λ

Logic Programming – Part II

Programmazione Funzionale

2024/2025

Università di Trento

Chiara Di Francescomarino

When you have time

You can find the link also in Moodle!

Join this Wooclap event

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
DQDQRX

When you have time

- Fill the feedback form about the course:
    - https://forms.gle/9btC4JBDAWumxeQ29

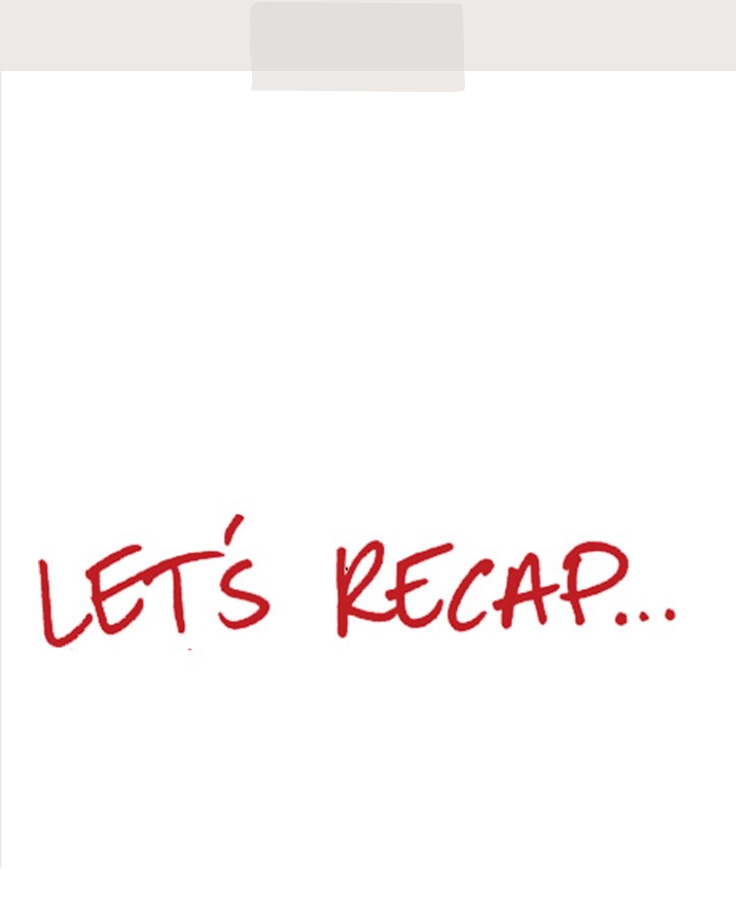
- The link is also available in Moodle

Next lectures

- Tomorrow: last tutoring slot
- Thursday May 29: exam simulation
- Last lecture: Tuesday June 3

Today

- Prolog
  - Functions
  - Search order and backtracking
  - Operator `cut` and `not`
- Clarifications on structures, signatures and functors



Recap

An example of Prolog program

lowercase for atoms

Clauses:
Fact and rules

uppercase for variables

```
% facts:
rainy(rochester).
rainy(seattle).
cold(rochester).

% rules for "X is snowy"
snowy(X):-rainy(X),cold(X).
```

Query or goal

```
?:- snowy (rochester)
```

Unification and resolution in Prolog

```
takes(alice, cs254)
takes(bob, cs254)
```

- 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`
- Resolution: the idea is unifying the goal with the head of a rule
  - If succeeds, clauses in body become subgoals
  - Continue until all subgoals are satisfied
    - If search fails, backtrack and try untried subgoals

Unification in Prolog

- The unification rules for Prolog are as follows:
  - A constant unifies only with itself
  - Two structures unify if and only if they have the same functor and the same number of arguments, and the corresponding arguments unify recursively
  - A variable unifies with anything
    - If the other thing has a value, then the variable is instantiated
    - If the other thing is an uninstantiated variable, then the two variables are associated in such a way that if either is given a value later, that value will be shared by both.

Arithmetic: built-in predicate `is`

- The built-in predicate `is` unifies its first argument with the arithmetic value of its second argument

```
?- is(X, 1+2).          prefix notation
X=3.


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

Lists: head and Tail

- Differently from ML, all sorts of even heterogenous Prolog terms can be elements of a list, e.g., `[a,c,2,'hi',[W,3]]`
- A non-empty list can be thought of as consisting of two parts
  - The head
  - The tail
- As in ML, the head is the first item in the list
- The tail is everything else
  - The tail is the list that remains when we take the first element away
  - The tail of a list is always a list

Vertical bar notation |

- Using this notation, `[a, b, c]` can be expressed as `[a | [b, c]]`, `[a, b | [c]]`, or `[a, b, c | []]`
- `[H|T]` is syntactically similar to ML `h::t`

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

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]; ...

Without arithmetic operations , we can have variables also as “operands”

Let’s try

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
OSLGGP

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)`

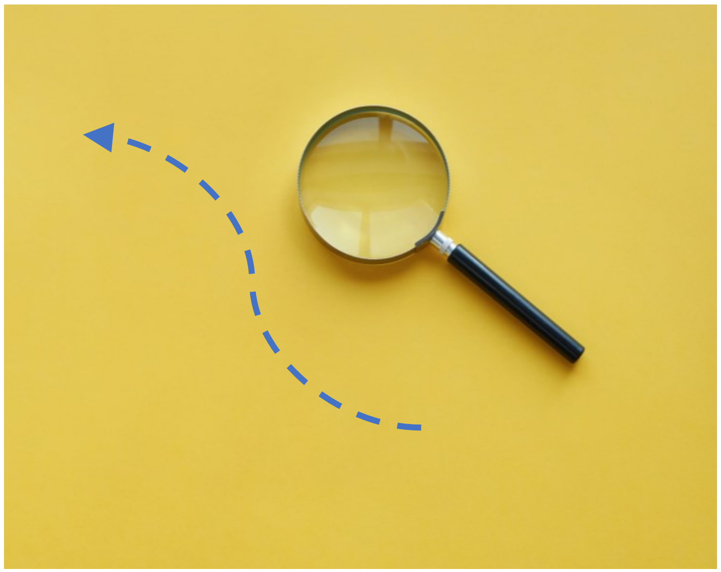
```
?- sort([2,1,3], R).
R= [1,2,3].
```

- `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



Search order and backtracking

Ground and nonground terms

- A goal or term where variables do not occur is called ground; otherwise it is called nonground (if there is at least a nonground variable)
  - `foo(a,b)` is ground;
  - `bar(X)` is nonground
- When resolving nonground goals, the interpreter will unify and instantiate the free variables with terms so that the ground predicate holds.

Substitution

- Two terms unify
  - if they are identical

    `?- takes(alice, cs254).` -> unifies directly with the fact
  - they can be made identical by substituting free variables

    `?- takes(alice, X).` -> variable `X` is instantiated with `cs254`
- Unifying two terms `s` and `t` means finding a substitution θ over their free variables.
- 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
- 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 θ

Question

- Which of these are ground terms?

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

ground
ground
nonground

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.

Search order: forward or backward?

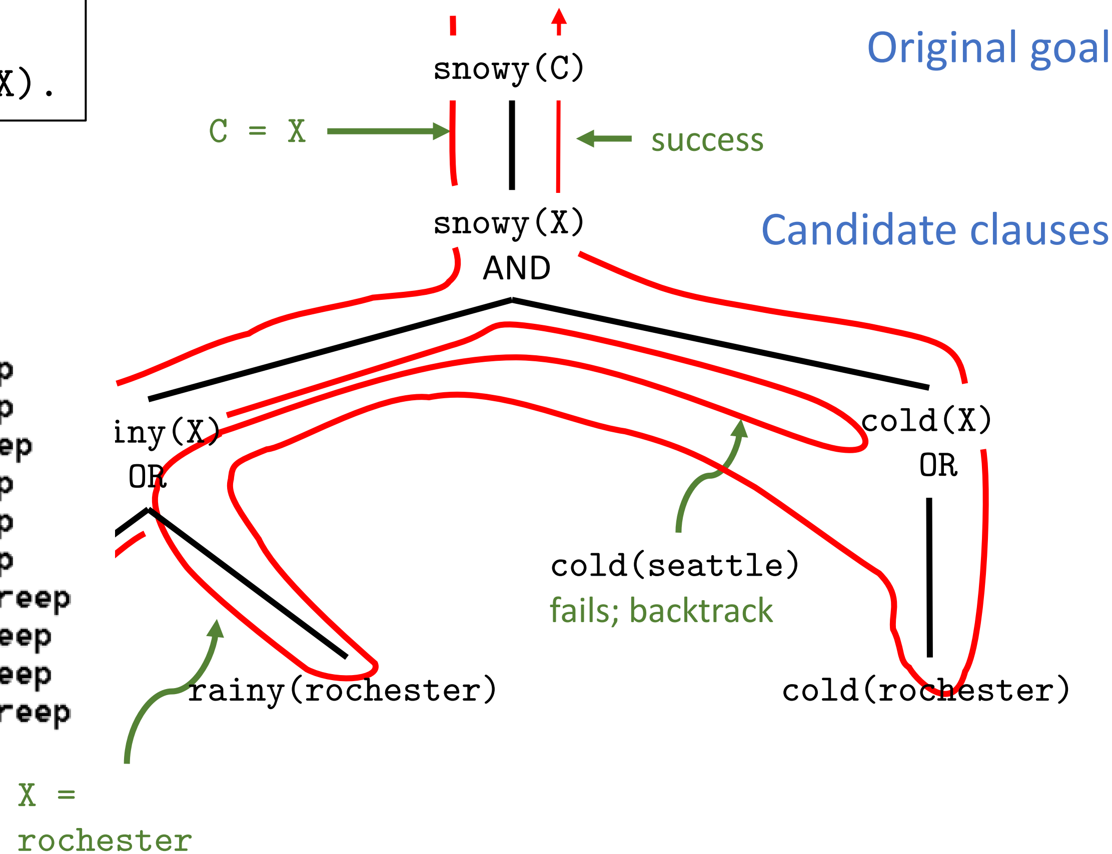
- If the number of existing rules is very large, but the number of facts is small, it is possible for forward chaining to discover a solution faster than backward chaining
- In most circumstances, however, backward chaining turns out to be more efficient
- Prolog is defined to use backward chaining.
- 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 (e.g., left to right).
- It can be described in terms of a tree of subgoals.

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: backtracking

- Whenever a unification operation is "undone" in order to pursue a different path through the search tree, variables that were given values or associated with another one 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

When does a goal fail?

- `G` will not fail unless all of its subgoals, and all of its siblings to the right in the search tree, have also failed
- 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.

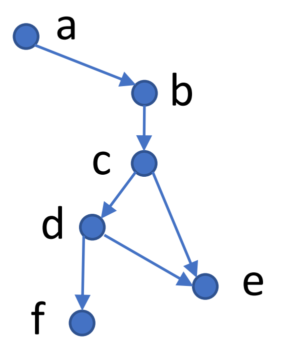
Deterministic and predictable programs

- 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`

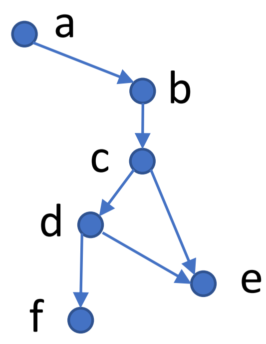
- If we query for `?path1(a,d).`

```
?- 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).
```



What if …

- … we reverse the order of the terms on the right-hand side of the final clause?

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

- 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`
- 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)
```

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

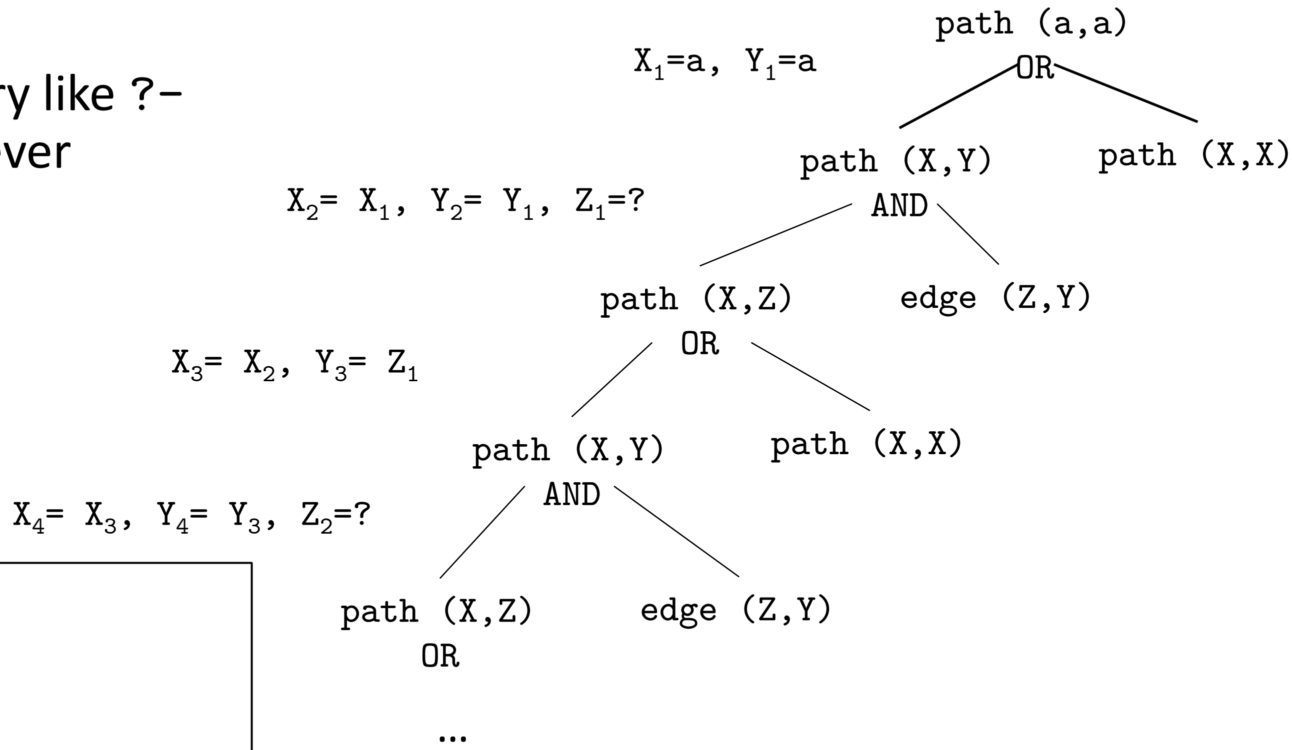
```
path3(X, Y) :- path3(X, Z), edge(Z, Y).
path3(X,X).
```

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:
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) ? 
```

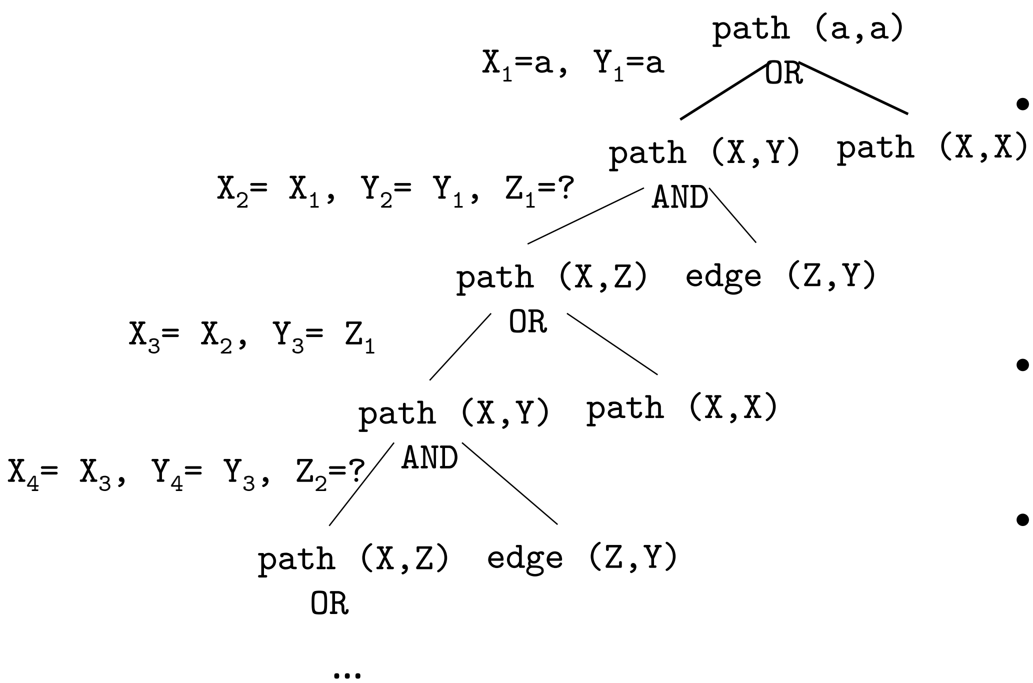
Non-termination

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
  `path(X, Y) :- path(X, Z), 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

Cut

Imperative control flow

- Besides simple ordering, Prolog provides the programmer with several explicit control flow features
- The most important is known as the cut: a zero-argument predicate written as an exclamation point !
- 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.

The cut operator

- 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* +1st clause.
  2. If the evaluation of B succeed, 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 well as are eliminated all the alternatives provided by the clauses from the *k*th to the *n*th to compute `p(t)`.

Why to use the `cut` operator?

- Let us consider

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

- If a given atom a appears in list `L` n times, then the goal `?- member(a,L)` can succeed n times

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

What's the problem?

These extra successes may not always be appropriate! … they can lead to wasted computation, particularly for long lists, when member is followed by a goal that may fail.

Why to use the `cut` operator?

- Let's assume we want to check whether a number is a prime candidate

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

- and `prime(X)` is expensive to compute.
- To determine whether `a` is a `prime` candidate, we first check whether it is a member of the candidates list, and then check 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

Saving time with the `cut` operator

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

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

The `cut`: other examples

- Similarly, if we have
  - `minimum2(X, Y, X) :- X < Y,!.`
  - `minimum2(X, Y, Y) :- X > Y.`

```
?- minimum2(3,5,X).
X = 3.
```

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:

```
?- minimum1(3,5,X).
X = 3 ;
false.
```

- 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 **`not`** operator

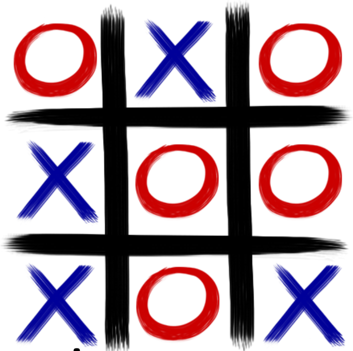
- An alternative way to ensure that `member(X,L)` succeeds no more than once is to embed a use of `not` in the second clause

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

```
?- member3(a,[a,a,b,c,a]).
true ;
false.
```

- 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

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).

full(A) :- x(A).
full(A) :- o(A).
empty(A) :- not(full(A)).

% strategy:
good(A) :- win(A).
good(A) :- split(A).
good(A) :- weak_build(A).
good(A) :- block_win(A).
good(A) :- strong_build(A).
```

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

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

These are our strategies in order of application

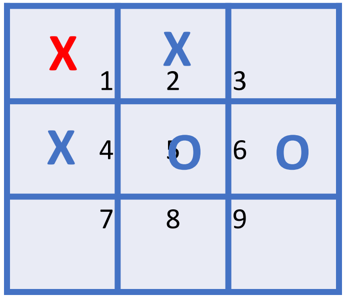
Strategy

```
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



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

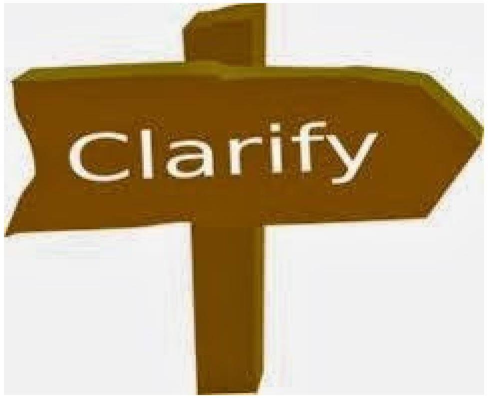
The fourth is moving toward three in a row, so that the obvious blocking won't allow our opponent to build toward three in a row

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).
```



This Photo by Unknown Author is licensed under CC BY-SA-NC

Clarifications on ML

Structures

- A structure is a module in ML.
- In order to access the structure components you have to use the name of the structure
- As an alternative you can use `open structureName`, however you have to be **very careful** opening structures – especially predefined ones, as default functions can be overwritten.

Example: accessing structure components

```
> structure BranchTree =
        struct
          datatype 'a T = BTEmpty | Branch of 'a * 'a T * 'a T
          fun countNodes BTEmpty = 0
          |countNodes (Branch(a,b,c)) = 1+countNodes(b) + countNodes(c);
end;

> val brtree = BTEmpty;
poly: : error: Value or constructor (BTEmpty) has not been declared Found near
BTEmpty
Static Errors
> val brtree = BranchTree.BTEmpty;
val brtree = BTEmpty: 'a BranchTree.T

> open BranchTree;
datatype 'a T = BTEmpty | Branch of 'a * 'a T * 'a T
val countNodes = fn: 'a T -> int
> val brtree = BTEmpty;
val brtree = BTEmpty: 'a T
```

Example: be careful with opening structures!

```
> 2 >1;
val it = true: bool

> open String;
val < = fn: string * string -> bool
val <= = fn: string * string -> bool
val > = fn: string * string -> bool
...

> 1 > 2;
poly: : error: Type error in function application.
   Function: > : string * string -> bool
   Argument: (1, 2) : int * int
   Reason:
      Can't unify int (*In Basis*) with string (*In Basis*)
         (Different type constructors)
Found near 1 > 2
Static Errors
```

Signatures

- A signature is a kind of interface in ML.
- When we restrict a structure to a signature, we are restricting external access only to types, data and functions in the signature.

Example: signature with abstract type

```
> signature BRANCHTREE =
          sig
          type 'a T
          val countNodes : 'a T -> int
end;

> structure BranchTree :> BRANCHTREE =
struct
    datatype 'a T = BTEmpty | Branch of 'a * 'a T * 'a T
    fun countNodes BTEmpty = 0
          |countNodes (Branch(a,b,c)) = 1+countNodes(b)+countNodes(c);
end;

> val brtree = BranchTree.BTEmpty;
poly: : error: Value or constructor (BTEmpty) has not been declared in structure
BranchTree
Found near BranchTree.BTEmpty
Static Errors
```

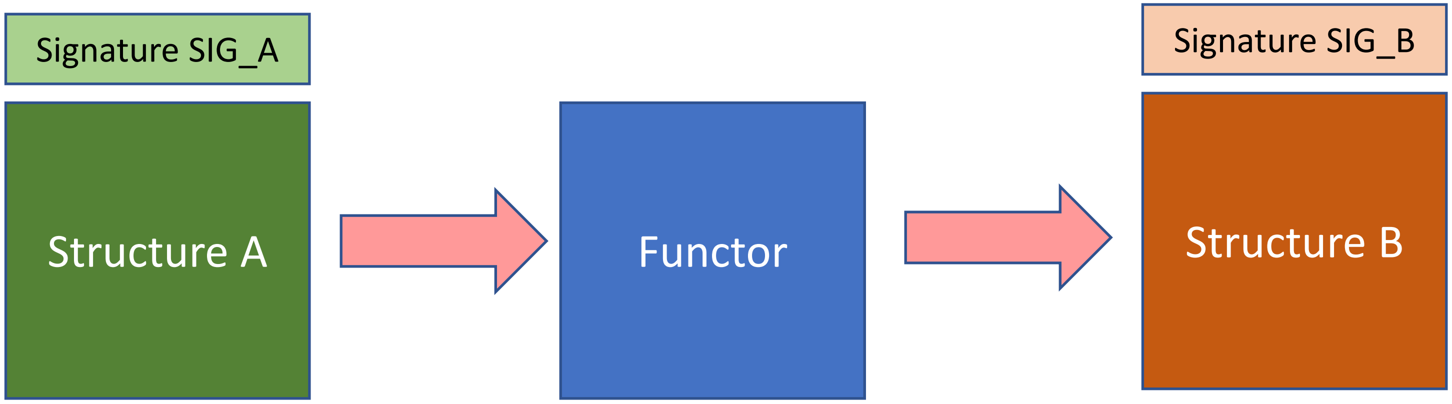
Example: signature with specified type

```
> signature BRANCHTREE =
         sig
         datatype 'a T = BTEmpty | Branch of 'a * 'a T * 'a T
         val countNodes : 'a T -> int
end;


> structure BranchTree :> BRANCHTREE =
struct
    datatype 'a T = BTEmpty | Branch of 'a * 'a T * 'a T
    fun countNodes BTEmpty = 0
         |countNodes (Branch(a,b,c)) = 1+countNodes(b)+countNodes(c);
end;

> val brtree = BranchTree.BTEmpty;
val brtree = BTEmpty: 'a BranchTree.T
```

Functors

- A functor is a structure that takes a structure and returns another structure
  - As a function takes a value and returns a new value a functor takes a structure and returns a new structure

```
functor <identifier> (<structure name>:
<signature>) = <structure definition>
```

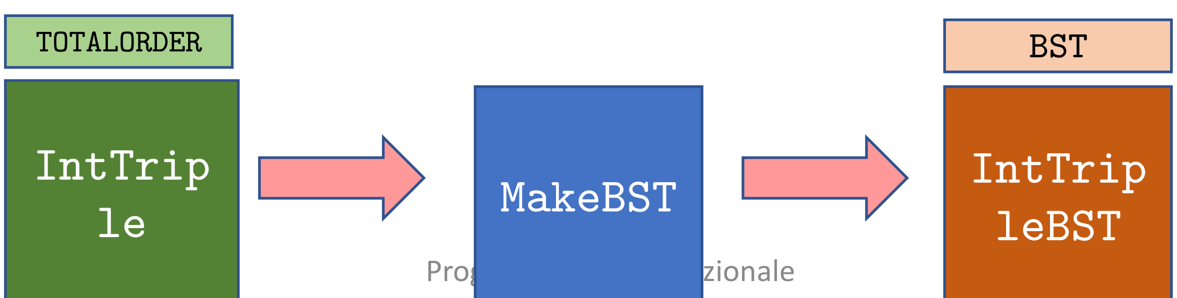


What we did

- The steps we took for building a functor `MakeBST` that allows for building a binary search tree with a customized ordering function:
  - Step 1: define a signature `TOTALORDER` that is satisfied by our functor inputs
  - Step 2: define a functor `MakeBST` that takes a structure `S` with signature `TOTALORDER` and produces a structure
  - Step 3: define structure `String` with signature `TOTALORDER` and with a comparison operator on strings
  - Step 4: apply `MakeBST` to String to produce the desired structure

Exercise 9.12

- Write a structure `IntTriple` implementing the signature `TOTALORDER` that, in place of `String`, defines the elements of a binary search tree as tuples of 3 integer numbers
- lt: (a, b, c) < (x, y, z) iff
  - a < x
  - a = x and b < y
  - a = x, b = y and c < z
- Use the functor `MakeBST` to get a structure that stores triples of integers in binary trees



Exercise 9.12

```
> signature TOTALORDER = sig
    type element;
    val lt : element * element -> bool
  end;
signature TOTALORDER = sig type element val lt: element * element -> bool end;

functor MakeBST(Lt: TOTALORDER):
  sig
    type 'label btree;
    exception EmptyTree;
    val create: Lt.element btree;
    val lookup: Lt.element * Lt.element btree -> bool;
    val insert: Lt.element * Lt.element btree -> Lt.element btree;
    val deletemin : Lt.element btree -> Lt.element * Lt.element btree;
    val delete : Lt.element * Lt.element btree -> Lt.element btree
  end
=
```

BST

Actually you can omit the signature here, although it is better to specify it

Exercise 9.12

```
struct
  open Lt;
  datatype 'label btree
      = Empty
       | Node of 'label * 'label btree * 'label btree;
   val create = Empty;
   fun lookup(x, Empty) = false
    | lookup(x, Node(y, left, right)) =
       if lt(x, y)
       then lookup(x, left)
       else if lt(y, x)
       then lookup(x, right)
       else (* x=y *) true;
```

Exercise 9.12

```
fun insert(x, Empty) = Node(x, Empty, Empty)
  | insert(x, T as Node(y, left, right)) =
     if lt(x, y)
     then Node(y, insert(x, left), right)
     else
       if lt(y, x)
       then Node(y, left, insert(x, right))
       else (* x=y *) T; (* do nothing; x was already there *)
exception EmptyTree;
fun deletemin(Empty) = raise EmptyTree
  | deletemin(Node(y, Empty, right)) = (y, right)
  | deletemin(Node(w, left, right)) = let
     val (y, L) = deletemin(left)
  in
     (y, Node(w, L, right))
  end;
```

Exercise 9.12

```
fun delete(x, Empty) = Empty
   | delete(x, Node(y, left, right)) =
      if lt(x, y)
      then Node(y, delete(x, left), right)
      else if lt(y, x)
      then Node(y, left, delete(x, right))
      else (* x=y *) case (left, right)
         of (Empty, r) => r
         | (l, Empty) => l
         | (l, r) => let
            val (z, r1) = deletemin(r)
            in
            Node(z, l, r1)
     end;
end;
```

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

- Prolog
  - Functions
  - Search order and backtracking
  - Operator `cut` and `not`
- Clarifications on structures, signatures and functors

Next time

- Introduction to Scala