Resolution Demo

Video by Dr Razvan Voicu:

http://www.youtube.com/watch?v=7-aKp-34iWE

List

List Manipulation in Prolog

- List in Prolog is denoted by square bracket with its elements separated by comma:

```
[mary, [], n(A), [1,2,3], X]
```

- Prefix syntax also possible:

```
[t1,t2,t3]  ≡ .(t1,.(t2,.(t3,[])))
```

- In order to break into head and tail, we can use either:

```
(i)      .(H,T)
(ii)     [H|T]
```

Append

- We can join two lists together by the following relation

```
append([],Y,Y).
append([X|Xs],Y,[X|Rs]) :- append(Xs,Y,Rs).
```

- This is structurally similar to a functional definition with a base and a recursive scenario.

```
append([],Y)       =  Y
append([X|Xs],Y) = [X|append(Xs,Y)]
```

- However, take note that the former is a relation, while the latter is a function.

Append

- In particular, relation can be executed in different ways.

- Joining two lists:

```
?- append([1,2,3],[4,5],Z).
➔ Z = [1,2,3,4,5]
```

- Computing the difference:

```
?- append([1,2,3],Y,[1,2,3,4,5]).
➔ Y = [4,5]
```

Append

- Splitting a List:

```
?- append(X,Y,[1,2]).
➔ X=[], Y=[1,2];
   X=[1], Y=[2];
   X=[1,2], Y=[].
```

- Prefix of a List:

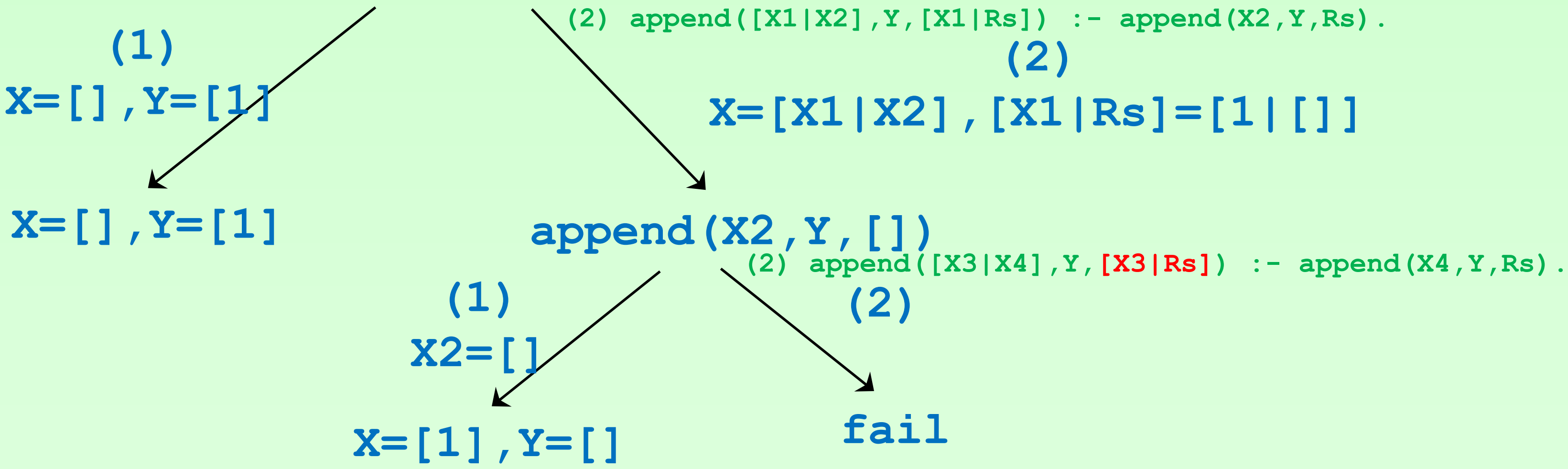
```
?- append(X,_,[1,2,3]).
➔ X=[]; X=[1]; X=[1,2]; X=[1,2,3].
```

Resolution/Search Tree

- We can represent the backtracking performed by resolution using a search tree.

```
(1) append([],Y,Y).
(2) append([X|Xs],Y,[X|Rs]) :- append(Xs,Y,Rs).
```

- Consider: `append(X,Y,[1])`



```
(1)
X=[],Y=[1]

X=[],Y=[1]

(2) append([X1|X2],Y,[X1|Rs]) :- append(X2,Y,Rs).
(2)
X=[X1|X2],[X1|Rs]=[1|[]]

append(X2,Y,[])

(1)
X2=[]

X=[1],Y=[]

(2) append([X3|X4],Y,[X3|Rs]) :- append(X4,Y,Rs).
(2)

fail
```

Reverse

- We can reverse a list, as follows

```
reverse([],[]).
reverse([X|Xs],Y) :- reverse(Xs,Y2),append(Y2,[X],Y).
```

- This is similar to a functional definition

```
reverse([])      =  []
reverse([X|Xs]) =  append(reverse(Xs),[X])
```

Reverse - Examples

- Reverse a List:

```
?- reverse([1,2,3],Y).
➔   Y=[3,2,1].
```

- Another way to reverse a list:

```
?- reverse(X,[1,2,3]).
➔  X=[3,2,1].
```

Reverse

- How about the following query? What does it compute?

```
?- reverse(X,X).
➔           X=[];
            X=[_V];
            …
```

Arithmetic

is-Operator

- Arithmetic expression is evaluated by the is-operator. This is akin to functional evaluation.

- The 2nd argument of is-predicate must be a ground expression (without any variables) to allow the expression to be evaluated.

```
 X is 3+4     ➔ succeed X=7

7 is 3+4     ➔ succeed

7 is X+4     ➔ fail
          (uninstantiated argument)
```

Comparator

- Relations `>, <, >=, =<, =\=, =:=` compares two arithmetic expressions that *must* be evaluated.

- The operator `=` is for term unification.

Factorial

- Computing factorial using a relation:

```
fact(0,1).
fact(N,R) :- N>0, M is N-1,
        fact(M,R1), R is N*R1.
```

- Can compute:

```
?- fact(5,R).
   ➔ R=120.
?- fact(15,R).
   ➔ R=1307674368000.
```

- But not :

```
?- fact(X,120).
 ERROR: >/2: Arguments are not
  sufficiently instantiated.
```

Factorial

- Computing factorial using a relation:

```
fact(0,1).
fact(N,R) :- N>0, M is N-1,
       fact(M,R1), R is N*R1.
```

- This above definition does not allow the first parameter **N** to be *unknown*.

Negation and Cut

Select

- List membership can be implemented as:

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

Note presence of non-linear pattern in LHS of Horn clause

- Functional version looks like:

```
mem(X,[])           = false
mem(X,[Y|T])        = if X=Y then true
                        else mem(X,T)
```

- If you query:

```
?- sel(X,[]).
   ➔ false
```

Fails immediately as both clauses are inapplicable.

Select

- List membership test:

```
?- sel(3,[1,3,5]).
   → true
?- sel(6,[1,3,5]).
   → false
```

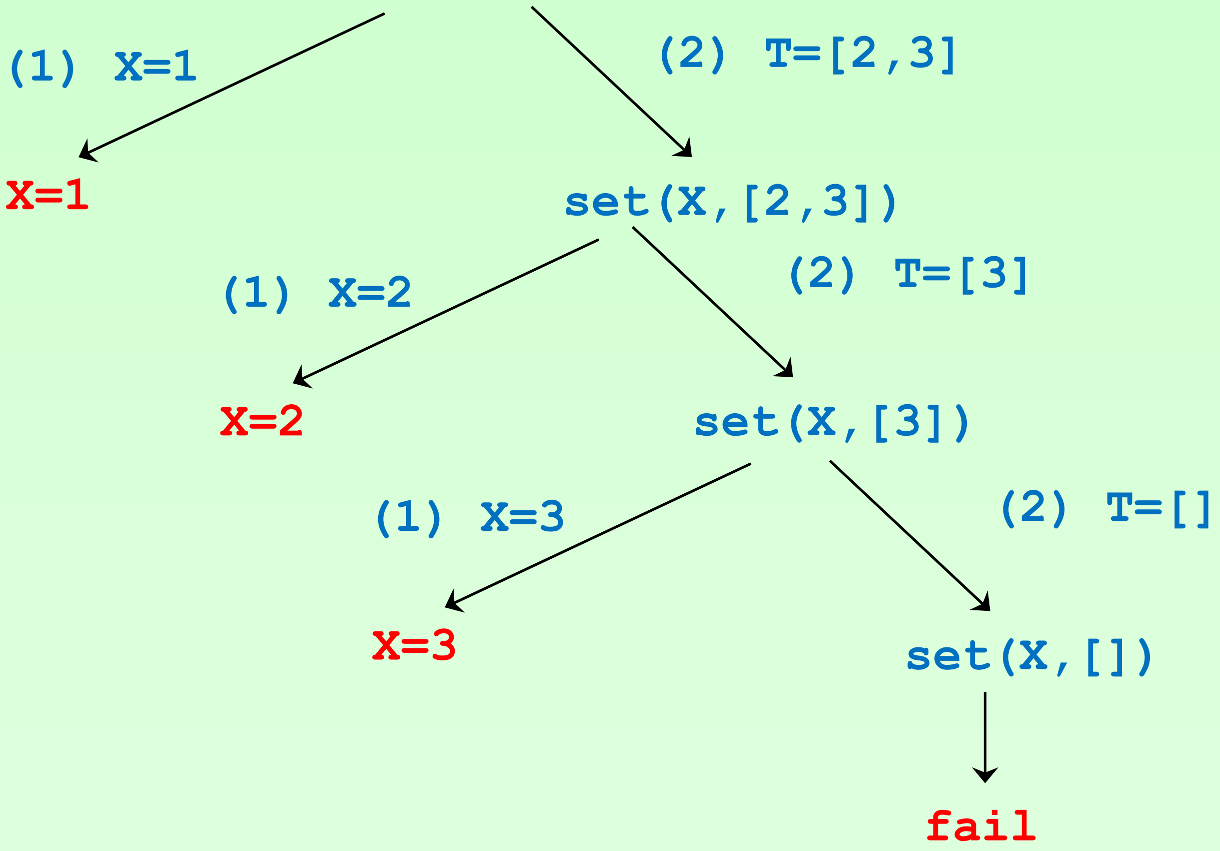
- Element generator:

```
?- sel(X,[1,3,5]).
   → X=1;  X=3;  X=5.
```

Resolution/Search Tree

```
(1) sel(X,[X|_]).
(2) sel(X,[_|T]) :- sel(X,T).
```

- Consider: `sel(X,[1,2,3])`



Negation as Failure

- Prolog is based on *closed* world assumption.

- Whatever can be proven is true.

- Whatever cannot be proven is *assumed* to be false.

Negation as Failure

- Consider.

```
father(john, mary).
mother(eva, mary).
father(john, tom).
mother(eva, tom).
father(kevin, john).
mother(cristina, john).
```

- Query.

```
?- not(father(john,kerry)).
   → true.
```

Negation as Failure

- Can use the clause.

```
male(X) :- not(female(X)).
```

- Need only specify facts on female.
- Above clause says if a person cannot be proven to be female, we shall assume the person is male.
- Negation as failure is sound only if the given clauses are complete.

Removing Duplicates

- To remove duplicate in a list.

```
remDupl([],[]).
remDupl([H|T],R) :- sel(H,T), remDupl(T,R).
remDupl([H|T],[H|R]) :- remDupl(T,R).
```

- Only partially correct:

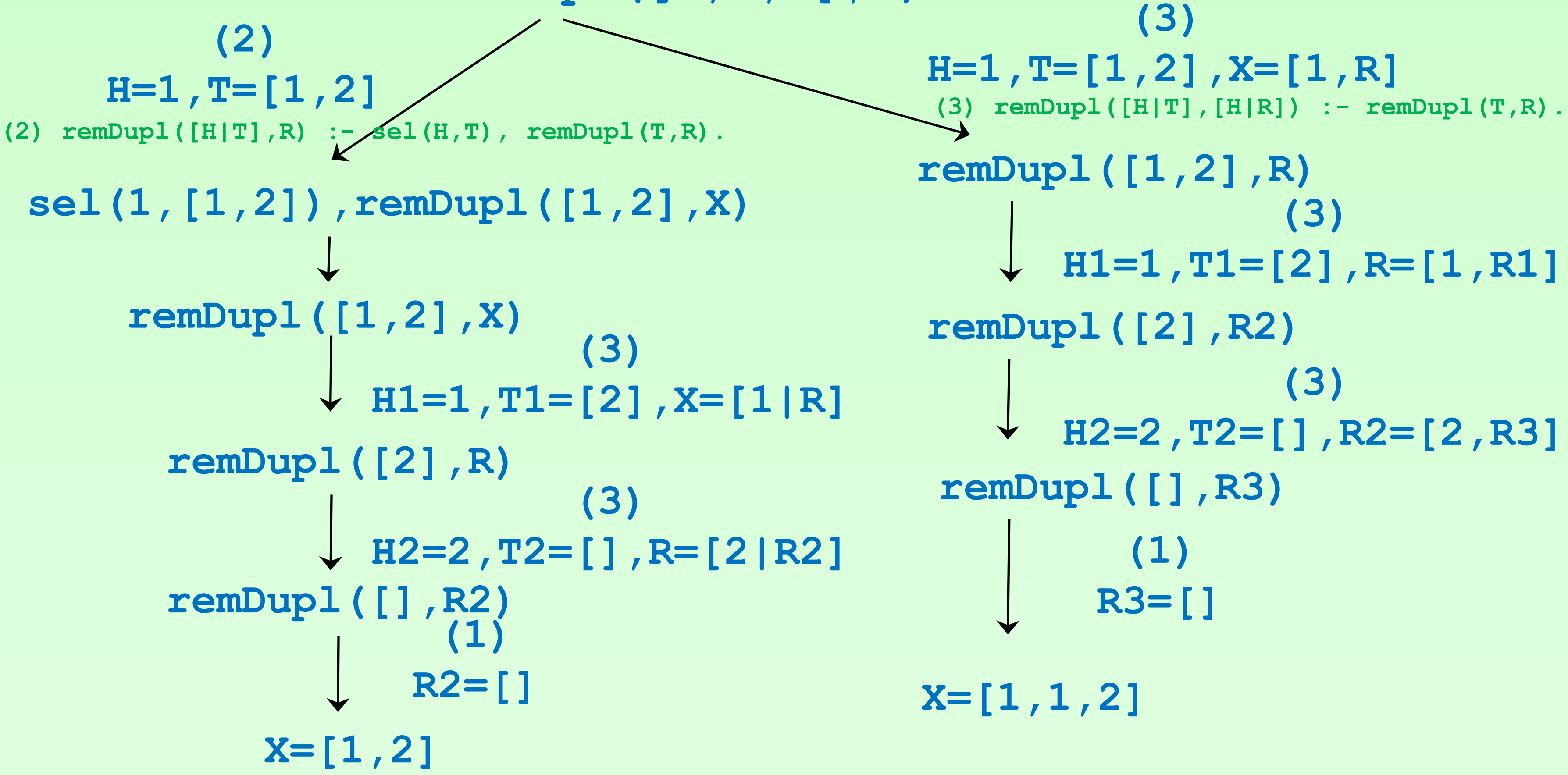
```
?- remDupl([1,1,2],R).
   ➔ R=[1,2];
     R=[1,1,2].
```

- First answer correct but not the second.

Resolution/Search Tree

```
(1) remDupl([],[]).
(2) remDupl([H|T],R) :- sel(H,T), remDupl(T,R).
(3) remDupl([H|T],[H|R]) :- remDupl(T,R).
```

- Consider: `remDupl([1,1,2],X)`



(2)

H=1,T=[1,2]

(2) remDupl([H|T],R) :- sel(H,T), remDupl(T,R).

sel(1,[1,2]),remDupl([1,2],X)

remDupl([1,2],X)

(3)

H1=1,T1=[2],X=[1|R]

remDupl([2],R)

(3)

H2=2,T2=[],R=[2|R2]

remDupl([],R2)

(1)

R2=[]

X=[1,2]

(3)

H=1,T=[1,2],X=[1,R]

(3) remDupl([H|T],[H|R]) :- remDupl(T,R).

remDupl([1,2],R)

(3)

H1=1,T1=[2],R=[1,R1]

remDupl([2],R2)

(3)

H2=2,T2=[],R2=[2,R3]

remDupl([],R3)

(1)

R3=[]

X=[1,1,2]

Using Negation

- Add a negation to 2nd clause.

```
remDup2([],[]).
remDup2([H|T],R) :- sel(H,T), remDup2(T,R).
remDup2([H|T],[H|R]) :- not(sel(H,T)),remDup2(T,R).
```

- One correct solution:

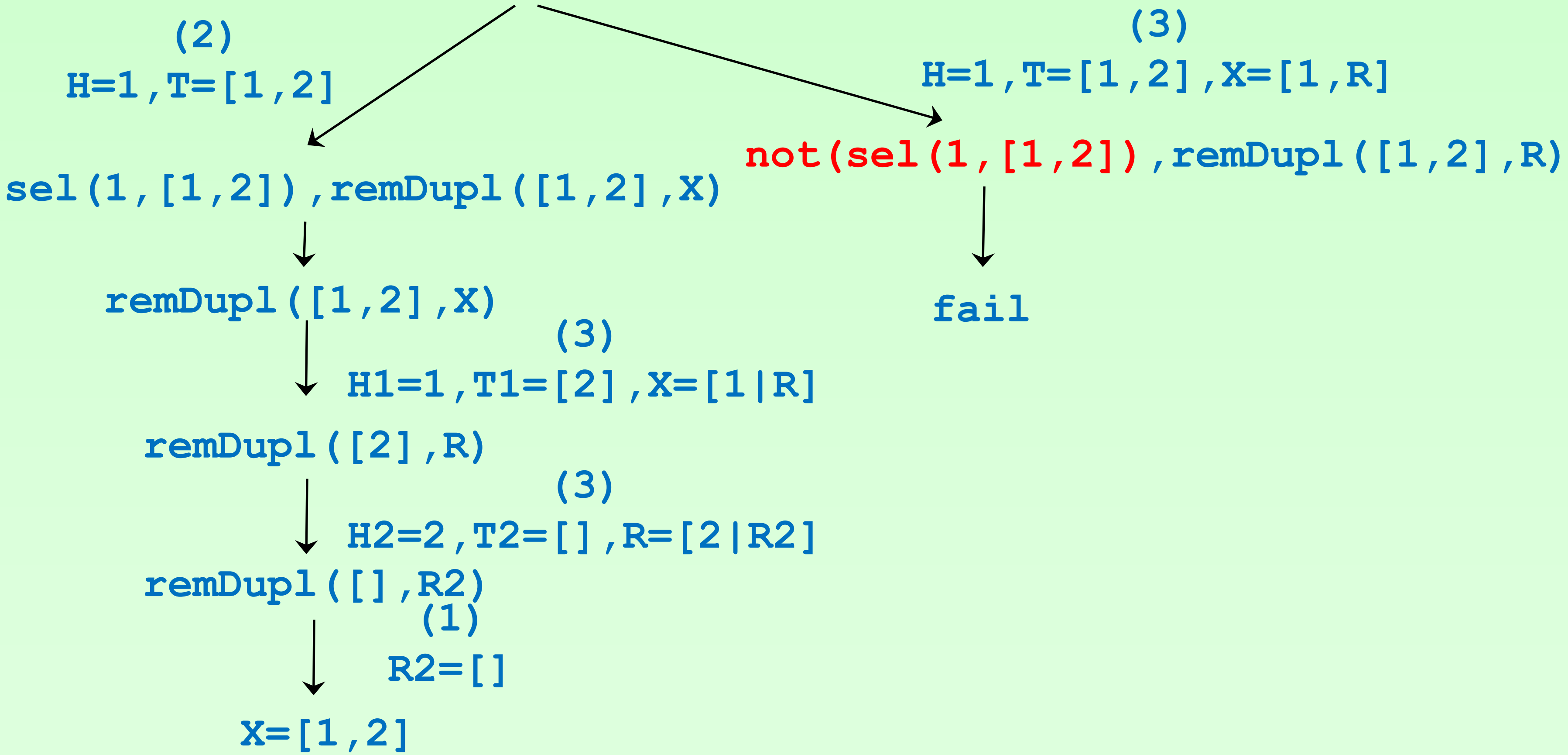
```
    ?- remDup2([1,1,2],R).
       ➔ R=[1,2].
```

- This ensure that we try either 2nd or 3rd clause and not both.

Resolution/Search Tree

```
(1) remDupl([],[]).
(2) remDupl([H|T],R) :- sel(H,T), remDupl(T,R).
(3) remDupl([H|T],[H|R]) :- not(sel(H,T)),remDupl(T,R).
```

- Consider: `remDupl([1,1,2],X)`



(2)
H=1,T=[1,2]

sel(1,[1,2]),remDupl([1,2],X)

remDupl([1,2],X)

(3)
H1=1,T1=[2],X=[1|R]

remDupl([2],R)

(3)
H2=2,T2=[],R=[2|R2]

remDupl([],R2)

(1)
R2=[]

X=[1,2]

(3)
H=1,T=[1,2],X=[1,R]

not(sel(1,[1,2]),remDupl([1,2],R)

fail

Using Cut to Limit Backtracking

- To avoid backtracking, we can add a cut operator !.

```
remDup3([],[]).
remDup3([H|T],R) :- sel(H,T), !, remDup3(T,R).
remDup3([H|T],[H|R]) :- remDup3(T,R).
```

- More efficient and one answer:

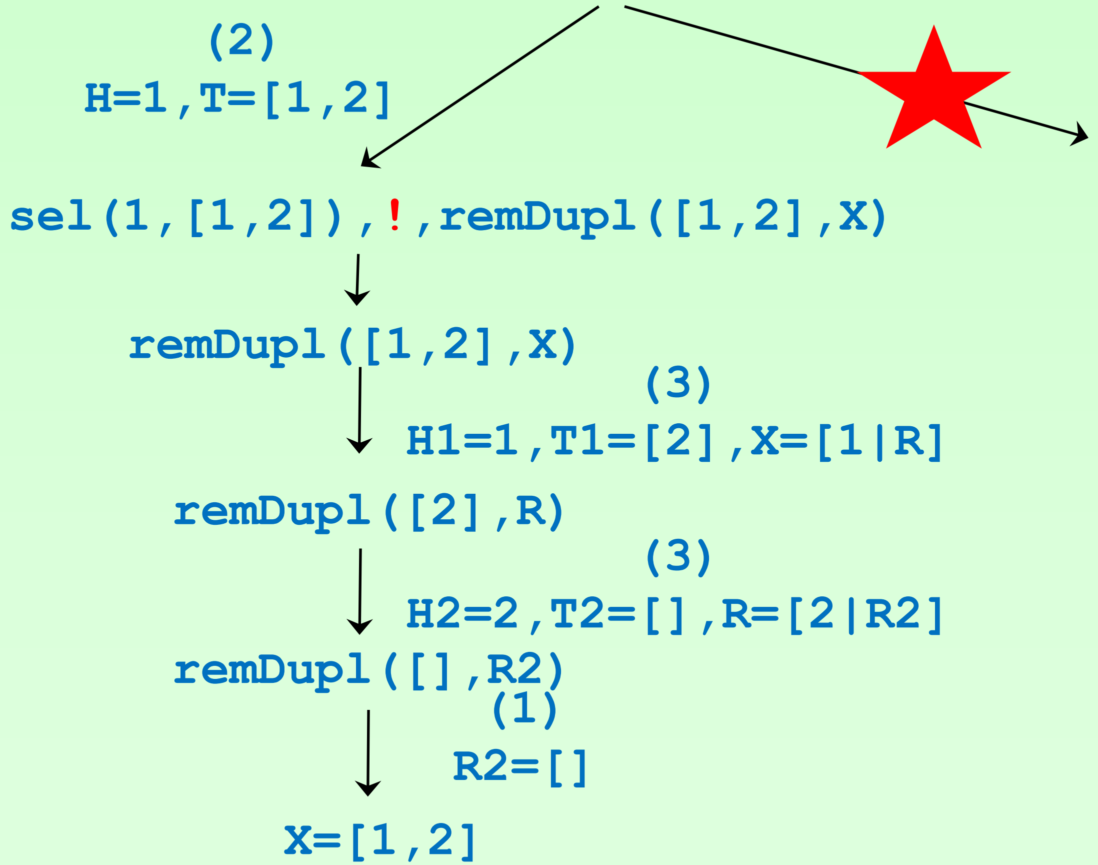
```
?- remDup3([1,1,2],R).
   ➔ R=[1,2].
```

- If `sel(H,T)` succeed, do not backtrack
- If `sel(H,T)` fails, backtrack to try 3rd clause.

Resolution/Search Tree

```
(1) remDupl([],[]).
(2) remDupl([H|T],R) :- sel(H,T),!, remDupl(T,R).
(3) remDupl([H|T],[H|R]) :- remDupl(T,R).
```

- Consider: `remDupl([1,1,2],X)`



```
(2)
H=1,T=[1,2]
sel(1,[1,2]),!,remDupl([1,2],X)
remDupl([1,2],X)
(3)
H1=1,T1=[2],X=[1|R]
remDupl([2],R)
(3)
H2=2,T2=[],R=[2|R2]
remDupl([],R2)
(1)
R2=[]
X=[1,2]
```

Impure Features

Impure Prolog

- **`atom(X)`** : succeeds if **`X`** is bound to an atom.
- **`var(X)`** : succeeds if **`X`** is a free variable.
- **`integer(X)`** : succeeds if **`X`** is bound to an integer.
- **`write(X)`** : output the binding of **`X`**.

Constraint Solving

Finite Constraint Solving

- Constraint solving based on finite domain.

- Essentially based on bounded arithmetic.

- Very powerful
  - Can solve puzzles.
  - Can be used for arithmetic relational.
  - Can be used for harder optimization problems.

Finite Domain Constraints

- Some basic constraints.

```
X #> 3.
   ➔ X in 4..sup

X #\= 10.
   ➔ X in inf..9\/11..sup

3*X #= 9.
   ➔ X=3

X*X #= 9.
   ➔ X=3\/-3
```

Finite Domain Constraints

- More advanced constraints.

```
Vs = [X,Y,Z], Vs ins 1..3, all_different(Vs),
  X = 1, Y #\= 2.
    ➔ Vs = [1, 3, 2], X = 1, Y = 3, Z = 2.
```

```
4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.
    ➔ X = 3, Y = 6.
```

```
X #= Y #<==> B, X in 0..3, Y in 4..5.
    ➔ B = 0, X in 0..3, Y in 4..5.
```

Factorial with Finite Constraints

- Defining factorial more generally using finite constraint solving:

```
cfact(0,1).
cfact(N,R) :- N#>0, M #= N-1,
       R #= N*R1, cfact(M,R1).
```

- Note recursive call comes last, or solver may loop.
- Can now compute:

```
?- cfact(5,R).
   ➔ R=120.
?- cfact(N,120).
   ➔ N=5.
```

Puzzle Solving

- Consider a coding system for alphabet that ensures the following:

```
S E N D + M O R E = M O N E Y
```

- A solution is to use:

```
S=9, E=5, N=6, D=7, M=1, O=0, Y=2
```

since we can show.

```
9 5 6 7 + 1 0 8 5 =  1 0 6 5 2
```

Puzzle Solving

- We can model this as a finite constraint problem using:

```
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
      Vars = [S,E,N,D,M,O,R,Y],
      Vars ins 0..9,
      all_different(Vars),
                S*1000 + E*100 + N*10 + D +
                M*1000 + O*100 + R*10 + E #=
      M*10000 + O*1000 + N*100 + E*10 + Y,
      M #\= 0, S #\= 0.
```

Puzzle Solving

- Querying with `puzzle(As+Bs=Cs)` gives partially solved answer:

```
As = [9, _G10134, _G10137, _G10140],
Bs = [1, 0, _G10155, _G10134],
Cs = [1, 0, _G10137, _G10134, _G10179],
_G10134 in 4..7,
all_different([9, _G10134, _G10137, _G10140, 1, 0,
  _G10155, _G10179]),
1000*9+91*_G10134+ -90*_G10137+_G10140+ -9000*1+ -
  900*0+10*_G10155+ -1*_G10179#=0,
_G10137 in 5..8,
_G10140 in 2..8,
_G10155 in 2..8,
_G10179 in 2..8.
```

Puzzle Solving

- Using `label(As)` that performs an *enumeration* gives unique solution!

```
puzzle(As+Bs=Cs),label(As).
As = [9, 5, 6, 7],
Bs = [1, 0, 8, 5],
Cs = [1, 0, 6, 5, 2] ;
```

Logic Puzzle

There are 5 houses, each of a different color, and inhabited by a person from a different country who has a different pet, drink, and make of a car.

(a) The English woman lives in the red house.

(b) The Spaniard owns the dog.

(c) Coffee is drunk in the green house.

(d) The Ukrainian drinks tea.

(e) The green house is immediately to the right of the ivory house.

(f) The BMW driver owns snails.

(g) The owner of the yellow house drives a Toyota.

(h) Milk is drunk in the middle house.

(i) The Norwegian lives in the first house of the left.

(j) The person who drives the Ford lives in the house next to the owner of the fox.

(k) The Toyota driver lives in the house next to the house where the horse is kept.

(l) The Honda owner drinks orange juice.

(m) The Japanese drives a Datsun.

(n) The Norwegian lives next to the blue house.

Logic Puzzle

There are 5 houses, each of a different color, and inhabited by a person from a different country who has a different pet, drink, and make of a car.

```
People = [English, Spaniard, Ukrainian, Norwegian, Japanese],
Colors = [Red, Green, Ivory, Yellow, Blue],
Drinks = [Tea, Milk, Orange, Coffee, Water],
Pets = [Dog, Snails, Fox, Horse, Zebra],
Cars = [BMW, Toyota, Ford, Datsun, Honda],
Houses = [House1=1, House2=2, House3=3, House4=4, House5=5]

L = [People,Colors,Drinks,Pets,Cars],
```

Logic Puzzle

Adding constraint on all_different

```
appendall(L, All),

All ins 1..5,

all_different(People), all_different(Colors),

all_different(Drinks), all_different(Pets),

all_different(Cars),
```

Logic Puzzle

(a) The English woman lives in the red house.

English = Red

(b) The Spaniard owns the dog.

Spaniard = Dog

(c) Coffee is drunk in the green house.

(d) The Ukrainian drinks tea.

(e) The green house is immediately to the right of the ivory house.

Green #= Ivory+1

(f) The BMW driver owns snails.

(g) The owner of the yellow house drives a Toyota.

(h) Milk is drunk in the middle house.

Logic Puzzle

(i) The Norwegian lives in the first house of the left.

(j) The person who drives the Ford lives in the house next to the owner of the fox.

abs(Ford-Fox) #= 1

(k) The Toyota driver lives in the house next to the house where the horse is kept.

(l) The Honda owner drinks orange juice.

(m) The Japanese drives a Datsun.

(n) The Norwegian lives next to the blue house.

Summary

- Logic programming and particularly with constraint-solving capability is an extremely powerful problem-solving mechanism.

- Two current weaknesses
  - Untyped language setting
  - Lack of higher-order functions

- Nevertheless, special language features allows harder problems to be more easily modelled.