PROLOG
Programming in Logic

Dr. Jozo Dujmović

The limits of your language are the limits of your world.

- *L. Wittgenstein*

Quoting *A. Perlis*

- A good programming language is a conceptual universe for thinking about programming.

- A language that doesn't affect the way you think about programming is not worth knowing.

Introduction

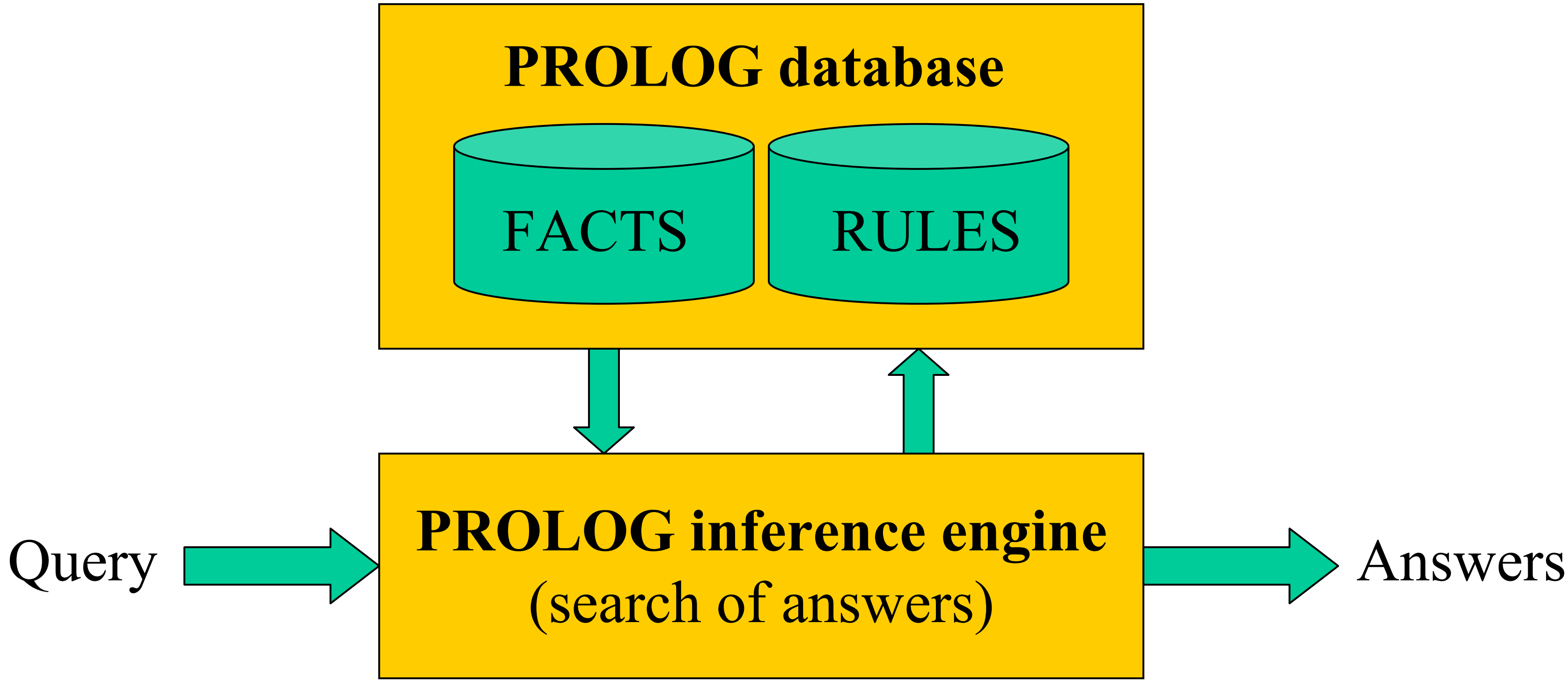
The Concept of Logic Programming

- Beginning: Alan Colmerauer et al. 1973, Marseille, France
- Concept: Programming in Logic
  - A program consists of a set of clauses.
  - Each clause is either a **fact** related to a given problem, or a **rule** about how the solution may relate to, or be inferred from, the given facts.
- Emphasis on nonprocedural programming [specify WHAT has to be done, without details HOW to do it]

Transforming WHAT to HOW

- Nonprocedural programming uses various means for generating solutions from the statement of a problem
- Examples:
  - A compiler transforms arithmetic expressions to assembly language procedures
  - Recursion is a mechanism of repetitive solving of the same problem of decreasing size
  - In Prolog, an inference engine generates solutions using a systematic search of a database of facts and inference rules

Concept of PROLOG system



Prolog inference engine uses query to search and combine facts and rules, and to automatically generate all answers that satisfy user's questions.

Generally, the query specifies what has to be done, but not how to do it.

Application Areas

- Artificial intelligence
- Relational databases
- Mathematical logic
- Abstract problem solving
- Architectural design
- Symbolic equation solving
- Biochemical structure analysis
- Business
- Security
- … and many others

PROLOG: installation and documentation

- SWI: **http://www.swi-prolog.org/**
- GNU: **http://gnu-prolog.inria.fr/**

SWI Prolog install and use

To get SWI-Prolog from the University of Amsterdam:

Go to http://www.swi-prolog.org/

To install:

Download Self-installing executable for Windows/Linux/MacOS

To select the type of PROLOG files:

Default type can be .pl or .pro - it is necessary to select .pro if .pl is reserved for Perl

To start:

Double click SWI-Prolog, or double click program name that has a default type (e.g. max.pro)

At the beginning:

```
consult(program_name).        % No extension: consult(max).
                              % do not use consult(max.pro).
```

At the end:

```
halt.
```

At the infinite loop:

Ctrl-C and then 'a' (for 'abort')

To see database:

```
listing.
```

Using GNU Prolog

The GNU Prolog compiler is available on host Libra. To use this compiler you need to update the PATH variable to include the path:

/usr/local/gprolog-1.2.1/bin

This environment variable should be set in the .cshrc file.

To get documentation, use the URL to the Gprolog home page:

http://gnu-prolog.inria.fr/

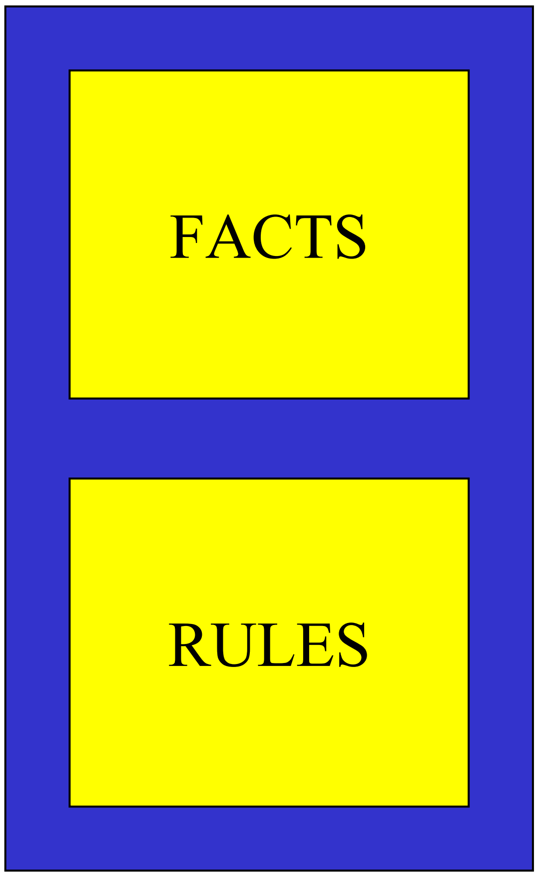
GNU Prolog start/stop:

```
libra: /afs/sfsu.edu/f1/jozo% gprolog
GNU Prolog 1.2.16
By Daniel Diaz
Copyright (C) 1999-2002 Daniel Diaz
| ?- halt.
libra: /afs/sfsu.edu/f1/jozo%
```

Facts and Rules

Prolog Programs

PROGRAM




```
% Facts

likes(jozo,wine).

likes(eisman,wine).


% Rules

friends(X,Y) :- likes(X,wine),

                likes(Y,wine).
```

Basic terminology

Comment

Atoms

Terminator

```
% Facts
likes(jozo,wine).
likes(eisman,wine).
```

Predicate

Variables

```
% Rules
friends(X,Y) :- likes(X,wine), likes(Y,wine).
```

Variables If And

Verbal interpretation of facts and rules

```
% Facts: Jozo likes wine

likes(jozo,wine).


% Rules: X and Y are friends if X likes wine
and Y likes wine

friends(X,Y) :- likes(X,wine), likes(Y,wine).
```

Prolog: Goals and Operation

- Automatic reasoning based on facts and rules.
- Non-procedural (declarative) programming.
- Applications in combinatorial problems (particularly in the area of artificial intelligence)
- Operation of Prolog inference engine is based on a mechanism of systematic search and pattern matching.
- In most cases, Prolog is an interpreter (it does not generate standalone executable code)

Facts and their syntax

```
likes(jozo,wine).
likes(eisman,wine).
```

Predicate: likes

Atoms: likes, eisman, jozo, wine

Terminator

Query: Who likes wine?

```
?- likes(X,wine).
X = jozo ;
X = eisman ;
no
```

Request to generate more solutions

No more solutions

Atoms

- Simplest terms are constants and atoms
- Constants: integer (-123), real (-12.3e-8), string (’this is a string’)
- Atoms: symbolic constants (all lower case words and special atoms):
  - likes (predicate), peter, food; e.g., likes(peter,food)
  - [ ] (empty list)
  - * , + , - , ! (operators)

Variables

- Syntax: variables start with uppercase letter or underscore: e.g., X, Sister, Mother, _abc, _ (underscore = anonymous [nameless] variable)
- Semantics: Variable = name of a dynamic object with unspecified data type, temporarily instantiated by binding to an atom of specific type.
- Binding based on unification (pattern matching):

  ?- peter + Brother = peter + john.

  Brother = john .... (binding)

  ?- 2/3 = X/Y.

  X = 2 .... (binding)

  Y = 3 .... (binding)

List of Facts

```
brother(john, mike).                ↓
brother(john, peter).               ↓
brother(john, george).              ↓
sister(john, mary).                 ↓
sister(john, ann).                  ↓
drives(john, ford, mustang).        ↓
drives(mary, toyota, camry).        ↓
drives(peter, honda, civic).        ↓
drives(ann, ford, focus).           ↓
```

Prolog searches the list of facts by going top-down through the list of facts. The search is based on pattern matching. When the match is detected the search engine stops search and inserts pointer to the line where the match is found. The search can later continue from that point.

Based on these facts, what questions could you ask?

Sample Queries

- Who are brothers of John?
- Who are sisters of John?
- Who are siblings of John?
- Who is female in this family?
- Who is male in this family? (John = ???)
- Who drives cars manufactured by Ford?
- Who drives a Camry?
- What manufacturer sold two cars to this family?
- Is George a brother of Ann?
- Who are brothers of Ann?
- … and so on, and so forth...

Yes and No

```
?- brother(john, george).
yes
?- brother(adam, george).
no
```

- Meaning:
  - Yes = search engine was able to find a match (to “prove the rule”)
  - No = no match can be found in current database

Simple Query Based On Facts

- Who is the brother of John?

```
?- brother(john, B).
```

or

```
?- brother(john, Brother).
```

B and **Brother** are variables. Variables are instantiated in the process of pattern matching (no matching between uninstantiated variables; e.g. X=Y yields no result but X=3, Y=X instantiates both X and Y to 3)

Pattern Matching

Matching two expressions:

```
?- brother(john, B).

   brother(john, peter).
```

Matching can be achieved only if **`B`** is instantiated as the atom **`peter`**. Prolog sets the value of **`B`** using a pointer to **`peter`**.

Pattern Matching With Multiple Variables

```
predicate(adam, B, clint, D).

predicate(A, boris, C   , dan).
```

Matching can be achieved only if all variables are instantiated with corresponding atoms. In the process of unification all variables (A, B, C, D) will be instantiated (binded to atoms).

Prolog Inference (Search) Engine

- Prolog inference engine searches solutions using the following simple mechanisms:
  - Top-down search through facts.
  - Left-right satisfaction of goals.
  - Pattern matching of facts and expressions.
  - Instantiation of variables to satisfy matching.
  - Selective backtracking controlled by the cut mechanism.

Query Based On Rules - Syntax

- General form of rule consists of simultaneous satisfaction of several conditions (goals).

  ```
  rule(X,Y) :- goal1(X), goal2(Y),
  goal3(X,Y).
  ```

- Verbal interpretation: X and Y satisfy condition `rule(X,Y)` if X satisfies `goal1`, Y satisfies `goal2`, and X and Y satisfy `goal3`.

Conjunctive Query (“,” = “and”)

- In this family, who is John’s sister who drives a toyota?

```
driver(S) :- sister(john, S),
             drives(S, toyota, _ ).
```

- Interpretation: the driver is S if S is the sister of John **and** S drives a Toyota.
- We are not interested which model S drives. Consequently, we use the anonymous variable “ _ ” that can be initialized with any car model. Initializing a model variable that is never used is considered an unacceptable practice; such a variable is called the **singleton variable**.

Dialog (user interaction with Prolog)

```
?-   driver(ann).     % Is Ann the driver?
no
?-   driver(mary).    % Is Mary the driver?
yes
?-   driver(Name).    % Find the name of the driver
Name = mary ;         % Is there anybody else?
no
```

Disjunctive Query (“;” = “or”)

- Who is a sibling of person P?

```
sibling(P,S)  :- brother(P,S) ;
                  sister(P, S).
```

- Interpretation: S is a sibling of P if S is a brother of P **or** if S is a sister of P.
- Alternative notation (using two rules):

```
sibling(P,S)  :- brother(P,S).
sibling(P,S)  :- sister(P, S).
```

Satisfying a Sequence of Goals

- Goals are satisfied going left to right. For each satisfied goal there is a pointer left in the database at the place where the goal was satisfied.
- If a goal is not satisfied, then the search engine performs a backtrack: it returns to a previous goal and continues search from the previous pointer position.
- If the rule cannot be satisfied, Prolog search engine returns the answer "**no**".

Rules

X and Y are friends if X likes wine and Y likes wine.

```
friends(X,Y) :- likes(X,wine),likes(Y,wine).
```

if

and

variables

Modified rule: X and Y are friends if they like the same thing.

```
friends(X,Y) :- likes(X,S),likes(Y,S).
```

Improving Rules

Show all friends:

```
?- friends(X , Y).

X = jozo
Y = jozo;

X = jozo
Y = eisman ;

X = eisman
Y = jozo ;

X = eisman
Y = eisman  ;
no
```

```
friends(X,Y) :- likes(X,S),likes(Y,S), X \= Y.
```

(now X and Y must be different)

Queries

Are jozo and eisman friends?

```
?- friends(jozo , eisman).

yes
```

Who is a friend of jozo?

```
?- friends(jozo, X).

X = eisman;

no
```
(cannot find more solutions)

```
likes(jozo,wine).

likes(eisman,wine).
```

% X and Y are friends if both of then like wine

```
friends1(X,Y) :- likes(X,wine), likes(Y,wine).
```

```
?- friends1(X,Y).

X = jozo
Y = jozo ;

X = jozo
Y = eisman ;

X = eisman
Y = jozo ;

X = eisman
Y = eisman ;

No
```

Sample program #1

```
likes(jozo,wine).

likes(eisman,wine).
```

Sample program #2

% X and Y are friends if they like the same thing

```
friends2(X,Y) :- likes(X,W), likes(Y,W).
```

```
?- friends2(X,Y).

X = jozo
Y = jozo ;

X = jozo
Y = eisman ;

X = eisman
Y = jozo ;

X = eisman
Y = eisman ;

No
```

```
likes(jozo,wine).
likes(eisman,wine).
```

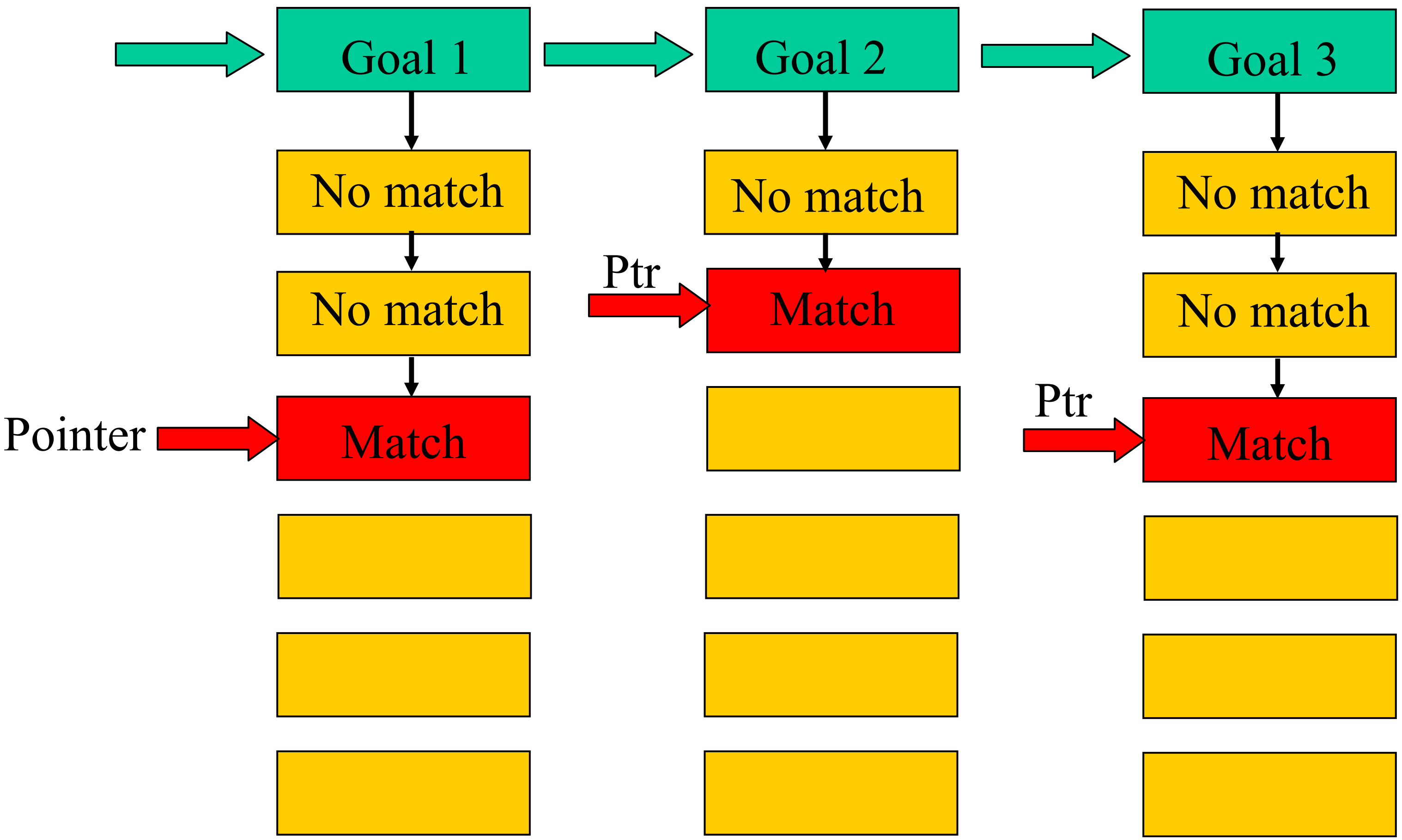
Sample program #3

% X and Y (different people) are friends if they like the same thing

```
friends3(X,Y) :- likes(X,W), likes(Y,W), X \= Y.
?- friends3(X,Y).

X = jozo
Y = eisman ;

X = eisman
Y = jozo ;

No
```

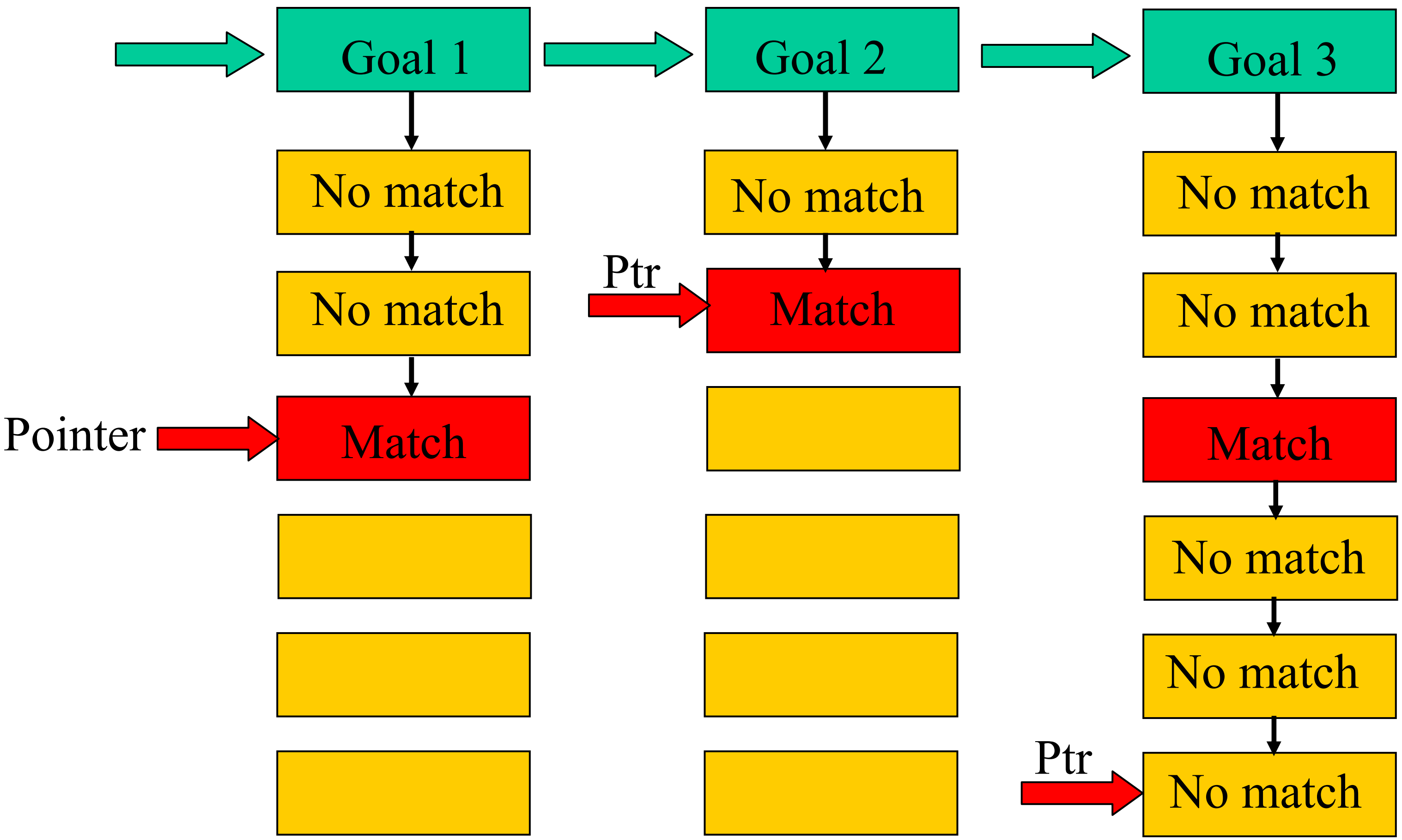
Prolog search mechanism

PROLOG search mechanism (1)



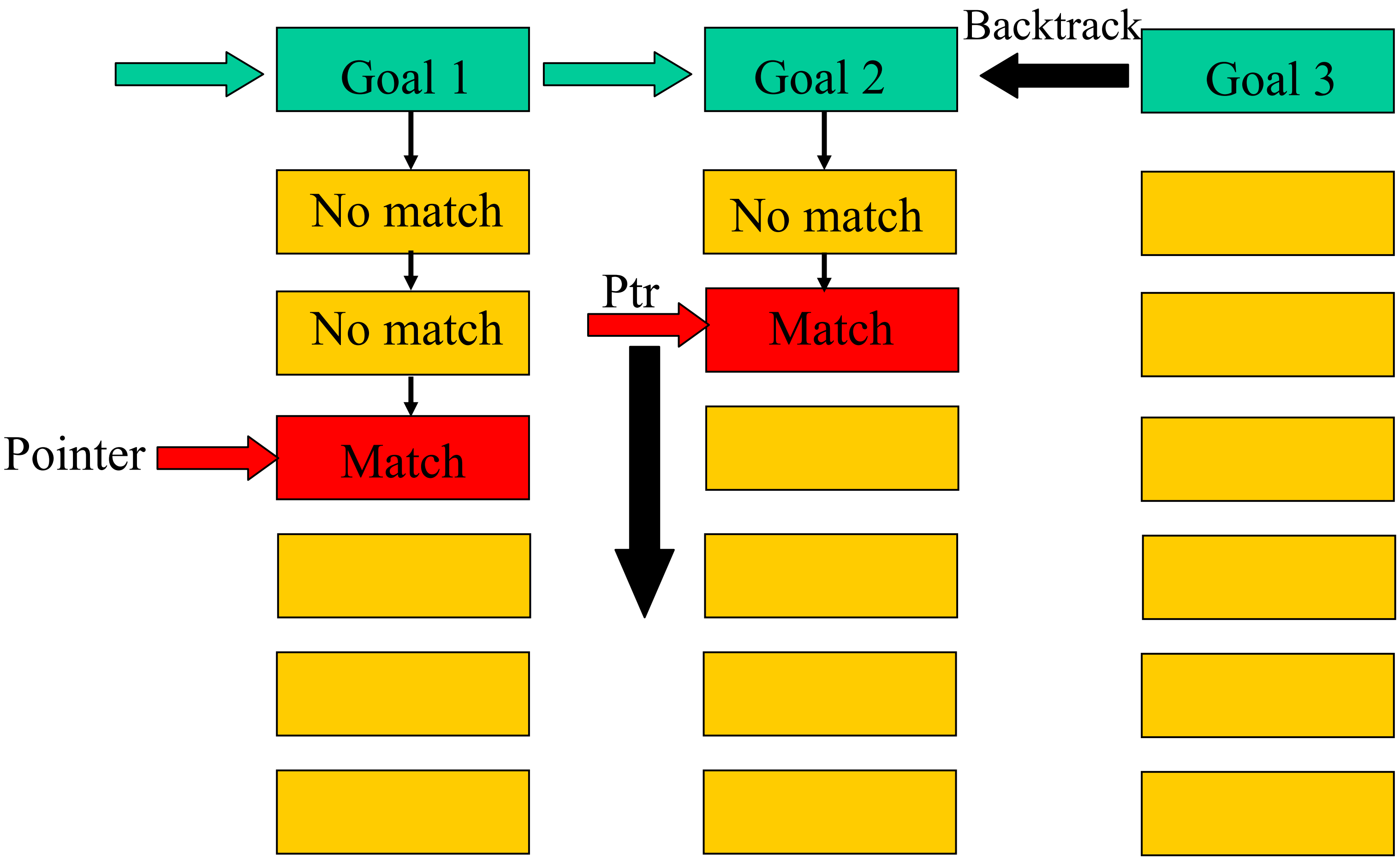
When the match is found the answer is shown to the user

PROLOG search mechanism (2)



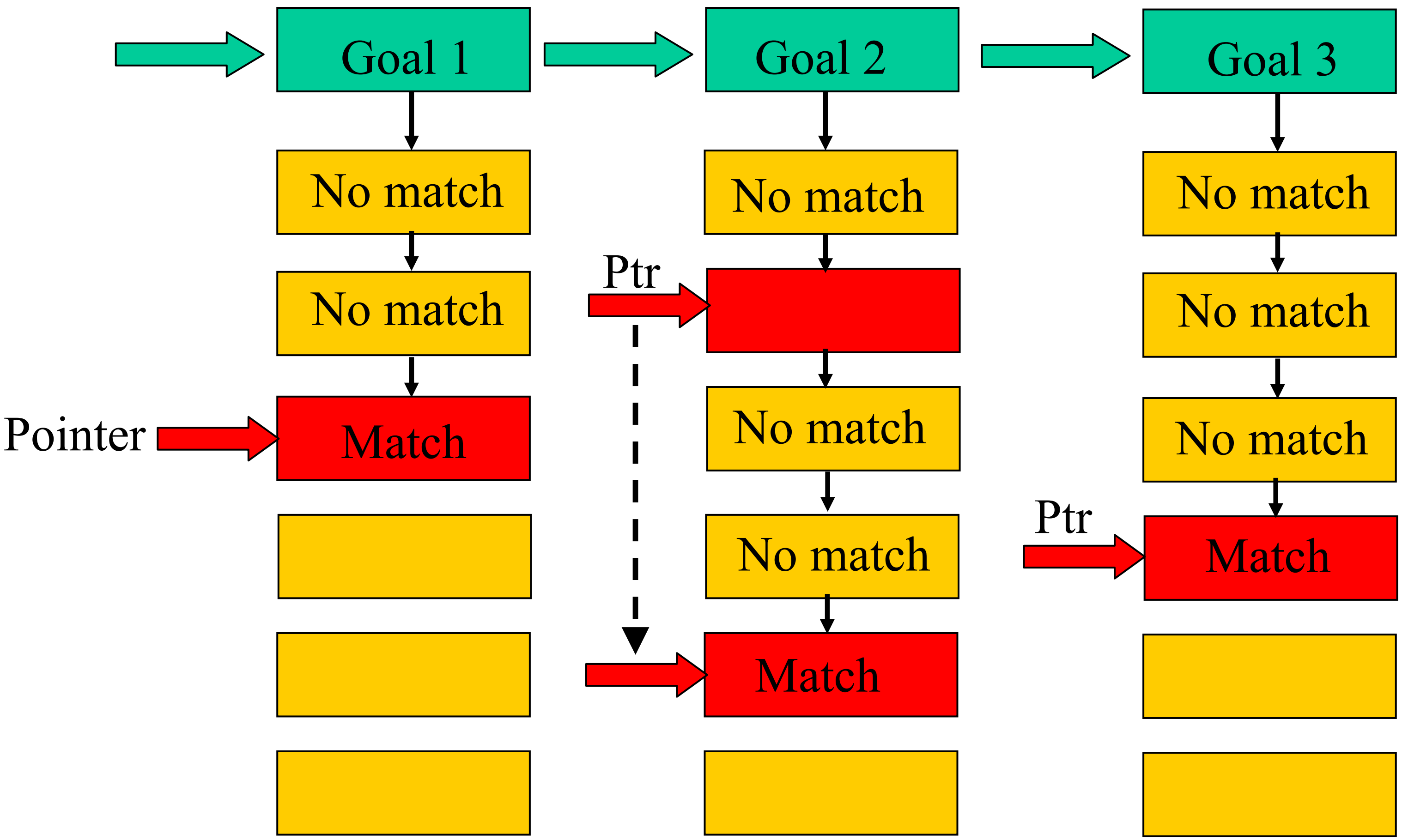
Unsuccessful continuation of search: no other match with the Goal 3

PROLOG search mechanism (3)



Backtrack and continue search from the match point of the previous goal

PROLOG search mechanism (4)



New combination of matching generates a new answer to the given query

Backtracking

- Backtracking is caused by a failure to find a match in the set of facts
- Backtracking is the process of returning back to the previous successful match and continuation of search from that position
- Each match is registered by a pointer
- Search mechanism with backtracking is implemented in the PROLOG search engine

Equalities and Inequalities

`X=Y`	X is equal to Y if X and Y match (for arbitrary X,Y)
`X\=Y`	succeeds if X and Y do not match (different patterns)
`X==Y`	X is literally equal to Y if X and Y are identical
`X\==Y`	succeeds if X is not literally equal to Y
`X=:=Y`	the value of X equals the value of Y (X,Y: arit.expr.)
`X=\=Y`	the value of X is not equal to the value of Y (X,Y are arithmetic expressions)
`X is Y`	X matches (or is assigned) the value of Y (X is a variable or a constant, and Y is an arithmetic expr.)

Assignments

```
?- X is 1 + 2 + 3 + 4.
X = 10 ;
No
?- 10 is 1 + 2 + 3 + 4.
Yes
?- 1 + 2 + 3 + 4 is 10.
No
?- 1 + 2 is 2 + 1.
No
?- 1 + 2 =:= 2 + 1.
Yes
```

```
?- X is 1/3.
X = 0.333333 ;
No
?- 0.33333333333333333 is 1/3.
No
?- 0.33333333333333333 =:= 1/3.
No
?- 1/3 =:= 1/3.
Yes
?- 1/3 = 1/3.
Yes
?- 1/3 == 1/3.
Yes
```

Sample program #4

```
likes(jozo,wine).
likes(eisman,wine).
```

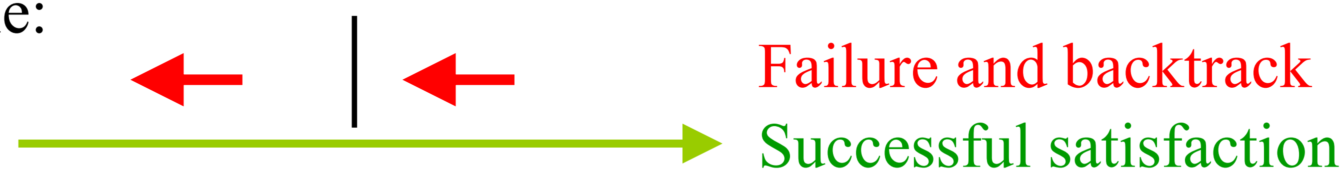
% X and Y (different people) are friends if they like the same thing.

% Stop search when the first pair of friends is detected.

```
friends4(X,Y) :- likes(X,W), likes(Y,W), X \= Y, !.
?- friends4(X,Y).

X = jozo
Y = eisman ;
No
```

Cut

Cut (!) is a goal that always succeeds and the system becomes committed to all choices made before the satisfaction of cut. For example:




```
r  :-  a , b , c , ! , d , e , f .

       ↓   ↓   ↓   -   ↓   ↓   ↓        ↓ : matching goal and fact

       ↓   -   ↓       -   ↓   -        - : satisfaction of goal

       -       -       ↓   -   ↓        = : end of search options

                       -       =
```

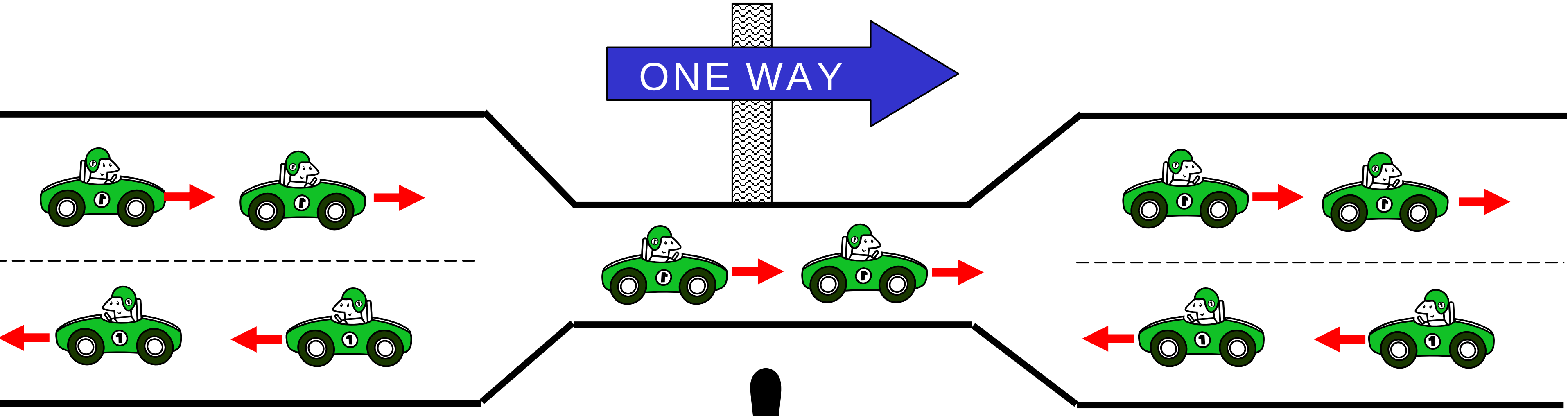
After satisfying a, b, c, and going over the cut (!) it is not possible to backtrack and try other combinations of a,b,c. The rule r will be satisfied with the first combination of a,b,c and all other combinations of d,e,f. Backtracking normally works in the d,e,f area.

Cut as a diode

a , b , c , ! , d , e , f .

a , b , c , , d , e , f .

A traffic interpretation of cut



a , b , c ,   !   , d , e , f .

When and why to use the cut?

- The goal of cut is to restrict or to terminate the search process
- Cut is used to improve performance (remove unnecessary search operations)
- Cut is a procedural mechanism in an otherwise nonprocedural world
- Cut is inconsistent with the nonprocedural approach to problem solving, but it is necessary to make efficient programs

Input and Output

read(X)	Read X from keyboard
write(X)	Display X on the screen
tab(N)	Display N spaces
nl	New line

```
likes(jozo,wine).

likes(eisman,wine).
```

Sample program #5

% X and Y (different people) are friends if they like the same thing.

% Stop search when the first pair of friends is detected, and

% generate a message.

```
friends5(X,Y) :- likes(X,W), likes(Y,W), X \= Y, !,
                 write(X), write(' and '), write(Y),
                 write(' are friends because they both
                 like '),  write(W), nl.
```

```
?- friends5(X,Y).
jozo and eisman are friends because they both like wine
X = jozo
Y = eisman ;
No
```

Sample program #6

% X and Y (different people) are friends if they like the same thing.

% Stop search when the first pair of friends is detected, and

% generate a message. Do not display the values of arguments

% X and Y.

```
friends6 :- likes(X,W), likes(Y,W), X \= Y, !,
            write(X), write(' and '), write(Y),
            write(' are friends because they both like '),
            write(W), nl.
```

Sample program #7

% X and Y (different people) are friends if they like the same thing.

% Show all solutions. Display the values of arguments X and Y.

```
friends7(X,Y) :- likes(X,W), likes(Y,W), X \= Y,
           write(X), write(' and '), write(Y),
           write(' are friends because they both like '),
           write(W), nl.
```

Sample program #8

% X and Y (different people) are friends if they like the same thing.

% Show all solutions. Do *not* display the values of arguments X and Y.

```
friends8 :- likes(X,W), likes(Y,W), X \= Y,
          write(X), write(' and '), write(Y),
          write(' are friends because they both like '),
          write(W), nl.
```

Sample program #9

% bigfriends are those who share two or more things that they like. Show the first solution only.

```
bigfriends1(X,Y) :- likes(X,W), likes(Y,W),
likes(X,F), likes(Y,F), X \= Y, W \= F, !.
```

% bigfriends are those who share two or more things that they like. Show all solutions.

```
bigfriends2(X,Y) :- likes(X,W), likes(Y,W),
likes(X,F), likes(Y,F), X \= Y, W \= F.
```

Eliminating Trivial Solutions

- Prolog search engine generates all solutions. They include:
  - repetition of the same result
  - permutation of previous results
- Trivial repetitions (X=same, Y=same) can be eliminated using the condition X \= Y; this condition can be used only after X and Y have been instantiated
- Permutation of solutions (X=a, Y=b and X=b, Y=a) can be eliminated using cut (!) that terminates the search after the first solution has been generated

Overloading

```
% Predicates are identified by both the name
% and the number of arguments
max(X,Y,Y) :- X =< Y.       % max(X,Y,Maximum)
max(X,Y,X) :- X > Y .

max(X,Y,Z,Max) :- max(X,Y,T), max(T,Z,Max).

max(W,X,Y,Z,Max) :- max(W,X,Y,T), max(T,Z,Max).

% max(X,Y) is a Prolog library function
max4(W,X,Y,Z,Max) :- P is max(W,X),Q is max(Y,Z),
                     Max is max(P,Q).   % or
max4(W,X,Y,Z,M) :- M is max(max(W,X), max(Y,Z))
```

Overloading: results

```
11 ?- max(1,2,M).
M = 2 ;
No
12 ?- max(1,3,2,M).
M = 3 ;
No
13 ?- max(1,3,2,-1,M).
M = 3 ;
No
14 ?- max4(1,3,2,-1,M).
M = 3 ;
No
```

Adding and Deleting Clauses

Database Manipulation

- Adding new clauses during execution

  assert(male(tom)). (add fact as the last fact of male group)

  asserta(male(tom)). (add fact as the first fact of male group)

  assertz(male(tom)). (same as assert)

- Deleting clauses during execution

  retract(male(tom)). (this fact is removed from the database)

Assert and Retract

```
?- assert(car(ford)).
Yes
?-
assert(car(honda)).
Yes
?- car(X).
X = ford ;
X = honda ;
No
```

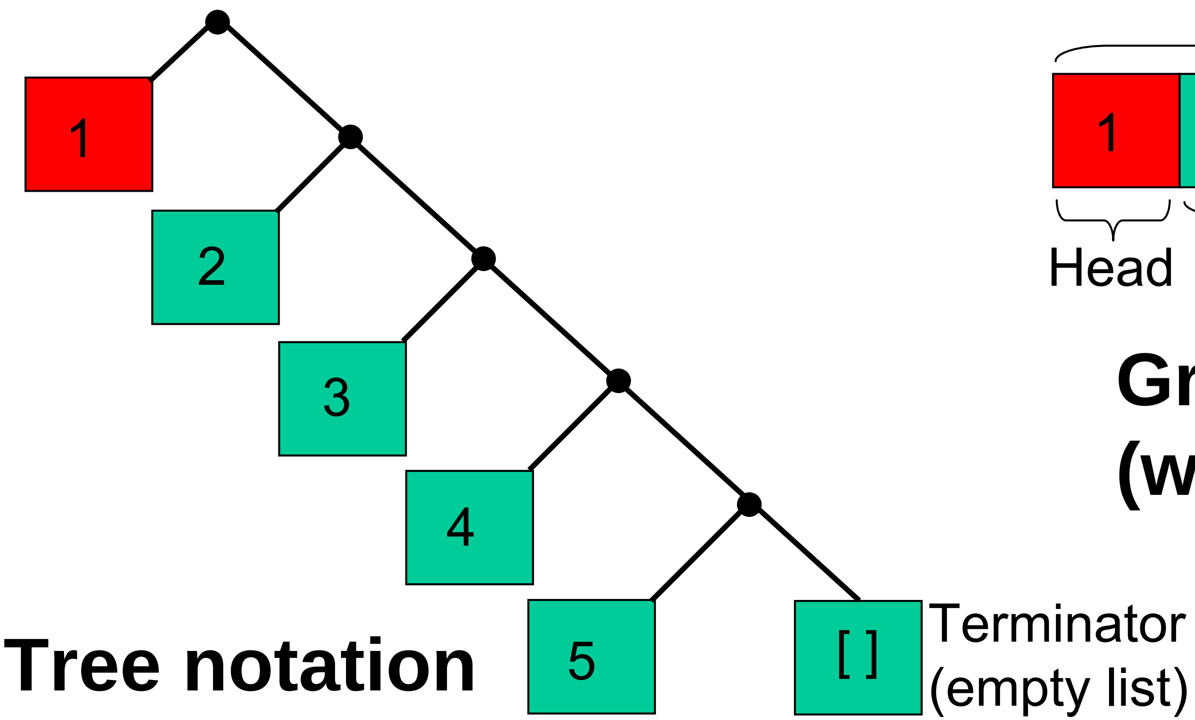
```
?- retract(car(ford)).
Yes
?- car(X).
X = honda ;
No
?-
```

```
14 ?- assert(max4(A,B,C,D,Max) :- Max is max(max(A,B), max(C,D))).
              % max(X,Y) is a Prolog library function

true.

15 ?- max4(4,3,2,1,M).

M = 4.

16 ?- max4(1,4,3,2,M).

M = 4.

17 ?- assert(max4(A,B,C,D,Max) :- Max is max(max(A,D), max(C,B))).

true.

18 ?- max4(1,4,3,2,M).

M = 4 .

19 ?- max4(4,3,2,1,M).

M = 4 ;

M = 4.
```

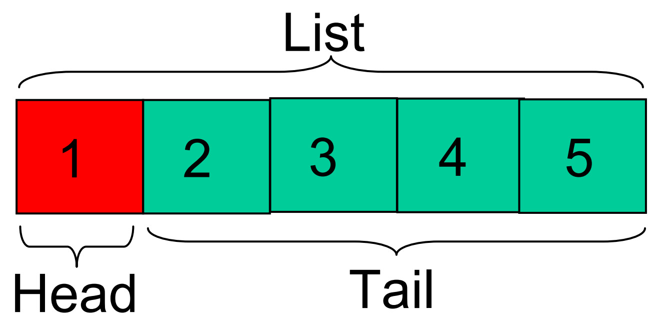
List Processing

List is a special form of binary tree (it must be terminated by an empty list)

List notation: [1 , 2 , 3 , 4 , 5]



Tree notation

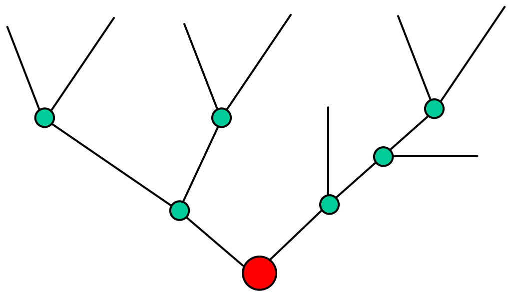


Graphic notation (w/o terminator)

The concept of recursion

- Recursion is a problem solving technique that creates a solution of problem of size *n* by combining solutions of the same problem with sizes that are less than *n*
- Object(n) = function(Object(n-1), Object(n-2))
  Object(1)=a, Object(2)=b; (a,b are constant boundary conditions)
- Recursive functions call themselves

Many problems are naturally recursive (e.g. a tree is obtained by combining two subtrees)



Examples of recursion

$$sum(n) = a_1 + \ldots + a_{n-1} + a_n = sum(n-1) + a_n$$

$$sum(n) = 1 + x + \ldots + x^{n-1} + x^n =$$

$$= sum(n-1) + x^n = 1 + x(sum(n-1))$$

$$\therefore \quad sum(n-1) = (1 - x^n)/(1 - x)$$

$$f(n) = n! = 1 \cdot 2 \cdots (n-1) \cdot n = f(n-1) \cdot n$$

$$f(1) = 1, \quad f(n-1) = f(n)/n, \quad f(0) = f(1)/1 = 1$$

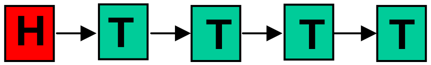
$$\max(n) = \max(a_1, \ldots, a_{n-1}, a_n) = \max(\max(n-1), a_n)$$

$$\max(k, n) = \max(a_k, a_{k+1}, \ldots, a_n) = \max(a_k, \max(k+1, n))$$

$$\max(n, n) = a_n$$

$$length(\text{list of size } n) = length(\text{list of size } n-1) + 1$$

$$length(\text{list}) = length(\text{tail}) + 1 \qquad (\text{list} = \text{head} + \text{tail})$$



Basic List Operations

```
[ ]   = empty list
[X]   = list containing a single element X
[H | T] = list containing a head element H and tail T
[_ | T] = list containing an anonymous head, and tail T
[H | _] = list containing head H and an anonymous tail
```

Anonymous variable (denoted as underscore _) is a variable that is not referenced (reused). It can be instantiated with any data object. Anonymous variables are used o avoid **singleton variables** (variables that are defined but never used).

Separator (vertical bar) operator ‘|’

- Primary function: separate the head from the tail of a list: [Head | Tail] ; Tail = sublist
- Additional function: separate several elements from the rest of a list:

[1 2 3]	Equivalent notations of the same list
[1 | [2 , 3]]	
[1 , 2 , | [3]]	
[1 , 2 , 3 | [ ]]	

- Note: right of the operator ‘|’ should be a list

```prolog
separate([Head | Tail]) :- write('List = '), write([Head | Tail]), nl,
                           write('Head = '), write( Head ), nl,
                           write('Tail = '), write( Tail) .
```

```
?- separate([1,2,3,4,5]).
List = [1, 2, 3, 4, 5]
Head = 1
Tail = [2, 3, 4, 5]
true.

?- separate([[]]).
List = [[]]
Head = []
Tail = []
true.

?- separate([[] | []]).
List = [[]]
Head = []
Tail = []
true.
```

```
?- separate([1,2,3 | [4,5]]).
List = [1, 2, 3, 4, 5]
Head = 1
Tail = [2, 3, 4, 5]
true.

?- separate([1,2,3 | 4, 5]).
List = [1, 2, 3| (4, 5)]
Head = 1
Tail = [2, 3| (4, 5)]
true.

?- separate([1, 2, 3 | 4]).
List = [1, 2, 3|4]
Head = 1
Tail = [2, 3|4]
true.

?- separate([1, 2 | 3]).
List = [1, 2|3]
Head = 1
Tail = [2|3]
true.
```

```
?- separate([2 | 3]).
List = [2|3]
Head = 2
Tail = 3
true.

?- islist([2 | 3]).  % Is not a list
false.

?- is_list([2 | 3]). % Library fun.
false.

?- .(_,_) = [2 | 3].  % Is tree
true.

?- separate([(1,2,3,4)]).
List = [ (1, 2, 3, 4)]
Head = 1, 2, 3, 4
Tail = []
true.
```

Boundary Conditions For Lists

```
proclist([ ]) :- <the case of empty list>.
proclist([X]) :- <the case of list with a single element X>
proclist([X|Rest]) :- <the case of one or more elements>
proclist([X,Y]) :- <the case of two elements X and Y>
proclist([X,Y | Rest]) :- <the case of two or more elements>
```

Examples:

```
allsame([_]).  % single (anonymous) element
allsame([X,Y|Rest]) :- X=Y, allsame([Y|Rest]). % two or more
sumlist([ ], 0). % empty list
sumlist([X|Rest],Sum) :- sumlist(Rest,SR),
                         Sum is X + SR. % one or more
```

Show List Elements

```
showlist( [ ] ) :- nl.
showlist([H|T]) :- write(H), tab(1),
                   showlist(T).
```

For an empty list display new line.

For nonempty list display the head of list, one space separator, and then display the tail of list.

```
1 ?- showlist([a,b,c,d,e]).
a b c d e
Yes
```

Test if an object is a list

```
islist([]).
islist([_|T]) :- islist(T).
```

Rules:

An empty list is a list.

A nonempty list is a list if its tail is a list

Similarly: `istree([_|_]).`

List Pattern Matching

```
?- [H|T]=[1,2].
H=1
T=[2]
?- [H|T]=[1].
H=1
T=[]
?- [H|T]=[ ].
No
?- .(H,T)=[ ].
No
```

The empty list has no head and no tail.

[ ] does not match with any nonempty list.

```
showlist( [ ] ) :- nl.
showlist([H|T]) :- write(H), tab(1), showlist(T).
?- showlist( [1,2,3] ).
1 2 3
Yes
?- showlist(123).
No
-------------------------------------------
showlst([H|T]) :- write(H), tab(1), showlst(T).
?- showlst([1,2,3]).
1 2 3
No
```

In this cases there is no boundary condition. At the end of recursive calls Prolog tries showlst([]), and this fails and produces the final “No”. As opposed to that, showlist([]) produces newline and “Yes”.

List Pattern Matching Using "|"

Usually, left of "|" are individual elements and right is a list

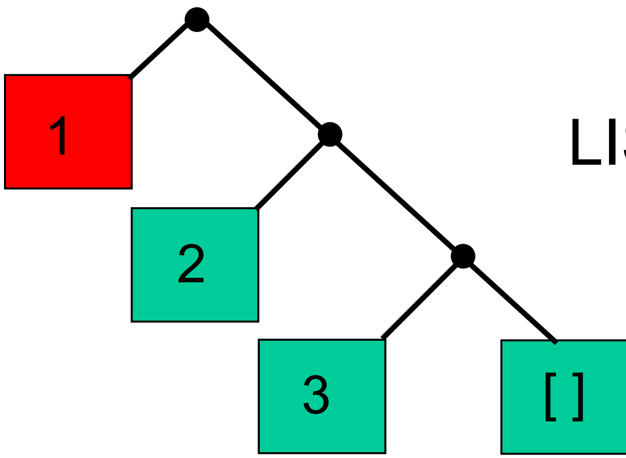
```
?- [A, B | C] = [1,2,3,4,5].
A=1
B=2
C = [3,4,5]

?- [1,2,3] = [A, B, C | D].
A = 1
B = 2
C = 3
D = [ ]
?- [1,2] = [A, B, C | D].
No
?- [1, 2 | 3, 4] = [A, B, C].
No
?- [1, [2 | 3], 4] = [A, B, C].
A = 1
B = [2 | 3]
C = 4
?- [1 | [2,3,4]] = [1,2,3,4].
Yes
?- [1, [2], 3] = [1, [A|B], 3].
A = 2
B = [ ]
```

```
?- [1 | [2|[3]]] = [A | B].
A = 1
B = [2, 3]

?- [1 | [2|[3]]] = [A , B].
No
```

```
53 ?- [H|T] = [1,2,3,4,5].
H = 1,
T = [2, 3, 4, 5].

54 ?- [H|T] = [1 | 2,3,4,5].
H = 1,
T = (2, 3, 4, 5).

55 ?- [H|T] = [1 | [2,3,4,5]].
H = 1,
T = [2, 3, 4, 5].
```

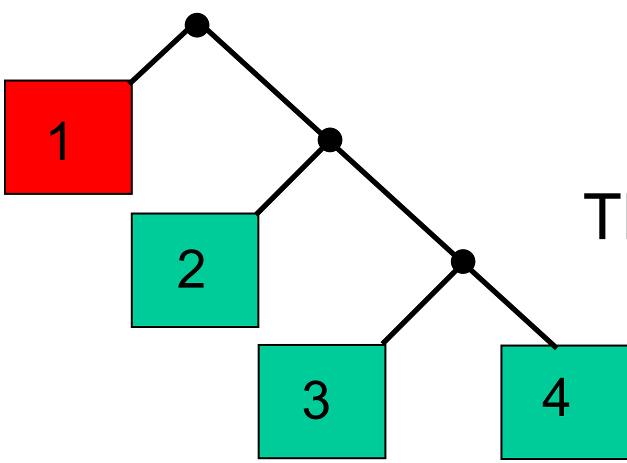
```
45 ?- islist([1, 2 | 3, 4]).
false.
46 ?- [1, 2 | 3, 4] = [A, B, C].
false.
47 ?- [1, 2 | 3, 4] = .(A, .(B, C)).
A = 1,
B = 2,
C = (3, 4).
48 ?- istree([1, 2 | 3, 4]).
true.
52 ?- [1, 2 | 3, 4] = .(A, B).
A = 1,
B = [2| (3, 4)].
```

Dot predicate notation of lists and trees



LIST: . (1, . (2, . (3, [ ]))) or [1, 2, 3]

Terminator
(empty list)



TREE: . (1, . (2, . (3, 4))) or [1, 2, 3 | 4]

(No terminator)

```
?- X = .(1, .(2, .(3, []))).
X = [1, 2, 3].
?- X = .(1, .(2, .(3, 4))).
X = [1, 2, 3|4].
?- is_list( .(1, .(2, .(3, 4))) ). % Library function
false.
?- is_list( .(1, .(2, .(3, []))) ).
true.
?- is_list( [1,2, 3|4] ).
false.
?- is_list( [1,2,3] ).
true.
?- is_list( [1,2,3 | []] ).
true.
```

```
?- [X,Y,Z|T]=[1,2,3,4].
X = 1,
Y = 2,
Z = 3,
T = [4].

?- [X,Y,Z|T]=[1,2,3|4].
X = 1,
Y = 2,
Z = 3,
T = 4.

?- [X,Y,Z,T]=[1,2,3|4].
false.
```

```
?- [X,Y] = [first | second].
false.
?- [X,Y]=[first, second | []].
X = first,
Y = second.
?- is_list([1,2,3|4]).
false.
?- is_list([1,2,3|[4]]).
true.
?- X=[1,2,3|[4]].
X = [1, 2, 3, 4].
```

```
?- [1|2] = .(1,2).
true.
?- [1,[2|3]|4] = .(1, .(.(2,3),4)).
true.
?- X = .(1, 2).
X = [1|2].
?- X = .(.(1,2) , .(3,4)).
X = [[1|2], 3|4].
?- istree( .(.(1,2) , .(3,4)) ).
true.
?- [H|T] = ( .(.(1,2) , .(3,4)) ).
H = [1|2],
T = [3|4].
?- is_list( .(.(1,2) , .(3,4)) ).
false.
```

```
istree([_|_]).
?- istree(17).
false.

?- istree([17]).
true.
```

Last Element in a List

```
% Last element of a list (List, Last element)
lastelement([X], X).
lastelement([_|T], X) :- lastelement(T, X).
```

If the list contains one element, this is the last element.

If the list contains more than one element, then the last element is the last in the tail of the list.

```
?- lastelement([a,b,c,d,e], X).
X = e ;
No
?- lastelement([a,b,c,d,X], e).
X = e ;
No
?- lastelement([a,b,c,d,[1,[2,3],4]], X).
X = [1, [2, 3], 4] ;
No
```

Two Adjacent Elements in a List

```
adjacent(First element, Second element, List)

adjacent(X, Y, [X,Y|_]).

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

Two first elements in a list are adjacent elements.

If the adjacent elements are not the first two elements, then they may be two adjacent elements in the tail of the list.

```
?- adjacent(X,Y, [1,2,3,4]).

X = 1
Y = 2 ;

X = 2
Y = 3 ;

X = 3
Y = 4 ;

No
```

```
?- adjacent(2,Y, [1,2,3,4]).

Y = 3 ;

No
?- adjacent(X,Y, [1,[2,3],4]).

X = 1
Y = [2, 3] ;

X = [2, 3]
Y = 4 ;

No
```

Member of a List

```
% member_of_list (Member, List)
member1(X, [X|_]).
member1(X, [_|T]) :- member1(X, T).
```

Member of the list is the head of list.

If X is not the head of list, then it is a member of the list if it is the member of the tail of the list.

```
?- member1(2,[1,2,3]).

Yes
?- member1(7,[1,2,3]).

No
?- member1(X, [1,2,3]).

X = 1 ;
X = 2 ;
X = 3 ;

No
```

```
?- member1(2,[1,X,3]).

X = 2 ;

No
?- member1(2,L).

L = [2|_G314] ;
L = [_G313, 2|_G317] ;
L = [_G313, _G316, 2|_G320]

Yes
10 ?- member1(2,[2]).

Yes
```

Library function member and the use of cut

% Without cut all results are available:

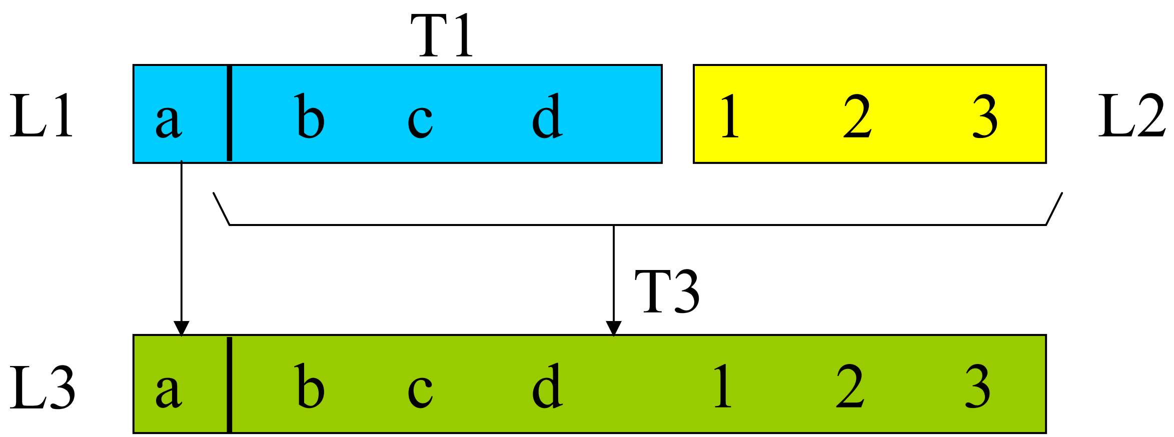
```
?- member(X, [1,2,3]).

X = 1 ;
X = 2 ;
X = 3 ;

false.
```

% With cut only the first result is available:

```
?- member(X, [1,2,3]), ! .

X = 1.

?-
```

Append List

```
append1([], L2, L2). % (first, second, result)
append1([X|T1], L2, [X|T3]):- append1(T1,L2,T3).
```



Appending an empty list to a list gives the unchanged list.

Appending L1 and L2 yields L3 where the head of L3 is the same as the head of L1, and the tail of L3 is the tail of L1 plus L2.

Flexibility

(interchangeability of inputs and outputs)

If the results are based on pattern matching, then every argument can be either input or output.

```
1 ?- append([1,2],[3,4],[1,2,3,4]).
Yes
2 ?- append(X,[3,4],[1,2,3,4]).
X = [1, 2] ;
No
3 ?- append([1,2],Y,[1,2,3,4]).
Y = [3, 4] ;
No
4 ?- append(X,Y,[1,2,3,4]).
X = []
Y = [1, 2, 3, 4] ;
X = [1]
Y = [2, 3, 4] ;
X = [1, 2]
Y = [3, 4] ;
X = [1, 2, 3]
Y = [4] ;
X = [1, 2, 3, 4]
Y = [] ;
No
```

Note: append is a library function

```
5 ?- append([1,2],[3,4],Z).
Z = [1, 2, 3, 4] ;
No
6 ?-
append([1,2],[3,4],[1,2,X,4]).
X = 3 ;
No
7 ?-
append([1,Y],[3,4],[1,2,X,4]).
Y = 2
X = 3 ;
No
8 ?-
append([1,Y],[3,Z],[1,2,X,4]).
Y = 2
Z = 4
X = 3 ;
No
9 ?-
append([X,2],[3,4],[Y,2,3,4]).
X = _G365
Y = _G365 ;
No
```

Three aspects of flexibility

- Prolog predicates can define relationships between objects. Such predicates are **flexible** if **any subset** of objects can be used as inputs, and remaining objects are used as output.
- Three characteristic cases are:
  - (1) All objects are used as inputs and Prolog answer is yes/no or true/false. E.g. friends(peter,john).
  - (2) A subset of objects is used as inputs and remaining object are outputs. E.g. friends(peter,X).
  - (3) All objects are used as outputs, generating all combinations or results. E.g. friends(X,Y).

An example of flexible predicate: friends

```
likes(peter, wine).
likes(john,wine).
likes(bill,wine).
likes(mary,food).

friends(X,Y) :- likes(X,S), likes(Y,S), X\=Y.
```

% Inputs only

```
1 ?- friends(peter,john).
true.
```

% Inputs and outputs

```
2 ?- friends(peter,X).
X = john ;
X = bill ;
false.
```

% Outputs only

```
3 ?- friends(X,Y).
X = peter,
Y = john ;
X = peter,
Y = bill ;
X = john,
Y = peter ;
X = john,
Y = bill ;
X = bill,
Y = peter ;
X = bill,
Y = john ;
false.
```

Remove Initial Part of a List

```
% List L1 minus list L2 equals list L3
listminuslist(L1, L2, L3) :-
                        append1(L2, L3, L1).
```

L3 = L1 – L2 (L2 must be the initial part of the list L1).

L3 = L1 – L2 is equivalent to L1 = L2 + L3

```
?- listminuslist([1,2,3,4,5,6],[1,2],L).
L = [3, 4, 5, 6] ;
No

?-
listminuslist([1,2,3,4,5,6],[2,3,4],L).
No
```

```
?- listminuslist([1,2,3,4,5,6],M,L).

M = []
L = [1, 2, 3, 4, 5, 6] ;
M = [1]
L = [2, 3, 4, 5, 6] ;
M = [1, 2]
L = [3, 4, 5, 6] ;
M = [1, 2, 3]
L = [4, 5, 6] ;
M = [1, 2, 3, 4]
L = [5, 6] ;
M = [1, 2, 3, 4, 5]
L = [6] ;
M = [1, 2, 3, 4, 5, 6]
L = [] ;

No
```

Delete an Element of List

```
% delete(Element, From list, Resulting list)
del(X, [X|T], T).
del(X, [H|T], [H|T1]) :- del(X, T, T1).
```

After deleting the head of list the resulting list is the tail.

Otherwise, the element X is deleted from the tail of the list. The head of list remains unchanged.

```
?- del(2, [1,2,3], L).
L = [1, 3] ;
No
?- del(7, [1,2,3], L).
No
?- del(X, [1,2,3], [1,3]).
X = 2 ;
No
```

```
?- del(X, [1,2,3], L).
X = 1
L = [2, 3] ;

X = 2
L = [1, 3] ;

X = 3
L = [1, 2] ;
No
```

Insert an Element in a List

```
% insert(Element, List, Expanded list)
insert(X, L, XL) :- del(X, XL, L).
```

Inserting X in L gives XL if deleting X from XL gives L. In other words, if X + L = XL, then L = XL – X.

Both Delete and Insert are flexible. Inserting and deleting are inverse operations; if the flexibility works, insert operation can be interpreted as inverse deleting.

```
?- insert(1, [2,3,4], L).

L = [1, 2, 3, 4] ;

L = [2, 1, 3, 4] ;

L = [2, 3, 1, 4] ;

L = [2, 3, 4, 1] ;

No
```

```
?- insert(E,L, [1,2,3]).

E = 1
L = [2, 3] ;

E = 2
L = [1, 3] ;

E = 3
L = [1, 2] ;

No
```

Select

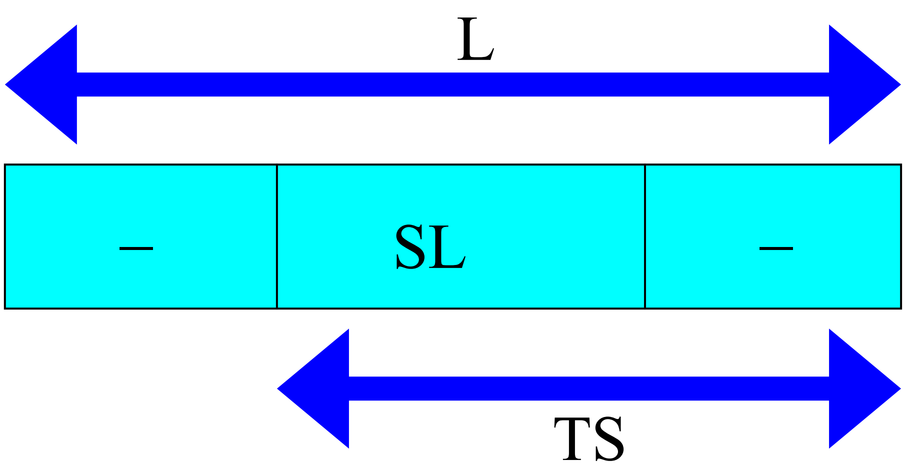
- Flexible remove of an element from list

```
?- select(e, [1,e,2,e,3], L).
L = [1, 2, e, 3] ;
L = [1, e, 2, 3] ;
No
?- select(e, L, [1,2,3]).
L = [e, 1, 2, 3] ;
L = [1, e, 2, 3] ;
L = [1, 2, e, 3] ;
L = [1, 2, 3, e] ;
No
```

Sublist of a List (1)

```
sublist(SL, L) :- append1(_, TS, L),
                  append1(SL, _, TS).
```

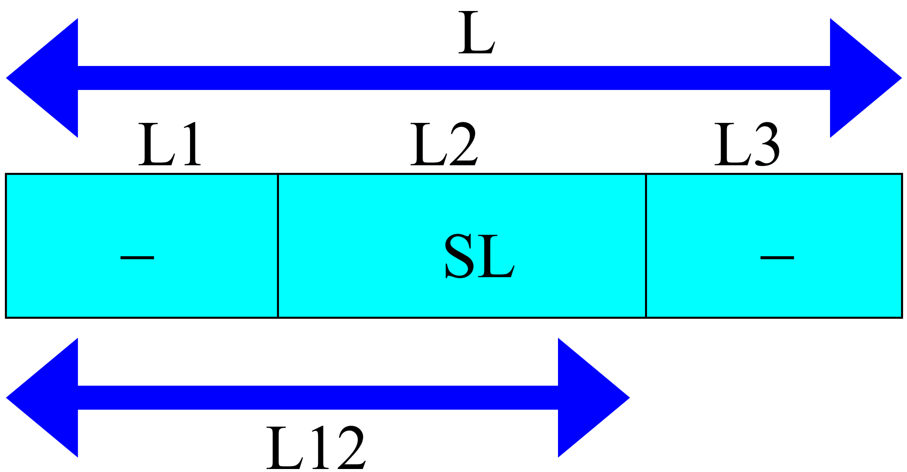
The sublist (SL) of a list is a head sublist of the tail sublist (TS).



Sublist of a List (2)

```
append3(L1, L2, L3, L) :- append1(L1, L2, L12),
                          append1(L12, L3, L).
sublist2(SL, L) :- append3(_, SL, _, L).
```

The sublist (SL) of a list L is defined as a middle component in appending three lists to produce the list L.



```
?- sublist(SL, [1,2,3]).

SL = [] ;
SL = [1] ;
SL = [1, 2] ;
SL = [1, 2, 3] ;
SL = [] ;
SL = [2] ;
SL = [2, 3] ;
SL = [] ;
SL = [3] ;
SL = [] ;

No
```

```
?- sublist2(SL, [1,2,3]).

SL = [] ;
SL = [1] ;
SL = [1, 2] ;
SL = [1, 2, 3] ;
SL = [] ;
SL = [2] ;
SL = [2, 3] ;
SL = [] ;
SL = [3] ;
SL = [] ;
ERROR: Out of global stack
```

(both system append and append1 generate this result)

Subset

```
% Subset:   sset(Subset, Set)
sset([],[]).
sset([Head | SubTail], [Head | Tail]) :- sset(SubTail, Tail).
sset(SS, [_ | Tail]) :- sset(SS, Tail).
powerset(Set, PS) :- findall(SS, sset(SS, Set), PS).
```

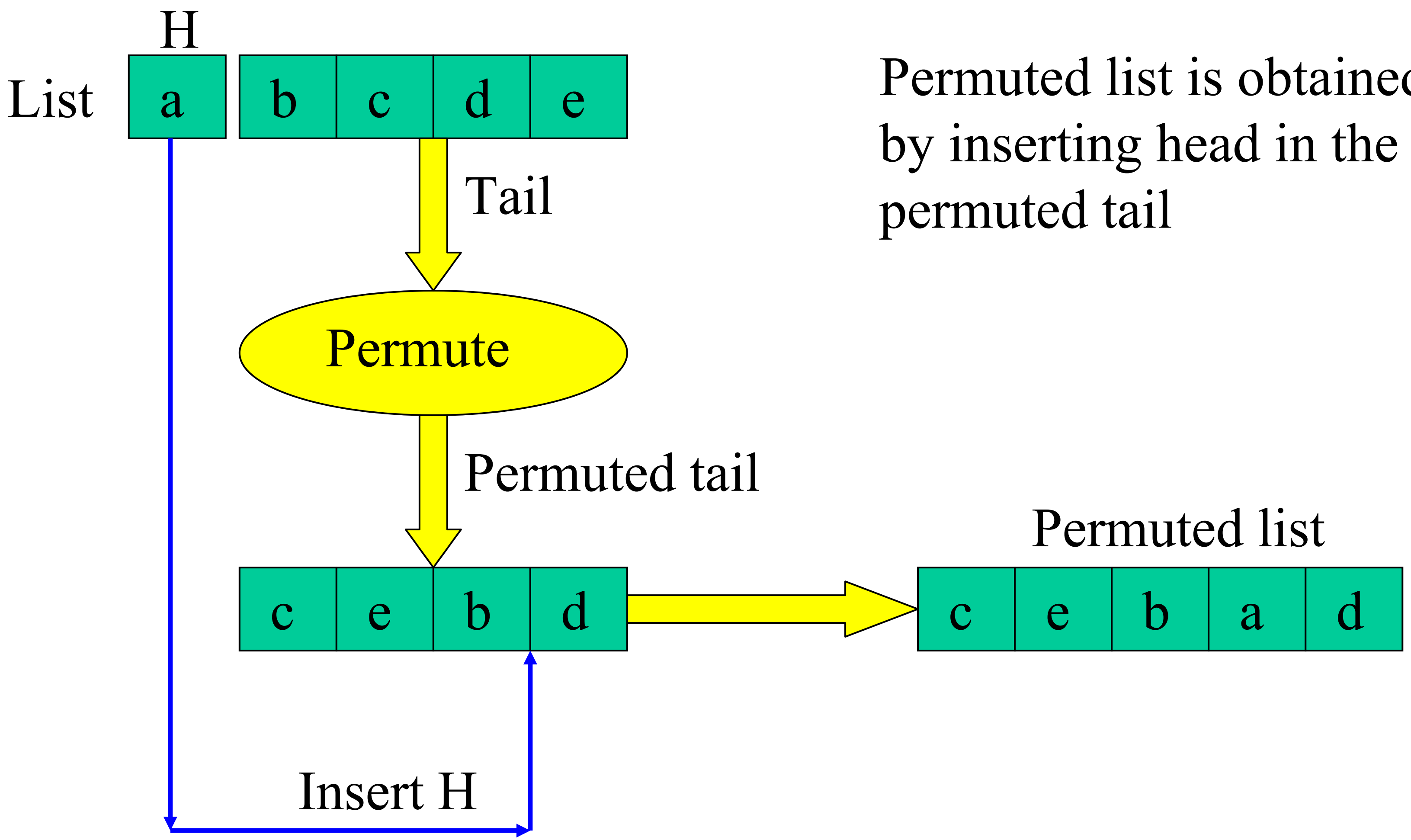
1. Subset of an empty set is the empty set
2. If the subset and the set have the same head, then the tail of subset must be a subset of the tail of the original set.
3. If the head of set is different form the head of subset, then the subset must be contained in the tail of set.

```
1 ?- sset(SS, [1,2,3]).
SS = [1, 2, 3] ;
SS = [1, 2] ;
SS = [1, 3] ;
SS = [1] ;
SS = [2, 3] ;
SS = [2] ;
SS = [3] ;
SS = [] ;
false.
```

```
3 ?- powerset([a,b],PS).
PS = [[a, b], [a], [b], []]
6 ?- powerset([1,2,3],PS).
PS = [[1, 2, 3], [1, 2],
[1, 3], [1], [2, 3], [2],
[3], []] .
```

```
2 ?- sset([1,2], S).
S = [1, 2] ;
S = [1, 2, _G354] ;
S = [1, 2, _G354, _G357] ;
S = [1, 2, _G354, _G357, _G360] ;
S = [1, 2, _G354, _G357, _G360, _G363] ;
S = [1, 2, _G354, _G357, _G360, _G363, _G366] .
```

Permutation of List (1)



Permuted list is obtained by inserting head in the permuted tail

Permutation of List (2)

```
%permute(Original list, Permuted list)
permute([], []).
permute([H|T], P) :- permute(T, T1),
                     insert(H, T1, P).
```

The permutation of an empty list is the same empty list.

For nonempty list a permutation of elements is obtained by inserting head of list in a permutation of tail elements.

```
?- permute([1,2,3], X).

X = [1, 2, 3] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [1, 3, 2] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;

No
```

```
?- permute(X, [1,2,3]).

X = [1, 2, 3] ;
X = [2, 1, 3] ;
X = [3, 1, 2] ;
X = [1, 3, 2] ;
X = [2, 3, 1] ;
X = [3, 2, 1] ;

Action (h for help) ? abort
% Execution Aborted
```

Prolog (library) permutation

```
?- permutation([1,2,3],L).
L = [1, 2, 3] ;
L = [2, 1, 3] ;
L = [2, 3, 1] ;
L = [1, 3, 2] ;
L = [3, 1, 2] ;
L = [3, 2, 1] ;
No
```

```
?- permutation(L,[1,2,3]).
L = [1, 2, 3] ;
L = [2, 1, 3] ;
L = [3, 1, 2] ;
L = [1, 3, 2] ;
L = [2, 3, 1] ;
L = [3, 2, 1] ;
No
```

Naïve reverse (inefficient list reverse program frequently used as a benchmark)

```
% reverse(Original list, Reversed list)

nreverse([],[]).
nreverse([H|T], Rev) :- nreverse(T,RT),
                        append(RT, [H], Rev).
```

To get a reversed list, append the head to the reversed tail.
List = [a,b,c,d]
Head = a
Tail = [b,c,d]
Reversed tail = [d,c,b]
Reversed tail + head = [d,c,b] + [a] = [d,c,b,a]

```
?- nreverse([a,b,c,d], R).
R = [d, c, b, a].
?- nreverse(R, [a,b,c,d]).
R = [d, c, b, a] ;
Action (h for help) ? abort
% Execution Aborted
?- nreverse(R, [a,b,c,d]).
R = [d, c, b, a] .
?- nreverse([1,2,3,4], [4,X,2,1]).
X = 3.
?- nreverse(R, [a,b,c,d]).
R = [d, c, b, a] .
```

```
% Library function reverse

?- reverse([a,b,c,d], R).
R = [d, c, b, a].
?- reverse([1,2,3,4],
[4,X,2,1]).
X = 3.
?- reverse(R, [a,b,c,d]).
R = [d, c, b, a] ;
false.
```

Tail Recursion

- Tail recursion is a problem solving technique that consists of separating the head and tail of a list using [H | T]
- When H and T are instantiated, the same procedure is recursively applied to the tail T (of course, T is shorter than the initial list)
- At the end of this process T is either empty ([ ]) or a single element ([X]). This simple case is then used as a boundary condition (and must be located as the first rule)

Example of tail recursion: list length

```
len([], 0).   % length(L, Len) is a Prolog library predicate
len([_|T], N) :- len(T, N1), N is 1 + N1.
```

Rules:
The length of an empty list is zero.
The length of a nonempty list is the length of its tail plus one.

Note: assignment is a procedural concept and the resulting predicates (both len and length) are not flexible

```
1 ?- len([1,2,_],L).
L = 3
Yes
2 ?- len([_,_,_],L).
L = 3
Yes
```

```
3 ?- len(List,3).
List = [_G318, _G321, _G324] ;
Action (h for help) ? abort
% Execution Aborted
4 ?- length(List,3).
List = [_G339, _G342, _G345]
Yes
5 ?-
```

Example of tail recursion: list maximum

```
maxlist([X],X).   % the case of single element
maxlist([H|T],Max) :- maxlist(T,Tmax),
                      Max is max(H,Tmax).
```

Rules:

The maximum of a list with one element is that element.

The maximum of a list with two or more elements is the largest of the following two values: the head and the maximum of the tail (max(X,Y) is a Prolog library function of 2 arguments)

```
?- maxlist([1,2,3,2,1], Max).
Max = 3 ;
No
```

Flexibility and inflexibility

- Prolog rules based on pattern matching mechanism define **relationships between data objects**. E.g. p(A, B, C, D) creates relationships between A, B, C, and D.
- In the case of flexible program there is no difference between inputs and outputs: any subset of variables can be input, and remaining variables are then the output.
- Flexibility is related to nonprocedural programming; some Prolog programs include procedural components and are not flexible (one such example is the system sort program: a tradeoff between flexibility and performance).
- **Flexibility:** interchangeability of inputs and outputs.
- **Inflexibility:** inputs and outputs are not interchangeable

NEGATION

not(X) succeeds if an attempt to satisfy goal X fails

```
not_member(_,[ ]).
not_member(X,[H|T]) :- X \= H, not_member(X,T).

is_set1([]).
is_set1([H|T]) :- not_member(H,T), is_set1(T).


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

isset([]).
isset([H|T]) :- not(is_member(H,T)), isset(T).
```

```
1 ?- isset([]).

Yes
2 ?- isset([1]).

Yes
3 ?- isset([1,a,v,3]).

Yes
4 ?- isset([3,1,a,v,3]).

No
5 ?- is_set1([]).

Yes
6 ?- is_set1([1]).

Yes
7 ?- is_set1([1,a,v,3]).

Yes
8 ?- is_set1([3,1,a,v,3]).

No
```

Prolog sort predicate

```
11 ?- sort([7,3,5,2,1,6,4], X).
X = [1, 2, 3, 4, 5, 6, 7] ;
No
12 ?- sort([7,3,5,2,1,6,4], [1,2,3,4,5,6,7]).
Yes
13 ?- sort(X, [1,2,3,4,5,6,7]).
ERROR: Arguments are not sufficiently
instantiated
```

Flexible Sort (1)

```
% Test whether a list is sorted
sorted1([_]).
sorted1([H,A|B]) :-  H =< A, sorted1([A|B]).

fsort1(List, Slist) :- permutation(List, Slist),
                       sorted1(Slist).
```

```
?- fsort1([4,3,2,1],S).
S = [1, 2, 3, 4] ;
No
```

```
?- fsort1(L,[1,2,3]).
L = [1, 2, 3] ;
L = [2, 1, 3] ;
L = [3, 1, 2] ;
L = [1, 3, 2] ;
L = [2, 3, 1] ;
L = [3, 2, 1] ;
No
```

(Flexible sort is extremely inefficient: O(n!))

Flexible Sort (2)

```
% Test whether a list is sorted
sorted(L) :- length(L,Len), Len < 2.
sorted([H|[A|B]]) :-  H =< A, sorted([A|B]).

fsort(List, Slist) :- permutation(List, Slist),
                      sorted(Slist).
```

```
?- fsort([4,3,2,1],S).
S = [1, 2, 3, 4] ;
No

?- fsort(L,[1,2,3]).
L = [1, 2, 3] ;
L = [2, 1, 3] ;
L = [3, 1, 2] ;
L = [1, 3, 2] ;
L = [2, 3, 1] ;
L = [3, 2, 1] ;
No
```

(Flexible sort is extremely inefficient: O(n!))

Flexible Sort (3)

```
le(a,b).
le(b,c).
le(c,d).
le(d,e).
le(a,a).                        %  Other combinations???
le(b,b).                        %  E.g. le(a,d).
le(c,c).
le(d,d).
le(e,e).
sorted2([_]).
sorted2([H,A|B]) :- le(H,A), sorted2([A|B]).
fsort2(List,Slist) :- permutation(List,Slist), sorted2(Slist).
```

```
11 ?- fsort2([a,b,c],[a,b,c]).
true .
12 ?- fsort2([d,c,b,a],S).
S = [a, b, c, d] ;
false.
13 ?- fsort2([e,b,a],L).
No
```

```
10 ?- fsort2(L,[a,b,c]).
L = [a, b, c] ;
L = [b, a, c] ;
L = [c, a, b] ;
L = [a, c, b] ;
L = [b, c, a] ;
L = [c, b, a] ;
false.
```

```
le(a,a).
le(b,b).
le(c,c).
le(d,d).
le(e,e).

le(a,b).
le(a,c).
le(a,d).
le(a,e).
le(b,c).
le(b,d).
le(b,e).
le(c,d).
le(c,e).
le(d,e).

%transitive_le(X,Y) :- le(X,Y).
%transitive_le(X,Y) :- le(X,Temp), transitive_le(Temp,Y).

sorted([_]).
sorted([H,A|B]) :- le(H,A), sorted([A|B]).
fsort(List,Slist) :- permutation(List,Slist), sorted(Slist).
```

Inflexible Bubble and Select Sorts

```
% Bubble sort
% Partition the list, find two adjacent elements in
% wrong order, swap them, and repeat until sorted
bsort(L,S) :- append(K, [A, B | T], L), B<A, !,
              append(K, [B, A | T], M), bsort(M,S).
% Deliver the sorted list
bsort(L,L).
```

```
% Select sort
% First put the minimum value in the first position
ssort(L,S) :- append([H1|T1],[H2|T2],L), H2<H1, !,
              append([H2|T1],[H1|T2],M), ssort(M,S).
% Then sort the tail
ssort([H|T],S) :- ssort(T,ST), append([H], ST, S),!.
% Deliver the sorted list
ssort(L,L).
```

```
?- bsort([5,3,4,2,3,1],S).
S = [1, 2, 3, 3, 4, 5] ;
No
?- ssort([5,3,4,2,3,1],S).
S = [1, 2, 3, 3, 4, 5] ;
No
?- bsort([1,3,5,7,9],S).
S = [1, 3, 5, 7, 9] ;
No
?- ssort([1,3,5,7,9],S).
S = [1, 3, 5, 7, 9] ;
No
?- bsort([1],S).
S = [1] ;
No
?- ssort([1],S).
S = [1] ;
No
```

```
?- bsort([ ],S).
S = [ ] ;
No

?- ssort([ ],S).
S = [ ] ;
No

?- ssort(L,[1,2,3]).
ERROR: Arguments are not sufficiently instantiated
   Exception: (8) ssort(_G302, [1, 2, 3]) ? abort
% Execution Aborted

?- bsort(L,[1,2,3]).
ERROR: Arguments are not sufficiently instantiated
   Exception: (8) bsort(_G302, [1, 2, 3]) ? abort
% Execution Aborted
?-
```

Loops and Repetitions (Procedures)

Using Recursion to Make a List of N Natural Numbers [1, 2, …, N]

```
makelist(N, []) :- N =< 0.
makelist(N, L ) :- N > 0, N1 is N-1,
                   makelist(N1, L1),
                   append1(L1, [N], L).
```

For zero elements (N=0) an empty list is created.

For N elements (N>0) the list is created by appending N to the list of N-1 elements

Same as Prolog library predicate numlist(MinInt, MaxInt, List).

Loop

```
loop :-
  repeat,                   % repeat always succeeds
  nl,                       % new line
  write('Enter an integer or "stop" to end the program: '),
  read(X),
  ( X = stop, !             % if X is "stop" end the program
    ;                       % ... or
    Y is X*X,               % compute and display results
    write('Square of '), write(X), write(' is '), write(Y),
    fail                    % fail causes backtrack to repeat
  ).
```

```
?- consult(loop).
% loop compiled 0.00 sec, 72 bytes
Yes
?- loop.
Enter an integer or "stop" to end the program: 3.
Square of 3 is 9
Enter an integer or "stop" to end the program: 32.
Square of 32 is 1024
Enter an integer or "stop" to end the program: 1024.
Square of 1024 is 1048576
Enter an integer or "stop" to end the program: stop.
Yes
?-
```

Loop with Recursion

```
looprec :-
  write('Enter an integer or "stop" to end the program: '),
  read(X),
  process(X).

  process(stop) :- !.

  process(N) :-
    Y is N*N,
    write('Square of '), write(N), write(' is '), write(Y),
    nl,
    looprec.
```

Output of this program is the same as the output of loop.

Test and debug 1

```
testappend :-
  % Perform an operation
  append(X,Y,[1,2,3]),

  % write results
  write(X), write(' '), write(Y), write('\n'),

  % repeat (continue) from the beginning
  fail.            % Backtrack to append
```

Test and debug 2

```
2 ?- testappend.

[] [1, 2, 3]

[1] [2, 3]

[1, 2] [3]

[1, 2, 3] []


No

3 ?-
```

Test and debug 3

```
del(X, [X|T], T).
del(X, [H|T], [H|T1]) :- del(X, T, T1).

insert(X, L, XL) :- del(X, XL, L).

permute([], []).
permute([H|T], P) :- permute(T, T1), insert(H, T1, P).

testpermute :- % Perform an operation
              permute([1,2,3],X),

              % write results
              write(X), write('\n'),

              fail.  % repeat (continue) from the beginning
              % This generates all results
```

Test and debug 4

```
1 ?- testpermute.

[1, 2, 3]

[2, 1, 3]

[2, 3, 1]

[1, 3, 2]

[3, 1, 2]

[3, 2, 1]


No
```

If-then-else

(

<condition> , <then part> ; <else part>

)

Example of if-then-else: maximum

```
max1(X,Y,Y) :- X =< Y, ! .    % max1,2,3 programs behave similarly.
max1(X,Y,X) :- X > Y .        % max1 and max2 are partially flexible:
                              % max1(1,Y,2). returns Y=2
max2(X,Y,Y) :- X =< Y.        % max2(X,2,2). returns error, but it
max2(X,Y,X) :- X > Y .        % works fine in other applications

max3(X,Y,M) :- (X =< Y, M is Y , !) ; M is X .
max4(X,Y,M) :- (X =< Y, M is Y) ; (X > Y, M is X) .
```

These functions are similar to the Prolog library function max(X,Y)

Results of max*

```
?- max1(1,7,M).
M = 7
Yes
?- max2(1,7,M).
M = 7
Yes
?- max3(1,7,M).
M = 7
Yes
?- max4(1,7,M).
M = 7 ;
No
```

```
?- max1(1,X,7).
X = 7 ;
No
?- max2(1,X,7).
X = 7 ;
No
?- max3(1,X,7).
ERROR: Arguments are not sufficiently
instantiated
?- max4(1,X,7).
ERROR: Arguments are not sufficiently
instantiated
?- max1(X,7,7).
ERROR: Arguments are not sufficiently
instantiated
   Exception: (7) max1(_G260, 7, 7) ?
creep
?- max2(X,7,7).
ERROR: Arguments are not sufficiently
instantiated
   Exception: (7) max2(_G260, 7, 7) ?
abort
% Execution Aborted
```

The role of cut

```
min2(X,Y,Min) :- X =< Y, Min=X, ! ; Min=Y.
minlist([X],X).    % The case of single element
minlist([H|T], Min) :- minlist(T, Tmin),
                       min2(H, Tmin, Min).
```

If we remove cut from min2 then **minlist** will generate multiple min values, or wrong results (try [1,2,1,2,1,2,1,2]):

10 ?- minlist([4,3,2,1,2,3,1], Min).

Min = 1 ;

Min = 1 ;

No

List maximum

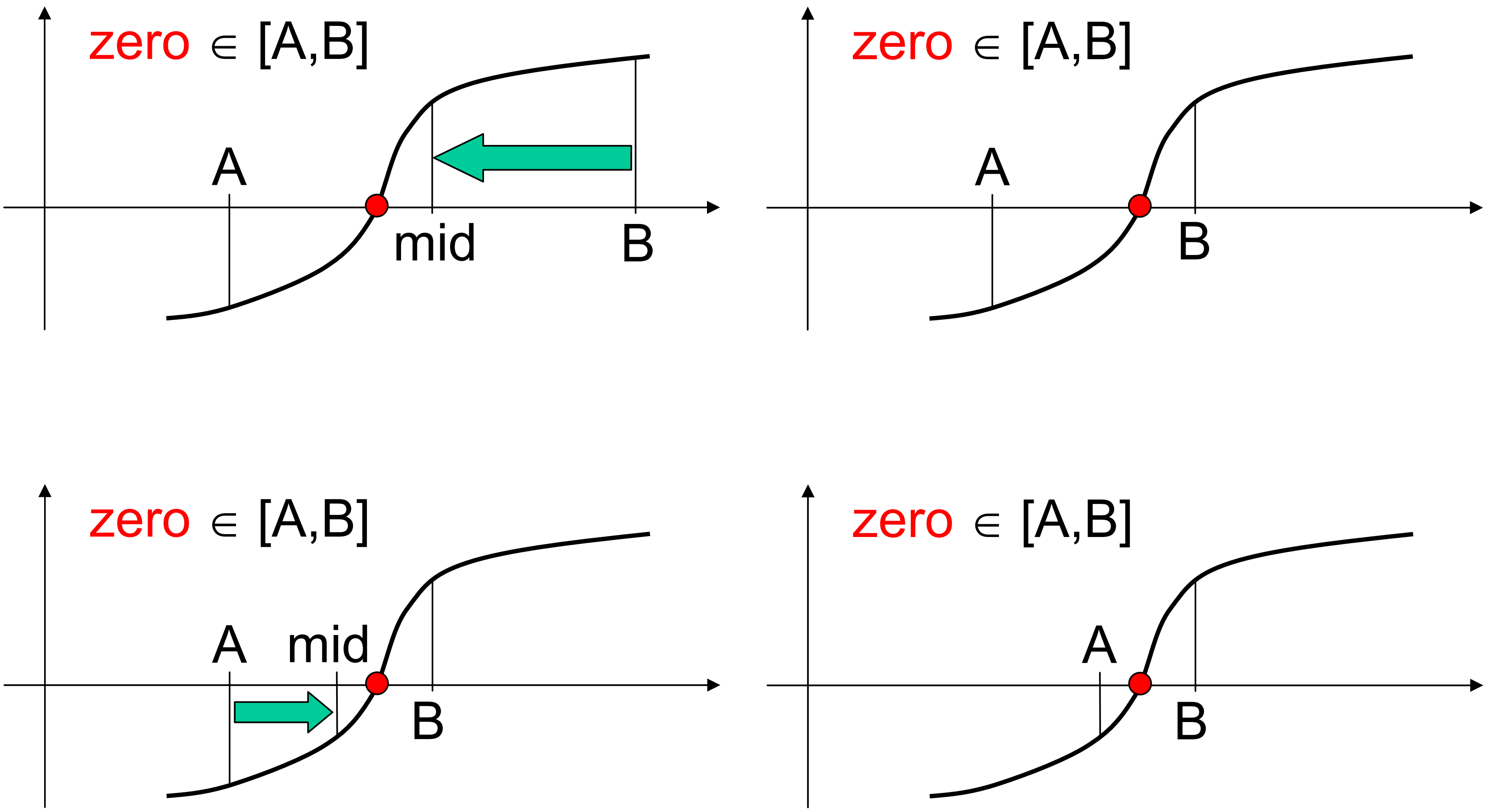
```
max1(X,Y,Y) :- X =< Y, ! .
max1(X,Y,X) :- X > Y .
max2(X,Y,Y) :- X =< Y.
max2(X,Y,X) :- X > Y .
max3(X,Y,M) :- (X =< Y, M is Y , !) ; M is X .
max4(X,Y,M) :- (X =< Y, M is Y) ; (X > Y, M is X) .

maxlist1([X],X).         % The case of single element
maxlist1([H|T],Max) :- maxlist1(T,Tmax), max1(H,Tmax,Max).
maxlist2([X],X).         % The case of single element
maxlist2([H|T],Max) :- maxlist2(T,Tmax), max2(H,Tmax,Max).
maxlist3([X],X).         % The case of single element
maxlist3([H|T],Max) :- maxlist3(T,Tmax), max3(H,Tmax,Max).
maxlist4([X],X).         % The case of single element
maxlist4([H|T],Max) :- maxlist4(T,Tmax), max4(H,Tmax,Max).
maxlist5([X],X).         % The case of single element
maxlist5([H|T],Max) :- maxlist5(T,Tmax), Max is max(H,Tmax).
```

List maximum: results

```
?- maxlist1([1,2,3,2,1], Max).
Max = 3 ;
No
?- maxlist2([1,2,3,2,1], Max).
Max = 3 ;
No
?- maxlist3([1,2,3,2,1], Max).
Max = 3 ;
No
?- maxlist4([1,2,3,2,1], Max).
Max = 3 ;
No
?- maxlist5([1,2,3,2,1], Max).
Max = 3 ;
No
```

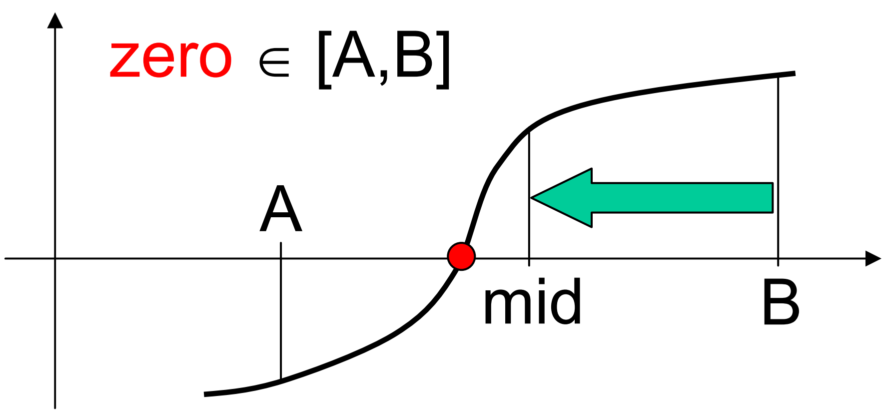
```
?- maxlist1([X,1,2,3,2,1], 7).
X = 7 ;
No
?- maxlist2([X,1,2,3,2,1], 7).
X = 7 ;
No
?- maxlist3([X,1,2,3,2,1], 7).
ERROR: Arguments are not sufficiently instantiated
?- maxlist4([X,1,2,3,2,1], 7).
ERROR: Arguments are not sufficiently instantiated
?- maxlist5([X,1,2,3,2,1], 7).
ERROR: Arguments are not sufficiently instantiated
^  Exception: (8) 7 is max(_G350, 3) ? abort
% Execution Aborted
?- maxlist1([1,2,3,2,1,X], 7).
ERROR: Arguments are not sufficiently instantiated
   Exception: (13) max1(1, _G368, _L190) ? abort
% Execution Aborted
```

Bisection method: f(x)=0, x=?



Recursive bisection method

```
double zero_rec(double A, double B)     // Bisection method
{                                       // for solving F(x)=0
  double mid=(A+B)/2.;
  const double eps = 0.000001;          // Max error of the
                                        // resulting root
  if(F(mid)==0. || B-A<eps) return mid; // Boundary condition
  if(F(A)*F(mid)>0.) A=mid; else B=mid;
  return zero_rec(A,B);                 // Recursive call
}
```



Computing the zero of function using the bisection method

```
f(X,Y) :- Y is X*(1-X).

zero(A,B,_) :- A >= B,
               write('Error: A >= B'),
               nl, !.

zero(A,B,_) :- f(A,Ya), f(B,Yb),
               Ya*Yb>0,
               write('Error: f(A)*f(B)>0'),
               nl, !.

zero(A,B,X) :- Mid is (A+B)/2.0,
               f(Mid,Y), Y =:= 0,
               X is Mid,
               write('Exact result'), !.
```

```
zero(A,B,X) :- B-A < 0.0000001,
               X is (A+B)/2.0, !.

zero(A,B,X) :- Mid is (A+B)/2.0,
               f(Mid,Ymid), f(A,Ya),
               (  % if-then-else
                  Ymid*Ya > 0,
                  zero(Mid,B,X)
               ;
                  zero(A,Mid,X)
               ).
```

Results of the bisection method

```
?- zero(0, 2, X).
Exact result
X = 1.0.

?- zero(1, 2, X).
X = 1.0.

?- zero(-10, 0, X).
X = -3.72529e-008 .

?- zero(-10, 0.5, X).
X = -9.31323e-009 .

?- zero(0.1, 100, X).
X = 1.0 .
```

```
?- zero(5, 10, X).
Error: f(A)*f(B)>0
true.

?- zero(10, 5, X).
Error: A >= B
true.
```

Forall predicate: testing all components

```
% Generally:  forall(binding, condition)

allPos(L) :- forall(member(E, L), E>0).

?- allPos([1,2,3,4]).

Yes

32 ?- allPos([1,2,-3,4]).

No

?- forall(member(E, [1,2,3,4]), E>0).

true.

?- forall(member(E, [1,2,-3]), E>0).

false.
```

prog(A,B,X) = Program that for input parameters A and B yields the result X.

> prog(a,b,X).
X = x1;
X = x2;
X = x3;
X = x4;
No

Multiple solutions

FIND ALL SOLUTIONS AND STORE THEM IN A LIST

If we want to process all values of X, then we can use the following method:

1. Use findall(X, prog(a,b,X), Xlist) to create the list of all solutions Xlist = [x1, x2, x3, x4].

2. Use process(Xlist) to process the list of all solutions

Example:

```
findalltest(A,B) :- findall(X, prog(A,B,X), Xlist), write(Xlist), nl, process(Xlist).
> findalltest(a,b).
  [x1,x2,x3,x4]      (All results of program process)
```

Findall Search: make a list of all solutions

```
findall(X, prog(a,b,X), Xlist).
```

Make Xlist that includes all X solutions from prog(a,b,X)

```
nel(L, E) :- member(E,L), E < 0.

negList(L, NegList) :- findall(E, nel(L,E), NegList).

?- nel([1,-2,3,-4,5,-6],NegEl).
NegEl = -2 ;
NegEl = -4 ;
NegEl = -6 ;
No
?- negList([1,-2,3,-4,5,-6], L).
L = [-2, -4, -6] ;
No
```

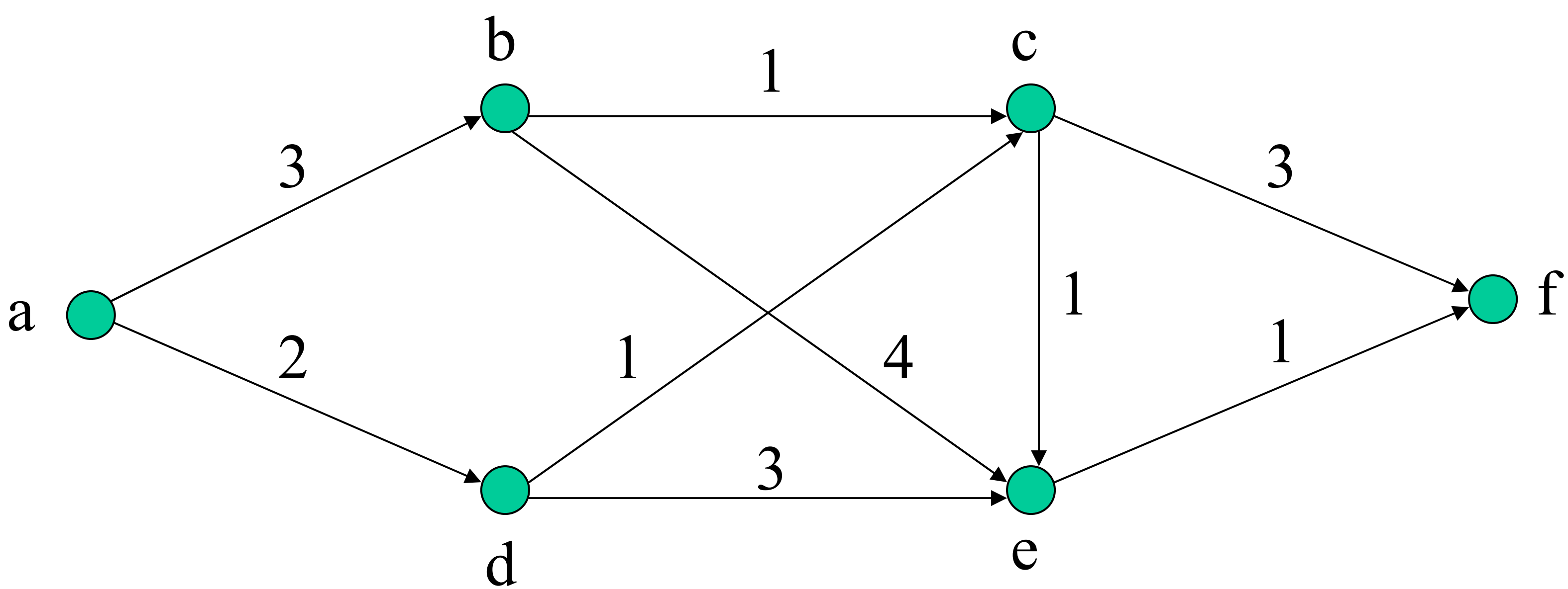
Setof Search: make a set of all different solutions

```
setof(X, prog(a,b,X), Xset).
```

Make Xset that includes all different X solutions from prog(a,b,X)

```
nel(L, E) :- member(E,L), E < 0.

allNeg(L, NegList) :- findall(E, nel(L,E), NegList).

setNeg(L, NegSet)  :- setof(E, nel(L,E), NegSet).

?- allNeg([1,-1,2,-1,3,-2,4,-1], L).

L = [-1, -1, -2, -1] ;

No

?- setNeg([1,-1,2,-1,3,-2,4,-1], L).

L = [-2, -1] ;

No
```

Graph analysis



Write a program that finds the shortest path between two points

```
% FACTS
link( a, b, 3 ).
link( a, d, 2 ).
link( b, c, 1 ).
link( b, e, 4 ).
link( c, e, 1 ).
link( c, f, 3 ).
link( d, c, 1 ).
link( d, e, 3 ).
link( e, f, 1 ).

% RULES
path(X,Y,LXY) :- link(X,Y,LXY).
path(X,Y,LXY) :- link(X,A,LXA),
                 path(A,Y,LAY),
                 LXY is LXA+LAY.
```

```
?- path(a,e,Length).
Length = 7 ;
Length = 5 ;
Length = 5 ;
Length = 4 ;
No

?- path(a,f,Length).
Length = 7 ;
Length = 6 ;
Length = 8 ;
Length = 6 ;
Length = 5 ;
Length = 6 ;
No
```

Find all paths between X and Y

```
pathList(X,Y,List) :-
      findall(LXY, path(X,Y,LXY), List).
?- pathList(a, e, L).
L = [7, 5, 5, 4] ;
No
?- pathList(a, f, L).
L = [7, 6, 8, 6, 5, 6] ;
No
```

Find the shortest path between X and Y

```
min([X], X).  % Minimum component of a list
min([H|T], M) :- min(T,M), M=<H, !.
min([H|_], H).

minpath(X,Y) :- pathList(X,Y,L), min(L,M), length(L,N),
                write('Number of different paths from '),
                write(X), write(' to '), write(Y),
                write(' is '), write(N), nl,
                write('Minimum length from '), write(X),
                write(' to '), write(Y), write(' is '),
                write(M), nl.
```

```
?- minpath(a,f).
Number of different paths from a to f is 6
Minimum length from a to f is 5

Yes

?- minpath(a,e).
Number of different paths from a to e is 4
Minimum length from a to e is 4

Yes

?- minpath(d,f).
Number of different paths from d to f is 3
Minimum length from d to f is 3

Yes
```

Complete graph analysis program

```
%  FACTS
link( a, b, 3 ).
link( a, d, 2 ).
link( b, c, 1 ).
link( b, e, 4 ).
link( c, e, 1 ).
link( c, f, 3 ).
link( d, c, 1 ).
link( d, e, 3 ).
link( e, f, 1 ).

% RULES
path(X,Y,LXY) :- link(X,Y,LXY).
path(X,Y,LXY) :- link(X,A,LXA),
                 path(A,Y,LAY),
                 LXY is LXA+LAY.

pathList(X,Y,List) :- findall(LXY, path(X,Y,LXY), List).

min([X], X).  % Minimum component of a list
min([H|T], M) :- min(T,M), M=<H, !.
min([H|_], H).

minpath(X,Y) :- pathList(X,Y,L), min(L,M), length(L,N),
                write('Number of different paths from '),
                write(X), write(' to '), write(Y),
                write(' is '), write(N), nl,
                write('Minimum length from '), write(X),
                write(' to '), write(Y), write(' is '),
                write(M), nl.
```

Event Search Program

Facts have the following format:

event(<year>, [<description>]).

Our program contains the following facts:

```
event(1941, [second, world, war]).
event(1914, [first, world, war]).
event(1889, [birth, of, charlie, chaplin]).
event(1977, [death, of, charlie, chaplin]).
event(1906, [san, francisco, earthquake]).
event(1756, [birth, of, wolfgang, amadeus, mozart]).
event(1791, [death, of, wolfgang, amadeus, mozart]).
```

Write the following rules:

- show(List): displays list elements
- alive(X): displays the life span of X
- life(X): displays how many years X lived
- search(Event,Year): search event/year
- find(X): find event or year defined by X

```
show([]).
show([H|T]) :- write(H), tab(1), show(T).

showln([])  :- nl.
showln([H|T]) :- write(H), tab(1), showln(T).
```

```
?- show([wolfgang, amadeus, mozart]).
wolfgang amadeus mozart

Yes
?- showln([wolfgang, amadeus, mozart]).
wolfgang amadeus mozart

Yes
```

(In this case show and showln generate the same output)

```
alive(X) :- event(Y1, [birth,_|Name]), member(X,Name),
            event(Y2, [death,_|Name]),
            show(Name), write(': '), write(Y1-Y2), nl.
```

```
?- alive(amadeus).
wolfgang amadeus mozart : 1756-1791
Yes
?- alive(war).
No
?- alive(Person).
charlie chaplin : 1889-1977
Person = charlie ;
charlie chaplin : 1889-1977
Person = chaplin ;
wolfgang amadeus mozart : 1756-1791
Person = wolfgang ;
wolfgang amadeus mozart : 1756-1791
Person = amadeus ;
wolfgang amadeus mozart : 1756-1791
Person = mozart ;
No
```

(multiple results are generated by the member function)

```
life(X) :- event(Y1, [birth,_|Name]), last(Name,X),
           event(Y2, [death,_|Name]),
           show(Name), write('lived '),
           Y is Y2-Y1, write(Y), write(' years'), nl.
```

```
?- life(amadeus).
No
?- life(mozart).
wolfgang amadeus mozart lived 35 years
Yes
?- life(Person).
charlie chaplin lived 88 years
Person = chaplin ;
wolfgang amadeus mozart lived 35 years
Person = mozart ;
No
```

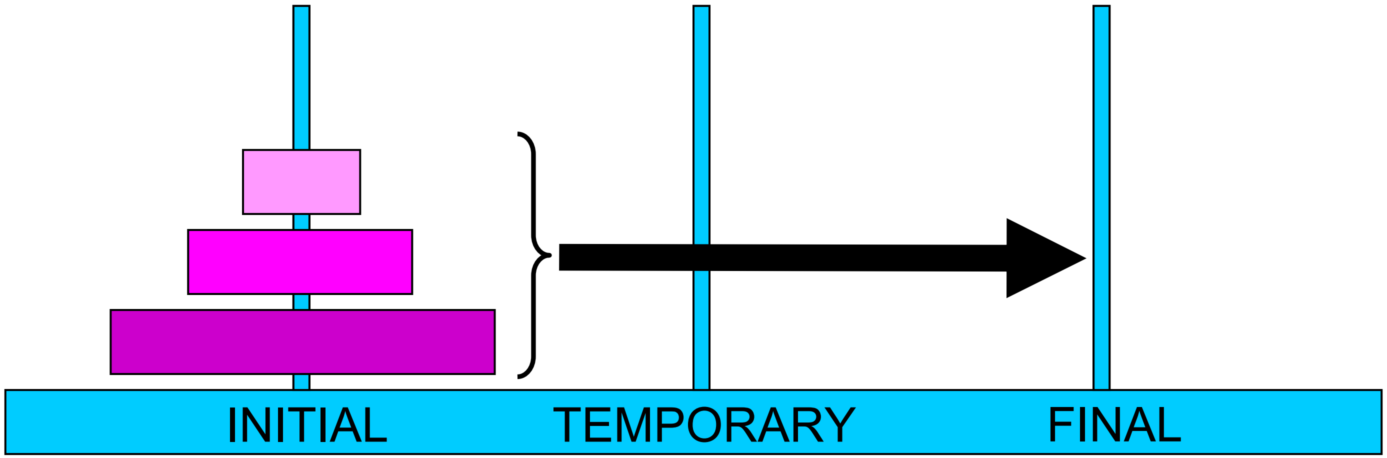
```
search(Event,Year) :- event(Year,L),
                      member(Event, L),
    nl, write(Year), write(': '), showln(L).
```

```
?- search(war,Y).
1941: second world war
Y = 1941 ;
1914: first world war
Y = 1914 ;
No
?- search(Event, 1914).
1914: first world war
Event = first
Yes
?- search(Event, Year).
1941: second world war                (vertical spaces removed)
Event = second
Year = 1941
Yes
```

```
find(Y) :- event(Y, L), write(Y), write(': '), showln(L).

find(X) :- event(Y,L),member(X,L),write(Y),write(': '),showln(L).
```

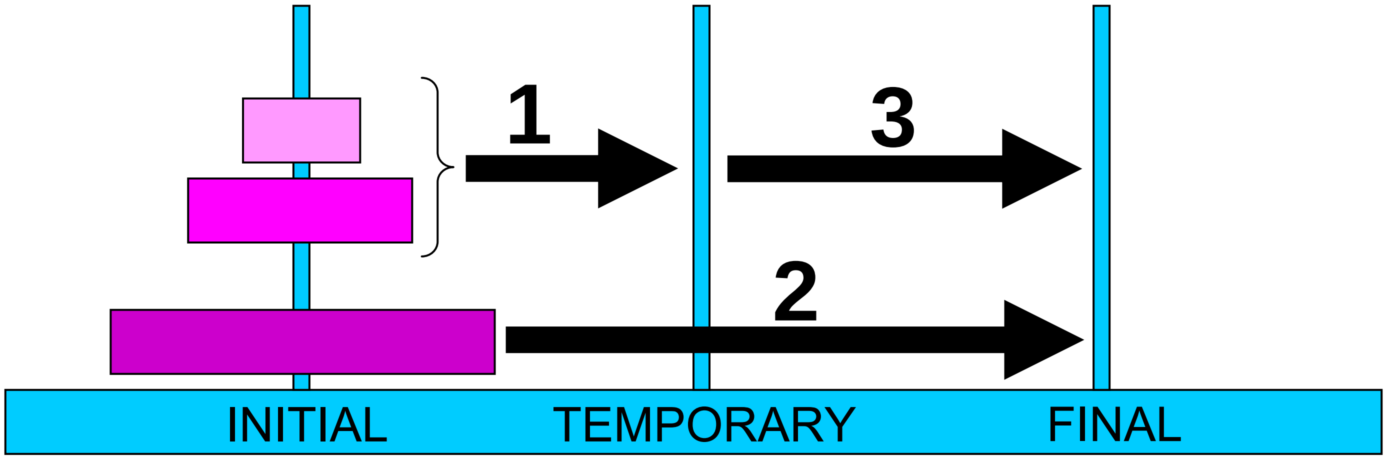
```
?- find(1941).
1941: second world war
Yes
?- find(war).
1941: second world war
Yes
?- find(amadeus).
1756: birth of wolfgang amadeus mozart
Yes
?- find(X).
1941: second world war
X = 1941 ;
1914: first world war
X = 1914 ;
1889: birth of charlie chaplin
X = 1889 ;
1977: death of charlie chaplin
X = 1977 ;
1906: san francisco earthquake
X = 1906
```

Towers of Hanoi: combining procedural and nonprocedural



- Transfer n disks from the initial position to the final position.
- Moves include only one disk
- On each peg disks must always be sorted

Solution



1. Move n-1 disks from the INITIAL position to the TEMPORARY position
2. Move the bottom disk from INITIAL to FINAL position
3. Move n-1 disks from the TEMPORARY position to the FINAL position

Towers of Hanoi - PROLOG version

```
%                |        |        |
%               =|=       |        |
%              ==|==      |        |
%             ===|===     |        |
% ---------------+--------+--------+---------------
%              left    middle   right

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          T O W E R S   O F   H A N O I
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% N      = number of disks to be moved from the
%          initial position to the final position
% left   = initial position of disks
% right  = final position of disks
% middle = auxiliary position of disks
%
% Method:  1. Move top N-1 disks from left to middle
%          2. Move the last disk from left to right
%          3. Move N-1 disks from middle to right
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Disks must always be sorted (larger disk must never come above a smaller disk)

```
%Towers of Hanoi

message([]) :- nl.
message([H|T]) :- write(H), tab(1), message(T).

% If there are no disks for moving terminate the program

move(0, _, _, _)  :-  !.

% Move disks from initial position to final position using
% the third position as auxiliary:

move(N, Initial, Final, Auxiliary) :-
        N1 is N - 1 ,
        move(N1, Initial, Auxiliary, Final),
        message([move, disk, from, Initial, to, Final]),
        move(N1, Auxiliary, Final, Initial).

hanoi(N) :- move(N, left, right, middle).
```

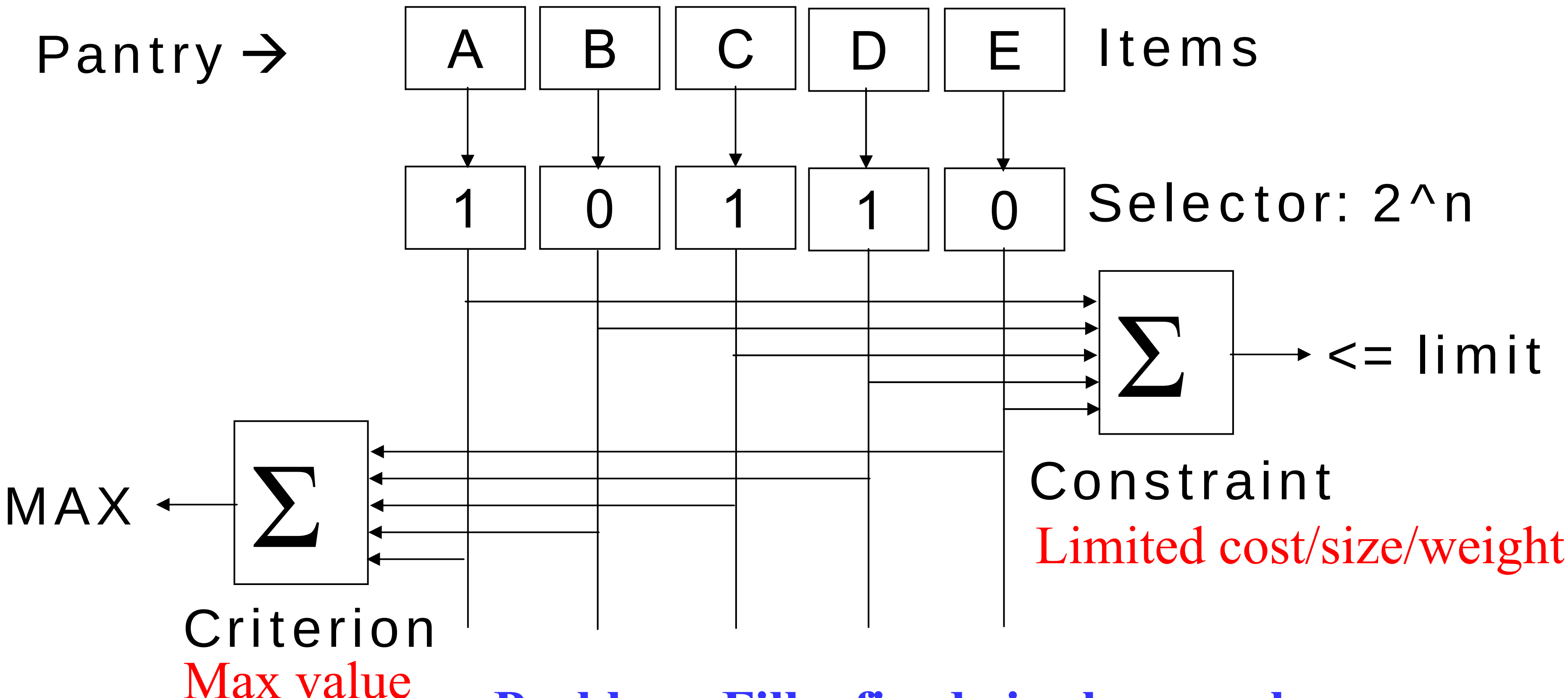
```
?- hanoi(0).

Yes
?- hanoi(1).
move disk from left to right

Yes
?- hanoi(2).
move disk from left to middle
move disk from left to right
move disk from middle to right

Yes
?- hanoi(3).
move disk from left to right
move disk from left to middle
move disk from right to middle
move disk from left to right
move disk from middle to left
move disk from middle to right
move disk from left to right

Yes
```

```
?- hanoi(4).
move disk from left to middle
move disk from left to right
move disk from middle to right
move disk from left to middle
move disk from right to left
move disk from right to middle
move disk from left to middle
move disk from left to right
move disk from middle to right
move disk from middle to left
move disk from right to left
move disk from middle to right
move disk from left to middle
move disk from left to right
move disk from middle to right

Yes
?-
```

Number of moves = 2^N - 1

Knapsack optimization problem

[<item>, <cost/size/weight>, <value>]



Problem: Fill a fixed-size knapsack with the most valuable items

BASIC STEPS:

- User manual
- Item definitions: [<name>,<cost/weight/size>,<value>]
- Display items in the pantry, one item per line
- Show the pantry and read the cost/weight/size limit
- Compute the total cost of selected knapsack
- Compute the total value of selected knapsack
- Generate subset of a given set: sset(Subset, Set). Use it to get knapsack as a subset of a pantry: sset(Knapsack,Pantry)
- Select a legal subset that is within the given cost limit
- Make a list of legal knapsacks
- Find the optimum knapsack (one that has the maximum value)
- Compute and display the optimum solution

```
/************************************************************\
|  KNAPSACK OPTIMUM SOLUTION            Jozo Dujmovic, 2010   |
|  Maximize knapsack value for a constrained cost/weight/size |
|  Exhaustive search O(2^n)                                   |
\************************************************************/

% User manual

man :- write('\nKNAPSACK OPTIMIZATION PROBLEM:'), nl,
       write('1. Items in the pantry are defined as the pantry list inside'),nl,
       write('   the program and displayed at the beginning of the program.'),nl,
       write('2. Cost/weight/size LIMIT is defined as a prompted user input.'),nl,
       write('3. The input LIMIT value must be followed by SPACE and DOT.'),nl,
       write('4. To activate this program use the command "go".'), nl,nl .

% Item definitions: [ <name>,<cost/weight/size>,<benefit> ]

pantry([ [alpha,4,9200], ['beta ',2,4500], [gamma,1,6700], [delta,3,6900] ]).
```

```
% Display items in the pantry, one item per line

showitems( []  ) :- nl.
showitems( [X] ) :- write(X), nl.
showitems([H|T]) :- write(H), nl, showitems(T).

% Show the pantry and read the cost/weight/size limit

getinputs(Limit) :- nl, write('KNAPSACK OPTIMIZATION PROBLEM'), nl,nl,
       write('---------------------------------'), nl,
       write('Input data: <name>,<cost>,<value>'), nl,
       write('---------------------------------'), nl,
       pantry(P), showitems(P),
       write('---------------------------------'), nl,nl,
       write('Enter the cost/weight/size limit : '), read(Limit), nl, !,
       write('OPTIMUM KNAPSACK CONTENTS'), nl.
```

```
% Compute the total cost of selected knapsack

cost([],0).
cost([[_,C,_] | Rest], Cost) :- cost(Rest,CR), Cost is C + CR.

% Compute the total value of selected knapsack

value([],0).
value([[_,_,V] | Rest], Value):- value(Rest,VR), Value is V + VR.

% Generate subset of a given set: sset(Subset, Set). Use it to
% get knapsack as a subset of a pantry: sset(Knapsack,Pantry)

sset([],[]).
sset([Head | SubsetTail], [Head | Tail]) :- sset(SubsetTail, Tail).
sset(SS, [_ | Tail]) :- sset(SS, Tail).

% Select a legal subset that is within the given cost limit

legalSubset(Pantry,Limit,Knapsack) :- sset(Knapsack,Pantry),
                                      cost(Knapsack,Cost), Cost =< Limit,
                                      value(Knapsack,Value), Value > 0.
```

```
% Make a list of legal knapsacks

knapsackList(LIST) :- getinputs(Limit), pantry(Pantry),
              findall(Knapsack,legalSubset(Pantry,Limit,Knapsack),LIST).

% Find the optimum knapsack (one that has the maximum value)

optimum([K],K,C,V)   :- cost(K,C), value(K,V).
optimum([K|T],K,C,V) :- optimum(T,_,_,TV), value(K,V), V>=TV, cost(K,C), !.
optimum([_|T],K,C,V) :- optimum(T,K,C,V).

% Compute and display the optimum solution

go :- knapsackList(LIST), optimum(LIST,K,C,V),
      write('Knapsack = '), write(K), nl,
      write('Cost     = '), write(C), nl,
      write('Value    = '), write(V), nl.
```

```
1 ?- man.

KNAPSACK OPTIMIZATION PROBLEM:
1. Items in the pantry are defined as the pantry list inside
   the program and displayed at the beginning of the program.
2. Cost/weight/size LIMIT is defined as a prompted user input.
3. The input LIMIT value must be followed by SPACE and DOT.
4. To activate this program use the command "go".

true.

2 ?- go.

KNAPSACK OPTIMIZATION PROBLEM

---------------------------------
Input data: <name>,<cost>,<value>
---------------------------------
[alpha, 4, 9200]
[beta , 2, 4500]
[gamma, 1, 6700]
[delta, 3, 6900]
---------------------------------

Enter the cost/weight/size limit : 4 .

OPTIMUM KNAPSACK CONTENTS
Knapsack = [[gamma, 1, 6700], [delta, 3, 6900]]
Cost     = 4
Value    = 13600
true.
```

A sequence of all solutions

- Knapsack weight has a range of possible values
- The minimum weight is the weight of the smallest item in the pantry
- The maximum weight is the sum of weights of all items in the pantry

```
% Compute the minimum cost of the pantry

clist(Clist) :- pantry(P), findall(C, member([_,C,_],P), Clist).

lmin([X],X).
lmin([H|T], H) :- lmin(T,M), H=<M, !.
lmin([_|T],M)  :- lmin(T,M).

mincost(Cmin) :- clist(Clist), lmin(Clist,Cmin).

% Display items in the pantry, one item per line

showitems( []  ) :- nl.
showitems( [X] ) :- write(X), nl.
showitems([H|T]) :- write(H), nl, showitems(T).

% Show all items in the pantry

showpantry :- nl, write('KNAPSACK OPTIMIZATION PROBLEM'), nl,nl,
         write('---------------------------------'), nl,
         write('Input data: <name>,<cost>,<value>'), nl,
         write('---------------------------------'), nl,
         pantry(P), showitems(P),
         write('---------------------------------'), nl,nl.
```

```
% Compute and display the optimum solution in a single step mode

go1 :- showpantry, mincost(Cmin), pantry(P), cost(P,Cmax),
numlist(Cmin,Cmax,CostList), !,
      member(Limit,CostList), write('\nOptimum solution for Limit = '),
write(Limit),nl,
      knapsackList(Limit,LIST), optimum(LIST,K,C,V),
      write('Knapsack   = '), write(K), nl,
      write('Cost       = '), write(C), nl,
      write('Value      = '), write(V), nl, RelValue is V/C,
      write('Value/Cost = '), write(RelValue), nl.

% Compute all costs from minimum to maximum

allcases(CostList,Cmin) :- showpantry, mincost(Cmin),
                           pantry(P), cost(P,Cmax), numlist(Cmin,Cmax,CostList).

% Show list and count elements 1,2,3...  Call: showlist(Cmin, List)

showlist(_,[]) :- !.
showlist(N,[Limit|T]) :- write('\nOptimum solution for Limit = '), write(Limit),nl,
      knapsackList(Limit,LIST), optimum(LIST,K,C,V),
      write('Knapsack   = '), write(K), nl,
      write('Cost       = '), write(C), nl,
      write('Value      = '), write(V), nl, RelValue is V/C,
      write('Value/Cost = '), write(RelValue), nl,
      N1 is N+1, showlist(N1,T).

% Compute and display all optimum solutions
```



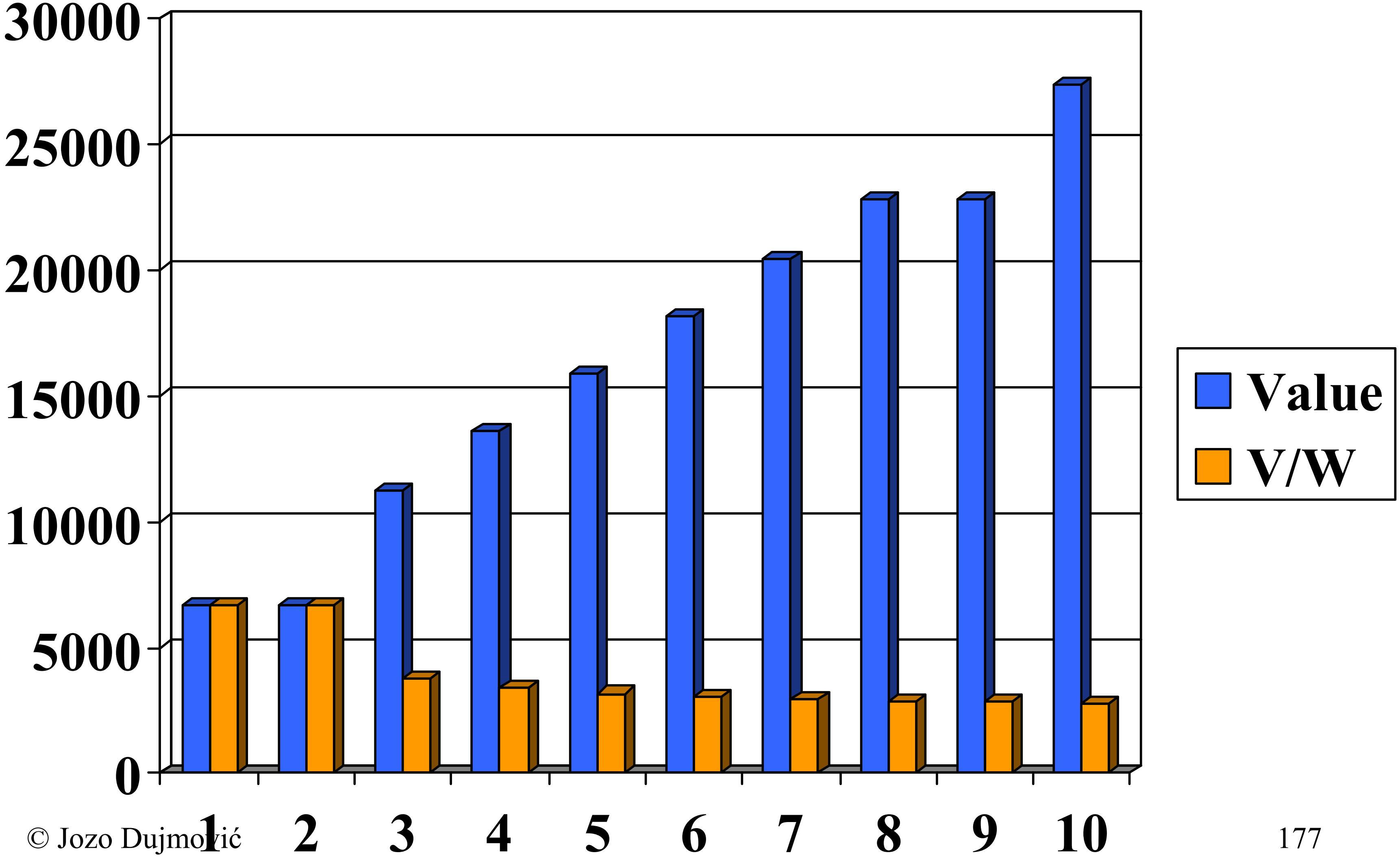
```
go :- allcases(CostList,Cmin), showlist(Cmin,CostList),!.
```

```
?- man.

KNAPSACK OPTIMIZATION PROBLEM:
1. Items in the pantry are defined as the pantry list inside
   the program and displayed at the beginning of the program.
2. Cost/weight/size LIMIT takes all possible values from a
   range of values computed using data from the pantry.
3. To get solutions in a single step mode use the command "go1".
4. To get all solutions use the command "go".

true.

?- go.

KNAPSACK OPTIMIZATION PROBLEM

---------------------------------
Input data: <name>,<cost>,<value>
---------------------------------
[alpha, 4, 9200]
[beta , 2, 4500]
[gamma, 1, 6700]
[delta, 3, 6900]
---------------------------------
```

```
Optimum solution for Limit = 1
Knapsack   = [[gamma, 1, 6700]]
Cost       = 1
Value      = 6700
Value/Cost = 6700

Optimum solution for Limit = 2
Knapsack   = [[gamma, 1, 6700]]
Cost       = 1
Value      = 6700
Value/Cost = 6700

Optimum solution for Limit = 3
Knapsack   = [[beta , 2, 4500], [gamma, 1, 6700]]
Cost       = 3
Value      = 11200
Value/Cost = 3733.33

Optimum solution for Limit = 4
Knapsack   = [[gamma, 1, 6700], [delta, 3, 6900]]
Cost       = 4
Value      = 13600
Value/Cost = 3400

Optimum solution for Limit = 5
Knapsack   = [[alpha, 4, 9200], [gamma, 1, 6700]]
Cost       = 5
Value      = 15900
Value/Cost = 3180
```

```
Optimum solution for Limit = 6
Knapsack   = [[beta , 2, 4500], [gamma, 1, 6700], [delta, 3, 6900]]
Cost       = 6
Value      = 18100
Value/Cost = 3016.67

Optimum solution for Limit = 7
Knapsack   = [[alpha, 4, 9200], [beta , 2, 4500], [gamma, 1, 6700]]
Cost       = 7
Value      = 20400
Value/Cost = 2914.29

Optimum solution for Limit = 8
Knapsack   = [[alpha, 4, 9200], [gamma, 1, 6700], [delta, 3, 6900]]
Cost       = 8
Value      = 22800
Value/Cost = 2850

Optimum solution for Limit = 9
Knapsack   = [[alpha, 4, 9200], [gamma, 1, 6700], [delta, 3, 6900]]
Cost       = 8
Value      = 22800
Value/Cost = 2850

Optimum solution for Limit = 10
Knapsack   = [[alpha, 4, 9200], [beta , 2, 4500], [gamma, 1, 6700], [delta, 3, 6900]]
Cost       = 10
Value      = 27300
Value/Cost = 2730
true.
```

Optimum value and value/weight of knapsack as functions of weight limit



Prolog Reference Card

Program manipulation

consult(filename).	Load program filename.pl (or filename.pro)
[filename].	Same as consult
[f1, f2, …].	Consult f1, then consult f2, etc.
make.	Reload all modified files (files edited after consulting)
listing.	Listing of all predicates
listing(p).	Listing of the specific predicate p
halt.	Terminate the program and exit

Note: In most Prolog implementations, build-in procedures are predefined and cannot be redefined by the user.

Constants

atom	symbol
123	integer
12.3	real number
'string'	string

Input/Output

read(X).	Read X from keyboard
write(X).	Display X on the screen
tab(N)	Display N spaces
nl	New line

Relations and Equalities

X = Y	X is equal to Y if X and Y match (for arbitrary X,Y)
X \= Y	Succeeds if X and Y do not match
X = = Y	X is literally equal to Y (if X and Y are identical)
X \= = Y	Succeeds if X is not literally equal to Y
Expr1 =:= Expr2	Succeeds if Expr1 and Expr2 evaluate to the same value
Expr1 =\= Expr2	Succeeds if Expr1 and Expr2 evaluate to different values
Expr1 > Expr2	Succeeds if Expr1 > Expr2 (expressions are evaluated)
Expr1 < Expr2	Succeeds if Expr1 < Expr2 (expressions are evaluated)
Expr1 >= Expr2	Succeeds if Expr1 >= Expr2 (expressions are evaluated)
Expr1 =< Expr2	Succeeds if Expr1 <= Expr2 (expressions are evaluated)
X is Expr	X matches the value of expression Expr (assignment)

Facts and Rules

Facts

`predicate(a, b, c).` %Relationship between a, b, and c

Rules

% Rules for the length of list

```
len([ ], 0).
len[[_ | T], N) :- len(T, K), N is 1+K.
```

% A and B are in order if A is less than or equal to B

```
order(A, B)  :-  A=<B.
```

`not(order(A, B))` succeeds if order(A,B) fails

List operations (1)

[ ]	Empty list
[a, b, c]	List with three elements
[H | T]	Separation of head (H) and tail (T)
[H | _]	List with head H and an anonymous tail
[X, Y, Z | T]	First three elements X, Y, Z, and tail (list) T
[X, Y, Z | 123]	Not a list (it is a binary tree)
is_list(L).	Succeeds if L is a list
length(L, Len).	The length of list L is Len
member(E, L).	E is an element of list L
append([1,2], [3,4], L).	L = [1,2,3,4] ; flexible append of two lists
delete([1, e, 2, e, 3], e, L).	L = [1, 2, 3]
select(e, [1, e, 2, e, 3], L).	L = [1, 2, e, 3] remove one instance of element from list
select(e, L, [1, 2, 3]).	L = [e, 1, 2, 3] insert one instance of element in list

List operations (2)

nth0(Ind, L, E).	Addressing vector L[0],L[1],…: creates E=L[Ind]
nth1(Ind, L, E).	Addressing vector L[1],L[2],…: creates E=L[Ind]
last(L, E).	Unify E with the last element of list L
permutation(L1, L2).	L2 is a permutation of L1 and vice versa
flatten([1,[2,[3,4],5],6], L).	L = [1, 2, 3, 4, 5, 6] ; makes L from sublist elements
sumlist (List, Sum).	Sum is the sum of all elements in the List
numlist(MinInt, MaxInt, List).	Make integer list List = [MinInt, MinInt+1, … , Maxint]
reverse(List, RevList).	Reverse List and create the reversed list RevList
sort([7,3,5,3], X).	X = [3,5,7] non flexible sort with elimination of duplicates
msort([7,3,5,3], X).	X = [3,3,5,7] non flexible sort of all elements
merge(L1, L2, L12).	Merge sorted L1 and L2 yielding sorted L12 (all duplicates are included)

Set operations

is_set(S).	Succeeds if S is a set (a list without duplicates)
list_to_set(L, S).	Makes set-list S by eliminating duplicates from L
union(Set1, Set2, U).	$U = Set1 \cup Set2$
intersection(Set1, Set2, I).	$I = Set1 \cap Set2$
subtract(Set, Del, Res).	Res = Set \ Del (remove all elements of set Del from Set)
subset(Subset, Set).	Succeeds if Subset $\subseteq$ Set
merge_set(S1, S2, S12).	Merge sorted sets S1 and S2 yielding sorted $S12 = S1 \cup S2$ (all duplicates in S12 are eliminated)

Control and search

findall(X, prog(a,b,X), Xlist).	Make Xlist that includes all X solutions from prog(a,b,X)
setof(X, prog(a,b,X), Xset).	Make Xset that includes all different X solutions from prog(a,b,X)
forall(member(E,List), E>0).	Check that all members of List are positive
a, b, !, c, d.	Cut: prevent backtracking from c to b (and a)
a, b, c, ! .	Single satisfaction of goals a, b, and c
fail	Always fail (and cause backtracking)
repeat	Always succeed (backtracking restart point)

Library Functions

//	Integer division
mod	Modulus
rem	Remainder of division (float_fractional_part)
abs, sign, min, max	Min and max of two variables only
random(N)	Random integer 0 <= random <N
round	Nearest integer
truncate(X)	Integer part of X
float_integer_part(X)	Same as truncate
float_fractional_part(X)	Fractional part of X
floor(X), ceiling(X)	Nearest integers <X and >X (X is an expression)
**>> , << , \/ , /\ , xor , **	Bitwise shr, shl, or, and, xor, not
sqrt, **, ^, exp, log, log10	Base**expo same as Base^expo
sin, cos, tan, asin, acos, atan	Trigonometric function
pi, e	Constants 3.141593 and 2.718282

Recognizers

var(X), nonvar(X)	Succeeds if X is (is not) a variable
integer, float, number, string	Recognition of basic data types
atom, atomic(X)	X is atom, string, integer or float

Comments

% comment	Comment to the end of line
/* comment */	Multiline comment

Database operations

assert(car(ford)).	Add car(ford) as the last fact of the car predicate
assertz(car(ford)).	Same as assert
asserta(car(ford)).	Add car(ford) as the first fact of the car predicate
retract(car(ford)).	Remove the fact car(ford)
retractall(car(_)).	Remove all car facts from the database