- FUNCTIONAL PROGRAMMING

- IMPERATIVE PROGRAMMING
  - single-assignment (SISAL, ID)
    > val x : int;
    > x := 5;  > x := 6;
    (good for scientific computation)
  - consistent-assignment
    > val x = [1,2,3];
    > val y = [1,2,?]; (* partial list *)
    > y := [1,2,3,?]; (* increases info *)
    > y := [1,5];  (* contradiction *)

======> LOGIC PROGRAMMING
Logic Programming

- FUNCTIONS: \( y = f(x) \)
- RELATIONS: \( R(x, y) \). Arguments and results are treated uniformly.

ML: 
```ml
fun append (nil, x) = x 
| append (hd::tl, x) = hd::append(tl, x);
append([1,2], [3,4]);
val it = [1,2,3,4] : int list
append relation (X,Y,Z):
  []  [1]  [1]
  [1,2] [3] [1,2,3]
  :  :  :
append([], [], []);
append([], [1], [1]);
append([1,2], [3], [1,2,3]);
append(x, [1], [1]);
append([1], x, [1]);
append([1], [], x);
```

Facts and Rules

`append([] , X , X).`

`append([a,b],[c,d],[a,b,c,d])`
   `if append([b],[c,d],[b,c,d]).`

Notation: `[a,b] = [a | [b]]`

`append([a | [b]],[c | [d]],[a | [b,c,d]])`
   `if append([b],[c | [d]],[b,c,d])].`

`append([H | X1],Y,[H | Z]) :-`
   `append(X1,Y,Z).`

- Horn Clauses:
  P if Q1 and Q2 ..... and Qn
append([], X, X).
append([H | X1], Y, [H | Z]) :-
    append(X1, Y, Z).

?- append([a, b], [c, d], [a, b, c, d]).
yes

?- append([a, b], [c, d], Z).
Z = [a, b, c, d] ?
yes

?- append(X, [c, d], [a, b, c, d]).
X = [a, b] ?
yes

?- append([a, b], Y, [a, b, c, d]).
Y = [c, d] ?
yes
append([],X,X).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).

| ?- append([a],[b,c,d],X).
X = [a,b,c,d] ? ;
no

| ?- append(X,Y,[a,b,c,d]).
X = [],
Y = [a,b,c,d] ? ;
X = [a],
Y = [b,c,d] ? ;
X = [a,b],
Y = [c,d] ? ;
X = [a,b,c],
Y = [d] ? ;
X = [a,b,c,d],
Y = [] ? ;
no
Horn clauses cannot represent negative information. That is, you cannot ask if a tuple is NOT in a the relation append. The system can come back with either yes or fail not with a yes/no answer.
- Prefix relation

? prefix(X,Z) :- append(X,Y,Z).

- Suffix relation

Logic programming:

- Use of facts and rules to represent the information

- Use deduction to answer queries.

- Kowalski definition:

\[
\text{ALGORITHM} = \text{LOGIC} + \text{CONTROL}
\]

\[
P : - Q_1, Q_2, \ldots, Q_n
\]

the goals $Q_1$ $Q_2$ ... $Q_n$ are executed from left to right.
FACTS

father(john, mary).
father(sam, john).
father(sam, kathy).

sicstus
SICStus 2.1 #8: Wed Apr 28 18:33:10 PDT 1993
| ?-[facts].
| ?- father(john, mary).
yes
| ?- father(sam, X).
X = john ? ;  (carriage return if you are not interested in the next answer)
X = kathy ? ;
no
| ?- father(X, mary).
X = john ? ;
no
father(john,mary).
father(sam,john).
father(sam,kathy).

?- father(X,john),father(X,kathy).
X = sam ? ;
no

| ?- father(X,john),father(X,Y).
X = sam,
Y = john ? ;
X = sam,
Y = kathy ? ;
no
| ?- father(X,john),father(X,Y),Y \(\leq\) john.
X = sam,
Y = kathy ? ;
no
father(john,mary).
father(sam,john).
father(sam,kathy).

<table>
<thead>
<tr>
<th>?- father(X,Y),father(X,Z).</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = john,</td>
</tr>
<tr>
<td>X = sam,</td>
</tr>
<tr>
<td>Y = john,</td>
</tr>
<tr>
<td>Z = john ? ;</td>
</tr>
<tr>
<td>X = sam,</td>
</tr>
<tr>
<td>Y = john,</td>
</tr>
<tr>
<td>Z = kathy ? ;</td>
</tr>
<tr>
<td>X = sam,</td>
</tr>
<tr>
<td>Y = kathy,</td>
</tr>
<tr>
<td>Z = john ? ;</td>
</tr>
<tr>
<td>X = sam,</td>
</tr>
<tr>
<td>Y = kathy,</td>
</tr>
<tr>
<td>Z = kathy ? ;</td>
</tr>
<tr>
<td>no</td>
</tr>
</tbody>
</table>
Rules

- `<term>` :
  - `<term1>`, `<term2>`, …., `<termn>`
  - HEAD
  - CONDITIONS

- A fact is a special rule with no conditions.

father(john, mary).
father(sam, john).
father(sam, kathy).

grandpa(X, Y) :- father(X, Z), father(Z, Y).
| ?- grandpa(X, Y).
X = sam,
Y = mary ? ;
no
Unification.

?- X is 2+3.
X = 5 ?

?- X = 2+3. (2+3=X)
X=2+3?

?- 5 = 2+3.
no

?- 2+3 = 2 + Y.
Y=3?

?- 2+3 = 3 + Z.
no

?- f(X,b)=f(a,Y).
X = a,
Y = b ?
FACT: identify(Z,Z).
| ?- identify(f(X,b),f(a,Y)).
X = a,
Y = b ?

- Instance: a term obtained by substituting subterms for the variables.
- f(a,b) is an instance of f(X,b)
- f(a,b) is an instance of f(a,Y)
- g(a,a) is an instance of g(X,X)
- g(h(a),h(b)) is not an instance of g(X,X).

- Deduction in Prolog is based in unification.
Two terms T1 and T2 are said to be unifiable if they have a common instance.

| ?- X is 2+3,X=5.
X = 5 ?

| ?- X is 2+3, X = 2 + 3.
no
| ?- 2+3 is X.
ERROR

| ?- X is 5, 6 is X+1.
X = 5 ?

| ?- X=5, 6 is X+1.
X = 5 ?

| ?- X=5, 6=X+1.
no

| ?- 6 is X+1, X=5.
ERROR

| ?- 2*X = Y*(3+Y).
X = 3+2,
Y = 2 ?
| ?- Y=2, 2*X is Y*(Y+3).
no

| ?- Y=2, Z is Y*(Y+3).
Y = 2,
Z = 10 ?

| ?- X=2+2.
X =
Prolog interruption (h for help)? a
{Execution aborted}

| ?- X=2+X.
X = 2+(2+(2+(2+(2+(2+(2+(2+(2+
Prolog interruption (h for help)? a
{Execution aborted}
LISTS

Notation:

\([a,b,c] = [a, b, c \mid \, \, \, ] = [a \mid [b,c]]\)

\(| \,- [\,H \mid T\,] = [a,b,c].\)

\(H = a,\)

\(T = [b,c] \, ?\)

\(| \,- [a \mid T] = [H, b, c].\)

\(H = a,\)

\(T = [b,c] \, ?\)
datatype bintree = empty | node of int*bintree*bintree;

fun member(k,empty) = false
| member(k,node(n,s,t)) =
    if k<n then member(k,s)
    else if k>n then member(k,t)
    else true;

member(K,node(K,_,_)).
member(K,node(N,S,)) :- K < N, member(K,S).
member(K,node(N,_,T)) :- K > N, member(K,T).
member(M, [M|_]).
member(M, [_ | T]) :- member(M, T).

overlap(X, Y) :- member(M, X), member(M, Y).

| ? overlap(Z, [a, b, c, d]), member(Z, [1, 2, c, d]).

(infinite computation)
| ?- X = [1, 2, 3], member(a, X).

no
| ?- member(a, X), X = [1, 2, 3].

(infinite computation)
Variables in terms

\[ \text{?- \quad L=[a,b \mid X].} \]
\[ \text{L = [a,b\mid X] \ ? \ ;} \]
\text{no}

\[ \text{?- \quad L=[a,b \mid X], \quad X = [C,Y].} \]
\[ \text{L = [a,b,C,Y],} \]
\[ \text{X = [C,Y] \ ? \ ;} \]
\text{no}

- Unification of a variable representing the end of the list is similar to an assignment to that variable.
Order in the rules

append([], Y, Y).
append([H | X], Y, [H | Z]) :-
    append(X, Y, Z).
prefix(X, Z) :- append(X, Y, Z).
suffix(Y, Z) :- append(X, Y, Z).

? suffix([a], L), prefix(L, [a, b, c]).
L=[a]?
suffix([a],L) if append(X’,[a],L)

? append(X’,[a],L),prefix(L,[a,b,c]).
  X’ -> []
  Y -> [a]
  L -> [a]

? prefix([a],[a,b,c]).

? append([a],Y,[a,b,c]).

? append([],Y,[b,c]).
? suffix([b],L),prefix(L,[a,b,c]).
? append(X,[b],L),prefix(L,[a,b,c]).
    X  -->  []
    Y  -->  [b]
    L  -->  [b]
? prefix([b],[a,b,c]).
? append([b],Y,[a,b,c]).  ***  BACKTRACK
? append(X,[b],L),prefix(L,[a,b,c]).
append([H' | X'], Y', [H' | Z']) :-
    append(X', Y', Z').
    X  -->  [H'|X']
    Y'--->  [b]
    L  -->  [H' | Z']
? append(X',[b],Z'),prefix([H' | Z'],[a,b,c]).
    X'  -->  []
    Y  -->  [b]
    Z'  -->  [b]
? prefix([H' | [b]],[a,b,c]).
? append([H' | [b]],Y,[a,b,c]).
append([H'' | X'''], Y'', [H'' | Z'']) :-
    append(X'', Y'', Z'')
? append([b],Y,[b,c]).
? append([],Y,[c]).
ORDER OF GOALS

? suffix([a],L), prefix(L,[a,b,c]).
? append(X,[a],L), prefix(L,[a,b,c]).
L = [a];

? prefix(L,[a,b,c]), suffix([a],L).
CUTS

- Prolog is not really logic programming

B :- C1, C2, !, C3
B :- !, C

Control backtrack past B without considering any remaining rules for B. Thus, the cut makes B fail if C fails.

b :- c.
b :- d.
b :- e.

? b, G.

b :- c.
b :- !, d.
b :- e. \textit{(will never be considered)}
a(1) :- b.
a(2) :- e.
b :- ! c.
b :- d.
d.
e.

?a(X)
X=1;
X=2:
no

?a(X)
X=2;
no
- Good use of CUT

member(K, node(K, _, _)).
member(K, node(N, S, _)) :- K < N, ! member(K, S).
member(K, node(N, _, T)) :- K > N, member(K, T).