LECTURE 21

Logic Programming

LOGIC PROGRAMMING

- Logic programming is a form of declarative programming.
- A program is a collection of axioms.
- Each axiom is a *Horn clause* of the form:

$$H \leftarrow B_1, B_2, \ldots, B_n.$$

where H is the head term and the B's are the body terms.

- Meaning: H is true if all B_i are true.
- A user states a goal (a theorem) to be proven.
- The logic programming system uses inference steps to prove the goal (theorem) is true, applying a logical *resolution* strategy.

RESOLUTION STRATEGIES

- To deduce a goal (theorem), the programming system searches axioms and combines sub-goals using a resolution strategy.
- For example, given the axioms:

  $C \leftarrow A, B.$ $A \leftarrow true.$
  $D \leftarrow C.$ $B \leftarrow true.$

- *Forward chaining* deduces first that C is true: $C \leftarrow A, B$
  and then that D is true: $D \leftarrow C$
- *Backward chaining* finds that D can be proven if sub-goal C is true: $D \leftarrow C$
  The system then deduces that the sub-goal is C is true: $C \leftarrow A, B$
  Since the system could prove C, it has proven D.

PROLOG

- Prolog uses backward chaining, which is more efficient than forward chaining for larger collections of axioms.
- Prolog is interactive (mixed compiled/interpreted).
- Example applications:
  - Expert systems.
  - Artificial intelligence.
  - Natural language understanding.
  - Logical puzzles and games.

SWI-PROLOG

SWI-Prolog is a popular prolog system.

- Login to linprog.cs.fsu.edu.
- `pl` (or `swipl`) to start SWI-Prolog.
- `halt.` to halt Prolog (period is the Prolog command terminator).

SWI-Prolog manual

DEFINITIONS: PROLOG CLAUSES

- A program consists of a collection of Horn clauses.
- Each clause consists of a head predicate and body predicates:

$$H \leftarrow B_1, B_2, \ldots, B_n.$$

- A clause is either a rule, e.g. `snowy(X) :- rainy(X), cold(X).`
  Meaning: "If X is rainy and X is cold then this implies that X is snowy."
- Or a clause is a fact, e.g. `rainy(rochester).`
  Meaning: "Rochester is rainy."
- This fact is identical to the rule with true as the body predicate:

```
rainy(rochester) :- true.
```

LOGICAL PREDICATES

- A *predicate* is a verb phrase template that describes a property of objects, or a relationship among objects represented by the variables.
- Let's say we have the sentences:
  - "The car Tom is driving is blue"
  - "The sky is blue"
  - "The cover of this book is blue"
- All of these come from the template "is blue" by placing an appropriate phrase before it.
- The phrase "is blue" is a *predicate* and it describes the property of being blue.
- Let's say we allow B to be the name for the predicate "is_blue", then we can express that x is blue by "B(x)", where x represents an arbitrary object. B(x) reads as "x is blue".

DEFINITIONS: QUERIES AND GOALS

A program basically is a database of facts and rules (clauses).

In SWI-Prolog:

- To load program: `?- ['prologfilename'].`
  - e.g. `?- ['example.pl']`   ← Note single quotes!
- To list all clauses: `?- listing.`
- To list clauses related to a name: `?- listing(name).`

After the program is loaded, the goals can be 'executed' by doing queries.

DEFINITIONS: QUERIES AND GOALS

- A *query* is interactively entered by a user after a program is loaded.
- A query has the form $?-G_1, G_2, \ldots, G_n$ where the G's are goals (logical predicates).
- A *goal* is a predicate to be proven true by the programming system.
- Example program (example1.pl) with two facts:

```
rainy(seattle).
rainy(rochester).
```

Query with one goal to find which city C is rainy (if any): `?- rainy(C).`

Response by the interpreter: `C = seattle`

Type a semicolon ; to get next solution: `C = rochester`

Typing another semicolon does not return another solution.

EXAMPLE1.PL

```
carnahan@linprog1.cs.fsu.edu:~/COP4020/proj3>pl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 5.6.62)

For help, use ?- help(Topic). or ?- apropos(Word).

?- ['example1.pl'].
% example1.pl compiled 0.00 sec, 1,288 bytes
true.

?- rainy(C).
C = seattle ;
C = rochester.

?-
```

EXAMPLE 2

Consider a program with three facts and one rule (example2.pl):

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```

```
?- ['example2.pl'].
% example1.pl compiled 0.00 sec, 1,984
bytes
true.

?- snowy(rochester).
true.

?- snowy(seattle).
false.

?- snowy(paris).
false.

?- snowy(C).
C = rochester.

?-
```

BACKWARD CHAINING WITH BACKTRACKING

Consider again:

```
?- snowy(C).
C = rochester
```

The system first tries C = seattle:

```
rainy(seattle)
cold(seattle)  fail
```

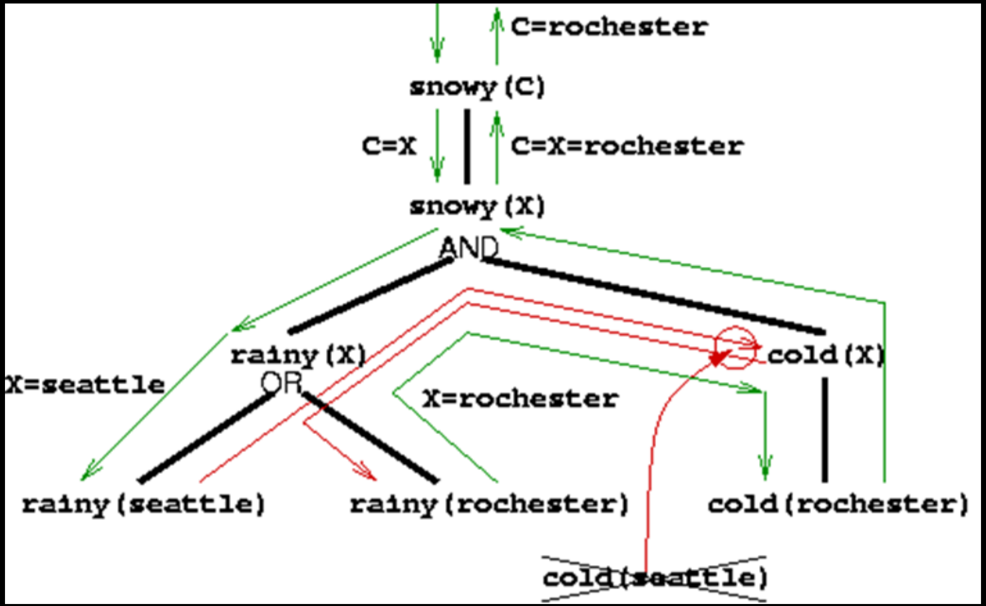
Then C = rochester:

```
rainy(rochester)
cold(rochester)
```

When a goal fails, backtracking is used to search for solutions.

The system keeps this execution point in memory together with the current variable bindings.

Backtracking unwinds variable bindings to establish new bindings.



An unsuccessful match forces backtracking in which alternative clauses are searched that match (sub-)goals.

EXAMPLE 3: FAMILY RELATIONSHIPS

Facts:

```
male(albert).
male(edward).
female(alice).
female(victoria).
parents(edward, victoria, albert).
parents(alice, victoria, albert).
```

Rule: `sister(X,Y) :- female(X), parents(X,M,F), parents(Y,M,F).`

Query: `?- sister(alice, Z).`

1. `sister(alice,Z)` **matches the rule:** `X=alice, Y=Z.`
2. **New goals:** `female(alice),parents(alice,M,F),parents(Z,M,F).`
3. `female(alice)` **matches 3rd fact.**
4. `parents(alice,M,F)` **matches 5th fact:** `M=victoria, F=albert.`
5. `parents(Z,victoria,albert)` **matches 6th fact:** `Z=edward.`

The system applies backward chaining to find the answer (example3.pl).

EXAMPLE 4: MURDER MYSTERY

```
% the murderer had brown hair:
      murderer(X) :- hair(X, brown).
% mr_holman had a ring:
      attire(mr_holman, ring).
% mr_pope had a watch:
      attire(mr_pope, watch).
% If sir_raymond had tattered cuffs then mr_woodley had the pincenez:
      attire(mr_woodley, pincenez) :-
      attire(sir_raymond, tattered_cuffs).
% and vice versa:
      attire(sir_raymond,pincenez) :-
      attire(mr_woodley, tattered_cuffs).
% A person has tattered cuffs if he is in room 16:
      attire(X, tattered_cuffs) :- room(X, 16).
% A person has black hair if he is in room 14, etc:
      hair(X, black) :- room(X, 14).
      hair(X, grey) :- room(X, 12).
      hair(X, brown) :- attire(X, pincenez).
      hair(X, red) :- attire(X, tattered_cuffs).
% mr_holman was in room 12, etc:
      room(mr_holman, 12).
      room(sir_raymond, 10).
      room(mr_woodley, 16).
      room(X, 14) :- attire(X, watch).
```

MURDER MYSTERY (CONT'D)

Question: who is the murderer?

```
        ?- murderer(X).
```

Execution trace (indentation shows nesting depth):

```
murderer(X)
   hair(X, brown) /* murderer(X) :- hair(X, brown). */
      attire(X, pincenez) /* hair(X, brown) :- attire(X, pincenez). */
         X = mr_woodley
         attire(sir_raymond, tattered_cuffs)
            room(sir_raymond, 16)
            FAIL (no facts or rules)
         FAIL (no alternative rules)
      REDO (found one alternative rule)
      attire(X, pincenez)
         X = sir_raymond
         attire(mr_woodley, tattered_cuffs)
            room(mr_woodley, 16)
            SUCCESS
         SUCCESS: X = sir_raymond
      SUCCESS: X = sir_raymond
   SUCCESS: X = sir_raymond
SUCCESS: X = sir_raymond
```

Check out example4.pl

TRACING

Use the “trace.” command.

```
?- ['example4.pl'] .
% example4.pl compiled 0.00 sec, 4,400 bytes
true.

?- trace.
Unknown message: query(yes)
[trace]  ?- murderer(X) .
   Call: (7) murderer(_G335) ? creep
   Call: (8) hair(_G335, brown) ? creep
   Call: (9) attire(_G335, pincenez) ? creep
   Call: (10) attire(sir_raymond, tattered_cuffs) ? creep
   Call: (11) room(sir_raymond, 16) ? creep
   Fail: (11) room(sir_raymond, 16) ? creep
   Fail: (10) attire(sir_raymond, tattered_cuffs) ? creep
   Redo: (9) attire(_G335, pincenez) ? creep
   Call: (10) attire(mr_woodley, tattered_cuffs) ? creep
   Call: (11) room(mr_woodley, 16) ? creep
   Exit: (11) room(mr_woodley, 16) ? creep
   Exit: (10) attire(mr_woodley, tattered_cuffs) ? creep
   Exit: (9) attire(sir_raymond, pincenez) ? creep
   Exit: (8) hair(sir_raymond, brown) ? creep
   Exit: (7) murderer(sir_raymond) ? creep
X = sir_raymond
```

DEFINITIONS: PROLOG TERMS

- Terms are symbolic expressions that are Prolog's building blocks.
- A Prolog program consists of Horn clauses (axioms) that consist of terms.
- Data structures processed by a Prolog program are terms.
- A term is either
  - A variable: a name beginning with an upper case letter.
  - A constant: a number or string.
  - An atom: a symbol or a name beginning with a lower case letter.
  - A structure of the form `functor(arg1, arg2, ..., argn)` where functor is an atom and the args are sub-terms.
  - A list of the form `[term1, term2, …, termn]`.
- Examples:
  - `X`, `Y`, `ABC`, and `Alice` are variables.
  - `7`, `3.14`, and `"hello"` are constants.
  - `foo`, `barFly`, and `+` are atoms.
  - `bin_tree(foo, bin_tree(bar, glar))` and `+(3,4)` are structures.

UNIFICATION AND VARIABLE INSTANTIATION

- A variable is instantiated to a term as a result of *unification*, which takes place when goals are matched to head predicates.
  - Goal in query: `rainy(C)`
  - Fact: `rainy(seattle)`
  - Unification is the result of the goal-fact match: `C=seattle`
- Unification is recursive:
  - An uninstantiated variable unifies with anything, even with other variables which makes them identical
  - An atom unifies with an identical atom.
  - A constant unifies with an identical constant.
  - A structure unifies with another structure if the functor and number of arguments are the same and the arguments unify recursively.
- Equality in Prolog is defined in terms of *unifiability*.

EXAMPLES OF UNIFICATION

- The built-in predicate `=(A,B)` succeeds if and only if `A` and `B` can be unified, where the goal `=(A,B)` may be written as `A = B`.

```
?- a = a.
yes
?- a = 5.
No
?- 5 = 5.0.
No
?- a = X.
X = a
?- foo(a,b) = foo(a,b).
Yes
?- foo(a,b) = foo(X,b).
X = a
?- foo(X,b) = Y.
Y = foo(X,b)
?- foo(Z,Z) = foo(a,b).
no
```

PROLOG LISTS

- A list is of the form: `[elt1, elt2, ..., eltn]` where elti are terms.
- The special list form `[elt1,elt2, ..., eltn | tail]` denotes a list whose tail list is tail.

Examples:

```
?- [a,b,c] = [a|T].
T = [b,c]

?- [a,b,c] = [a,b|T].
T = [c]

?- [a,b,c] = [a,b,c|T].
T = []
```

LIST OPERATIONS: LIST MEMBERSHIP

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

```
?- member(b, [a,b,c]).
```

Execution:

- `member(b, [a,b,c])` does not match `member(X, [X|T])`.
- `member(b, [a,b,c])` matches predicate `member(X1, [H1|T1])` with `X1 = b, H1 = a,` and `T1 = [b,c]`.
- Sub-goal to prove: `member(b, [b,c])`
- `member(b, [b,c])` matches predicate `member(X2, [X2|T2])` with `X2=b` and `T2=[c]`.
- The sub-goal is proven, so `member(b,[a,b,c])` is proven (deduced).
- Note: variables can be "local" to a clause (like the formal arguments of a function).
- Local variables such as `X1` and `X2` are used to indicate a match of a (sub)-goal and a head predicate of a clause.

PREDICATES AND RELATIONS

- Predicates are *not* functions with distinct inputs and outputs.
- Predicates are more general and define relationships between objects (terms).
- `member(b,[a,b,c])` relates term `b` to the list that contains `b`.

```
?- member(X, [a,b,c]).
X = a ;     % type ';' to try to find more solutions
X = b ;     % ... try to find more solutions
X = c ;     % ... try to find more solutions
no

?- member(b, [a,Y,c]).
Y = b

?- member(b, L).
L = [b|_G255]
```

where `L` is a list with `b` as head and `_G255` as tail, where `_G255` is a new variable.

EXAMPLE: LIST APPEND

- `L3 = L1 || L2.`
- How do we write this in Prolog?
- Predicates do not have return value
- Implication?
- `L1, L2, L3` must be arguments to a predicate, the programming will then specify the relation.
- `Append(L1, L2, L3).`
  - Define this recursively
  - Case 1: when L1 is empty.
  - Case 2: when L1 is not empty.
- Prolog has no if constructs, how do we handle two cases?

EXAMPLE: LIST APPEND

- `append(L1, L2, L3). % L3 = L1 || L2`
- Define this recursively
  - Case 1: when L1 is empty.
    - `append([], L2, L2).`
  - Case 2: when L1 is not empty.
    - `append([H | T], L2, [H | L]) :- append(T, L2, L).`
- Final solution:

```
append([], A, A).
append([H|T], A, [H|L]) :- append(T, A, L).
```

EXAMPLE: LIST APPEND

If we issued the query `?- append([a,b,c],[1,2,3],X) .`

```
append([a, b, c], [1, 2, 3], _G518)
append([b, c], [1, 2, 3], _G587)
append([c], [1, 2, 3], _G590)
append([], [1, 2, 3], _G593)
append([], [1, 2, 3], [1, 2, 3])
append([c], [1, 2, 3], [c, 1, 2, 3])
append([b, c], [1, 2, 3], [b, c, 1, 2, 3])
append([a, b, c], [1, 2, 3], [a, b, c, 1, 2, 3])

X = [a, b, c, 1, 2, 3]
yes
```

EXAMPLE: LIST APPEND

List append predicate definitions:

```
append([], A, A).
append([H|T], A, [H|L]) :- append(T, A, L).
```

Prolog append has more power then the append function in other languages.

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

?- append(Y, [d,e], [a,b,c,d,e]).
Y = [a,b,c]

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

?- append([a,b],[],[a,b,c]).
No

?- append([a,b],[X|Y],[a,b,c]).
X = c
Y = []
```

EXAMPLE: REMOVING AN ITEM FROM A LIST

- predicate prototype: `takeout(Item, SrcL, DstL)`

Three cases:

- `SrcL` is empty
- `SrcL` is not empty: the first element is `Item` (take out this item).
- `SrcL` is not empty: the first element is not `Item` (keep this item).

EXAMPLE: REMOVING AN ITEM FROM A LIST

- example6.pl
- Item may or may not be in `SrcL`
- Three cases:
- SrcL is empty.
  - `takeout(Item, [], []).`
- `SrcL` is not empty: the first element is `Item` (take out this item)
  - `takeout(Item, [Item | L], L).`
- `SrcL` is not empty: the first element is not `Item` (keep this item)
  - `takeout(Item, [X | L], [X | L1]) :- takeout(Item, L, L1).`
- Item must be in `SrcL`?

EXAMPLE: PERMUTING A LIST

```
perm(L1, L2). %L2 is a permutation of L1
```

- `perm([], []).`
- `perm([X|Y], Z) :- perm(Y, W), takeout(X, Z, W).`

[example7.pl]

EXAMPLE: SORTING

- Interface: `sort(List, Sortedlist).`
  - Remember! All variables must start with a capital letter.
- Naïve_sort (very inefficient sorting)
  - `naive_sort(List,Sorted) :- perm(List,Sorted), is_sorted(Sorted).`

How do we write `is_sorted`?

EXAMPLE: SORTING

- How do we write `is_sorted`?
- `is_sorted([]).`
-
```
is_sorted([_]).   % Underscore signifies anonymous
                  % variable.
```
- `is_sorted([X, Y | L]) :- X < Y, is_sorted([Y|L]).`

PROLOG I/O

Input:

- `seeing(File)` /* save current file for later use – not necessary for stdin*/
- `see(File)` /* open File */
- `read(Data)` /* read data */
- `seen` /* close file */
- `get_char( C )` /* read one character */
- Read `end_of_file` when reaching the end of the file.

PROLOG I/O

```
browse(File) :-
          seeing(Old),       /* save for later */
          see(File),         /* open this file */
          repeat,
          read(Data),        /* read from File */
          process(Data),
          seen,              /* close File */
          see(Old).          /*  previous read source */

process(end_of_file) :- !.
process(Data) :-  write(Data), nl, fail.
```

PROLOG I/O

test.txt

```
rainy(seattle).
rainy(rochester).
```

```
?- ['ioexample.pl'].
% ioexample.pl compiled 0.00 sec, 1,544 bytes

Yes
?- browse('test.txt').
rainy(seattle)
rainy(rochester)

Yes
```

PROLOG I/O

Output:

- `telling(File)` /* save for later use – not necessary for stdout*/
- `tell(File)` /* open file */
- `write(Data)` or `writeq(Data)` /* output data */
- `nl` /* output newline */
- `told` /* close file */
- `writef("%s", [A])` or `format(“~s”, [A])` /* A is a string */

PROLOG I/O

```
takeout(Item, [Item | L], L).
takeout(Item, [X | L], [X | L1]) :- takeout(Item, L, L1).
remove_write_list(Item, L, File) :-
    tell(File),
    takeout(Item, L, L2),
    write(L2),
    told,
    write('Wrote list to file.').
```

PROLOG I/O

```
?- ['ioexample2.pl'].
% ioexample2.pl compiled 0.00 sec, 1,328 bytes

Yes
?- remove_write_list(3, [1,2,3,4,5], 'output.txt').
Wrote list to file.

Yes
?- halt.
carnahan@diablo:~/COP4020/proj3>more output.txt
[1, 2, 4, 5]
```

PROLOG I/O

```
?- ['hello.pl'].
% hello.pl compiled 0.00 sec, 1,020 bytes
Yes
?- chat.
Hello, what is your name?
|: Caitlin.
Have a great day,_L137!
?- chat.
Hello, what is your name?
|: caitlin.
Have a great day,caitlin!
?- chat.
Hello, what is your name?
|: "Caitlin".
Have a great day, [67, 97, 105, 116, 108, 105, 110]!
```

```
chat :-
    write('Hello, what is your name? '),
    nl,
    read(Name),
    write('Have a great day,'),
    write(Name),
    write('!').
```

PROLOG I/O

```
?- ['hello.pl'].
% hello.pl compiled 0.00 sec, 1,148 bytes

Yes
?- chat.
Hello, what is your name?
|: "Caitlin".
Have a great day,Caitlin!

Yes
```

```
chat :-
    write('Hello, what is your name? '),
    nl,
    read(Name),
    write('Have a great day,'),
    format('~s', [Name]),
    write('!').
```

PROLOG ARITHMETIC

- Arithmetic is needed for computations in Prolog.
- The `is` predicate evaluates an arithmetic expression and instantiates a variable with the result.
- For example:

```
X is 2*sin(1)+1
```

instantiates `X` with the results of `2*sin(1)+1`.

ARITHMETIC EXAMPLES

- A predicate to compute the length of a list:
  - `length([], 0).`
  - `length([H|T], N) :- length(T, K), N is K + 1.`
- where the first argument of length is a list and the second is the computed length.
- Example query:

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

- Defining a predicate to compute GCD:

```
gcd(A, A, A).

gcd(A, B, G) :- A > B, N is A-B, gcd(N, B, G).

gcd(A, B, G) :- A < B, N is B-A, gcd(A, N, G).
```

CONTROL FEATURES

- Prolog offers built-in constructs to support a form of control-flow
- `\+ G` negates a (sub-)goal `G`.
- `!` (cut) terminates backtracking for a predicate.
- `fail` always fails (so, trigger backtracking).

```
?- \+ member(b, [a,b,c]).
no
?- \+ member(b, []).
yes
```

We can (re)define:

```
if(Cond, Then, Else) :- Cond, !, Then.
if(Cond, Then, Else) :- Else.
?- if(true, X=a, X=b).
X = a ; % try to find more solutions
no
?- if(fail, X=a, X=b).
X = b ;
no
```

FACTORIAL EXAMPLE

```
fact(1, 1).
fact(N, X) :- M is N-1, fact(M, Y), X is Y*N.
```

SUMMING THE ELEMENTS OF A LIST

```
?- sum([1,2,3], X).
X=6

sum([], 0).
sum([X|T], Y) :- sum(T, Y1), Y is X + Y1.
```

GENERATING A LIST OF COPIES

```
?- fill(3, x, Y)
Y=[x,x,x]

fill(0, _, []).
fill(N, X, [X|T]):- M is N-1, fill(M, X, T).
```

REPLACE X WITH Y IN LIST XS

```
?- subst(3, 0, [8,2,3,4,3,5], X).
X=[8,2,0,4,0,5]

replace1(_, _, [], []).
replace1(X, Y, [X|T], [Y|T1]) :- replace1(X, Y, T, T1).
replace1(X, Y, [Z|T], [Z|T1]) :- replace1(X, Y, T, T1).
```

SEARCHING FOR AN ITEM IN A LIST

Search for an item in a list, if found, return original list; otherwise, insert at the beginning of the list.

```
search(_, [], 0).
search(X, [X|_], 1).
search(X, [_|T], C) :- search(X, T, C).
searchinsert(X, Y, Y):- search(X, Y, 1).
searchinsert(X, Y, [X|Y]) :- search(X, Y, 0).
```

REMOVE FROM LIST

Remove all items in a list equal to X.

```
remove(_, [], []).
remove(X, [X|T], T1):- remove(X, T, T1).
remove(X, [Y|T], [Y|T1]) :- remove(X, T, T1).
```

INSERT INTO SORTED LIST

```
insertsorted(X, [], [X]).
insertsorted(X, [Y|T], [X, Y|T]) :- X =< Y.
insertsorted(X, [Y|T], [Y|T1]) :- X > Y, insertsorted(X, T, T1).
```

TERM MANIPULATION

- Built-in predicates `functor` and `arg`, for example:
  - `?- functor(foo(a,b,c), foo, 3). % Is foo(a,b,c) a functor called foo with arity 3?`
    `Yes`
  - `?- functor(bar(a,b,c), F, N).`
    `F = bar`
    `N = 3`
  - `?- functor(T, bee, 2).`
    `T = bee(_G1,_G2)        % unified! And new locations created for the args.`
  - `?- functor(T, bee, 2), arg(1, T, a), arg(2, T, b).`
    `T = bee(a,b)`
- The “univ” operator =..
  - Break up a term into a list or form a term from a list.
  - `?- foo(a,b,c) =.. L`
    `L = [foo,a,b,c]`
  - `?- T =.. [bee,a,b]`
    `T = bee(a,b)`

CONTROL FEATURES

- Prolog offers built-in constructs to support a form of control-flow
  - G1 ; G2 forms an “or”: try G1 then G2 if G1 failed
  - G1 -> G2 ; G3 forms an if-then-else: if G1 then G2 else G3
  - true is a true goal and acts like a no-op
  - repeat always succeeds (for looping)

We can use an “or” instead of two clauses:

```
rainy(C) :- ( C = rochester ; C = seattle ).
```

Using an if-then-else:

```
guess(X) :- (  X = heads
            -> format(“~s~n”, [“Guessed it!”])
            ;  format(“~s~n”, [“Sorry, no luck”])
            ).
```

Repeat drawing a random X until X=1 (and then cuts backtracking):

```
do_until_one :- repeat, X is random(10), X = 1, !.
```

META-LOGICAL

Meta-logical predicates perform operations that require reasoning about terms.

- X == Y is true if X is identical to Y (vice versa X \= Y).
- var(X) true if X is uninstantiated (vice versa nonvar(Y)).
- ground(X) true if X is a ground term, has no variables.
- atom(X) true if X is an atom.
- functor(X, F, N) true if X is a functor F with N arguments.
- X =.. [F, A1, A2, …, An] true if X a functor F with arguments.

Pure logic has no ordering constraints, e.g. X “or” Y = Y “or” X

- X=3, var(X). ↔ var(X), X=3.
- Meta-logical predicates are order sensitive and influence control.