

Logic Programming – Part II

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

Agenda



1.

2.

3.

Today

- Recap
- Prolog
 - Lists
 - Search order and backtracking
 - Cut
 - Built-in predicates

Lectures

- No class on Thu April 25
- Next lecture on **Tue April 30**

ML Challenge

- Teams of 3 students
- Program in ML
- Possibly on Thu May 9 (to be confirmed)

Intermediate feedback form

Fill in the form at:

<https://forms.gle/i7mH13wRDv61etyN6>

- It will remain open for about one week

LET'S RECAP...

Recap

Different characteristics and languages

- Imperative (how to do?)
 - Classical: Fortran, Pascal, C
 - Object-oriented: Smalltalk, C++, Java
 - Scripting: Perl, Python, Javascript
- Declarative (what to do?)
 - Functional: ML, Ocaml
 - Logic: Prolog

```
int main(){  
    printf("Hello World");  
    return 0;  
}
```

```
public class HelloWorld{  
    public static void  
    main(String[] args) {  
        System.out.println("He  
llo World"); }}
```

```
print '“Hello, world!\n”'
```

```
output = program (input)
```

```
program (input, output)
```

Logic Programming Concepts

- The programmer **states a collection of axioms** from which theorems can be proven
- The programmer states a **goal**, and the language implementation attempts to find **a collection of axioms and inference steps** (including choices of values for variables) **that together imply the goal**
- Prolog is the most widely used such language

Horn clauses

- A **Horn clause** consists of a **head**, or **consequent term H**, and a **body** consisting of terms B_i
- We write
$$H :- B_1, B_2, \dots, B_n$$
- When the B_i are all true, we can deduce that H is true as well
- If $n=0$, the clause is said to be a **unit** or a **fact**
- In order to derive new statements, a logic programming system combines existing statements, through a process known as **resolution**
 - If we know that A and B imply C , and that C implies D , we can deduce that A and B imply D
 - During resolution, free variables may acquire values through **unification with expressions** in matching terms

```
C :- A, B;  
D :- C;  
-----  
D :- A, B
```

```
p(X) :- q(X);  
q(1);  
-----  
p(1)
```

Prolog

- A Prolog interpreter runs in the context of a **database of clauses** (Horn clauses) that are assumed to be true
- Each **clause** is composed of **terms**
- A **term** may be:
 - a **constant**: atom or number – an identifier starts with a lower-case letter
 - a **variable**: an identifier starting with an upper-case letter
 - a **structure**: a logical predicate (a functor and a list of arguments, e.g., `teaches(scott, cs254)`) or a data structure

Clauses: facts and rules

- The **clauses** in a Prolog database can be classified as
 - **facts** or
 - **rules**
- Both end with a **period**
- A **fact** is a Horn clause without a right-hand side, e.g., `rainy(rochester)` .
- A **rule** has a right-hand side, e.g., `snowy(X) :- rainy(X), cold(X)` .

Queries (or goals)

- **Goal**: the predicate we wish to prove to be true.
- Clause with an empty left-hand side
 - Queries do not appear in Prolog programs
 - Rather, one builds a database of facts and rules and then initiates execution by giving the Prolog interpreter (or the compiled Prolog program) a query to be answered (i.e., a goal to be proven)
 - E.g., given the facts

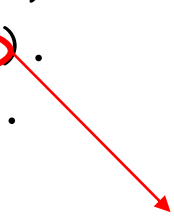
```
rainy(seattle).  
rainy(rochester).  
?- rainy(C).
```

Resolution

- The **resolution principle** (Robinson) says that if C1 and C2 are Horn clauses and the head of C1 matches one of the terms in the body of C2, then we can replace the term in C2 with the body of C1

```
    takes(alice, his201).  
C1  takes(alice, cs254).  
    takes(bob, art302).  
    takes(bob, cs254).  
C2  classmates(X, Y) :- takes(X, Z), takes(Y, Z).
```

The body is empty



- If we let X be alice and Z be cs254, we can replace the first term on the right-hand side of the last clause with the (empty) body of the second clause, yielding the new rule

```
classmates(alice, Y) :- takes(Y, cs254).
```

Unification

- The pattern-matching process used to associate `X` with `alice` and `Z` with `cs254` is known as **unification**
- Variables that are given values as a result of unification are said to be **instantiated**
- Two terms unify
 - if they are **identical**
`?- takes(alice, cs254).` -> unifies directly with the fact
`takes(alice, cs254).`
 - they can be **made identical by substituting variables**
`?- takes(alice, X).` -> variable `X` is instantiated with `cs254`

Equality in Prolog

- Equality in Prolog is defined in terms of “unifiability”
- Examples:

```
?- a = a.  
true. % constant unifies with itself
```

```
?- a = b.  
false. % but not with another constant
```

```
?- foo(a, b) = foo(a, b).  
true.
```

```
?- X = a.  
X = a. % only one possibility
```

```
?- foo(a, b) = foo(X, b).  
X = a. % arguments must unify only one possibility
```

Arithmetic

- The usual arithmetic operators are available in Prolog, but **they play the role of predicates**, not of functions
- `?- (2 + 3) = 5.`
`false.`
- To handle arithmetic, Prolog provides a built-in predicate, **is**, that unifies its first argument with the arithmetic value of its second argument

```
?- is(X, 1+2).
```

```
X=3.
```

```
?- X is 1+2.
```

```
X = 3.
```

```
?- 1+2 is 4-1.
```

```
false.
```

```
?- X is Y.
```

```
<error> Arguments are not sufficiently instantiated
```

```
?- Y is 1+2, X is Y.
```

```
Y=X, Y = 3.
```


Function parameters and return value

- `increment(X,Y) :- Y is X+1.`

- `?-increment(1,Z).`

`Z=2.`

- `?-increment (1,2).`

`True.`

- `?-increment(Z,2).`

Arguments are not sufficiently instantiated.

X+1 cannot be evaluated
since X has not yet been
instantiated.

- `addN(X,N,Y):- Y is X+N.`

- `?- addN(1,2,Z)`

`Z = 3.`

Question 1

- Facts

```
mario(A,B,C,D):-  
    A=C,  
    B=D,  
    C=2+3.
```

- Query

```
?- mario(5,1,5,1).
```

A.true.
B.false.

Answer question 1

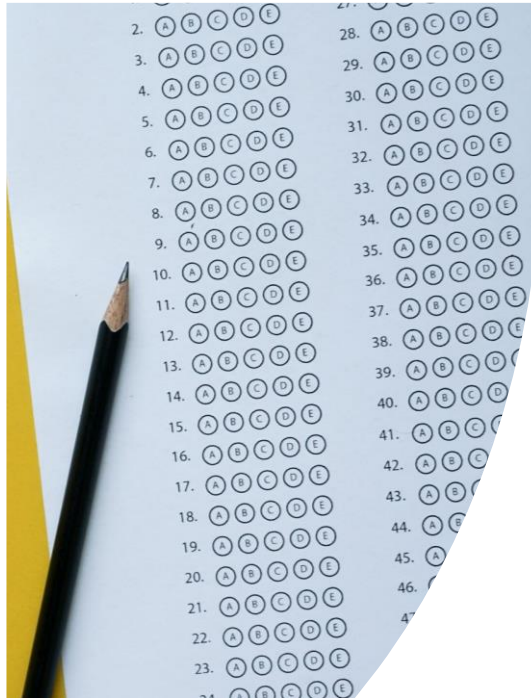
- Facts

```
mario(A,B,C,D):-  
    A=C,  
    B=D,  
    C=2+3.
```

- Query

```
?- mario(5,1,5,1).
```

A. true.
B. false.



Lists

Lists

- Example: `[a, c, 2, 'hi', [W, 3]]`
- The construct `[a, b, c]` is shorthand for the **compound** structure `.(a, .(b, .(c, [])))`, where `[]` is an **atom** (the empty list) and `.` is a built-in cons-like predicate.
- How does it work matching?

`?- [X, 1, Z] = [a, _, 17]`

`X=a,`

`Z=17`

Vertical bar notation

- Prolog adds an extra convenience: an optional vertical bar that delimits the tail of the list
- Using this notation, $[a, b, c]$ can be expressed as $[a \mid [b, c]]$, $[a, b \mid [c]]$, or $[a, b, c \mid []]$
- $[H \mid T]$ is syntactically similar to ML $h :: t$
 $?-[Head \mid Tail] = [a, b, c].$
 $Head = a.$
 $Tail = [b, c].$
- The vertical bar notation is particularly useful when the tail of the list is a variable

Examples

- $?-[X, Y, Z] = [1, 2, 3]$.
 $X=1$.
 $Y=2$.
 $Z=3$.
- $?-[1, 2, 3, 4] = [_ , X | _]$.
 $X = 2$.
- $?-[1, 2 | X] = [1, 2, 3, 4, 5]$.
 $X = [3, 4, 5]$.

Defining more complex predicates: member and sorted

```
member(X, [X|T]).
```

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

```
sorted([]). % empty list is sorted
```

```
sorted([X]). % singleton is sorted
```

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

```
% compound list is sorted if first two elements  
are in order and the remainder of the list  
(after first element) is sorted
```

- Here `=<` is a built-in predicate that operates on numbers

append (or concatenate)

- `append(L1, L2, L3)` succeeds when `L3` unifies with `L2` appended at the end of `L1`, that is `L3` is the concatenation of `L1` and `L2`.

- Given this definition:

```
append([], L2, L2). /*if L1 is empty,  
then L3 = L2 */
```

```
append([H | L1], L2, [H | L3]) :-  
append(L1, L2, L3) /*prepending a new  
element to L1, means prepending it to L3  
as well*/
```

Examples

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

Question 2

- Query

```
?- [3,4,'papaya',blueberry].
```

A.true.
B.false.

Answer question 2

- Query

```
?- [3,4,'papaya',blueberry].
```

A. true.
B. false.

Question 3

- Query

```
?- [a|T]=[a,b,c,[d,e],[3,4],list].
```

A. `T=[b,c,[d,e],[3,4],list].`

B. `false.`

C. `T=[d,a].`

D. `T=list.`

Answer question 3

- Query

```
?- [a|T]=[a,b,c,[d,e],[3,4],list].
```

A. $T=[b,c,[d,e],[3,4],list].$

B. $false.$

C. $T=[d,a].$

D. $T=list.$

Question 4

- What is the result of `unknown (A,L)`?

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

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

- A. Evaluates to false if A is contained in list L.
- B. Evaluates to true if A is contained in list L.
- C. Assigns the last element in L to A.
- D. Assigns one element in L to A.

Answer question 4

- What is the result of `unknown (A,L)`?

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

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

- A. Evaluates to false if A is contained in list L.
- B. Evaluates to true if A is contained in list L.**
- C. Assigns the last element in L to A.
- D. Assigns one element in L to A.

Question 5

- What is the result of `unknown (A,B)`?

```
unknown (L1,L2) :-  
    L1 = [H|T1],  
    L2 = [H,H|T2] .
```

- A. true if A and B have equal lengths
- B. true if the first element in A is equal to the first and the last element in B.
- C. true if the first element in A is equal to the first and the second element in B.
- D. true if the first element in A is equal to the last element in B.

Answer question 5

- What is the result of `unknown (A,B)`?

```
unknown (L1,L2) :-  
    L1 = [H|T1],  
    L2 = [H,H|T2] .
```

- A. true if A and B have equal lengths
- B. true if the first element in A is equal to the first and the last element in B.
- C. true if the first element in A is equal to the first and the second element in B.
- D. true if the first element in A is equal to the last element in B.

Some built-in predicates

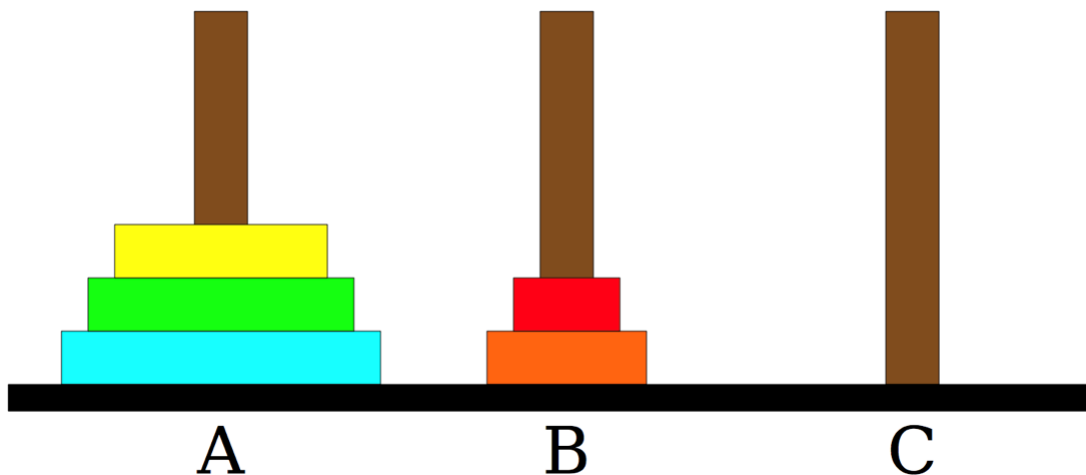
- `length(List,Length)`
 `?- length([a, b, [1,2,3]], Length).`
 `Length = 3.`
- `member(Elem,List)`
 `?- member(duey, [huey, duey, luey]).`
 `true.`
 `?- member(X, [huey, duey, luey]).`
 `X = huey; X = duey; X = luey.`
- `append(List1,List2,Result)`
 `?- append([duey], [huey, duey, luey], X).`
 `X = [duey, huey, duey, luey].`

Some built-in predicates

- `sort(List, SortedList)`
 ?- `sort([2,1,3], R).`
 R= `[1,2,3].`
- `findall(Elem, Predicate, ResultList)`
 ?- `findall(E, member(E, [huey, duey, luey]), R).`
 R= `[huey, duey, luey].`
- See documentation for more
 <http://www.swi-prolog.org/pldoc/man?section=builtin>

Example – Towers of Hanoi

- Problem
 - Move stack of disks between pegs
 - Can only move top disk in stack
 - Only allowed to place disk on top of larger disk



Example – Towers of Hanoi

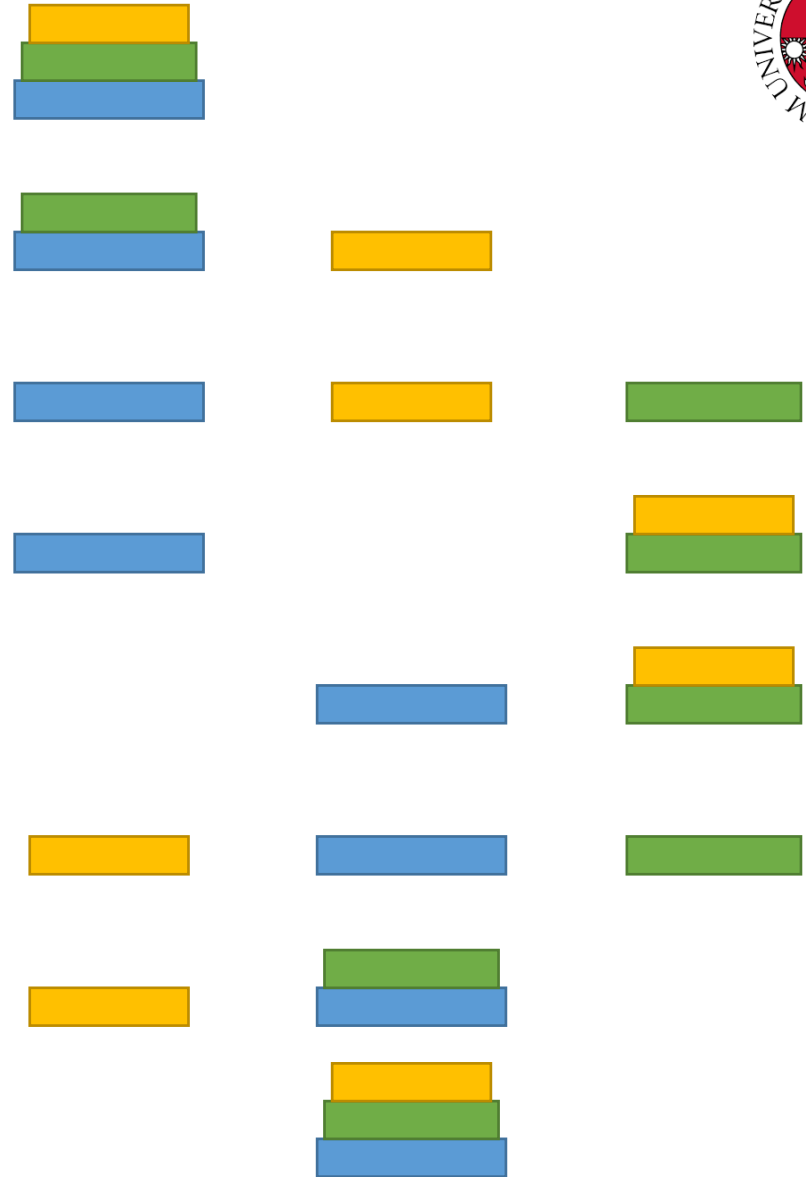
- How to move a stack of n disks from peg A to B?
- Base case
 - If $n = 1$, move disk from A to B
- Recursive step
 1. Move top $n-1$ disks from A to 3rd peg (C)
 2. Move bottom disk from A to B
 3. Move top $n-1$ disks from 3rd peg (C) to B

Towers of Hanoi

```
hanoi(N):-move(N,left,central,right).  
move(1,X,Y,_) :-  
    write('Move top disk from '),  
    write(X), write(' to '), write(Y), nl.  
move(N,X,Y,Z) :- N>1, M is N-1,  
    move(M,X,Z,Y), move(1,X,Y,_,  
    move(M,Z,Y,X).
```

Towers of Hanoi

```
?- hanoi(3).  
Move top disk from left to central  
Move top disk from left to right  
Move top disk from central to right  
Move top disk from left to central  
Move top disk from right to left  
Move top disk from right to central  
Move top disk from left to central  
true
```





Search order and backtracking

Prolog terminology

- A goal or term where variables do not occur is called **ground**; else it is called **nonground**
 - `foo(a,b)` is ground;
 - `bar(X)` is nonground
- When querying terms with variables the interpreter instantiates the free variables with terms so that the ground predicated holds.

Prolog terminology

- A **substitution** θ is a partial map from variables to terms where $domain(\theta) \cap range(\theta) = \emptyset$
 - Variables are terms, so a substitution can map variables to other variables, but not to themselves
- Unifying two terms s and t means finding a substitution θ over their free variables.
- Two terms unify
 - if they are **identical**
`?- takes(alice, cs254).` -> unifies directly with the fact
 - they can be **made identical by substituting variables**
`?- takes(alice, X).` -> variable X is instantiated with `cs254`
- Variables that are given values as a result of unification are said to be **instantiated**, that is A is an instance of B if there is a substitution such that $A = B \theta$

Question 6

- Which of these are ground terms?

```
cat(tom)  
mouse(jerry)  
dog(X)
```

Question 6

- Which of these are ground terms?

```
cat(tom)
mouse(jerry)
dog(X)
```

ground
ground
not ground

Search/execution order

- How does Prolog answer a query?
- It needs a sequence of resolution steps that will build the goal out of clauses in the database, or a proof that no such sequence exists
- In formal logic, there are two principal search strategies:
 - Start with existing clauses and work forward, attempting to derive the goal. This strategy is known as **forward chaining**.
 - Start with the goal and work backward, attempting to “unresolve” it into a set of preexisting clauses. This strategy is known as **backward chaining**.

What is the best strategy?

- If the number of existing rules is very large, but the number of facts is small, it is possible for forward chaining to discover a solution faster than backward chaining
- In most circumstances, however, backward chaining turns out to be more efficient
- Prolog is defined to use **backward chaining**.

Search order

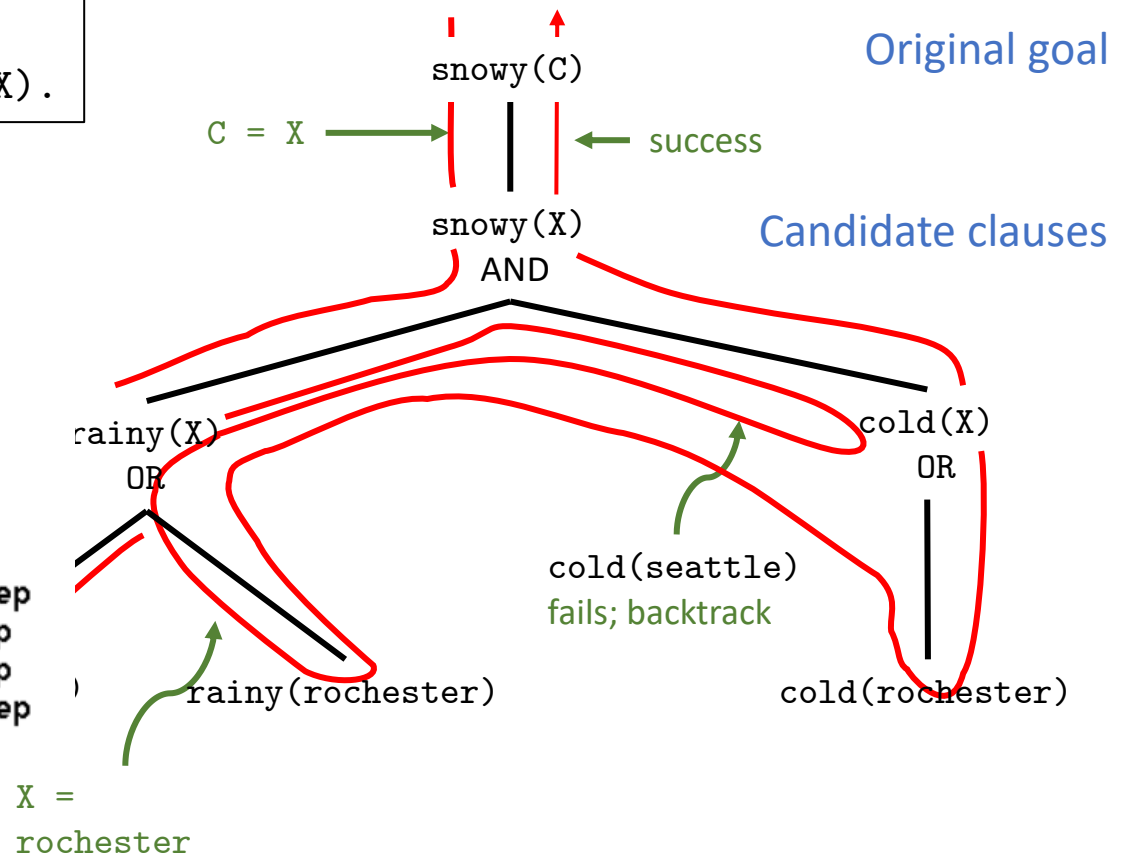
- In Prolog we unify terms on the right hand side with the heads of other clauses by obtaining a **tree of subgoals**.
- Since resolution is associative and commutative, a backward-chaining theorem prover can limit its search to sequences of resolutions in which terms on the right-hand side of a clause are unified with the heads of other clauses one by one in some particular order.
- How does Prolog explore the tree of subgoals?
- **Depth-first and left to right.**

Search order

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

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

```
[trace] ?- snowy(C).
Call: (10) snowy(_18738) ? creep
Call: (11) rainy(_18738) ? creep
Exit: (11) rainy(seattle) ? creep
Call: (11) cold(seattle) ? creep
Fail: (11) cold(seattle) ? creep
Redo: (11) rainy(_18738) ? creep
Exit: (11) rainy(rochester) ? creep
Call: (11) cold(rochester) ? creep
Exit: (11) cold(rochester) ? creep
Exit: (10) snowy(rochester) ? creep
C = rochester.
```



Evaluation

- The Prolog interpreter explores this tree depth first, from left to right
- It starts at the beginning of the database, searching for a rule R whose head can be unified with the top-level goal
- It then considers the terms in the body of R as subgoals, and attempts to satisfy them, recursively, left to right
- If at any point **a subgoal fails** (cannot be satisfied), the interpreter returns to the previous subgoal and attempts to satisfy it in a different way (i.e., try to unify it with the head of a different clause).

Evaluation

- The process of returning to previous goals is known as **backtracking**
- Whenever an unification operation is “undone” in order to pursue a different path through the search tree, variables that were given values or associated with one another as a result of that unification are returned to their uninstantiated or unassociated state
- In the example, the binding of `X` to `seattle` is broken when we backtrack to the `rainy(X)` subgoal
- The effect is similar to the breaking of bindings between actual and formal parameters in an imperative language, except that Prolog describes the bindings in terms of unification rather than subroutine calls

Backtracking

- Space management for backtracking search in Prolog usually uses a single **stack** (both for the execution and for choice backtracking)
- The interpreter pushes a frame into its stack every time it begins to pursue a new subgoal G
- If G **fails**, the frame is popped from the stack and the interpreter begins to backtrack
- If G **succeeds**, control returns to the “caller”, i.e., the parent in the search tree

Backtracking

- G will not fail unless all of its subgoals, and all of its siblings to the right in the search tree, have also failed, implying that there is nothing above G's frame in the stack
- At the top level of the interpreter, a semicolon typed by the user is treated the same as failure of the most recently satisfied subgoal.

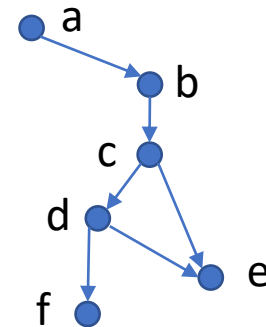
Backtracking

- The fact that clauses are ordered, and that the interpreter considers them from first to last, means that the results of a Prolog program are **deterministic and predictable**
- The combination of ordering and depth-first search means that the Prolog programmer must often consider the order to ensure that recursive programs terminate

Example

- Suppose the database describes a directed acyclic graph:

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



```
path1(X, X).
```

```
path1(X, Y) :- edge(X, Z), path1(Z, Y)
```

These tell us how to determine whether there is a path from X to Y

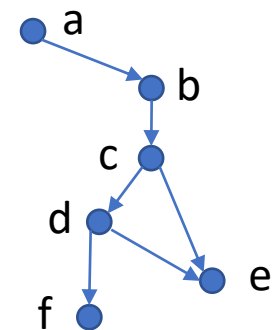
- If we query for `?path1(a,d)`.

```
?- path1(a,d).
true ;
false.
```

Backtracking path1

```
[trace]  ?- path1(d,B).
  Call: (10) path1(d, _42780) ? creep
  Exit: (10) path1(d, d) ? creep
B = d ;
  Redo: (10) path1(d, _42780) ? creep
  Call: (11) edge(d, _47212) ? creep
  Exit: (11) edge(d, e) ? creep
  Call: (11) path1(e, _42780) ? creep
  Exit: (11) path1(e, e) ? creep
  Exit: (10) path1(d, e) ? creep
B = e ;
  Redo: (11) path1(e, _42780) ? creep
  Call: (12) edge(e, _53592) ? creep
  Fail: (12) edge(e, _53592) ? creep
  Fail: (11) path1(e, _42780) ? creep
  Redo: (11) edge(d, _47212) ? creep
  Exit: (11) edge(d, f) ? creep
  Call: (11) path1(f, _42780) ? creep
  Exit: (11) path1(f, f) ? creep
  Exit: (10) path1(d, f) ? creep
B = f ;
  Redo: (11) path1(f, _42780) ? creep
  Call: (12) edge(f, _62402) ? creep
  Fail: (12) edge(f, _62402) ? creep
  Fail: (11) path1(f, _42780) ? creep
  Fail: (10) path1(d, _42780) ? creep
false.
```

```
edge(a, b).
edge(b, c).
edge(c, d).
edge(d, e).
edge(b, e).
edge(d, f).
path1(X, X).
path1(X, Y) :- edge(X, Z), path1(Z, Y)
```



Backtracking

- If we reverse the order of the terms on the right-hand side of the final clause, then Prolog will search for a node Z that is reachable from X before checking to see whether there is an edge from Z to Y

```
path2(X, X).
```

```
path2(X, Y) :- path2(X, Z), edge(Z, Y).
```

- The program will work, but it will be less efficient and won't stop if you continue asking paths

```
?- path2(a,d).
true ;
ERROR: Stack limit (1.0Gb) exceeded
ERROR: Stack sizes: local: 0.9Gb, global: 84.2Mb, trail: 0Kb
ERROR: Stack depth: 11,027,909, last-call: 0%, Choice points: 4
ERROR: Probable infinite recursion (cycle):
ERROR: [11,027,908] user:path2(_22065580, d)
ERROR: [11,027,907] user:path2(_22065600, d)
```

- If we also reverse the order of the last two clauses:

```
path3(X, Y) :- path3(X, Z), edge(Z, Y).
```

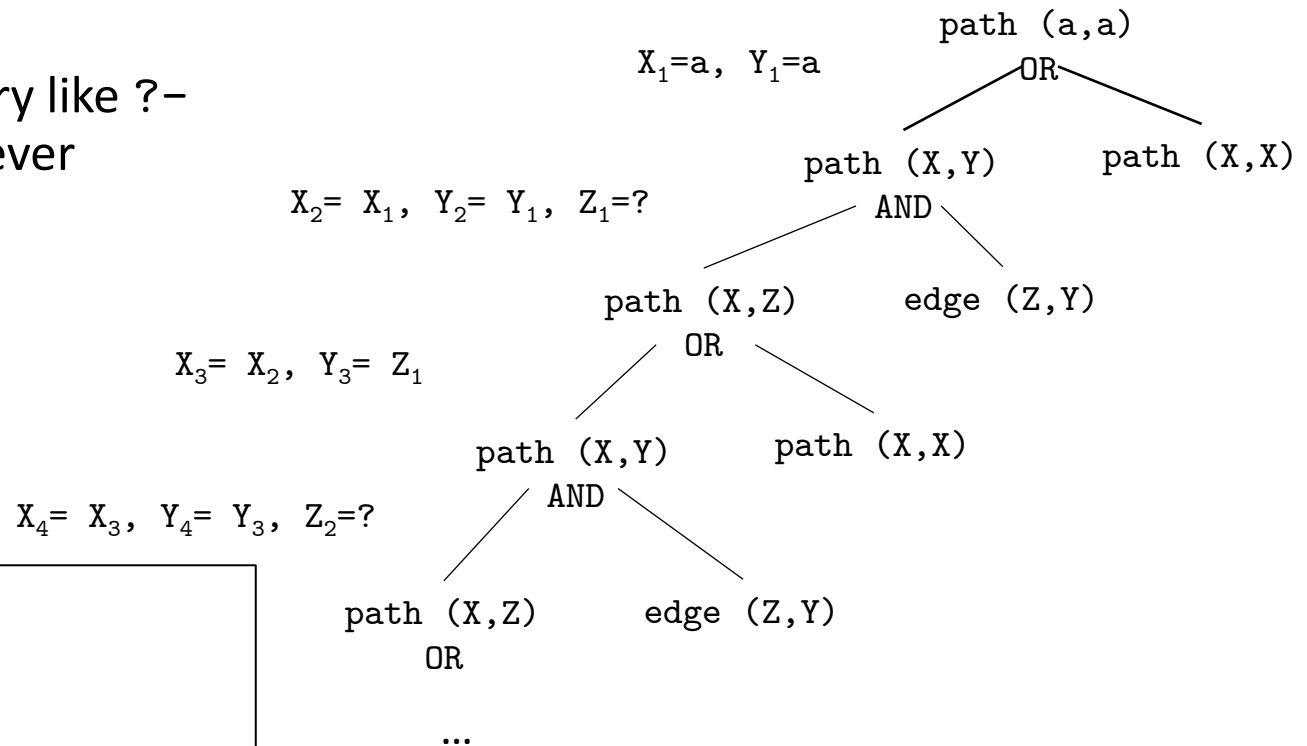
```
path3(X,X).
```

logically, they are the same, but Prolog will no longer be able to find answers

```
?- path3(a,d).
ERROR: Stack limit (1.0Gb) exceeded
ERROR: Stack sizes: local: 0.9Gb, global: 48.4Mb, trail: 0Kb
ERROR: Stack depth: 6,339,224, last-call: 0%, Choice points: 6,339,217
ERROR: In:
ERROR: [6,339,224] user:path3(_12679480, d)
ERROR: [6,339,223] user:path3(_12679500, d)
ERROR: [6,339,222] user:path3(_12679520, d)
ERROR: [6,339,221] user:path3(_12679540, d)
ERROR: [6,339,220] user:path3(_12679560, d)
ERROR: Use the --stack_limit=size[KMG] command line option or
ERROR: ?- set_prolog_flag(stack_limit, 2_147_483_648). to double the limit.
Exception: (6,339,223) path3(_12679394, d) ?
```

Non-termination

Even a simple query like ?-
path(a, a) will never
terminate



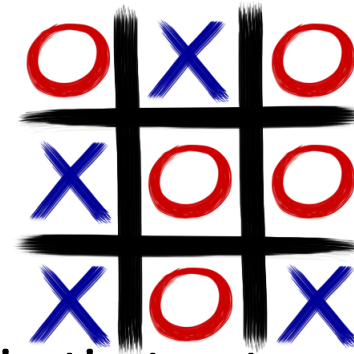
```

edge(a, b).
edge(b, c).
edge(c, d).
edge(d, e).
edge(b, e).
edge(d, f).
path(X, Y) :- path(X, Z),
               edge(Z, Y).
path(X, X).
    
```

Non-termination

- The interpreter first unifies `path(a, a)` with the left-hand side of
$$\text{path}(X, Y) \text{ :- path}(X, Z), \text{ edge}(Z, Y).$$
- It then considers the goals on the right-hand side, the first of which, `path(X,Z)`, unifies with the left-hand side of the very same rule, leading to an infinite loop
- Prolog gets lost in an infinite branch of the search tree, and never discovers the finite branches to the right
- We could avoid this problem by exploring the tree in breadth-first order, but that strategy was rejected by Prolog's designers because of its expense

Example: Tic-Tac-Toe



- Consider the problem of making a move in tic-tac-toe
- Number the squares from 1 to 9 in row-major order
- We use the Prolog fact $x(n)$ to indicate that player X has placed a marker in square n , and $o(m)$ to indicate that player O has placed a marker in square m
- Assume that the computer is player X , and that it is X 's turn to move
- We want to issue a query $?-move(A)$ that will cause Prolog to choose a good square for the computer to occupy next.

Tic-Tac-Toe

- We need to be able to tell whether three given squares lie in a row.

- This can be expressed as

```
ordered_line(1, 2, 3).
```

```
ordered_line(7, 8, 9).
```

```
ordered_line(2, 5, 8).
```

```
ordered_line(1, 5, 9).
```

```
...
```

```
line(A, B, C) :- ordered_line(A, B, C).
```

```
line(A, B, C) :- ordered_line(A, C, B).
```

```
line(A, B, C) :- ordered_line(B, A, C).
```

```
line(A, B, C) :- ordered_line(B, C, A).
```

```
line(A, B, C) :- ordered_line(C, A, B).
```

```
line(A, B, C) :- ordered_line(C, B, A).
```

- There is no winning strategy for the game

Tic-tac-toe

- Assume that our program is playing against a less-than-perfect opponent.
- Our task is never to lose, and to maximize our chances of winning if our opponent makes a mistake

```
move(A) :- good(A), empty(A).
```

We can satisfy `move(A)` by choosing a good and empty square

```
full(A) :- x(A).
```

```
full(A) :- o(A).
```

```
empty(A) :- not(full(A)).
```

Square `n` is empty if we cannot prove that it is full, that is neither `x(n)` nor `o(n)` are in the db

```
% strategy:
```

```
good(A) :- win(A).
```

```
good(A) :- split(A).
```

```
good(A) :- weak_build(A).
```

```
good(A) :- block_win(A).
```

```
good(A) :- strong_build(A).
```

These are our strategies in order of application

Strategy

```
win(A) :- x(B), x(C), line(A, B, C).
```

```
block_win(A) :- o(B), o(C), line(A, B, C).
```

```
split(A) :- x(B), x(C), different(B, C),
line(A, B, D), line(A, C, E), empty(D), empty(E).
same(A, A).
different(A, B) :- not(same(A, B)).
```

The first choice is to win, i.e., completing a line!

The second is preventing our opponent from winning

The third is creating a split, that is a situation in which our opponent cannot prevent us from winning on the next move

X	X	
1	2	3
X	O	O
4	5	6
7	8	9

Strategy

```
win(A) :- x(B), x(C), line(A, B, C).
```

```
block_win(A) :- o(B), o(C), line(A, B, C).
```

```
split(A) :- x(B), x(C), different(B, C),  
line(A, B, D), line(A, C, E), empty(D), empty(E).  
same(A, A).  
different(A, B) :- not(same(A, B)).
```

```
strong_build(A) :- x(B), line(A, B, C), empty(C),  
not(risky(C)).  
risky(C) :- o(D), line(C, D, E), empty(E).
```

```
weak_build(A) :- x(B), line(A, B, C), empty(C),  
not(double_risky(C)).  
double_risky(C) :- o(D), o(E), different(D, E),  
line(C, D, F),  
line(C, E, G), empty(F), empty(G).
```

The first choice is to win, i.e., completing a line!

The second is preventing our opponent from winning

The third is creating a split, that is a situation in which our opponent cannot prevent us from winning on the next move

The fourth is moving toward three in a row, so that the obvious blocking won't allow our opponent to build toward three in a row

The fifth is moving toward three in a row, so that the obvious blocking won't allow our opponent to create a split

Last part of the strategy

- If none of these goals can be satisfied, our final, default choice is to pick an unoccupied square, giving priority to the center, the corners, and the sides in that order:

good(5) .

good(1) .

good(3) .

good(7) .

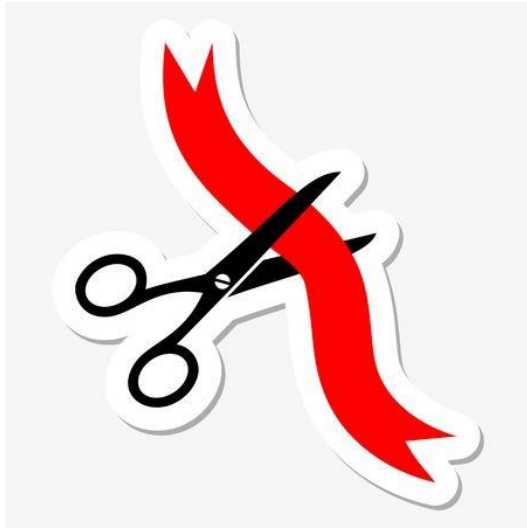
good(9) .

good(2) .

good(4) .

good(6) .

good(8) .



Cut

Imperative control flow

- The ordering of clauses and of terms in Prolog is significant, with consequences for **efficiency**, **termination**, and **choice among alternatives**
- Besides simple ordering, Prolog provides the programmer with several explicit control flow features
- The most important is known as the **cut**
- The cut is a zero-argument predicate written as an exclamation point: **!**

Cut

- It allows the programmer to eliminate some of the possible alternatives produced during evaluation, so as to improve efficiency
- As a subgoal it always succeeds, but with the **crucial side effect** that it commits the interpreter to whatever choices have been made since unifying the parent goal with the left-hand side of the current rule, including the choice of that unification itself.

Why do we need the cut predicate?

- Example

```
member(X, [X|T]).
```

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

- If a given atom *a* appears in list *L* *n* times, what happens if we ask multiple solutions (;) for the goal

```
?- member(a,L).
```

- It will succeed *n* times

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

Why do we need the cut predicate?

- These extra successes can lead to wasted computation, particularly for long lists, when `member` is followed by a goal that may fail and is expensive to compute:

```
prime_candidate(X) :- member(X, candidates), prime(X).
```

- To determine whether *a* is a prime candidate, we first check to see whether it is a member of the candidates list, and then check to see whether it is prime
- If `prime(a)` fails, Prolog backtracks and attempts to satisfy `member(a, candidates)` again
- If *a* is in the `candidates` list more than once, then the subgoal will succeed again, leading to reconsideration of the `prime(a)` subgoal, even though that subgoal will fail

The cut

- We can save time by cutting off all further searches for a after the first is found

```
member1(X, [X|T]) :- !.
```

```
member1(X, [_|T]) :- member1(X, T)
```

- The cut on the right-hand side of the first rule says that if X is the head of L, we should not attempt to unify member(X,L) with the left-hand side of the second rule; the cut commits us to the first rule

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

Example of the cut predicated

Without Cut

```
[trace] ?- member2(a,[a,a,b,c,a]).
  Call: (10) member2(a, [a, a, b, c, a]) ? creep
  Exit: (10) member2(a, [a, a, b, c, a]) ? creep
true ;
  Redo: (10) member2(a, [a, a, b, c, a]) ? creep
  Call: (11) member2(a, [a, b, c, a]) ? creep
  Exit: (11) member2(a, [a, b, c, a]) ? creep
  Exit: (10) member2(a, [a, a, b, c, a]) ? creep
true ;
  Redo: (11) member2(a, [a, b, c, a]) ? creep
  Call: (12) member2(a, [b, c, a]) ? creep
  Call: (13) member2(a, [c, a]) ? creep
  Call: (14) member2(a, [a]) ? creep
  Exit: (14) member2(a, [a]) ? creep
  Exit: (13) member2(a, [c, a]) ? creep
  Exit: (12) member2(a, [b, c, a]) ? creep
  Exit: (11) member2(a, [a, b, c, a]) ? creep
  Exit: (10) member2(a, [a, a, b, c, a]) ? creep
true ;
  Redo: (14) member2(a, [a]) ? creep
  Call: (15) member2(a, []) ? creep
  Fail: (15) member2(a, []) ? creep
  Fail: (14) member2(a, [a]) ? creep
  Fail: (13) member2(a, [c, a]) ? creep
  Fail: (12) member2(a, [b, c, a]) ? creep
  Fail: (11) member2(a, [a, b, c, a]) ? creep
  Fail: (10) member2(a, [a, a, b, c, a]) ? creep
false.
```

With Cut

```
[trace] ?- member(a,[a,a,b,c,a]).
  Call: (10) member(a, [a, a, b, c, a]) ? creep
  Exit: (10) member(a, [a, a, b, c, a]) ? creep
true.
```


The cut: other examples

- Similarly, if we have

```
minimum2(X, Y, X) :- X < Y, !.
```

```
minimum2(X, Y, Y) :- X > Y.
```

the cut expresses the fact that once the first clause has been used, there is no need to consider the second, that would instead result in:

```
[trace] ?- minimum2(3,5,X).
  Call: (10) minimum2(3, 5, _10852) ? creep
  Call: (11) 3<5 ? creep
  Exit: (11) 3<5 ? creep
  Exit: (10) minimum2(3, 5, 3) ? creep
X = 3.
```

```
[trace] ?- minimum1(3,5,X).
  Call: (10) minimum1(3, 5, _22218) ? creep
  Call: (11) 3<5 ? creep
  Exit: (11) 3<5 ? creep
  Exit: (10) minimum1(3, 5, 3) ? creep
X = 3 ;
  Redo: (10) minimum1(3, 5, _22218) ? creep
  Call: (11) 3>5 ? creep
  Fail: (11) 3>5 ? creep
  Fail: (10) minimum1(3, 5, _22218) ? creep
false.
```

The cut: other examples

- Similarly for `fact(X, Y)`

- Using cut

```
fact1(0,1):-!.
```

```
fact1(N,X):-M is N-1,fact1(M,Y),X is Y*N.
```

```
?- fact1(3,X).  
X = 6.
```

- Without cut

```
fact2(0,1).
```

```
fact2(N,X):- M is N-1,fact2(M,Y),X is Y*N.
```

```
?- fact2(3,X).  
X = 6 ;  
ERROR: Stack limit (1.0Gb) exceeded
```

The cut

- In general, if we have n clauses to define the predicate p

$p(S1) :- A1.$

...

$p(Sk) :- B, !, C.$

...

$p(Sn) :- An.$

- If we find the k th clause in the list being used, we have the following cases:
 1. If an evaluation of B fails, then we proceed by trying the $k + 1$ st clause.
 2. If the evaluation of B succeeds, then $!$ is evaluated. It succeeds and the evaluation proceeds with C . In the case of backtracking, however, **all the alternative ways of computing B are eliminated**, as all the alternatives provided by the clauses from the k th to the n th to compute $p(t)$.

The negation

- An alternative way to ensure that `member(X,L)` succeeds no more than once is to embed a use of `not` in the second clause

```
member3(X, [X|T]).
```

```
member3(X, [H|T]) :- not(X = H), member(X, T).
```

- This code will do the same, but is slightly less efficient, as Prolog will actually consider the second rule, abandoning it only after (re)unifying `X` with `H` and reversing the sense of the test

```
[trace] ?- member3(a,[a,a,b,c,a]).
  Call: (10) member3(a, [a, a, b, c, a]) ? creep
  Exit: (10) member3(a, [a, a, b, c, a]) ? creep
true ;
  Redo: (10) member3(a, [a, a, b, c, a]) ? creep
^  Call: (11) not(a=a) ? creep
^  Fail: (11) not(user:(a=a)) ? creep
  Fail: (10) member3(a, [a, a, b, c, a]) ? creep
false.
```

```
[trace] ?- member(a,[a,a,b,c,a]).
  Call: (10) member(a, [a, a, b, c, a]) ? creep
  Exit: (10) member(a, [a, a, b, c, a]) ? creep
true.
```

Usage of negation

```
direct_flight(bologna, paris).
direct_flight(bologna, amsterdam).
direct_flight(paris, bombay).
direct_flight(amsterdam, moscow).

flight(X, Y):- direct_flight(X, Y).
flight(X, Y):- direct_flight(X, Z),
               flight(Z, Y).
indirect_flight(X, Y) :-
    flight(X, Y),
    not(direct_flight(X, Y)).
```

```
?- flight(bologna, bombay).
true
?- direct_flight(bologna,
moscow)
false
?- flight(bologna, X)
X = paris
• How to express that it does not
  exist a direct flight?
?- indirect_flight(bologna,
bombay)
true
?- indirect_flight(bologna,
paris)
false
```

How does negation work?

```
[trace] ?- indirect_flight(bologna,moscow).
  Call: (10) indirect_flight(bologna, moscow) ? creep
  Call: (11) flight(bologna, moscow) ? creep
  Call: (12) direct_flight(bologna, moscow) ? creep
  Fail: (12) direct_flight(bologna, moscow) ? creep
  Redo: (11) flight(bologna, moscow) ? creep
  Call: (12) direct_flight(bologna, _15512) ? creep
  Exit: (12) direct_flight(bologna, paris) ? creep
  Call: (12) flight(paris, moscow) ? creep
  Call: (13) direct_flight(paris, moscow) ? creep
  Fail: (13) direct_flight(paris, moscow) ? creep
  Redo: (12) flight(paris, moscow) ? creep
  Call: (13) direct_flight(paris, _20374) ? creep
  Exit: (13) direct_flight(paris, bombay) ? creep
  Call: (13) flight(bombay, moscow) ? creep
  Call: (14) direct_flight(bombay, moscow) ? creep
  Fail: (14) direct_flight(bombay, moscow) ? creep
  Redo: (13) flight(bombay, moscow) ? creep
  Call: (14) direct_flight(bombay, _25236) ? creep
  Fail: (14) direct_flight(bombay, _25236) ? creep
  Fail: (13) flight(bombay, moscow) ? creep
  Fail: (12) flight(paris, moscow) ? creep
  Redo: (12) direct_flight(bologna, _15512) ? creep
  Exit: (12) direct_flight(bologna, amsterdam) ? creep
  Call: (12) flight(amsterdam, moscow) ? creep
  Call: (13) direct_flight(amsterdam, moscow) ? creep
  Exit: (13) direct_flight(amsterdam, moscow) ? creep
  Exit: (12) flight(amsterdam, moscow) ? creep
  Exit: (11) flight(bologna, moscow) ? creep
^ Call: (11) not(direct_flight(bologna, moscow)) ? creep
  Call: (12) direct_flight(bologna, moscow) ? creep
  Fail: (12) direct_flight(bologna, moscow) ? creep
^ Exit: (11) not(user:direct_flight(bologna, moscow)) ? creep
  Exit: (10) indirect_flight(bologna, moscow) ? creep
true ■
```

- To evaluate `not (G)`, it evaluates `G`.
- If the evaluation of `G` terminates (possibly after backtracking) with failure, then the goal `not (G)` succeeds.
- If `G` has a computation that terminates with success, then that of `not (G)` fails
- If the evaluation of `G` does not terminate, then `not (G)` fails to terminate (negation as failure).

The negation

- `not` is implemented by a combination of the `cut` and two other built-in predicates, `call` and `fail`

```
not(P) :- call(P), !, fail.
```

```
not(P).
```

Call and fail

- The **call** predicate takes a term as argument and attempts to satisfy it as a goal
- The **fail** predicate always fails
- In principle, we could replace uses of the cut with uses of `not`, using the cut only in the implementation of `not`
- Doing so it often makes a program easier to read, but also makes it **less efficient**
- Explicit use of the cut may actually make a program more difficult to read

If-then-else effect

- $B \rightarrow C1;C2.$
- In general, the cut can be used whenever we want the effect of **if-then-else**

statement $:-$ condition, !, then_part.

statement $:-$ else_part.

Question 7

- What does this query return?

```
s(c).  
s(m).  
s(d).  
solve(X):-s(X),!.  
solve(other_solution).  
?- solve(X).
```

- A. $X = c; X = m; X = d; X = \text{other_solution}.$
- B. $X = c$
- C. $X = c; X = m; X = d;$
- D. true

Answer question 7

- What does this query return?

```
s(c) .  
s(m) .  
s(d) .  
solve(X):-s(X),!.  
solve(other_solution).  
?- solve(X).
```

A. $X = c$; $X = m$; $X = d$; $X = \text{other_solution}$.

B. $X = c$.

C. $X = c$; $X = m$; $X = d$;

D. true

Question 8

- What does this query return?

```
check(_, []) :- !.  
check(E, [H|T]) :- E > H, check(E, T).  
?- check(10, [4, 3, 2]).
```

- A. false.
- B. true; false.
- C. true.
- D. false; true.

Answer question 8

- What does this query return?

```
check(_, []) :- !.  
check(E, [H|T]) :- E > H, check(E, T).  
?- check(10, [4, 3, 2]).
```

- A. false.
- B. true; false.
- C. true.
- D. false; true.

Why use cuts?

- Save time and space, or eliminate redundancy
 - Prune useless branches in the search tree
 - If sure these branches will not lead to solutions
 - These are **green cuts**
- Guide the search to a different solution
 - Change the meaning of the program
 - Intentionally returning only subsets of possible solutions
 - These are **red cuts**

Question 9

- Is this a green or red cut?

```
s(c) .  
s(m) .  
s(d) .  
solve(X):-s(X),!.  
solve(other_solution).  
?- solve(X).
```

A. green
B. red

Answer question 9

- Is this a green or red cut?

```
s(c) .  
s(m) .  
s(d) .  
solve(X):-s(X),!.  
solve(other_solution).  
?- solve(X).
```

A. green

B. red

Question 10

- What does this query return?

```
check(_, []) :- !.  
check(E, [H|T]) :- E > H, check(E, T).  
?- check(10, [4, 3, 2]).
```

A. red
B. green

Answer question 10

- What does this query return?

```
check(_, []) :- !.  
check(E, [H|T]) :- E > H, check(E, T).  
?- check(10, [4, 3, 2]).
```

A. red

B. green

Tic-tac-toe example

- If we type semicolons at the program, it will continue to generate a series of increasingly poor moves from the same board position, even though we only want the first move
- We can cut off consideration of the others by using the cut:

```
move(A) :- good(A), empty(A), !.
```

- To achieve the same effect with `not` would be harder



Built-in predicates

I/O built-in predicates

- Prolog provides some I/O features.
 - `write` and `nl`: print to the current output file
 - `read`: read terms from the current input file
 - `get` and `put`: individual characters are read and written
 - `consult` and `reconsult`: to read database clauses from a file

repeat

- The `repeat` predicate can succeed an arbitrary number of times
- It behaves as if it were implemented with the following pair of rules.

```
repeat.
```

```
repeat :- repeat.
```

- In general, `repeat` allows us to turn any predicate with side effects into a generator.
- For instance

```
?- repeat, write(a).
```

```
a
```

```
true ;
```

```
a
```

```
true ;
```

```
a
```

```
true ■
```

Database

- The database can modify itself.
- A running Prolog program can add clauses to its database with the built-in predicate `assert`, or remove them with `retract`

```
?- rainy(X).  
X = rochester ;  
X = seattle.
```

```
?- assert(rainy(syracuse)).  
true.
```

```
?- rainy(X).  
X = rochester ;  
X = seattle ;  
X = syracuse.
```

```
?- retract(rainy(rochester)).  
true.
```

```
?- rainy(X).  
X = seattle ;  
X = syracuse.
```

The complete Tic-toc-toe example

- It uses `assert`, `retract`, the `cut`, `fail`, `repeat`, and `write` to play an entire game
- Moves are added to the database with `assert`.
- They are cleared with `retract` at the beginning of each game.

The complete Tic-tac-toe example (1)

`:- dynamic o/1.` The predicate `dynamic` allows
`:- dynamic x/1.` you to use `assert` and `retract`

```
x(_).  
o(a).  
  
ordered_line(1, 2, 3).  
ordered_line(4, 5, 6).  
ordered_line(7, 8, 9).  
ordered_line(1, 4, 7).  
ordered_line(2, 5, 8).  
ordered_line(3, 6, 9).  
ordered_line(1, 5, 9).  
ordered_line(3, 5, 7).  
line(A, B, C) :- ordered_line(A, B, C).  
line(A, B, C) :- ordered_line(A, C, B).  
line(A, B, C) :- ordered_line(B, A, C).  
line(A, B, C) :- ordered_line(B, C, A).  
line(A, B, C) :- ordered_line(C, A, B).  
line(A, B, C) :- ordered_line(C, B, A).  
  
move(A) :- good(A), empty(A).  
  
full(A) :- x(A).  
full(A) :- o(A).  
empty(A) :- not(full(A)).
```

The complete Tic-tac-toe example (2)

```
win(A) :- x(B), x(C), line(A, B, C).
block_win(A) :- o(B), o(C), line(A, B, C).
split(A) :- x(B), x(C), different(B, C),
line(A, B, D), line(A, C, E), empty(D), empty(E).
same(A, A).
different(A, B) :- not(same(A, B)).
strong_build(A) :- x(B), line(A, B, C), empty(C), not(risky(C)).
risky(C) :- o(D), line(C, D, E), empty(E).
weak_build(A) :- x(B), line(A, B, C), empty(C),
not(double_risky(C)).
double_risky(C) :- o(D), o(E), different(D, E), line(C, D, F),
line(C, E, G), empty(F), empty(G).
```

% strategies:

```
good(A) :- win(A).
good(A) :- split(A).
good(A) :- block_win(A).
good(A) :- weak_build(A).
good(A) :- strong_build(A).
good(5).
good(1).
good(3).
good(7).
good(9).
good(2).
good(4).
good(6).
good(8).
```

If block win occurs too late among the strategies, it is likely that it lose

The complete Tic-tac-toe example (3)

```

all_full :- full(1), full(2), full(3), full(4), full(5), full(6),
full(7), full(8), full(9).

done :- line(A,B,C), o(A), o(B), o(C), write('I won'), nl.
done :- all_full, write('You lost'), nl.
% User's move
getmove :- repeat, write('Please enter a move: '), read(X), empty(X),
asserta(o(X)).
makemove :- move(X), !, makemove.
makemove :- all_full.
% Computer's move
respond :- line(A,B,C), o(A), o(B), o(C),
printboard, write('You won.'), nl. %Shouldn't ever happen!
respond :- makemove, printboard, done.

printsquare(N) :- o(N), write( o ).
printsquare(N) :- x(N), write( x ).
printsquare(N) :- empty(N), write(' ').
printboard :- printsquare(1), printsquare(2), printsquare(3), nl,
printsquare(4), printsquare(5), printsquare(6), nl,
printsquare(7), printsquare(8), printsquare(9), nl.

```

Repeats the rest of the clause

write prints strings, nl prints newlines

Assert a clause in the db as first clause

read reads strings

The complete Tic-tac-toe example (4)

```
% Clear everything to start a new game.
clear :- retractall(x(_)), retractall(o(_)),
write('Board:'), nl,
write(' 1 '), write(' 2 '),write(' 3 '), nl,
write(' 4 '), write(' 5 '),write(' 6 '), nl,
write(' 7 '), write(' 8 '),write(' 9 '), nl,
write('You will be o, computer will be x. '), nl,
write('Enter moves by giving a number (1-9) followed by a period. '),nl,
nl.

% main goal:
play :- clear, repeat, getmove, respond.
```

retracts all

Characteristics of logic programming

- We only need to define the logic specifications
- Pros of “computation as deduction”
 - ability to use a program in more than one way (both input and output)
 - Possibility of obtaining several solutions
 - Possibility of looking at a program as a logical formula
- Cons
 - Backtracking can be inefficient
 - Absence of types and modules
 - Not very well developed programming environments

Readings

- Chapter 12 of the reference book
 - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill
- Few slides from the University of Maryland



Summary

- Lists
- Search order and backtracking
- Cut
- Built-in predicates

SUMMARY



Next time



- Lambda calculus