Recitation - 10

Jahnavi - jp5867@nyu.edu

Datatypes in Prolog:

1. **Atoms**: Atoms are used to represent constants. An atom is a sequence of letters, digits, and underscore characters, beginning with a lowercase letter.

   Examples of atoms in Prolog include hello, world, foo, and bar.

2. **Numbers**: Prolog has two kinds of numbers: integers and floating-point numbers. Integers are represented by sequences of digits, and can be positive or negative. Floating-point numbers are represented as sequences of digits with a decimal point.

   Examples of numbers in Prolog include 42, -123, 3.14, and -0.01.

3. **Variables**: Variables are used to represent unknown values. A variable is a sequence of letters, digits, and underscore characters, beginning with an uppercase letter or an underscore.

   Examples of variables in Prolog include X, Y, _Temp, and _.

Datatypes in Prolog:

1. **Lists**: Lists are a fundamental data structure in Prolog, and are represented as sequences of terms enclosed in square brackets. The elements of a list can be any Prolog term, including other lists.

   For example, [1, 2, 3] is a list of integers, and [a, [b, c], d] is a list of atoms and another list.

2. **Structures**: Structures are compound terms composed of a functor (which is an atom) and one or more arguments. Structures are represented as functor terms enclosed in parentheses, with the arguments separated by commas.

   For example, point(3, 4) is a structure with functor point and arguments 3 and 4, and person(john, smith, 42) is a structure with functor person and arguments john, smith, and 42.

Clause:

A clause in Prolog is a syntactic construct that defines a logical fact or rule. Facts simply state information, while rules define relationships between terms.

Facts: A fact is a clause that simply states a piece of information. It takes the form of a predicate with no arguments, or a predicate with one or more arguments.

```
john_is_cold.                /* john is cold */
raining.                     /* it is raining */
john_Forgot_His_Raincoat.    /* john forgot his raincoat */
fred_lost_his_car_keys.      /* fred lost is car keys */
peter_footballer.            /* peter plays football */
```

```
eats(fred,oranges).          /* "Fred eats oranges" */
eats(fred,t_bone_steaks).    /* "Fred eats T-bone steaks" */
eats(tony,apples).           /* "Tony eats apples" */
eats(john,apples).           /* "John eats apples" */
eats(john,grapefruit).       /* "John eats grapefruit" */
```

Clause:

Rules: A rule is a clause that defines a relationship between terms. It takes the form of a predicate with one or more arguments, followed by the symbol :-, followed by a list of one or more predicates.

```
fun(X, Y):- sub_1(X), sub_2(Y).
/* A query satisfies goal fun if that query
could both satisfy subgoals sub_1 and sub_2 */
```

A goal in the rule is the head of the rule and subgoal is a goal in the body of the rule.

Operations on goals:

1. Conjunction (logical and, ∧): separating the subgoals by commas.

```
computer(X):- hasHardware(X), hasSoftware(X).
/* X is a computer if it contains hardware and software. */
```

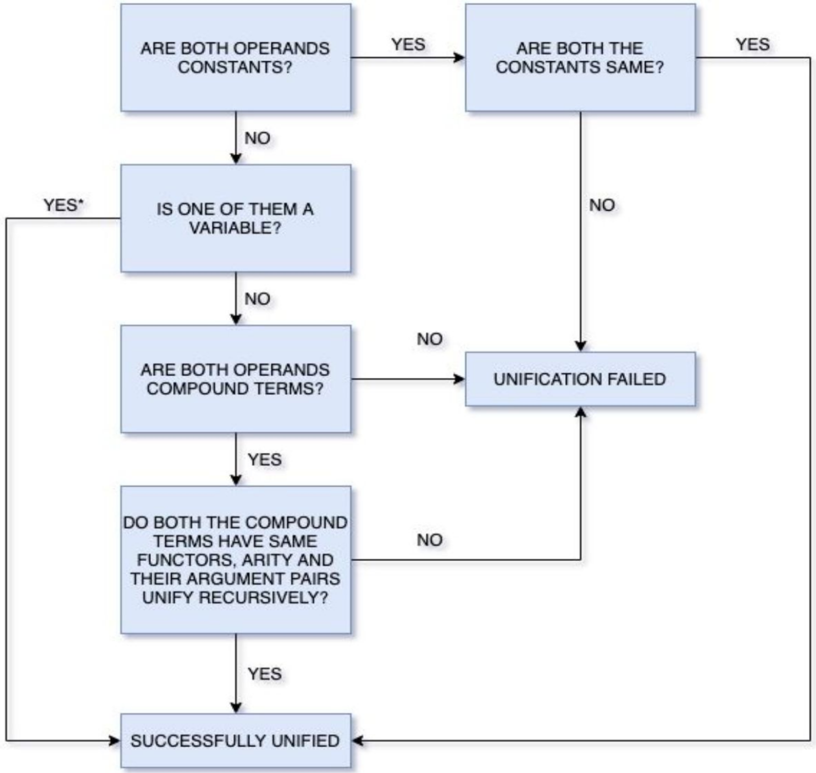
2. Disjunction (logical or, ∨): separating the subgoals by semicolons or defining by separated clauses

```
weather(X):- isSunny(X).
weather(X):- isRainy(X).
/* weather X is either sunny or rainy. */
```

evaluation is short circuit

Unification:

```
?- 3=5.
false.
?- 3=3.
true.
?- X=3.
X = 3.
?- X=f(Y).
X = f(Y).
?- 3=f(Y).
false.
?- f(A)=g(A).
false.
?- f(A)=f(A,B).
false.
?- f(A,B)=f(1,C).
A = 1, B = C.
?-foo(f(c ,d), k(r)) = foo(f(c ,d), k(r))
true.
```



Unification:

```
?- % use comma to unify more than two term
|    love(X,me)= love(you, Y), love(X,me) =love(u, Y).
false.

?- % dot doesn't work
|    love(X,me)= love(you, Y). love(X,me) =love(u, Y).

X = you,
Y = me.

X = u,
Y = me.
```

In Prolog, unify_with_occurs_check/2 is a built-in predicate that can be used to unify two terms while performing an occurs check to ensure that variables are not unified with terms that contain them.

?- unify_with_occurs_check(X, f(X)).
false.

?- unify_with_occurs_check(X, f(Y)).
X = f(Y).

Occurs check: If we try to unify two expressions we must generally avoid situations where the unification process tries to build infinite structures.

Consider: data(X,name(X)).
and try:
?- data(Y,Y).

First we successfully match the first arguments and Y is bound to X. Now we try to match Y with name(X). This involves trying to unify name(X) with X. What happens is an attempt to identify X with name(X) which yields a new problem ---to match name(X) against name(name(X)) and so on. We get a form of circularity which most Prolog systems cannot handle.

To avoid this it is necessary, that, whenever an attempt is made to unify a variable with a compound term, we check to see if the variable is contained within the structure of the compound term.

This check is known as the occurs check. If we try to unify two terms and we end up trying to unify a variable against a term containing that variable then the unification should fail.

Backtracking:

Given a goal (query) with some rules, backtracking is a way to backtrace and

find all possible solutions that satisfy all subgoals.

a. The interpreter tries to match facts and rules by the order of their definition.
b. If a subgoal cannot be satisfied, Prolog will try another way.
c. Consider this example, we check if an item exists in list or not:

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

d. By using the backtracking mechanism, we could find all possible solutions:

```
?- mem(1,[2,3,1,1]).
true ;
true ;
false.
```

Backtracking:

To check backtracking in Prolog, you should type trace. to open trace mode and observe the behavior based on your query. Once you have done, type nodebug. to exit the trace mode.

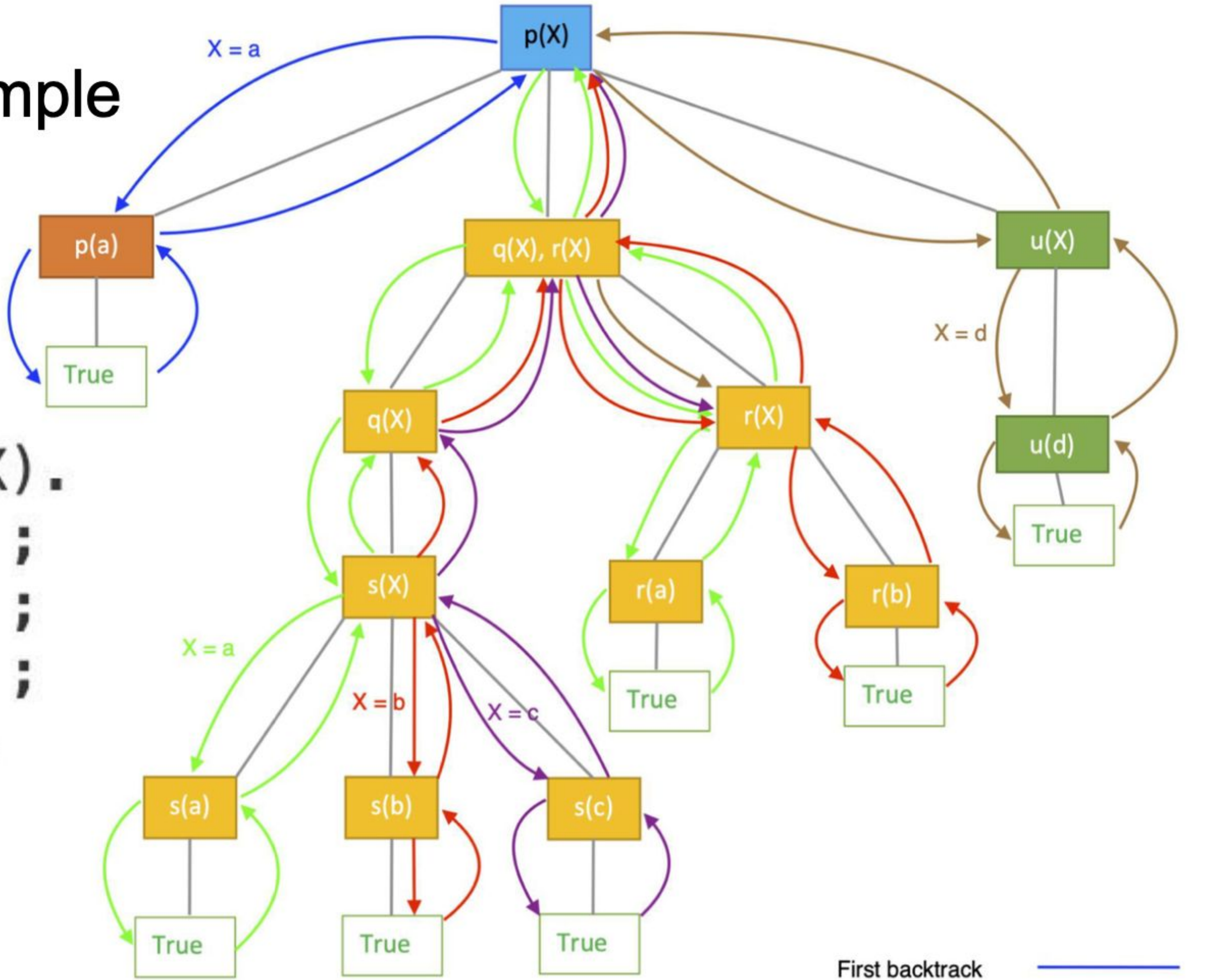
```
?- trace.
true.

[trace]  ?- mem(1,[2,3,1,1]).
   Call: (8) mem(1, [2, 3, 1, 1]) ? creep
   Call: (9) mem(1, [3, 1, 1]) ? creep
   Call: (10) mem(1, [1, 1]) ? creep
   Exit: (10) mem(1, [1, 1]) ? creep
   Exit: (9) mem(1, [3, 1, 1]) ? creep
   Exit: (8) mem(1, [2, 3, 1, 1]) ? creep
true ;
   Redo: (10) mem(1, [1, 1]) ? creep
   Call: (11) mem(1, [1]) ? creep
   Exit: (11) mem(1, [1]) ? creep
   Exit: (10) mem(1, [1, 1]) ? creep
   Exit: (9) mem(1, [3, 1, 1]) ? creep
   Exit: (8) mem(1, [2, 3, 1, 1]) ? creep
```

```
true ;
   Redo: (11) mem(1, [1]) ? creep
   Call: (12) mem(1, []) ? creep
   Fail: (12) mem(1, []) ? creep
   Fail: (11) mem(1, [1]) ? creep
   Fail: (10) mem(1, [1, 1]) ? creep
   Fail: (9) mem(1, [3, 1, 1]) ? creep
   Fail: (8) mem(1, [2, 3, 1, 1]) ? creep
false.

[trace]  ?- nodebug.
true.
```

Backtracking example

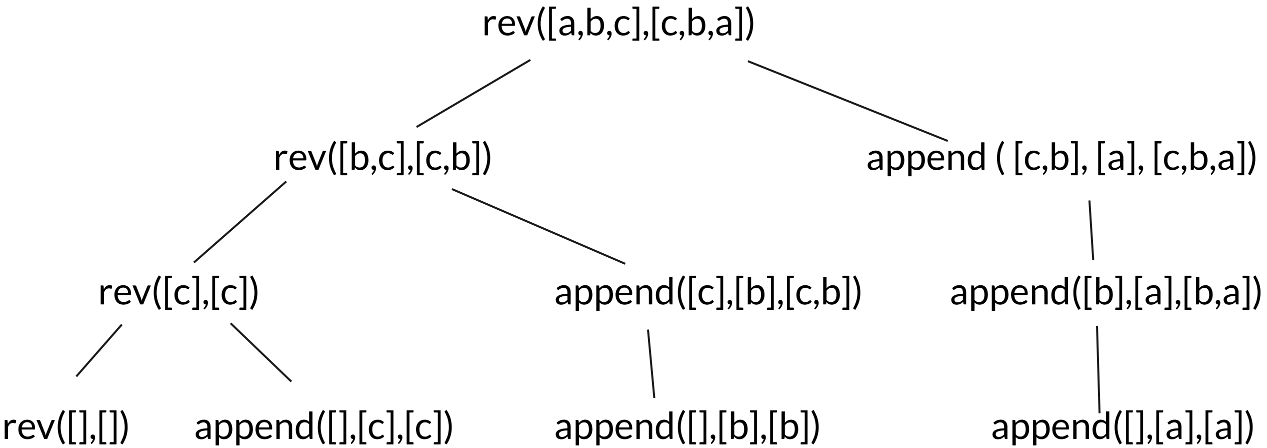
```
/* Define p */
p(a).
p(X) :- q(X), r(X).
p(X) :- u(X).

/* Define q */
q(X) :- s(X).

/* Define r */
r(a).
r(b).

/* Define s */
s(a).
s(b).
s(c).

/* Define u */
u(d).
```

```
?- p(X).
X = a ;
X = a ;
X = b ;
X = d.
```



Backtracking:

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




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

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

The first fact represents the base case, which states that appending an empty list to any list gives that same list. The second fact represents the recursive case, which states that appending a non-empty list [H|T] to another list X produces a new list [H|Z] where Z is the result of appending T to X. This recursive call to append continues until the first list is empty, at which point the base case is triggered and the result is returned.

Cut operator example:

A way to stop backtracking

Consider a new version of mem with cut operator

```
|: mem_noback(X,[X|_]):- !.
|: mem_noback(X,[_|T]):-mem_noback(X, T).
```

By using these rules, we only consider the first fact that satisfy the goal:

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

```
[trace]  ?- mem_noback(1,[2,3,1,1]).
   Call: (8) mem_noback(1, [2, 3, 1, 1]) ? creep
   Call: (9) mem_noback(1, [3, 1, 1]) ? creep
   Call: (10) mem_noback(1, [1, 1]) ? creep
   Exit: (10) mem_noback(1, [1, 1]) ? creep
   Exit: (9) mem_noback(1, [3, 1, 1]) ? creep
   Exit: (8) mem_noback(1, [2, 3, 1, 1]) ? creep
true.
```

Cut Operator:

Suppose we have the following facts

```
teaches(dr_fred, history).        studies(alice, english).
teaches(dr_fred, english).        studies(angus, english).
teaches(dr_fred, drama).          studies(amelia, drama).
teaches(dr_fiona, physics).       studies(alex, physics).
```

Then consider the following queries and their outputs:

```
?- teaches(dr_fred, Course), studies(Student, Course).
Course = english,
Student = alice ;
Course = english,
Student = angus ;
Course = drama,
Student = amelia.
```

```
teaches(dr_fred, history).          studies(alice, english).
teaches(dr_fred, english).          studies(angus, english).
teaches(dr_fred, drama).            studies(amelia, drama).
teaches(dr_fiona, physics).         studies(alex, physics).
```

Then consider the following queries with cut operator and their outputs:

```
?- teaches(dr_fred, Course), !, studies(Student, Course).
false.
```

This time Course is initially bound to history, then the cut goal is executed, and then studies goal is tried and fails (because nobody studies history). Because of the cut, we cannot backtrack to the teaches goal to find another binding for Course, so the whole query fails.

```
teaches(dr_fred, history).          studies(alice, english).
teaches(dr_fred, english).          studies(angus, english).
teaches(dr_fred, drama).            studies(amelia, drama).
teaches(dr_fiona, physics).         studies(alex, physics).
```

Then consider the following queries with cut operator in different location and their outputs:

```
?- teaches(dr_fred, Course), studies(Student, Course), !.
Course = english,
Student = alice.
```

Here the teaches goal is tried as usual, and Course is bound to history, again as usual. Next the studies goal is tried and fails, so we don't get to the cut at the end of the query at this point, and backtracking can occur. Thus the teaches goal is re-tried, and Course is bound to english. Then the studies goal is tried again, and succeeds, with Student = alice. After that, the cut goal is tried and of course succeeds, so no further backtracking is possible and only one solution is thus found.

Example:

Defining a parent relationship and finding a parent:

```
% facts

parent(john, sarah).

parent(john, jim).

parent(sue, sarah).

parent(sue, jim).

parent(jim, tom).
```

```
% rule
ancestor(X, Y) :-
  parent(X, Y).

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

Example:

Remove duplicates from the list

```
% Define the base case: the unique elements of an empty list is an empty list

unique([], []).

% Define the recursive case: the unique elements of a non-empty list is the unique elements of the tail of the list with the head of the list removed, if the head of the list is not already in the unique elements of the tail of the list

unique([H|T], [H|Rest]) :-
    not(member(H, T)),
    unique(T, Rest).

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

In this example, unique/2 is a predicate that takes a list L and returns a new list with all the duplicates removed. The first rule is the base case, which states that the unique elements of an empty list is an empty list. The second rule is the recursive case, which states that the unique elements of a non-empty list is the unique elements of the tail of the list with the head of the list removed, if the head of the list is not already in the unique elements of the tail of the list. The third rule is also a recursive case, which states that if the head of the list is already in the unique elements of the tail of the list, we do not include it in the result.