Declarative Programming

Programming Paradigms

- **Imperative (Non-Declarative) Programming**
  - Fortran, C, Cobol, Pascal
  - Object-Oriented Programming
    - C++, Java, C#
- **Declarative Programming**
  - Functional Programming
    - ML, Lisp, Haskel, Scheme, F#
  - Logic Programming
    - Prolog (Sicstus, SWI, GNU, YAP, Ciao)

Imperative Programming

- How to Solve, rather than what to solve
- Requires the programmer to specify an algorithm to be run
- Sequence of statements
- Makes the algorithm explicit and leaves the goal implicit

For example …

Imperative Programming (Contd.)

For example …

```
int Function (int n) {
   int t = 1;
   while (n > 0) {
      t = t * n;
      n = n – 1;
   }
   return t;
}
```

Computes the factorial of n

Declarative Programming

- What to Solve, rather than how to solve
- Requires the programmer to specify just the problem, the language compiler figures the algorithm
- Sequence of definitions (functions or predicates)
- Makes the goal explicit and leaves the algorithm implicit

For example …

Declarative Programming (Contd.)

For example …

fac(0) = 1
fac(n) = n * fac(n-1)

Computes the factorial of n

SEND + MORE = MONEY Puzzle

- Each letter represents a unique digit from 0 to 9.
- Two letters cannot represent the same digit.
- What digit each letter represents to satisfy the simple

equation below?

Solution:

	S	E	N	D
+	M	O	R	E
-----	-----	-----	-----	-----
M	O	N	E	Y

	9	5	6	7
+	1	0	8	5
-----	-----	-----	-----	-----
1	0	6	5	2

SEND + MORE = MONEY

A Declarative CLP Program:

```
solve(Digits) :-
      Digits = [S,E,N,D,M,O,R,Y],
      Digits :: [0..9],
      alldifferent(Digits),
                         1000*S + 100*E  + 10*N + D
                      + 1000*M + 100*O + 10*R + E
        #= 10000*M + 1000*O + 100*N  + 10*E + Y,
      labeling(Digits).
```

? – solve(X)

X = [9,5,6,7,1,0,8,2]

Declarative Vs Imperative Programming

Algorithm = Logic + Control

- Imperative programming needs both logic & control whereas Declarative programming needs just logic, it figures control on its own.
- Declarative programming is a higher level programming paradigm than Imperative.

Then why Declarative Programming is not so popular?

- Declarative programming is not as efficient as Imperative programming as it needs to figure out the control part of the algorithm on its own.

- Imperative programming is more closer to the popular (Von Neumann) architecture of a computer, whereas Declarative Programming is independent of the architecture, so not as optimized.

Applications of Declarative Programming

- Artificial Intelligence, Machine Learning
- Knowledge Representation, Semantic Web
- Deductive Databases (DataLog)
- Modeling and Simulation
- Verification and Validation
- Game Development
- Resource Allocation & Scheduling
- Many more …

Declarative Paradigms

- ***Functional Programming***
  - Based on λ (lambda) calculus
  - ML, Lisp, Haskell, Scheme, F#

- ***Logic Programming***
  - Based on First Order Logic
  - Prolog (Sicstus, SWI, GNU, YAP, Ciao)

  - Constraint Logic Programming (CLP)
  - Answer Set Programming (ASP)

Logic Programming

- Use of mathematical logic for computer programming
- Treats implications as goal-reduction procedures
  - B_1 and … and B_n implies H, is interpreted as
    to show/solve H, show/solve B_1 and … and B_n

- For example, it treats the implication (rule):
  - If you press the alarm signal button,
    then you alert the driver of the train of a possible emergency

  as the procedure:
  - To alert the driver of the train of a possible emergency,
    press the alarm signal button.

Logic Programming

- a.k.a. ***Rule-based Programming***

- ***Prolog (PROgramming in LOGic)***
  - the most representative LP language
  - H :- B_1, …, B_n.

- ***Algorithm = Logic + Control***
  - where “Logic” represents a logic program and “Control” represents different theorem-proving strategies, which led to various extensions of Logic Programming

Prolog

- Prolog programs define relations and allow you to query them to extract various tuples from the relations

- Relations are defined using predicates. Example:
  - *square(A,B) is true if B is A*A*
  - *pred(B,H,A) is true if A is ½ B*H*

- Prolog uses Horn clauses for explicit definition (facts) and for rules

Directionality

- Parameters are not directional (in, out)
  - Prolog programs can be run “in reverse”

- (2,4), (3,9),(4,16), (5,25),(6,36),(7,49), ... “square”
  - can ask square(X,9)

    “what number, when squared, gives 9”
  - can ask square(4,X)

    “what number is the square of 4”
  - can ask square(4,16)

    “is 16 the square of 4”

Prolog Syntax

- Variables: start with uppercase character (or “_”), may include “_” and digits:

  Examples: X, Ys, A_num, _, _x, _22

- Constants: lowercase first character, may include “_” and digits. Also, numbers and some special characters. Any quoted string.

  Examples: a, dog, a_big_cat, 23, 'Hungry man', []

- Structures: a functor (like a constant name) followed by a fixed number of arguments between parentheses:

  Example: date(monday, Month, 1994)

- Arguments can in turn be variables, constants and structures.

- *Arity:* is the number of arguments of a structure. Functors are represented as *name/arity* . A constant can be seen as a structure with arity zero.

- Variables, constants, and structures as a whole are called *terms* (they are the terms of a “first–order language”): the *data structures* of a logic program.

Database Programming

- Database: A collection of Prolog facts
- Prolog draws knowledge from these facts
- Programmer is responsible for the accuracy of the facts
- Questions are answered based on these facts

Database Programming - Example

Facts in the database:

```
likes(john,apple).
likes(mary,apple).
likes(john,fish).
likes(joe,mary).
```

Questions:

```
?-likes(joe, money).
no
?-likes(joe,mary).
yes
?-likes(john,X).
X = apple  ;
X = fish ;
no.
```

Deductive Databases

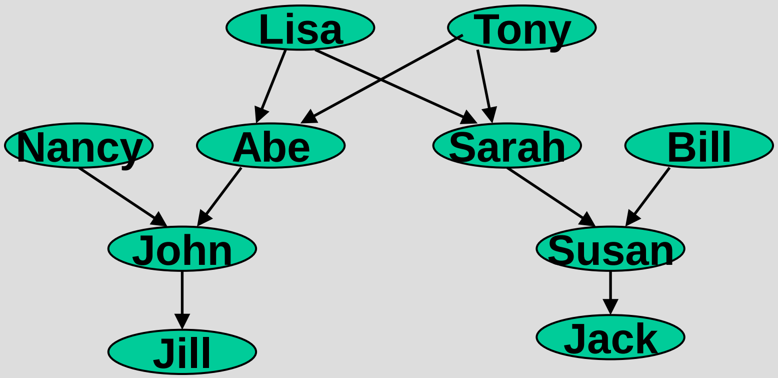
- "Deductive databases" uses these ideas to develop *logic-based databases*.

- They have syntactic restrictions (i.e., a subset of definite programs) are used

  (e.g. "Datalog" – no functors, no existential variables).

Database programming - Example

mother(lisa, abe).
mother(lisa, sarah).
mother(nancy, john).
mother(sarah, susan).
mother(susan, jack).
father(tony, abe).
father(tony, sarah).
father(abe, john).
father(bill, susan).
father(john, jill).

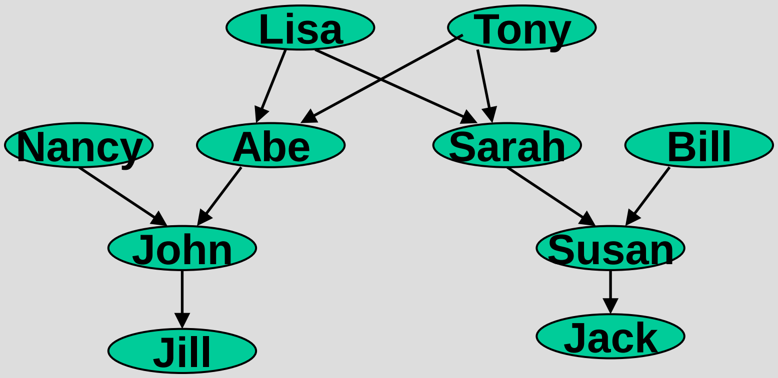


% pl (SWI Prolog)
% sicstus (Sicstus Prolog)
?- consult(‘family.pl’).

?- mother(lisa, abe).   - - > query
yes
?- mother(lisa, X).   - - > query
X = abe;
X= sarah;

Database programming - Example

mother(lisa, abe).
mother(lisa, sarah).
mother(nancy, john).
mother(sarah, susan).
mother(susan, jack).
father(tony, abe).
father(tony, sarah).
father(abe, john).
father(bill, susan).
father(john, jill).



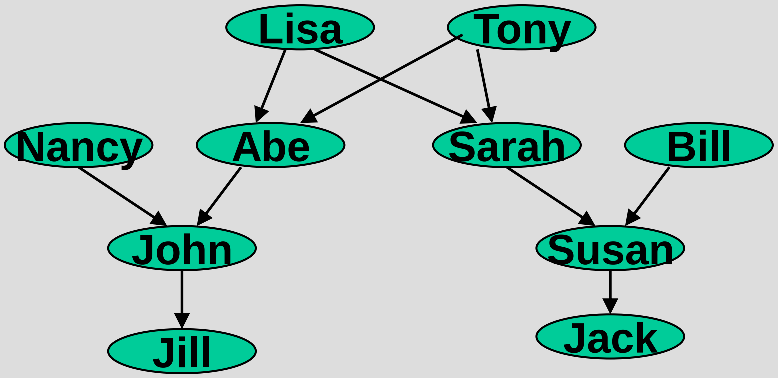
?- mother(lisa, john). - - > query
no

?- mother(X, sarah). - - >
X = lisa;

?- mother(X, Y). - - > query
5 answers

parent rule

mother(lisa, abe).
mother(lisa, sarah).
mother(nancy, john).
mother(sarah, susan).
mother(susan, jack).
father(tony, abe).
father(tony, sarah).
father(abe, john).
father(bill, susan).
father(john, jill).

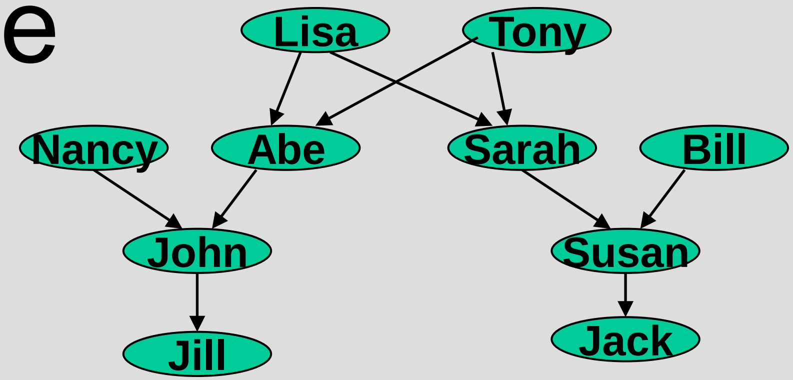


parent(X,Y) is true if X is parent of Y

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

?- parent(X, sarah).
X = lisa
X = tony

grandparent rule



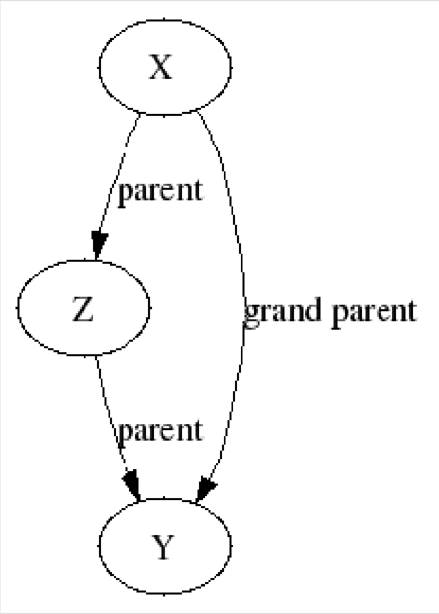
X is the grand parent of Y if:

X is the parent of Z and

Z is the parent of Y.

grandparent(X,Y):-

parent(X,Z), parent(Z,Y).



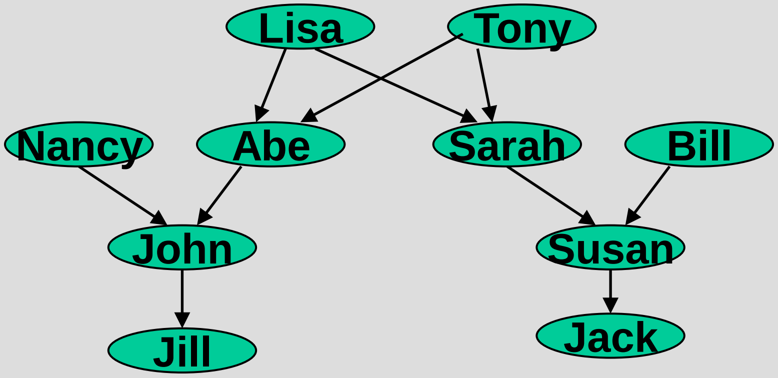
Who is the grandparent of john?

?- grandparent(X,john).

X = lisa;

X = tony

greatgrandparent rule

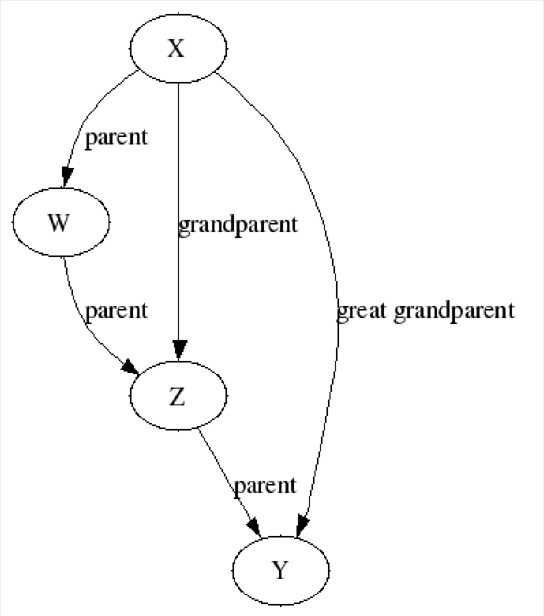


Define greatgrandparent using existing rules:

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

Who is the greatgrandparent of jill?

```
?- greatgrandparent(X,jill).
X = lisa;
X = tony
```



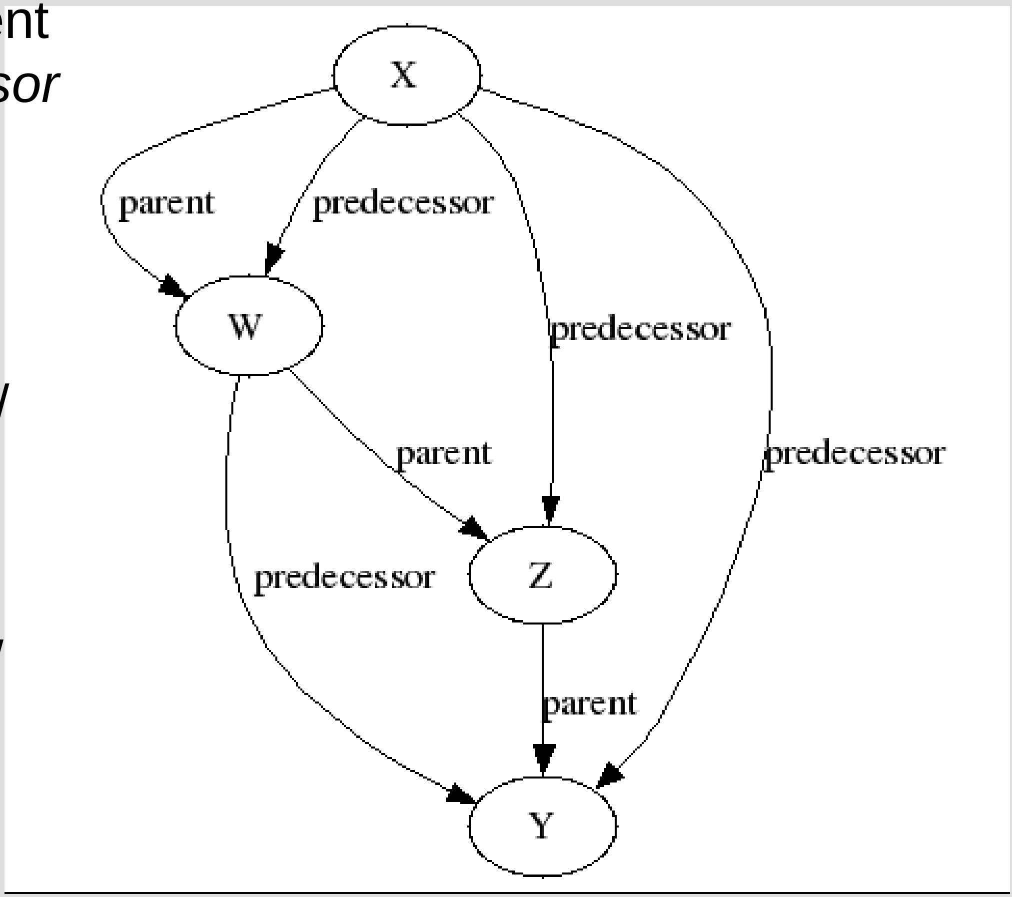
predecessor rule

Grandparent and great grandparent are specializations of a *predecessor* relation.

Definition of predecessor rule:

```
/* rule 1: the terminate condition */
predecessor(X,Z) :-
              parent(X,Z).

/* rule 2: the continue  condition */
predecessor(X,Z) :-
            parent(X,Y),
            predecessor(Y,Z).
```



Lists

- Common data structure in nonnumeric programming.
- Ordered sequence of elements that can have any length.
- Ordered: The order of elements in the sequence matters.
- Elements of a list are terms:
  - Constants
  - Variables
  - Structures
  - Lists.
- Can represent practically any kind of structure used in symbolic computation.

List Manipulation

Splitting a list L into head and tail:

- Head of L — the first element of L.
- Tail of L — the list that consists of all elements of L except the first.

Special notation for splitting lists into head and tail:

- [X|Y], where X is the head (term) and Y is the tail (list)
- [a|[b,c,d]] is the list [a,b,c,d]

Lists – ‘islist’ predicate

- Check if a given input is a list:
  - Empty list is a list
  - Non-empty list, check the first element and then recursively check if the remaining tail is a list

```
islist([ ]).
islist([Head|Tail]) :- islist(Tail).
```

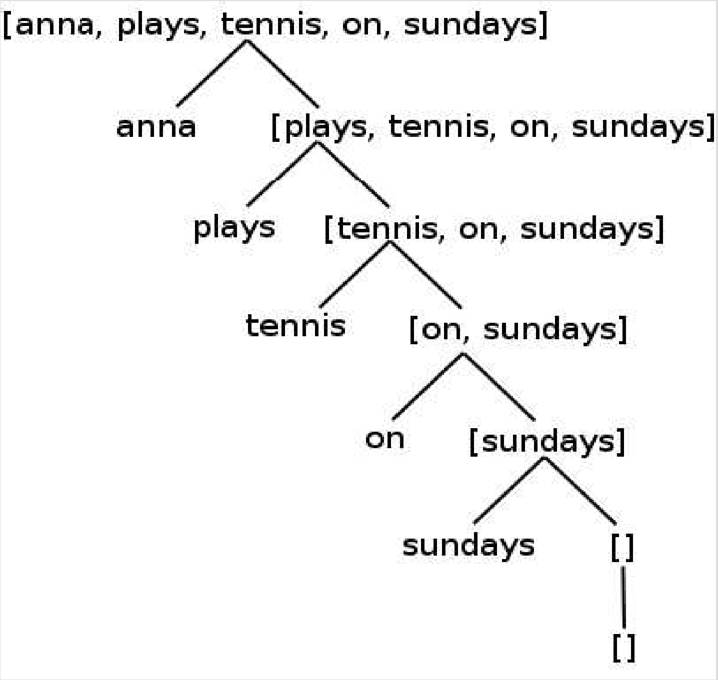
- Examples:

```
?- islist([1,3]).
yes

?- islist(f(a)).
no
```

Lists represented as a tree

- Tree representation of a list



Lists – ‘member’ predicate

member(X,Y) is true when X is a member of the list Y.

- X is a member of the list if X is the same as the head of the list Y
- X is a member of the list if X is a member of the tail of the list Y

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

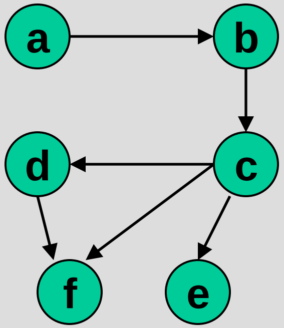
Lists – ‘sorted’ predicate

Define sorted(X)
– checks if X is a sorted list (ascending order)

- An empty list is sorted
- A list with a single element is sorted
- A compound list is sorted if the first 2 elements are in order and the remaining list (after the first element) is sorted

```
sorted([ ]).
sorted([X]).
sorted([A, B | T]) :- A =< B, sorted ([B|T]).
```

Recursion – ‘connected’ predicate



edge(a, b). edge(b, c).
edge(c, d). edge(c, e).
edge(c, f). edge(d, f).

Define connected (X, Y)

- is true if node X is connected to node Y
- there is a sequence of edges starting at node X, and finishing at node Y, which together define a path from X to Y.

```
connected(X, Y) :- edge(X,Y).
connected(X, Y) :- edge(X, Z), connected(Z, Y).
```

- **base case** of the recursion: the simplest way in which there is a path between two nodes is if they are directly connected to each other by an edge.
- **recursive case**: expresses that for two nodes X and Y to be connected, is if there is some node Z, to which X is connected, which is in turn connected to Y.

Recursion - Termination problems

- Avoid circular definitions. The following program will loop on any goal involving parent or child:

```
parent(X,Y) :- child(Y,X).
child(X,Y) :- parent(Y,X).
```

- Use left recursion carefully. The following program will loop on for the query ?- person(X)

```
person(X) :- person(Y), mother(X,Y).
person(adam).
```

Recursion - Termination problems

- Order of the Rules matter.
- General heuristics: Put facts before rules whenever possible.
- Sometimes putting rules in a certain order works fine for goals of one form but not if goals of another form are generated:

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

- works for goals like islist([1,2,3]), islist ([ ]), islist(f(1,2)) but loops for islist(X).
- What will happen if you change the order of islist clauses?

Recursion & Lists – length predicate

Predicate listlen(L,N) - succeeds if the length of list L is N.

- (Boundary condition) The empty list has length 0
- (Recursive case) The length of a nonempty list is obtained by adding one to the length of the tail of the list.

Program:

```
listlen([ ],0).
listlen([H|T],N):- listlen(T,N1), N is N1 + 1.
```

Recursion – ‘append’ predicate

Predicate append(L1, L2, L3)

- is true if L3 is the result of appending L2 to L1

– if L1 is the empty list, then L3 is L2, or
– if L1 is a nonempty list, then the head of L3 is the head of L1 and the tail of L3 is L2 appended to the tail of L1.

Program:

```
append([ ],L,L).
append([X|T1], L2, [X|T3]) :- append(T1, L2, T3).
```

Recursion – ‘append’ predicate

```
?- append([a,b,c], [2,1], [a,b,c,2,1]).
Yes

?- append([a,b,c], [2,1], X).
X = [a,b,c,2,1]

?- append(X, [2,1], [a,b,c,2,1]).          %prefix
X = [a,b,c]

?- append([a,b,c], X, [a,b,c,2,1]).        %suffix
X = [2,1]

?- append(X,Y,[a,b,c,2,1]).                %generating
X = [ ], Y = [a,b,c,2,1];
X = [a], Y = [b,c,2,1];
X = [a,b], Y = [c,2,1];
X = [a,b,c], Y = [2,1];
X = [a,b,c,2], Y = [1];
X = [a,b,c,2,1], Y = [];
no
```

List – ‘reverse’ predicate

- Predicate reverse(L, R)

  - succeeds if R is the reverse of list L

- Program:

```
reverse([ ], [ ]).
reverse([H|T], R) :- reverse(T, R1), append(R1, [H], R).
```

List – ‘delete’ predicate

- Predicate delete (X, L1, L2) (deletes a given element from the list)
  - succeeds when L2 is the list obtained by deleting element X from list L1

  - (termination condition) If the list L1 is empty, then resultant list L2 is also empty
  - If X is the variable we want to delete, and it is the head of the list L1, then recursively delete X from the tail of the list L1
  - If X is the variable that we want to delete and it is different from the head ‘H’ of L1, then ‘H’ becomes the head of the second list L2, and recursively delete X from tail of L1

- Program:

```
del(_, [ ], [ ]).
del(X, [X|T], L) :- del(X,T,L).
del(X, [H|T1], [H|T2]) :- X \= H, del(X,T1,T2).
```

List – ‘prefix’ and ‘suffix’ predicate

- Predicate prefix(P, L)

  - succeeds when P is the prefix of the list L
- Program:

```
prefix([ ], _).
prefix([P|Pt], [P|T]) :- prefix(Pt, T).
```

- Predicate suffix(S, L)

  - succeeds when S is the suffix of the list L
- Program:

```
suffix(S, S).
suffix(S, [H|T]) :- suffix(S,T).
```

‘select’ predicate

- select(X, L, NewList) – NewList is obtained by removing one occurrence of X from L. If X is not found, it fails.

```
select(X, [X|T], T).
select(X, [Y|T], [Y|R]) :- select(X, T, R).
```

- delete(X, L, NewList) – NewList is obtained by removing all occurrence of X from L. If X is not found NewList is same as L

```
del(_, [ ], [ ]).
del(X, [X|T], L) :- del(X,T,L).
del(X, [H|T1], [H|T2]) :- X \= H, del(X,T1,T2).
```

‘permutation’ predicate

- permutation(X, Xp) - Xp is the permutation of list X

```
permutation([ ],[ ]).
permutation(L, [H|T]) :- select(H, L, R),
                         permutation(R, T).
```

- Generates all possible permutations of list X

Permutation Sort

- permsort(X, Y) – Y is the ordered permutation of X

```
permsort(X, Y) :- permutation(X, Y),
                  ordered(Y).

ordered([ ]).
ordered([X]).
ordered([A, B | T]) :- A =< B, ordered([B|T]).
```

- Naïve sorting program, uses generate-and-test paradigm

Insertion Sort

- insertsort(X, Y) – Y is the sorted permutation of X
  (sorted using insertion sort)

- The first element is removed from the list, and remaining list is recursively sorted. Then the first element is inserted preserving the sorted order of the list

```
insertsort([ ], [ ]).
insertsort([H|T], Y) :- insertsort(T, Z), insert(H, Z, Y).

insert(X, [ ], [X]).
insert(X, [Y|T], [Y|Z]) :- X > Y, insert(X, T, Z).
insert(X, [Y|T], [X,Y|T]) :- X =< Y.
```

Quicksort

- qsort(L, R) – R is the sorted permutation of L
  (using quick sort)

- List is split into two by choosing a pivot element
  - one list containing elements smaller than the chosen pivot
  - other list containing elements larger than the chosen pivot

- Then the split lists are recursively sort and their results are appended

```
qsort([ ], [ ]).
qsort([X|L], R) :- partition(L, X, Littles, Bigs),
                   qsort(Littles, Ls),
                   qsort(Bigs, Bs),
                   append(Ls, [X|Bs], R).

partition([ ], _X, [ ], [ ]).
partition([X|Xs], Y, [X|Ls], Bs) :- X =< Y, partition(Xs, Y, Ls, Bs).
partition([X|Xs], Y, Ls, [X|Bs]) :- X > Y, partition(Xs, Y, Ls, Bs).
```

SEND + MORE = MONEY Puzzle

- Each letter represents a unique digit from 0 to 9.
- Two letters cannot represent the same digit.
- What digit each letter represents to satisfy the simple equation below?

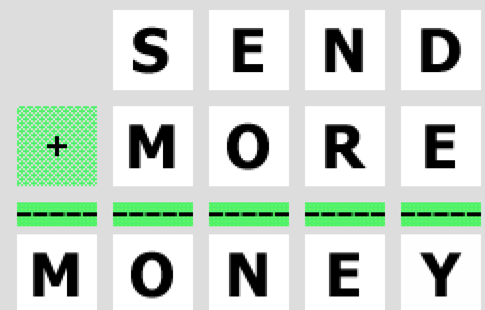
Solution:

	S	E	N	D
+	M	O	R	E
-----	-----	-----	-----	-----
M	O	N	E	Y

	9	5	6	7
+	1	0	8	5
-----	-----	-----	-----	-----
1	0	6	5	2

SEND + MORE = MONEY Puzzle

```
solve([S,E,N,D,M,O,R,Y]) :-  M is 1,
        select(D, [0,2,3,4,5,6,7,8,9], R1),
        select(E, R1, R2),
        Y is (D+E) mod 10,
        C1 is (D+E) // 10,
        select(Y, R2, R3),
        select(N, R3, R4),
        select(R, R4, R5),
        E is (N+R+C1) mod 10,
        C2 is (N+R+C1) // 10,
        select(O, R5, R6),
        N is (E+O+C2) mod 10,
        C3 is (E+O+C2) // 10,
        select(S, R6, R7),
        O is (S+M+C3) mod 10,
        M is (S+M+C3) // 10.
```



SEND + MORE = MONEY

A Declarative CLP Program:

```
solve(Digits) :-
      Digits = [S,E,N,D,M,O,R,Y],
      Digits :: [0..9],
      alldifferent(Digits),
                            1000*S + 100*E  + 10*N + D
                         + 1000*M + 100*O + 10*R + E
         #= 10000*M + 1000*O + 100*N  + 10*E + Y,
      labeling(Digits).
```

? – solve(X)

X = [9,5,6,7,1,0,8,2]