The Logic Programming Paradigm

Logic programming is based on the notion that a program implements a relation rather than a mapping. It is potentially higher-level than imperative or functional programming.

Conventional logic programming is based on the Horn clause subset of logic. The representative language is Prolog (**Pro**gramming in **Log**ic).

An Illustrative Example

Consider the following logic program defining some of the family relations.

```
brother(X,Y) :- male(X), father(Z,X), father(Z,Y).
sister(X,Y) :- female(X), father(Z,X), father(Z,Y).

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

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

In logic programming, the set of ***clauses*** defining parent/2 is called the ***procedure*** for parent/2. So are those for brother/2, *etc.*

```
brother(X,Y) :- male(X), father(Z,X), father(Z,Y).
sister(X,Y) :- female(X), father(Z,X), father(Z,Y).

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

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

Clauses without **bodies** are also called ***facts***.

```
father(john, mary).
father(john, tom).
mother(mary, tony).
male(john).
male(tom).
female(mary).
...
```

```
father(john, mary).
father(john, tom).
mother(ann, tom).
male(john).
male(tom).
female(mary).
...
```

A program is executed when the system tries to answer a ***query***.

```
?- sister(mary, tom).
yes.

?-parent(X, tom).
X = john;
X = ann
```

Logical Foundation of Logic Programming

if and end of clause

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

clause head clause body

<u>*Meaning:*</u>
X is an ancestor of Y **if** X is a parent of Z **and** Z is a parent of Y.

► *What are the possible values for X, Y, Z (and other variables?*

Logic Programs Talk about Terms

Variables in a logic program take values from the set of terms.

```
?- sister(mary, tom).
yes.

?-parent(X, tom).
X = john;
X = ann
```

john,
mary,
tom,
ann, …

What are terms?

A ***term*** is a

- a constant, or
- a variable, or
- a structure of the form $f(t_1, t_2, \ldots, t_n)$ where f is a functor (like a tag) and $t_1, t_2, \ldots, t_n$ are terms.

Examples:

john, mary, tom, ann, …	(constants)
X, Y, Z, …	(variables)
f(W, a), p(2, q(3, x, addr)), …	(structures)

A term that contains no variable is a **ground term**. The set of all ground terms is the domain of discourse of a logic program.

Horn Clauses

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

A clause is a Horn clause if there is only one ***atom*** on the left-hand-side and there are zero or more atoms on the right-hand-side.

```
brother(X,Y) :- male(X), father(Z,X), father(Z,Y).
sister(X,Y) :- female(X), father(Z,X), father(Z,Y).
father(john, mary).
father(john, tom).
```

Example of general clauses (not Prolog!):

```
brother(X,Y), sister(X,Y) :- father(Z,X), father(Z,Y).
```

Programming in Logic

Logic Programming conventionally refers to programming in the Horn Clause subset of logic. All clauses in a logic program are considered to form a conjunction.

```
brother(X,Y) :- male(X), father(Z,X), father(Z,Y).   ∧
sister(X,Y) :- female(X), father(Z,X), father(Z,Y).   ∧
parent(X,Y) :- father(X,Y).   ∧
parent(X,Y) :- mother(X,Y).   ∧
ancestor(X,Y) :- parent(X,Y).   ∧
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).   ∧
father(john, mary).   ∧          father(john, tom).   ∧
mother(ann, tom).   ∧
male(john).   ∧                  male(tom).   ∧
female(mary).
```

Execution of Prolog Programs

```
brother(X,Y) :- male(X), father(Z,X), father(Z,Y).
father(john, mary).
father(john, tom).
male(john).
male(tom).

?- brother(tom, mary).
yes.
?- brother(mary, mary).
no.
?- brother(W, mary).
W = tom
```

How are the search trees different?

```
brother(X,Y) :- male(X), father(Z,X), father(Z,Y).
father(john, mary).
father(john, tom).
male(john).
male(tom).
```

?- brother(W, mary).

?- male(W), father(Z,W), father(Z,mary).

{W=john}

?- father(Z,john), father(Z,mary).

{Z=john, john=mary} **! Cannot proceed. next.**

{Z=john, john=tom} **! Cannot proceed. next.**

Backtrack.

?- male(W), father(Z,W), father(Z,mary).

{W=tom}

?- father(Z,tom), father(Z,mary).

{W=tom}{Z=john}

?- father(john,mary).

?- □.

W=tom {W=tom}{Z=john}

```
brother(X,Y) :- male(X), father(Z,X), father(Z,Y).
father(john, mary).
father(john, tom).
male(john).
male(tom).
```

?- brother(W, mary).

?- male(W), father(Z,W), father(Z,mary). **Deep backtracking.**

W = john | W = tom

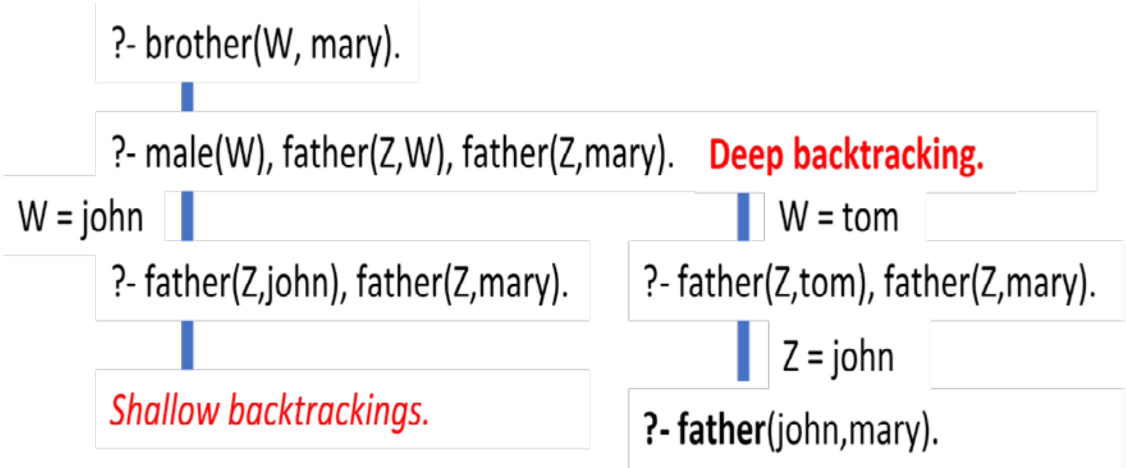
?- father(Z,john), father(Z,mary). | ?- father(Z,tom), father(Z,mary).

Shallow backtrackings. | Z = john

?- **father**(john,mary).

What do we observe from the *Search Tree*?

- Prolog selects ***goals*** to solve from left to right. It always selects and solves the left-most goal.
- Prolog finds clauses to use from top to bottom. The first applicable clause is always applied.
- Prolog backtracks when the selected goal cannot be solved.
- Prolog returns the answer found (or 'yes') during the search iff all goals are solved.



Unification as 2-way Pattern Matching

Consider the following case:

Goal: ?- p(X, a), …

Program p(a, b) :- q(…), …
p(b, Y) :- q(…), …
p(Z, Z) :- q(…), …

The second rule is applicable with ***unifier*** {X=b, Y=a} and the third with ***unifier*** {Z=a, X=a}.

Unification as 2-way Pattern Matching

p(X, a), …
p(a, b) :- … **! No unifier**

p(X, a), …
p(b, Y) :- …
➔ {X=b, Y=a}   Both become p(b,a).

p(X, a), …
p(Z, Z) :- …
➔ {X=Z, Z=a}   Both become p(a,a).

Unification as 2-way Pattern Matching

More examples.

p(f(Y), W, g(Z))
p(U, U, V).
➔ {U = f(Y), U = W, V = g(Z)}
Both become p(f(Y), f(Y), g(Z))

p(f(Y), W, g(Z))
p(V, U, V).
➔ **No unifier!**

Unification as 2-way Pattern Matching

More examples.

p(a, X, f(g(Y)))
p(Z, h(Z, W), f(W)).

➔ {Z = a, X = h(Z, W), W = g(Y)}
Both become p(a, h(a, g(Y)), f(g(Y))).

Unification is performed in each step of search. The ***most general unifier*** is found.

Consider the query and the program shown below:

```
?- p(X, a), …

p(a, b)  :- q(…), …
p(b, Y)  :- q(…), …
p(Z, Z)  :- q(…), …
```

What are the possible unifiers and which of them are the most general unifiers?

Lists in Prolog

```
allZero([]).
allZero(L) :- L = [0|T], allZero(T).
```

?- allZero([]).
yes.

?- AllZero([1]).
{[1]=[]} **! Cannot proceed. next.**
Why is this a shallow backtracking?

{L=[1]}

?- [1]=[0|T], allZero(T).
no.

Lists in Prolog

```
allZero([]).
allZero(L) :- L = [0|T], allZero(T).
```

```
?- allZero([0]).
{[0]=[]} ! Cannot proceed.  next.
                                                    {L=[0]}
?- [0]=[0|T], allZero(T).
                                               {L=[0]}{T=[]}
?- allZero([]).
?- □.
yes.                                           {L=[0]}{T=[]}
```

Lists in Prolog

```
allZero([]).
allZero(L) :- L = [0|T], allZero(T).
```

?- allZero([0,0]).

{[0,0]=[]} **! Cannot proceed. next.**

{L=[0,0]}

?- [0,0]=[0|T], allZero(T).

{L=[0,0]}{T=[0]}

?- allZero([0]).

{[0]=[]} **! Cannot proceed. next.**

{L=[0,0]}{T=[0]}{L_1=[0]}

?- [0]=[0|T_1], allZero(T_1).

{L=[0,0]}{T=[0]}{L_1=[0]}{T_1=[]}

?- allZero([]).

?- □.

yes. {L=[0,0]}{T=[0]}{L_1=[0]}{T_1=[]}

Lists in Prolog

```
allZero([]).
~~allZero(L) :- L = [0|T], allZero(T).~~
allZero([0|T]) :- allZero(T).
```

```
?- allZero([0]).
{[0]=[]} ! Cannot proceed.  next.
                                        {T=[]}
?- allZero([]).
?- □.
yes.                                    {T=[]}
```

Lists in Prolog

```
allZero([]).
~~allZero(L) :- L = [0|T], allZero(T).~~
allZero([0|T]) :- allZero(T).
```

?- allZero([0,0]).

{[0,0]=[]} **! Cannot proceed. next.**

{T=[0]}

?- allZero([0]).

{[0]=[]} **! Cannot proceed. next.**

{T=[0]}{T_1=[]}

?- allZero([]).

?- ☐.

yes. {T=[0]}{T_1=[]}

Lists in Prolog

```
allZero([]).
allZero([0|T]) :- allZero(T).
```

?- allZero([A,B,A]).

{[A,B,A]=[]} **! Cannot proceed. next.**

We shall not show shallow backtrackings from now on.

{A=0, T=[B,A]}

?- allZero([B,0]).

{A=0, T=[B,A]}{B=0, T_1=[]}

?- allZero([]).

?- □.

yes. {A=0, T=[B,A]}{B=0, T_1=[]}

A=0

B=0

(We say the **binding** of A is 0 and the **binding** of B is 0.)

Some list manipulation functions

```
append([], N, [N]).
append([X|List1], N, [X|List2]) :- append(List1, N, List2).
```

?- append([1,2], 3, Y).

(Shallow backtrackings are not shown.)

{X=1,List1=[2],N=3,Y=[X|List2]}

?- append([2], 3, List2).

{X=1,List1=[2],N=3,Y=[X|List2]} $\{X_1=2, List1_1=[], N_1=3, List2=[X_1|List2_1]\}$

?- append([], 3, $List2_1$).

{X=1,List1=[2],N=3,Y=[X|List2]}$\{X_1=2, List1_1=[], N_1=3, List2=[X_1|List2_1]\}\{N_2=3, List2_1=[N_2]\}$

?- □.

yes.

Y=[1,2,3]

(Note: Y = [X|List2] = $[1,X_1|List2_1]$ = $[1,2|[N_2]]$ = $[1,2,N_2]$ = [1,2,3].)

Some list manipulation functions

```
append([], N, [N]).
append([X|List1], N, [X|List2]) :- append(List1, N, List2).
```

```
?- append(L, 3, [1,2,3]).
L=[1,2]

?- append([1,2], N, [1,2,3]).
N=3
```

Some list manipulation functions

```
append([], N, [N]).
append([X|List1], N, [X|List2]) :- append(List1, N, List2).
```

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

{L=[X|List1],N=3,X=1,List2=[2,3]}

?- append(List1, 3, [2,3]).

{L=[X|List1],N=3,X=1,List2=[2,3]} {List1=[X_1|$List1_1$],N_1=3,X_1=2,$List2_1$=[3]}

?- append($List1_1$, 3, [3]).

{L=[X|List1],N=3,X=1,List2=[2,3]} {List1=[X_1|$List1_1$],N_1=3,X_1=2,$List2_1$=[3]} {$List1_1$=[],N_2=3}

?- $\square$.

yes.

L=[1,2]

(Note: L = [X|List1] = [1,X_1|$List1_1$] = [1,2|[]] = [1,2].)

Some list manipulation functions

```
concat([], List1, List1).
concat([N|List1], List2, [N|List3]) :- concat(List1, List2, List3).
```

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

?- concat([1,X],[5,6],[1,X,X,Y]).
X=5
Y=6

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

Some list manipulation functions

How is LP different from FP?

```
append([], N, [N]).
append([X|List1], N, [X|List2]) :- append(List1, N, List2).
```

```
?- append([1,2], 3, X).
?- append(L, 3, [1,2,3]).
?- append([1,2], N, [1,2,3]).
```

```
- fun append([],n:int) = n::[]
=   | append(x::list1,n) = x::append(list1,n);
val append = fn : int list * int -> int list
```

Another Prolog Example

The following Prolog program defines tree insertion (*more precisely, it defines the relation between the new item, the tree before insertion and the tree after insertion*).

```
insert(Newitem, emptyTree, node(emptyTree, Newitem, emptyTree).
insert(Newitem, node(Left, Olditem, Right), node(New_left, Olditem, Right) :-
        Newitem =< Olditem, insert(Newitem, Left, New_left).
insert(Newitem, node(Left, Olditem, Right), node(Left, Olditem, New_right) :-
        Newitem > Olditem, insert(Newitem, Right, New_right).
```

How is the tree in this Prolog program implemented?

```
?- insert(5, node(node(emptyTree, 2, emptyTree), 10, emptyTree), X).
```

Negation and the Closed-World Assumption

```
good(a).
good(b).

?- good(a).
yes.
?- good(c).
no.
?- not(good(c)).
yes.
```

The not/1 predicate returns true if its argument returns false (*i.e.*, cannot be proved). It returns false otherwise. → **CWA**.

```
good(a).
good(b).
colourful(a).
colourful(c).

?- good(X).
X=a;
X=b.

?- not(good(X)).
no.  ← Nothing is not good?
?- not(good(c)).
yes.  ← But c is not good?
?- colourful(X), good(X).
X = a
?- colourful(X), not(good(X)).
X = c
?- not(good(X)), colourful(X).
no.
```

The not/1 in Prolog is problematic.

Arithmetic in Prolog

In Prolog, ‘=’ is used for unification and ‘is’ for evaluation of arithmetic expressions.

```
sum([], 0).
sum([N|List], S) :- sum(List, S1), S is N + S1.
```

Test what happens for the following program (exercise):

```
sum([], 0).
sum([N|List], S) :- sum(List, S1), S = N + S1.
```

Also, consider the following program:

```
sum([], 0).
sum([N|List], S) :- S is N + S1, sum(List, S1).
```

Defining Natural Numbers and Arithmetic Operations in Logic

Extra-logical constants are not strictly necessary in Logic Programming. However, they greatly improve efficiency.

```
natural(0).
natural(s(X)) :- natural(X).

add(0, X, X).
add(s(N), X, s(Y)) :- add(N, X, Y).
```

Exercise: define *subtract*, *multiply* and *divide*.