Logic Programming

Rodica Potolea (A&Eng)
Camelia Lemnaru (B)
CS @ UTCN

Lecture #1

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA
Computer Science

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Agenda

- What this course is/is NOT about
- Logic in Programming
- Main LP elements
- LP program
  - Syntax
  - Semantic
  - Execution tree

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

What this course is about?

- **Course on Logic Programming**
  - Define the LP **syntax** and **semantics**
  - Understand the **execution mechanism**
  - Learn the main **LP paradigms**
  - ***Language as executable specifications***
  - Practice in a **LP language** (Prolog and "dialects")
- **General CS practice**
  - Basic **problem solving**
  - *Revise* **Data Structures and Algorithms** from the LP perspective

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

In a nutshell…

- Programming so far:
  - Imperative (procedural languages)
    - assumes the presence of an assignment (:=) operator
    - specific control structures used
    - presence of side effects
- Declarative languages (symbolic languages)
  - Functional and *logic* languages
    - Assignment operation NOT present
    - Control removed from programmer (*mostly*; in a pure LP style, all of it)
    - No side effects (*in pure LP;* specific mechanisms implementing side effects exist)

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

In a nutshell…

- Program = Algorithm + **Data Structure**
  - Wirth (book title)
- Algorithm = ***Logic*** + **Control**
  - Kowalsky's statement

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

LP philosophy

- Program:
  - In conventional programming languages - describing the algorithm using the language's syntax
  - In LP - describing formal relationships occurring among objects (here objects NOT in the OO meaning)
  - Focus on the descriptive perspective rather than the prescriptive one
  - A LP program = collection of known facts and relationships between components, rather than a sequence of steps to be taken
  - Computation is carried out by the specification (with logic embedded in the declarative semantic + new facts inferred) and just partially by explicit control information supplied by the program
- LP main characteristics:
  - **Logic variables**
  - **Shared structures**
  - **Unification** mechanism (matching)
  - Search for **alternative solutions** (when writing *nondeterministic* solutions)

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

LP syntax

- We'll use PROLOG to generically denote Logic Languages
  - PROLOG = PROgramming LOGic
  - For a brief history check references and Prolog Heritage
- Matching basic operation (evaluation rare, just by **explicit request**)
- Ex: **A=a** means A unifies (matches) with a and NOT that A takes the value of a
- Unification = the attempt to make the two sides the same
  - Successful matching (TRUE) – they are the SAME from now on
  - Unsuccessful matching (FALSE or fail in Prolog jargon) – backtracking mechanism installed (see later)
- NO keywords (just a very few; we'll learn them on the fly)
- NO statements
- The program is just what you write
- **Case sensitive** language
  - Variables – UPPER FIRST case (Variable, VARIABLE, VaRiAbLe are all variables; variable is constant)
  - Constants – lower first case (constant, cONSTANT, cOnStAnT are all constants; Constant is variable)

LP syntax

- Program = collection of predicates
- Predicate = a description (in Prolog syntax) of theorems
  - a collection of clauses
  - name of a predicate = a constant (starts with lower case)
- The general form of a clause is represented by:

```
p(X):-q1(Y),q2(X,Y),…, qn(X,Z).
```

p, qi, i=1…n represent names of predicates (first letter lower case)

X, Y, Z arguments (variables = start with upper case)

X	= parameter
`Y,Z`	= local variables (hidden variables; explained later)
`p(X)`	= head of the clause
`qi, i=1…n`	= body of the clause (sub-query)
`:-`	= if
`,`	= and (conjunction)
`.`	= full stop (end of clause). Dot is mandatory.

Prolog PREDICATES (clauses)

- The general form of a **clause** is represented by:

```
p(X):-q1(Y),q2(X,Y),…, qn(X,Z).
Left hand side (lhs) IF right hand side (rhs).
head :- body.
```

- Clause
  - a **theorem** = a representation in Prolog syntax of the relationships (body) among the objects (variables) occurring in the clause; it has a standalone meaning
  - Can be viewed as a procedure (function, method) from imperative (procedural, OO) languages
  - Meaning: the head is true (p; lhs) **if** the body (rhs) is true (qi, i=1…n; rhs)
  - The conclusion of the theorem (head, lhs) is true if all the hypotheses of the theorem (rhs) are true
  - `p` - head
  - `qi`, i=1…n - body
- Predicate = a set of one or several clauses
- Prolog program = a set of one or several predicates

Prolog PREDICATES
(rules, facts, queries)

- The general form of a clause is represented by:

```
p(X):-q1(Y),q2(X,Y),…, qn(X,Z).
```

- Particular form of clauses:
  - General form – above (**rule**)
  - Particular forms: **facts** and **queries**
- **Fact**

  = a clause without body

  = a theorem WITHOUT hypothesis = AXIOM (no need for proof)

  `p(a).` // meaning *p of a is true, without condition*.
- **Query**

  = a clause without head

  = a theorem WITHOUT conclusion = only hypotheses which need proof = if true, will be a new axiom

  ?- `q2(a,b).` // meaning *check if q2 of a and b is true*.

This defines the **declarative semantics** of Prolog = interpretation of the statements

The first LP program

```
man(john).
.
.
.
parent(john,david).
parent(john,edward).
.
.
.
father(X,Y):-
        man(X),
        parent(X,Y).
?-father(john,Z).
```

The first LP program

```
man(john).              %john, lower case start is a man.
.
.
.
parent(john,david).  %john is david's parent
parent(john,edward). % john is edward's parent
.                       % could be the other way around
.                       %  david is john's parent
.                       % is up to us; depends on how WE define it!
father(X,Y):-           % X is Y's father IF (by :- sign)
        man(X),         % X is man AND (by ,)
        parent(X,Y).    % X is Y's parent. That's all.
?-father(john,Z).       % whose father is john? Return me the result in Z.
```

Execution of the first LP program: Questions to answer

- How is it working?
  - Matching mechanism
  - Match the query against the head of the clause of the predicate with the same name (`father` in our case) = *query/head unification*
- Some questions need to be answered:
  - **Q1**: **what if query/head unification have several alternatives** (translates into: predicate has several clauses)
  - **Q2: what if query/head unification succeeds and we have some body** (translates into: we have a complete clause we matched with)
  - **Q3: what if query/head unification fails** (translates into: we have an unsuccessful matching)
- The answers define the **procedural semantics** of PROLOG = interpreting the statements as actions to take (call and run) (also named **operational semantics**)

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Execution of the first LP program answers to questions

- **Q1**: what if query/head **unification** have several alternatives (translated: predicate with several clauses)
  - **A1**: first clause first = **top-down**
- **Q2:** what if query/head unification succeeds and we have some body (translated: complete clause)
  - **A2**: first subgoal (sub-body) first = **left-right**
- **Q3:** what if query/head unification fails (translated: unsuccessful matching)
  - **A3**: unification fails = **backtracking**
  - Backtracking is a build-in mechanism = it automatically launches in case of failure

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

LP operational semantics

- Defined by the answers from before:
  - **Selection** (computation) rule: the top/down approach (A1)
  - **Search** rule: the left/right approach (A2)
  - **Backtracking** by default

OF CLUJ-NAPOCA
Computer Science

Execution of the first LP program

```
m1 man(john).
.
.
.
p1 parent(john,david).
p2 parent(john,edward).
.
.
.
f1 father(X,Y):-
        man(X),          f11
        parent(X,Y).     f12
q ?-father(john,Z).
```

A1: top-down
A2: left-right
A3: backtrack upon failure

A1: (1) **q/f1:** `father(john,Z)/father(X,Y) => X=john, Z=Y`

It results the CONCLUSION could be true in case all conjunction in the body is true as well

the body of the clause becomes the NEW goal (or query) to execute.

So, the NEW goal is an instance of f11,f12, that is:

f11,f12 = `man(X),parent(X,Y).`

Where X and Y got instantiated already (X and Y are shared variables; shared as they are THE SAME as in the head of the rule)

A2 tells us f11 gets executed before f12; moreover, f12 is executed **iff** f11 is TRUE.

Execution of the first LP program – contd.

A1: top-down
A2: left-right
A3: backtrack upon failure

`?- man(X),parent(X,Y).` % **f11, f12**

A2 tells us to *SOLVE f11 and PUT on hold f12* (what happens: the entire query is put on the exe. stack, f12 on bottom, f11 on top); the exec. pops the top of the stack

f11: `?- man(X).` is the new query to execute => Q1 => **A1** =>

(2) f11/m1: `man(X)/man(john) => X=john` succeeds.

Is a fact => unconditionally true => f11 IS TRUE.

In a query/rule match, HOW DO WE CONTINUE? We must check 3 conditions (for a **final** successful unification) in the specific order:

- **C1: current goal succeeds?**
- **C2: is a fact?**
- **C3: is the execution stack empty?**

Continuation depends on how the conditions are (not) met.

(remember: the execution stack contains the goals to execute)

Execution mechanism – conditions to check

- C1: current goal succeeds? If yes, go C2, else, backtrack.
- C2: is it a fact? If yes, go C3, else, go take and execute the entire body.
- C3: is the stack empty? If yes, over (the whole execution ends successfully), else pop the top of the execution stack and continue execution (from C1)
- Backtrack means WHAT?
  - UNDO the latest unification (that is, REMOVE latest matching) and
  - for the given goal, try a different alternative (next one, if any)
  - If again fail (or no other alternative), the backtracking mechanism propagates in REVERSE order of the execution.

Execution of the first LP program – contd.

`?- man(X),parent(X,Y).` % **f11, f12**

Solved f11; so?

- C1: current goal succeeds? YES => unification succeeds => successful node in the execution tree
- C2: is a fact? YES => node is a leaf in the tree
- C3: is the stack empty? NO => pop the top and it becomes the right sibling of the latest node

In our case, pop -> **f12:** `?- parent(X,Y).` is the new query to execute => Q1 => A1 =>

(3) f12/p1: `parent(X,Y)/ parent(john,david).`

BUT, from the previous query we had => `X=john`, and `X` is shared (among the body and head), thus here we actually have:

`john=john` (T) and `Y=david` (T)

So, successful matching.

Execution of the first LP program – contd.

f12 = `parent(X,Y).`

Solved f12; so?

- C1: current goal succeeds? YES => unification succeeds => successful node in the execution tree
- C2: is a fact? YES => node is a leaf in the tree
- C3: is the stack empty? YES => the whole execution ends successfully

So, successful matching. Where is the answer?

Following the deduction tree, we have the entire unification chain (next slides)

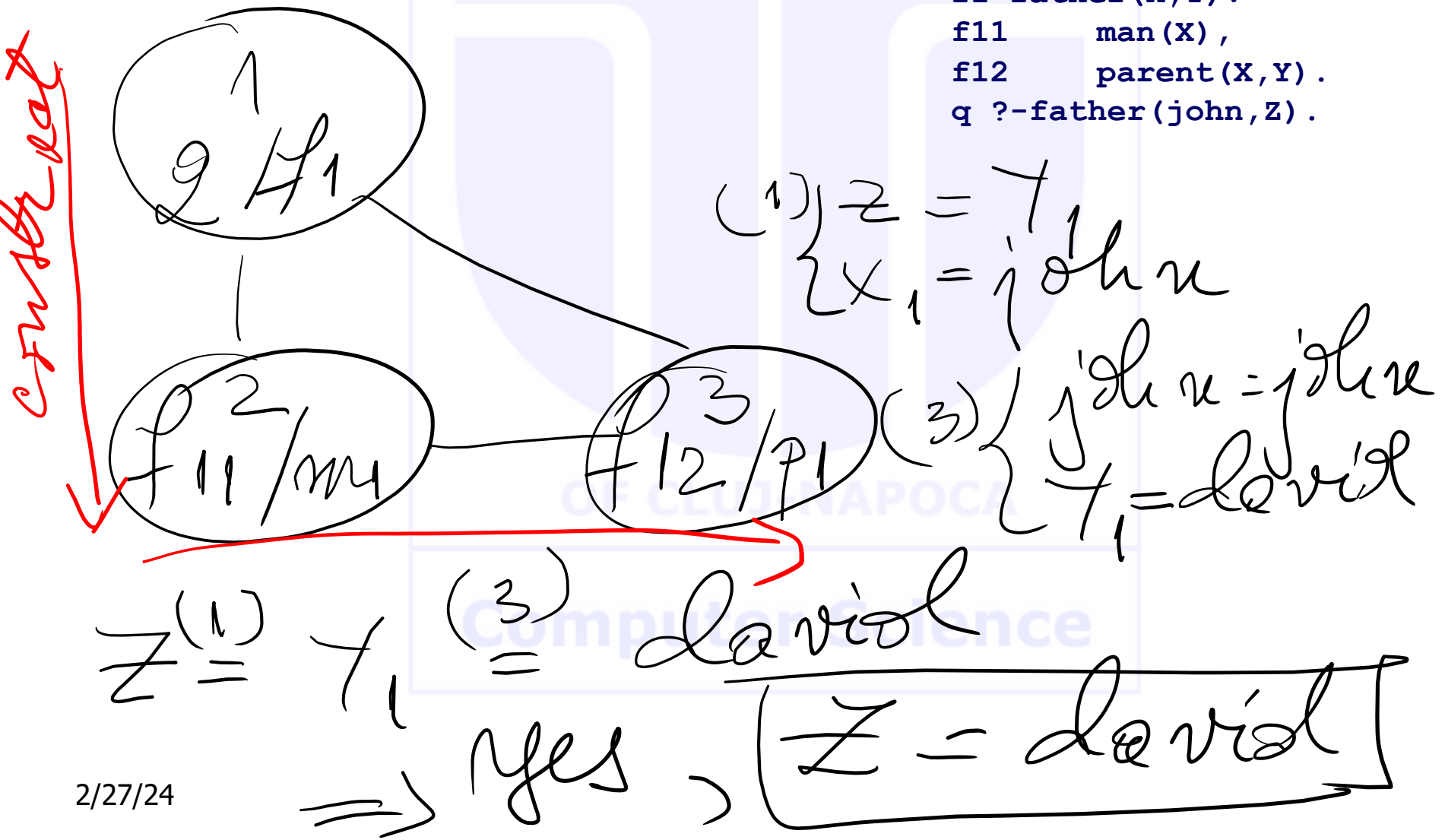
Here goes the deduction tree

```
m1 man(john).
.
p1 parent(john,david).
p2 parent(john,edward).
.
f1 father(X,Y):-
f11     man(X),
f12     parent(X,Y).
q ?-father(john,Z).
```

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA
Computer Science

Here goes the deduction tree

```
m1 man(john).
.
p1 parent(john,david).
p2 parent(john,edward).
.
f1 father(X,Y):-
f11     man(X),
f12     parent(X,Y).
q ?-father(john,Z).
```



TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Deduction Tree

- The final successful shape of an execution tree forms the **deduction** tree
- It stores the query/head matchings along with the unifications (formal vs actual arguments) made during the execution
- **Nondeterministic** behavior of Prolog = ability to find alternative solutions (if any) of the problem at hand
- It is performed via launching the **backtracking** mechanism (built-in)
- It refers to **repeating** the query; repeating the query is NOT asking the SAME query (if so, the SAME answer is provided)
- repeating the query is: for the SAME query AND in the context of the ALREADY built deduction tree, what OTHER possible answers are there?
- Syntactically, it is specified by ":" (at command prompt) (or ";" depending on the language), and it says: in the tree you already have, take the nodes in reverse order and try to find an alternative matching
- So, the LAST built node attempts to backtrack and only after it has no other choice it asks its predecessor node to backtrack.

Backtracking

- Evolution backwards in a tree already built
- The last built, the first to backtrack
- We had: (1)

(1) q/f1
(2) f12/m1
(3) f12/p1 backtrack on this => ERASE THE NODE and try **(3') f12/p2**

`parent(X,Y)/ parent(john,edward).`

BUT, from the previous query (2) we had => `X=john,` and `X` is shared (among the body and head), thus here we actually have:

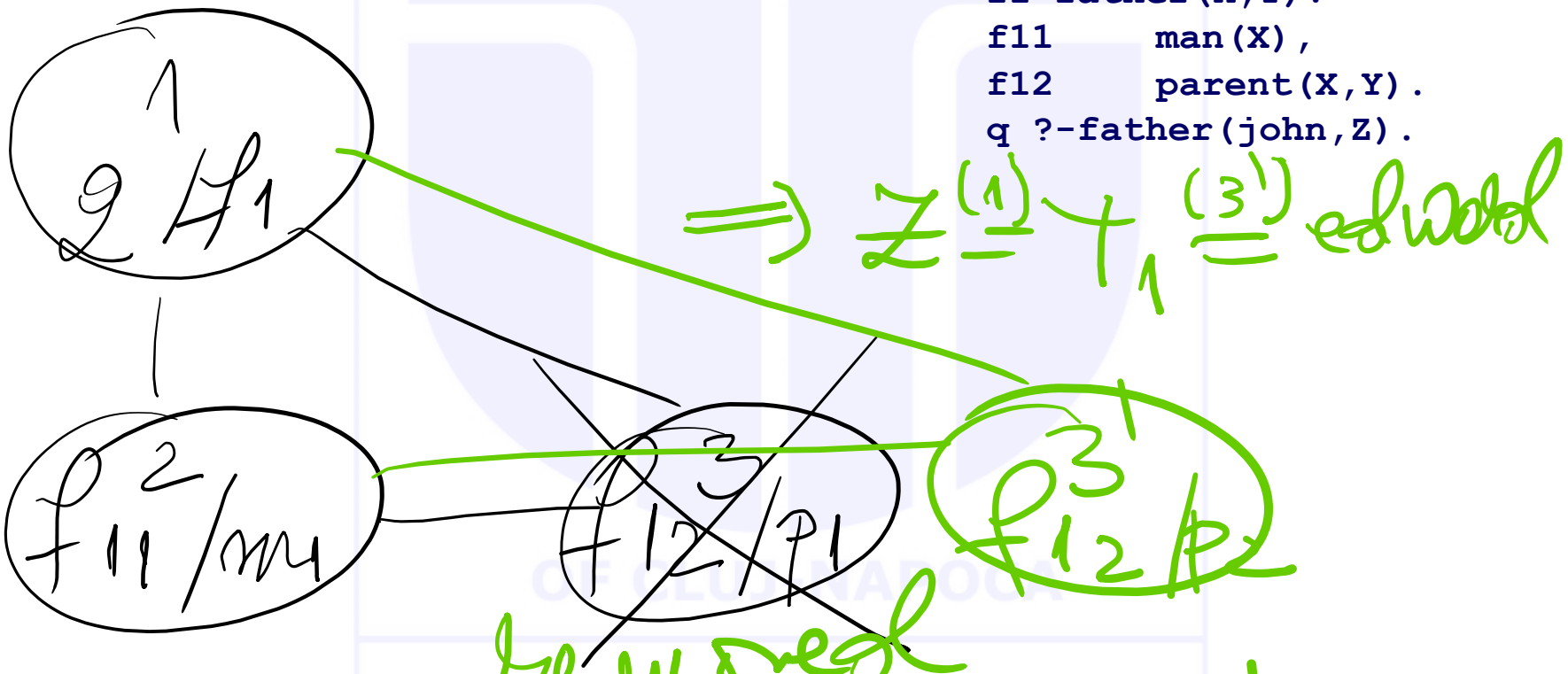
`john=john` (T) and `Y=edward` (T)

So, successful matching.

Following the new deduction tree we have the whole unification chain:

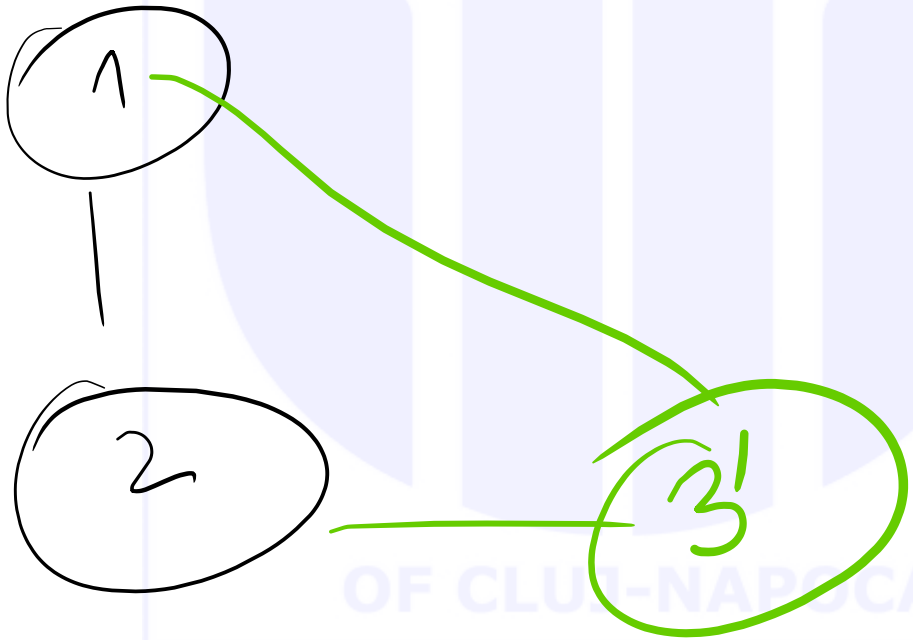
Here goes the deduction tree

```
m1 man(john).
.
p1 parent(john,david).
p2 parent(john,edward).
.
f1 father(X,Y):-
f11     man(X),
f12     parent(X,Y).
q ?-father(john,Z).
```

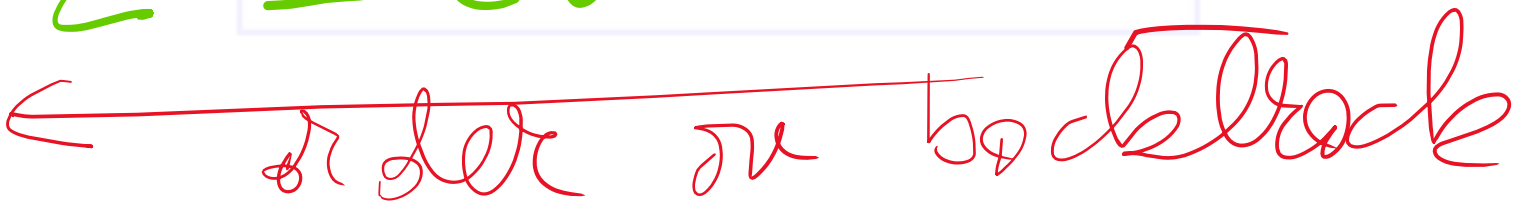


Backtracking – the tree

```
m1 man(john).
.
p1 parent(john,david).
p2 parent(john,edward).
.
f1 father(X,Y):-
f11       man(X),
f12       parent(X,Y).
q ?-father(john,Z).
```



Z = edward



TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Backtracking until final failure

- If we need ANOTHER answer, the evolution backwards in the new shape of the tree (1, 2 and 3')
- The last built, the first to backtrack
- We had:

(1) q/f1

(2) f12/m1

(3') f12/p2 backtrack on this => ERASE THE NODE and try **(3") f12/p3**
Assume p3 has as first argument something different from `john`

(3") fails to build, (3") is erased and the backtracking propagates on to 2:

(2') f12/m2 fails (as m2 has something different from `john`, and 2 constants unify iff they represent the same constant)

(2') fails to build, (2') is erased and the backtracking propagates on to 1:

(1') q/? No other clause exists => FINAL FAILURE

Meaning: NO other solution exists

Action: the whole tree is removed from memory (*go to the prev page to see*)

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Execution mechanism – review

- Rule 1: **Search** rule:

  the left to right evaluation of goals in conjunction
- Rule 2: **Selection** (computation) rule:

  the top-down selection of clauses from the program
- Rule 3: **Backtracking** by default:

  (see later lectures mechanisms for avoiding it when not necessary)

OF CLUJ-NAPOCA
Computer Science

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

First conclusions

- a Prolog **program** is a set of **theorems** (complete clauses) and axioms (facts)
- Executing a Prolog program means **proving a new theorem** (the query/goal) from the existing ones (program)
- The execution relies on **solving** a set of **systems of linear equations**
- The number of systems = number of nodes in the deduction tree
- Each system has a number of equations equal to the number of arguments of the goal executed in the corresponding node

List representation and decomposing lists

Lists: [1,2,3] – comma separated elements

- NO type required (heterogeneous lists by default)
- [a,b,c] how is it different from [A,B,C]?
- [] empty list
- [H|T] Head/Tail decomposition pattern with meaning:
  - H= the first element of the list
  - T = the rest of the list (= the list with the first element removed)
- [H|T] =[a,b,c] unification/matching
  - Before, H, T free variables
  - Now, bound to specific constants: H=a, T=[b,c]
  - [H1,H2|T]=[H1|[H2|T]] just rewriting rule
- [H|T] =[a]
  - H=a, T=[]
- [H|T] =[]
  - Fails, as H CANNOT unify the first element of the list (as there is NO element at all)
- In **LP basic operator is matching**

  (as opposed to FP where evaluator is the basic operation)

Example 2 – concatenating lists

```
append(L1,L2,L3):-
        L1=[],          //L2 empty
        L3=L2.          //L3 gets L2
append(L1,L2,L3):-
        L1=[H1|T],      //decomposition pattern; dec L1 into head and tail
        append(T,L2,R), //recurse on tail to get the partial result
        H3=H1,
        L3=[H3|R].      //recomposition pattern; rec L3 from head and tail
```

Order of clauses?

impact on correctness? Why?

impact on efficiency? How?

Example 2 contd

```
append(L1,L2,L3):-
        L1=[],          //L2 empty
        L3=L2.          //L3 gets L2
append(L1,L2,L3):-
        L1=[H1|T],      //decomposition pattern; dec L1 into head and tail
        append(T,L2,R), //recurse on tail to get the partial result
        H3=H1,
        L3=[H3|R].      //recomposition pattern; rec L3 from head and tail
```

Here we have explicit unifications.
By replacing them with **default** unifications, we get:

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

Example 2 execution

a1: append([],L,L).

a2: append([H|T],L,[H|R]):-

a21: append(T,L,R).

q:?-append([1,2],[3,4],Result).

Here goes equations and tree.

(1) **q/a1**[1,2]=[] fails, default backtracking

(1')**q/a2**[H1|T1]=[1,2] =>H1=1, T1=[2]

L1=[3,4]

Result=[H1|R1] =>Result=[1|R1]

a21: append(T1,L1,R1) (why?) and translates into append([2],[3,4],R1)

(2)**a21/a1** fails

(2')**a21/a2**[H2|T2]=[2] =>H2=2, T2=[]

L2=[3,4]

R1=[H2|R2] =>R1=[2|R2]

a21: append(T2,L2,R2)

append([],[3,4],R2)

(3)**a21/a1** []=[] L3=[3,4] R2=L3

Result=[1|R1]=[1|[2|R2]]=[1,2|R2]=[1,2|L3]=[1,2|[3,4]]=[1,2,3,4]

Example 2 execution

a1: append([],L,L).

a2: append([H|T],L,[H|R]):-

a21: append(T,L,R).

q:?-append([1,2],[3,4],Result).

Here goes equations and tree.

(1) **q/a1**[1,2]=[] fails, default backtracking

(1')**q/a2**[H1|T1]=[1,2] =>H1=1, T1=[2]

L1=[3,4]

Result=[H1|R1] =>Result=[1|R1]

a21: append(T1,L1,R1) (why?) and translates into append([2],[3,4],R1)

(2)**a21/a1** fails

(2')**a21/a2**[H2|T2]=[2] =>H2=2, T2=[]

L2=[3,4]

R1=[H2|R2] =>R1=[2|R2]

a21: append(T2,L2,R2) append([],[3,4],R2)

(3)**a21/a1** []=[] L3=[3,4] R2=L3

Result=[1|R1]=[1|[2|R2]]=[1,2|R2]=[1,2|L3]=[1,2|[3,4]]=[1,2,3,4]

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Lists patterns rephrasing

[H1,H2|T]=[H1|[H2|T]] just rewriting

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science