

PROLOG

- PROLOG is a programming language
- PROLOG: PROgramming in LOGic
- Invented by Alain Colmerauer in 1973 in Marseille (France)
- Use of mathematical logic to represent knowledge
- Declarative language



Syntax

- **Constants** are character strings that begin with minuscule letters
 - e.g. `alice`, `edward`, `james_bond_007`
- **Variables** are character strings that begin with a capital or a ‘ ’
 - e.g. `X`, `_G341`, `Stranger`, `The_Older...`
- **Clauses** begin by the positive literals, then the sign ‘:-’ and the sequence of negative literals separated by comma (‘,’’); it always end with a dot (‘.’’)

```
soeur(X, Y) :- femme(X),  
             parents(X, Mere, Pere),  
             parents(Y, Mere, Pere).
```

```
soeur(X, Y)  $\vee$   $\neg$  femme(X)  $\vee$   
  $\neg$  parents(X, Mere, Pere)  $\vee$   $\neg$  parents(Y, Mere, Pere).
```

```
femme(X)  $\wedge$  parents(X, Mere, Pere)  $\wedge$   
 parents(Y, Mere, Pere)  $\Rightarrow$  soeur(X, Y)
```

- **Mode “consult”**

```
femme(alice).  
femme(victoria).  
homme(albert).  
homme(edward).  
parents(edward, victoria, albert).  
parents(alice, victoria, albert).
```

Set of clauses

- **Mode “query”**

```
?- femme(alice).  
Yes  
?- homme(X).  
X = albert  
Yes  
?- femme(X), parents(X, U, I).  
X = alice  
U = victoria  
I = albert ;  
No
```

Data Bases in PROLOG

Atom Conjunction

• Mode “consult”

femme(alice).

femme(victoria).

homme(albert).

homme(edward).

parents(edward, victoria, albert).

parents(alice, victoria, albert).

**soeur(X, Y) :- femme(X), parents(X, Mere, Pere),
parents(Y, Mere, Pere).**

• Mode “query”

?- soeur(U, edward).

U = alice ;

No

?- soeur(U, V).

U = alice

V = edward ;

U = alice

V = alice ;

No

Set of Horn clauses

Deduction in
PROLOG

Atom conjunction

Definition: a **positive Horn clause** is a clause (i.e. A disjunction of literals) that owns one and only one positive literal

Examples:

Positive Horn Clauses

femme(victoria) .

homme(edward) .

parents(alice, victoria, albert) .

soeur(X, Y) :- femme(X), parents(X, Mere, Pere),
parents(Y, Mere, Pere) .

Are Horn clauses

\neg barber(X) \vee shave(X, Y) \vee shave(Y, Y) which means:

“the barbers shave those who don't shave by themselves” is not a Horn clause

\neg barber(X) \vee \neg shave(X, Y) \vee \neg shave(Y, Y) which means *“there is no barber who shaves someone who shaves himself”* is not a positive Horn clause

Declarative programming

- Description of predicate properties:

```
nombre(0).
```

```
nombre(s(X)) :- nombre(X).
```

```
addition(0, X, X).
```

```
addition(s(X), Y, s(Z)) :- addition(X, Y, Z).
```

- Call in query mode:

```
?- nombre(s(s(0))).
```

Yes

```
?- addition(s(s(0)), s(s(s(0))) Z).
```

Output

```
Z = s(s(s(s(s(0)))))
```

Yes

Result



- **Mode “consult”**

nombre(0). **nombre(0) →** Set of Horn clauses = Rules

nombre(s(X)) :- nombre(X). **nombre(s(X)) → nombre(x)**

addition(0, X, X). **addition(0, X, X) →**

addition(s(X), Y, s(Z)) :- addition(X, Y, Z).

addition(s(X), Y, s(Z)) → addition(X, Y, Z)

- **Mode “query”**

Atom conjunction

?- nombre(s(s(s(0))).

?- addition(s(s(0)), s(s(s(0))), X).

- **Work of the Interpreter**

*The query is rewritten using rules
until it becomes empty.*

PROLOG for
“babies”

PROLOG strategy

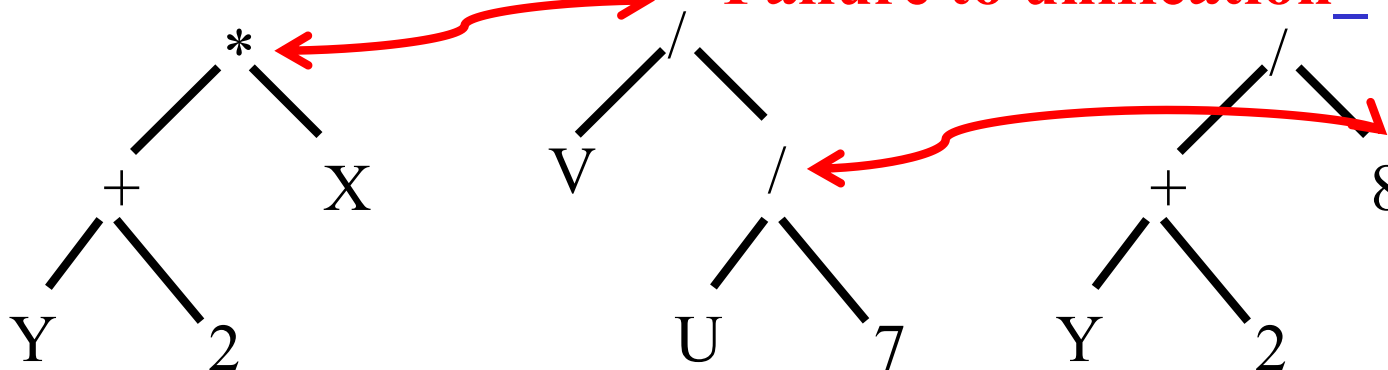
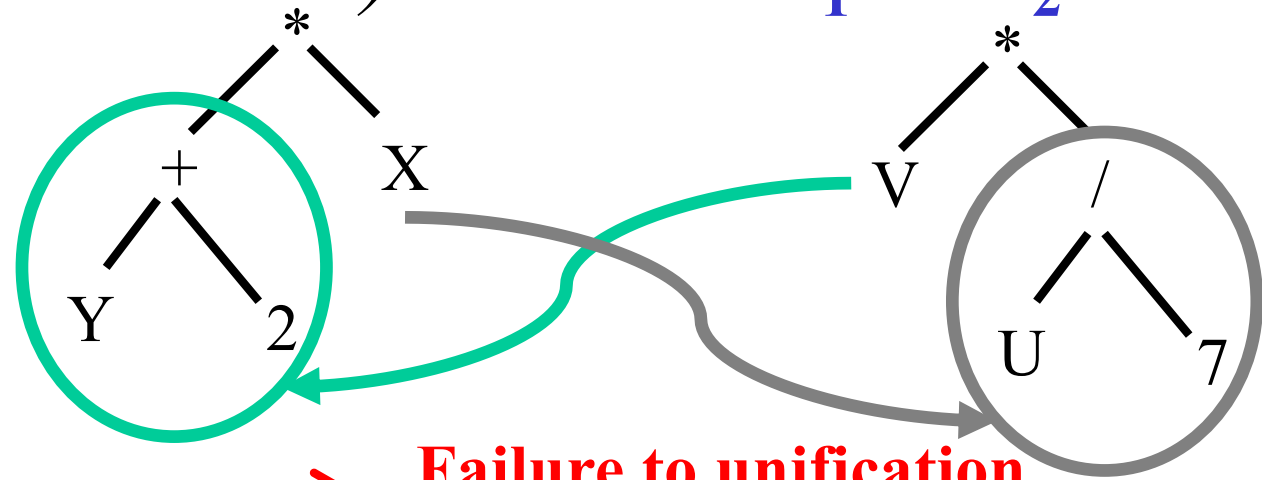
- PROLOG try to apply the “rules” (i.e. the clause) on the atoms of the “query”
- PROLOG makes use of the “unification” mechanism
- The atoms of the “query” are solved sequentially, from left to right.
- The clauses are considered one after one, from the first until the last.



Unification

Unify: to make unique

Unification: two terms t_1 and t_2 unify if and only if there exists a substitution σ (*i.e. a list of variable replacements*) such that: $t_1\sigma = t_2\sigma$



Example: addition

```
:- addition(s(s(0)), s(s(s(0))), z)
```

C1

C2

```
:- addition(s(0), s(s(s(0))), z1)
```

C1

C2

```
:- addition(0, s(s(s(0))), z2)
```

C1

C2



C1: addition(0, X, X).

C2: addition(s(X), Y, s(Z)) :- addition(X, Y, Z).

Function Inversion

- Description of predicate properties:

```
nombre(0).
```

```
nombre(s(X)) :- nombre(X).
```

```
addition(0, X, X).
```

```
addition(s(X), Y, s(Z)) :- addition(X, Y, Z).
```

- Call in query mode:

```
?- addition(s(s(0)), U, s(s(s(s(s(0)))))).
```

```
U = s(s(s(0)))
```

```
Yes
```

```
?- nombre(X).
```

```
X = 0 ;
```

```
X = s(0) ;
```

```
X = s(s(0)) ;
```

```
X = s(s(s(0))) ;
```

```
X = s(s(s(s(0))))
```



How PROLOG interprets clauses?

C1: femme(alice).

C2: femme(victoria).

C3: homme(albert).

C4: homme(edward).

C5: parents(edward, victoria, albert).

C6: parents(alice, victoria, albert).

C7: soeur(X, Y) :- femme(X),
parents(X, Mere, Pere),
parents(Y, Mere, Pere).

- **Mode “consult”**

nombre(0). **nombre(0) → Set of Horn clauses = Rules**

nombre(s(X)) :- nombre(X). **nombre(s(X)) → nombre(x)**

addition(0, X, X). **addition(0, X, X) →**

addition(s(X), Y, s(Z)) :- addition(X, Y, Z).

addition(s(X), Y, s(Z)) → addition(X, Y, Z)

- **Mode “query”**

Atom conjunction

?- nombre(s(s(s(0))).

?- addition(s(s(0)), s(s(s(0))), X).

- **Work of the Interpreter**

The query is rewritten using rules until it becomes empty.

PROLOG for
“babies”

PROLOG strategy

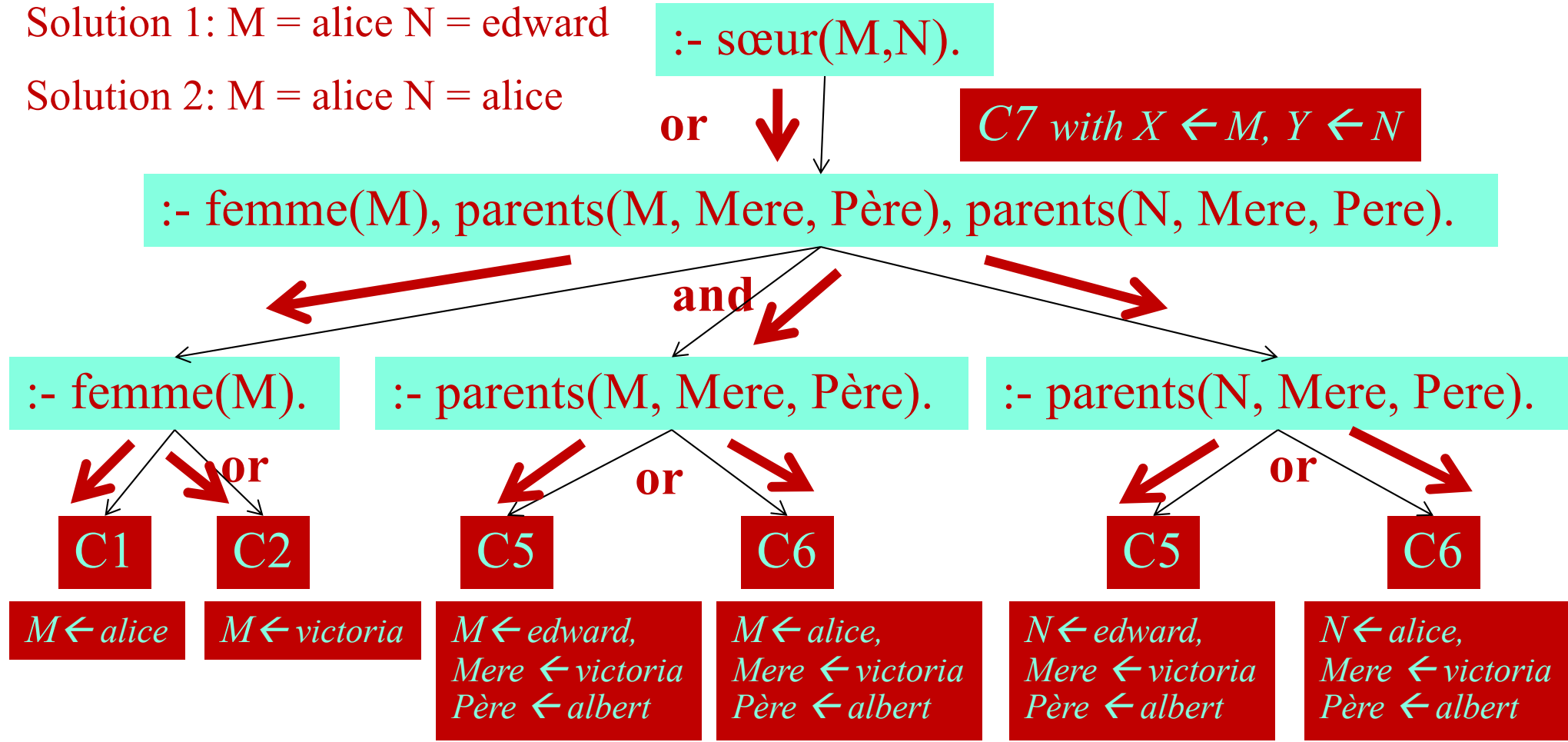
- PROLOG try to apply the “rules” (i.e. the clause) on the atoms of the “query”
- PROLOG makes use of the “unification” mechanism
- The atoms of the “query” are solved sequentially, from left to right.
- The clauses are considered one after one, from the first until the last.



Procedural interpretation: and/or trees

Solution 1: M = alice N = edward

Solution 2: M = alice N = alice

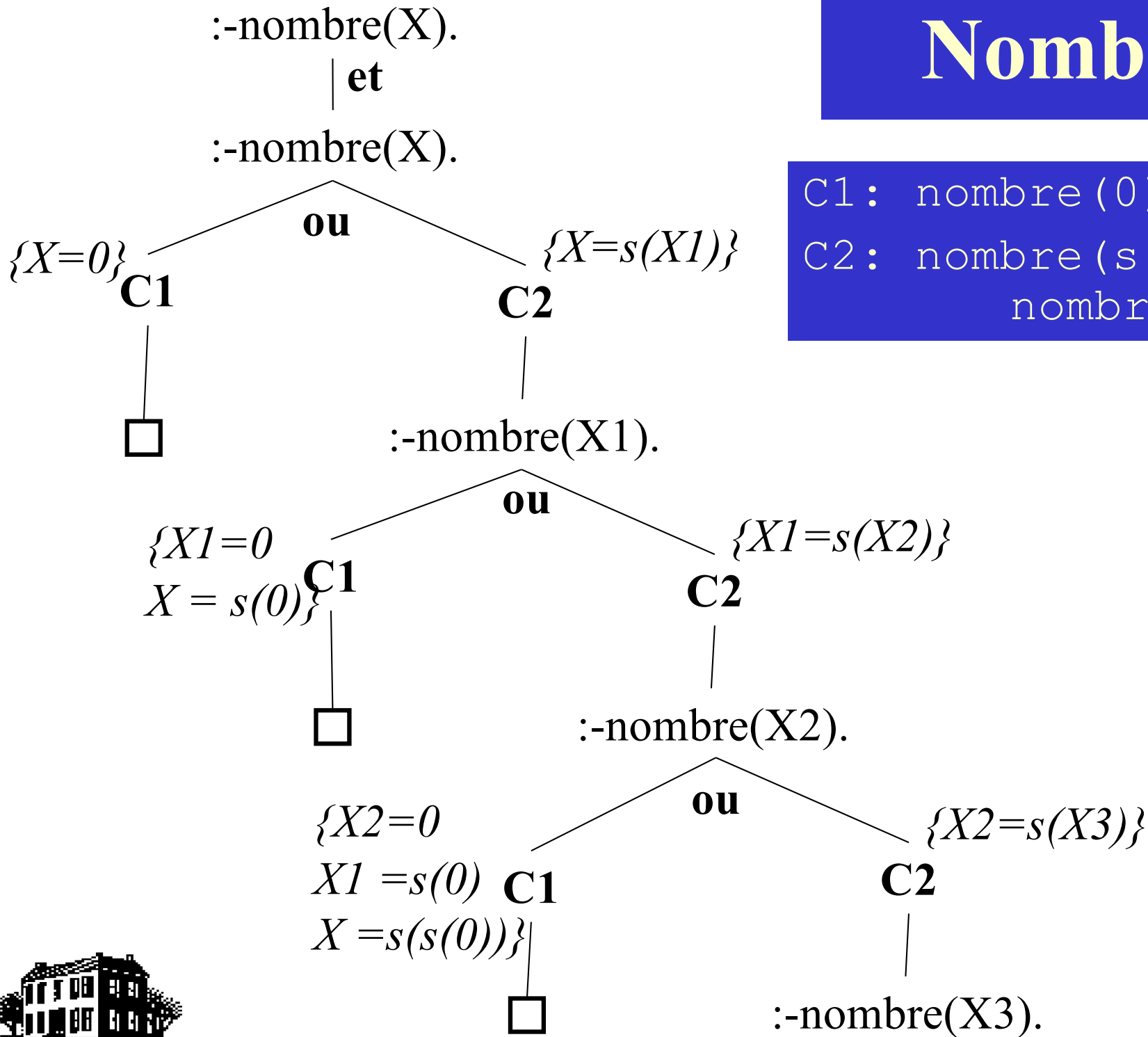


- C1: femme(alice).
- C2: femme(victoria).
- C3: homme(albert).
- C4: homme(edward).

- C5: parents(edward, victoria, albert).
- C6: parents(alice, victoria, albert).
- C7: soeur(X, Y) :- femme(X),
 parents(X, Mere, Pere), parents(Y, Mere, Pere).

Nombre

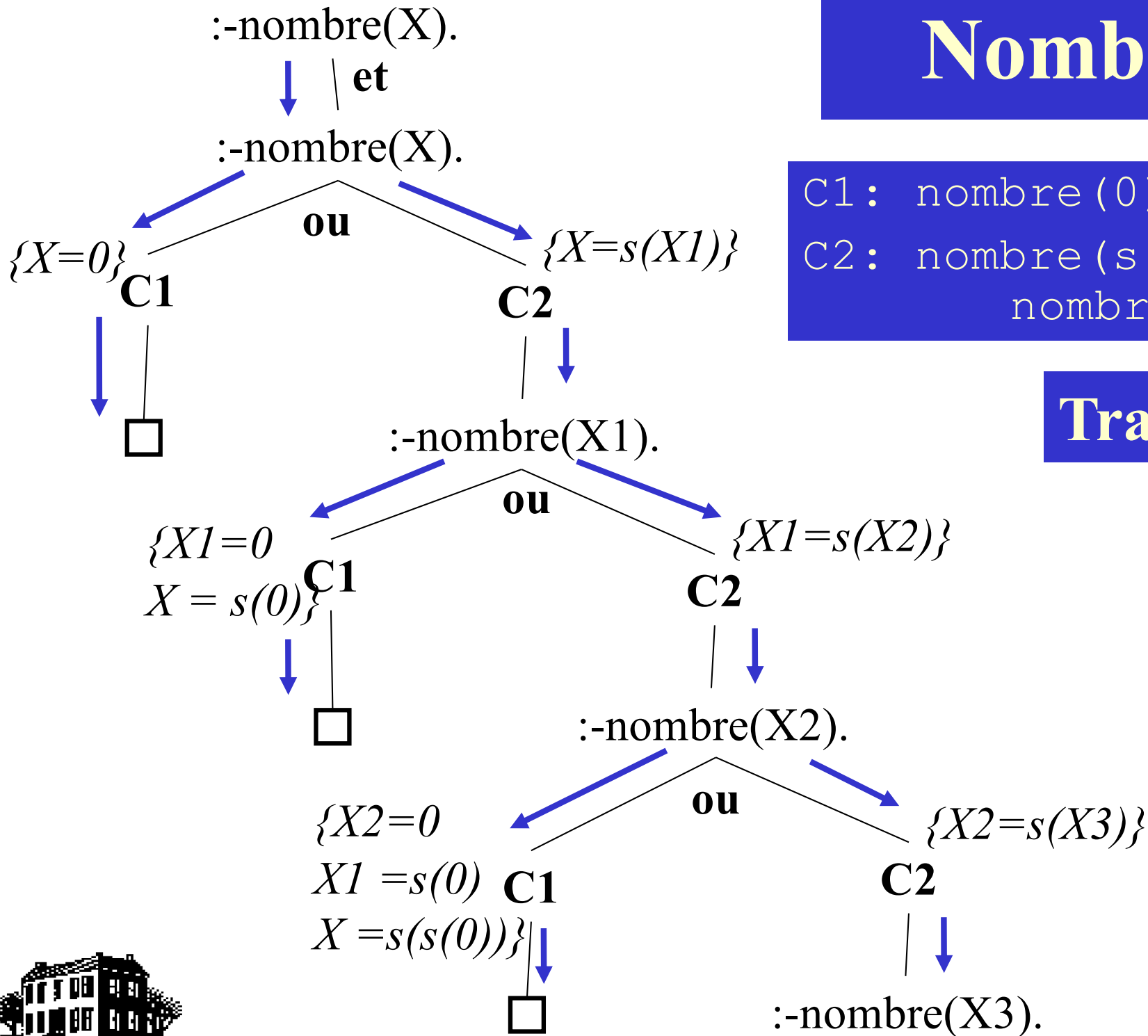
C1 : nombre (0) .
C2 : nombre (s (X)) :-
 nombre (X) .



Nombre

C1 : nombre (0) .
C2 : nombre (s (X)) :-
nombre (X) .

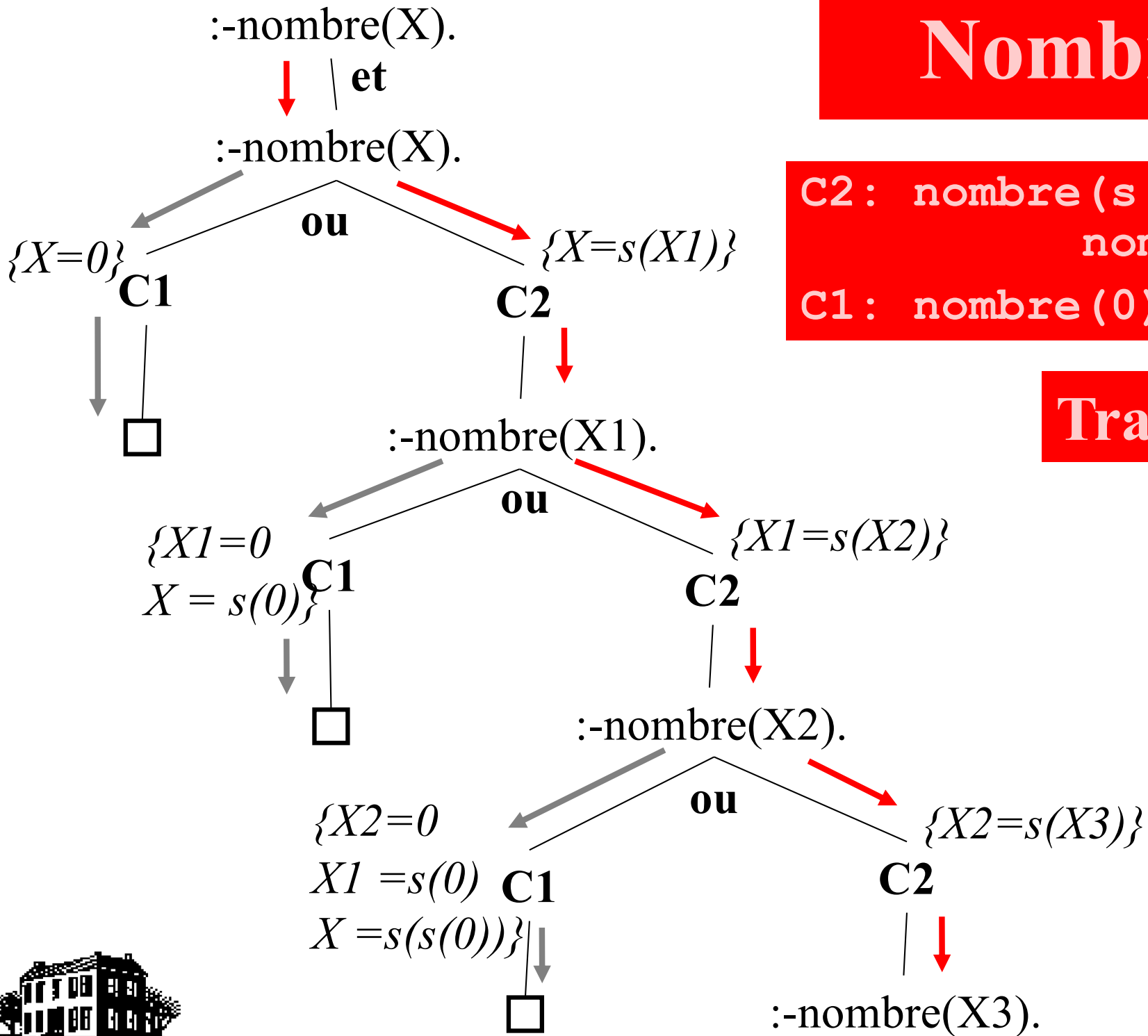
Travel 1



Nombre

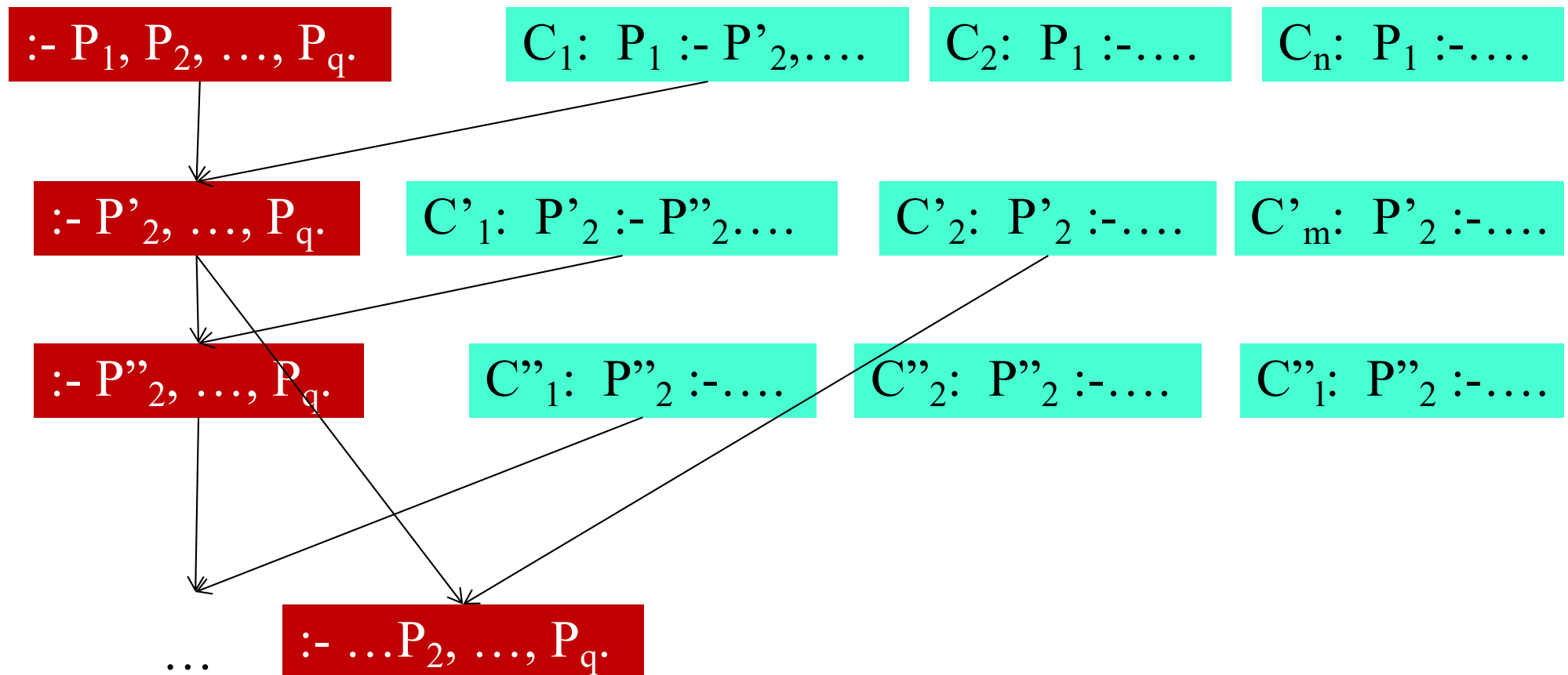
C2 : nombre (s (X)) :-
 nombre (X) .
C1 : nombre (0) .

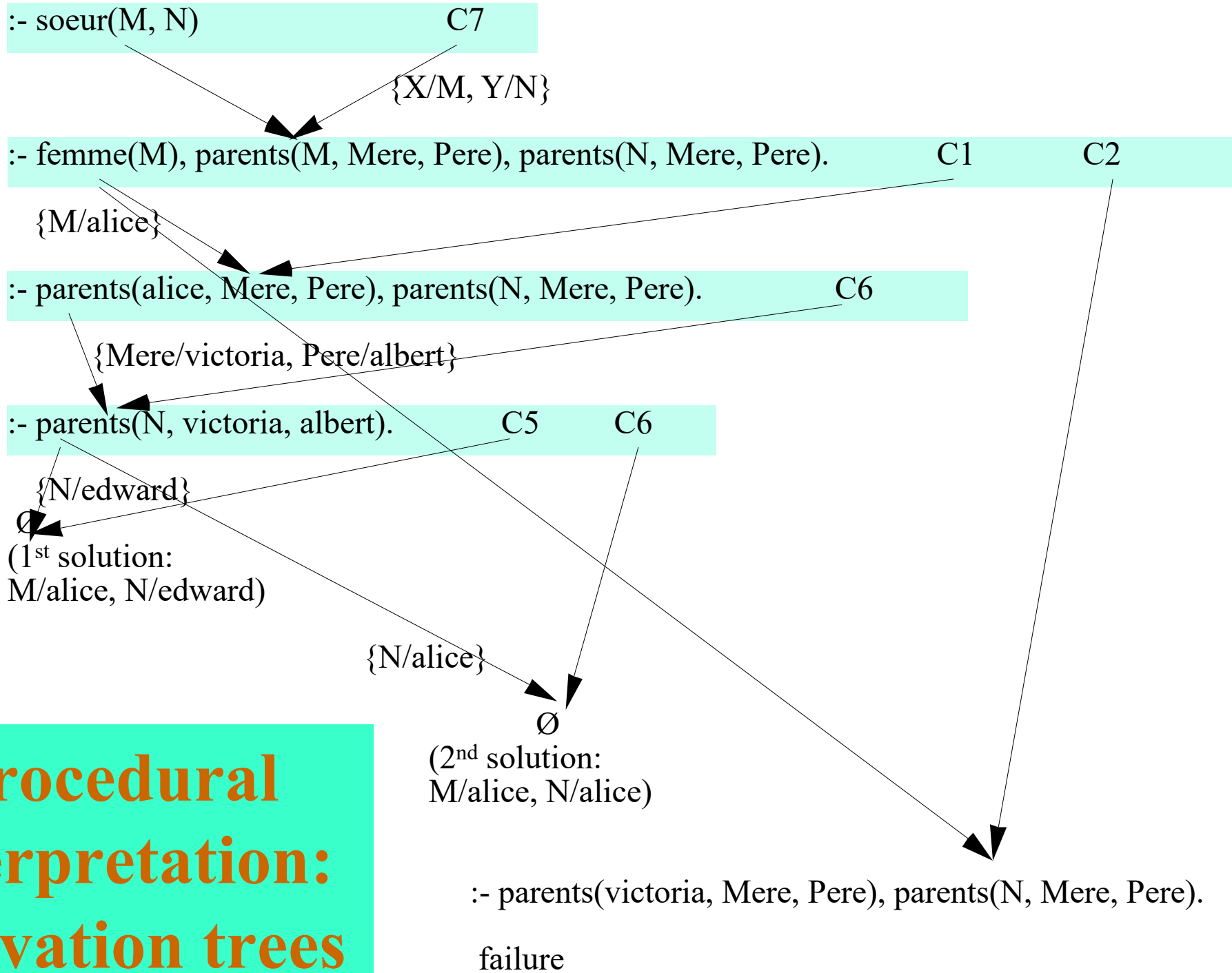
Travel 2



Procedural interpretation: *derivation tree*

Query

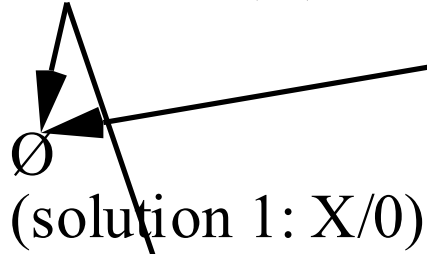




Procedural interpretation: derivation trees

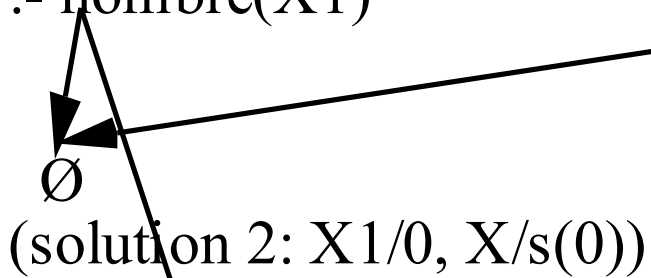
Nombre

`:- nombre(X).` `nombre(0)`



`nombre(s(X1)):- nombre(X1).`

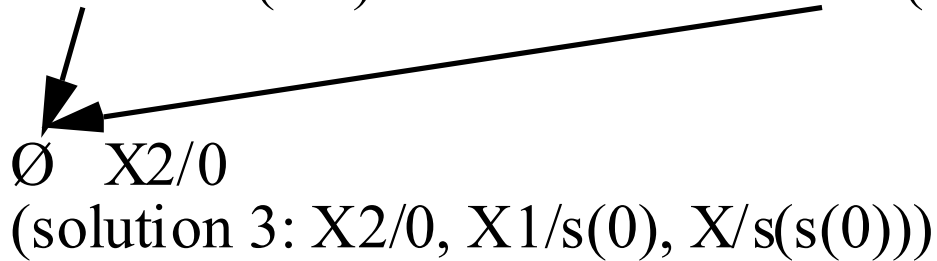
`:- nombre(X1)` `X/0 X/s(X1)`
`nombre(0)`



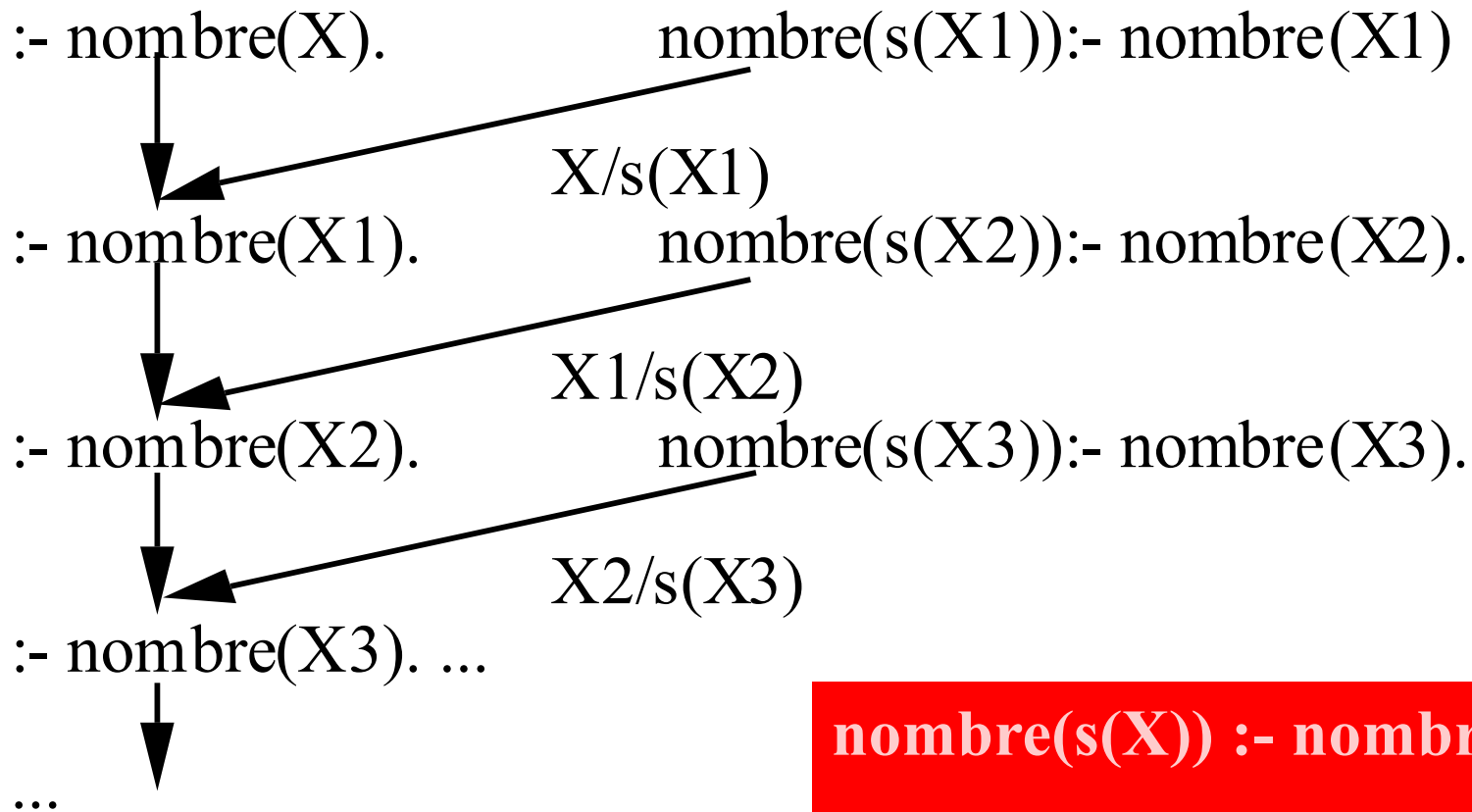
`nombre(0).`
`nombre(s(X)) :- nombre(X).`

`nombre(s(X2)):- nombre(X2).`

`:- nombre(X2)` `X1/0 X1/s(X2)`
`nombre(0) ...`



Nombre (2)



`nombre(s(X)) :- nombre(X).`
`nombre(0).`



Program

- A program is a set of clauses of which head literal has the same predicate name and the same arity

- Examples

- Program addition: *[addition(X, Y, Z) means $X+Y=Z$]*

```
addition(0, X, X).
```

```
addition(s(X), Y, s(Z)) :- addition(X, Y, Z).
```

- Program multiplication: *[multiply(X, Y, Z) means $Z=X.Y$]*

```
multiply(0, X, 0).
```

```
multiply(s(N), P, R) :- multiply(N, P, Q),  
                        addition(P, Q, R).
```

- Programme factorial: *[fact_s(N, P) means $P = N!$]*

```
fact_s(0, s(0)).
```

```
fact_s(s(0), s(0)).
```

```
fact_s(s(N), R) :- fact_s(N, Q),  
                  multiply(s(N), Q, R).
```



Procedure call

- A call to a procedure is an atom in the “query” mode:

Examples :

```
?- fact_s(s(s(s(0))), U).
```

```
U = s(s(s(s(s(s(0))))))
```

Yes

```
?- multiply(s(s(0)), s(s(s(0))), R).
```

```
R = s(s(s(s(s(s(0))))))
```

Yes

```
?- multiply(s(s(0)), H, s(s(s(s(s(s(0)))))).
```

```
H = s(s(s(0)))
```

Yes

Output

Input



Example addition and/or tree

`:- addition(s(s(0)), s(s(s(0))), Z)`

C2 $\{X1 = s(0),$
 $Z = s(Z1)\}$

`:- addition(s(0), s(s(s(0))), Z1)`

C2 $\{X2 = 0,$
 $Z = s(Z1) = s(s(Z2))\}$

`:- addition(0, s(s(s(0))), Z2)`

C1 $\{Z2 = s(s(s(0)))\}$

1st solution:
 $Z = s(s(Z2)) = s(s(s(s(0))))$



C1: `addition(0, X, X).`

C2: `addition(s(X), Y, s(Z)) :- addition(X, Y, Z).`

Example addition: derivation tree

`:- addition(s(s(0)), s(s(s(0))), z)`

C1

C2

`:- addition(s(0), s(s(s(0))), z1)`

C1

C2

`:- addition(0, s(s(s(0))), z2)`

C1

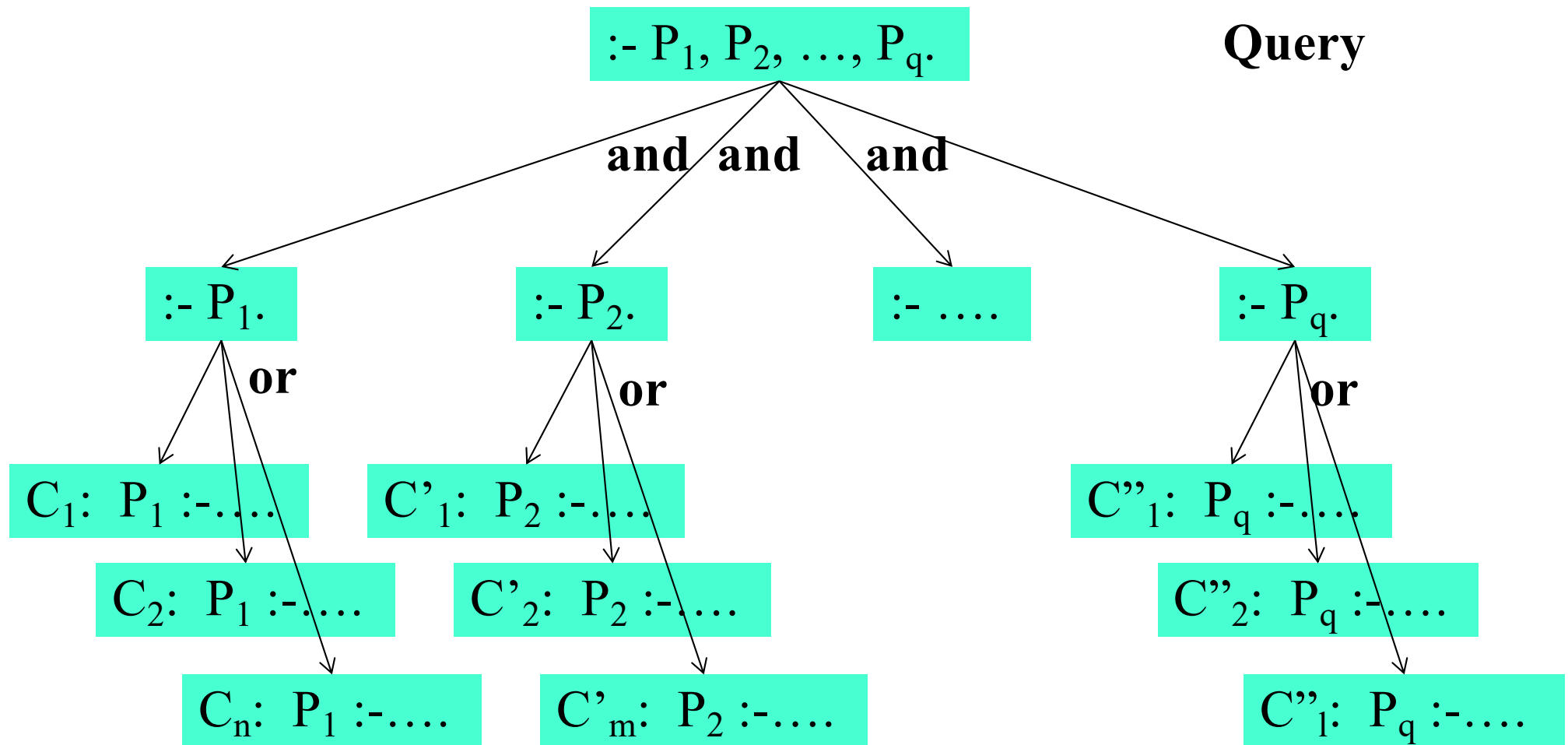
C2



C1: `addition(0, X, X).`

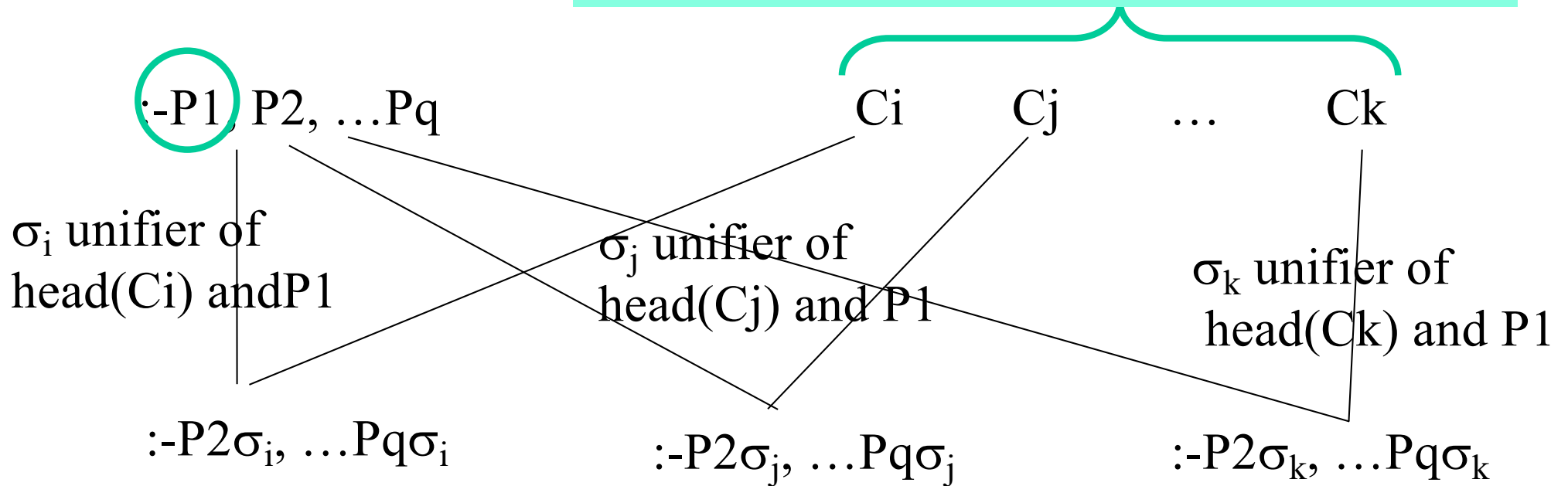
C2: `addition(s(X), Y, s(Z)) :- addition(X, Y, Z).`

Procedural interpretation: and/or tree



Procedural interpretation: derivation tree

Clauses of which head literal unify with P1



Data structures

- Terms:
 - Variables
 - Function with arguments: $f(a, g(h, I), K)$
- Atoms:
 - Character strings (beginning with a minuscule)
 - Numbers
- Lists:
 - Empty lists, list sequences
- Strings
- Numbers



List structure

- Defined by the *constructor*, **cons** and by the *empty list*, **nil**
 - Form: **cons**(<head>, <queue>)
 - <head> is any literal
 - <queue> is a list

Examples:

- The list (a b c) can be represented by:
cons(a, cons(b, cons(c, nil)))
- The tree (+ (- b c) (* d e)) can be represented by:
**cons(+, cons(cons(-, cons(b, cons(c, nil))),
cons (cons(*, cons(d, cons(e, nil))),
nil),
nil),
nil)**



List structure (2)

- Simplification:
 - The list (a b c) is written [a, b, c], which means $\text{cons}(a, \text{cons}(b, \text{cons}(c, \text{nil})))$
 - In the same way, (+ (- b c) (* d e)) is written [+, [-, b, c], [*, d, e]]
- List constructor: '|'

Example: the list (a, b, c) is written:

- [a, b, c]
- [a | [b, c]]
- [a | [b | [c | []]]]
- [a, b | [c]]



Program example

test if an element belong to a list

- Program « **appartient** » with terms:

```
appartient_t(A, cons(A, B)).  
appartient_t(A, cons(B,C)) :-  
    appartient_t(A, C).
```

- Program « **appartient** » with PROLOG representation :

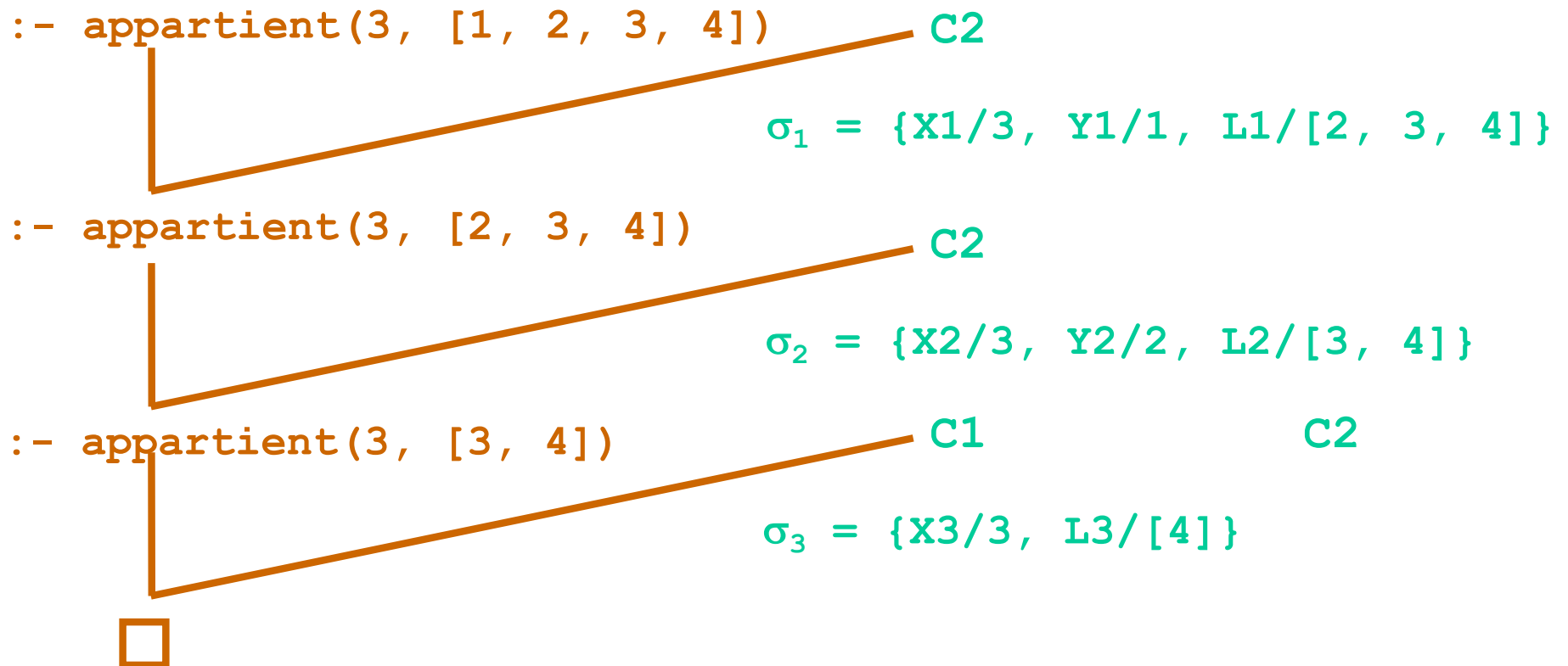
```
appartient0(A, [A|B]).  
appartient0(A, [B|C]) :- appartient0(A, C).
```



Execution

C1: appartient(X, [X|L]).

C2: appartient(X, [T|L]) :- appartient(X, L).



Other examples with lists

Concatenation of two lists:

```
concat([], M, M).
```

```
concat([A|L], M, [A|N]) :- concat(L, M, N).
```

List inversion:

```
miroirnaif([], []).
```

```
miroirnaif([A|Q], R) :- miroirnaif(Q, P),  
                        concat(P, [A], R).
```



Execution

C1: `concat([], M, M).`

C2: `concat([A|L], M, [A|N]) :- concat(L, M, N).`

`:- concat([a, b], [c, d, e], R)`

C2

$\sigma_1 = \{A1/a, L1/[b], M1/[c, d, e], R/[A1|N1]\}$

`:- concat([b], [c, d, e], N1)`

C2

$\sigma_2 = \{A2/b, L2/[], M2/[c, d, e], N1/[A2|N2]\}$

`:- concat([], [c, d, e], N2)`

C1

$\sigma_3 = \{N2=M3=[c, d, e]\}$



$R = [A1|N1] = [a|[A2|N2]] = [a | [b | [c, d, e]]] = [a, b, c, d, e]$