Lecture notes 5.1

Datalog

COMP 2411, session 1, 2004

Atoms, integers and variables

The fact "`person(socrates).`", the fact "`sum(2,3,Y).`" and the rule "`mortal(X) :- person(X).`" are built from the following sequences of symbols:

- atoms: "`person`", "`socrates`", "`sum`", "`mortal`" and "`:-`"
- integers: "2" and "3"
- variables: "X" and "Y"

plus some special characters: "`(`", "`)`", "`.`" and "`,`".

A constant is either an atom or an integer.

Atoms

An atom names specific individuals (like "`socrates`"), or nonlogical properties or relationships (like "`mortal`"), or logical properties or relationships (like "`:-`" which means *if*).

Formally, an atom is either:

- a sequence of alphanumeric characters and the underscore symbol that begin with a lower-case letter, like "`george_w_bush`", or "`agent_007`", or
- a sequence of signs, like "`@-->`" or "`:-`", or
- a sequence of alphanumeric characters, the underscore symbol and signs enclosed in single quotes, like "`'a#23&'`"

where a sign is any of:

```
+ - * / \ ~ ^ < > : . ? @ # $ &
```


Variables

A variable denotes an unspecified individual.

Formally, a variable is a sequence of alphanumeric characters and the underscore symbol that begins with an upper-case letter or the underscore.

The following are examples of variables:

```
X  R10  George  _  _R10
```

Structures

"`person(socrates)`", "`sum(2,3,Y)`" and "`mortal(father(X))`" are called compound terms or structures, built from functors and components or arguments.

- In "`person(socrates)`", "`person`" is the functor and "`socrates`" is the argument.
- In "`sum(2,3,Y)`", "`sum`" is the functor and "2", "3" and "Y" are the arguments.
- In "`mortal(father(X))`", "`mortal`" is the functor and "`father(X)`" is the argument, which is itself a compound term built from the functor "`father`" and the argument "X".

Operators

Operators offer an alternative syntax to functors for writing structures.

- "+" is a functor in "`+(2,3)`", but an infix operator in the alternative syntax "2+3".
- "`-`" is a functor in "`-(4)`", but an prefix operator in the alternative syntax "`-4`".
- It is also possible to write functors as postfix operators.

You can declare or redefine operators at will:

```
:- op(650, xfy, +),     % exclusive or
:- op(650, xfy, <->),   % equivalence
:- op(640, xfy, ->),    % implication
```


Datalog programs

Datalog programs are simple Prolog programs: the arguments of any structure in a Datalog program are either constants or variables (not structures).

Example (1)

Consider the following statements.

- Everybody is a descendant of Gaia.
- Poseidon and Zeus are offsprings of Cronus, and Hermes is an offspring of Zeus.
- Someone's offspring is a descendant of that person.
- A descendant of someone's offspring is a descendant of that person.
- Everyone Hermes is a descendant of is a god.
- Poseidon is a god.

Example (2)

They can be expressed as a Datalog program:

```
descendant(gaia,_).
descendant(X,Y) :- offspring(X,Y).
descendant(X,Z) :- offspring(X,Y),
                   descendant(Y,Z).

offspring(cronus,poseidon).
offspring(cronus,zeus).
offspring(zeus,hermes).

god(X) :- descendant(X,hermes).
god(poseidon).
```

- "`descendant/2`" is defined from 1 fact and 2 rules.
- "`offspring/2`" is defined from 3 facts.
- "`god/1`" is defined from 1 rule and 1 fact.

Simple queries

Answering a simple query requires solving a single goal. A simple query can contain variables.

```
?- god(cronus).

Yes
?- descendant(cronus,X).

X = poseidon ;

X = zeus ;

X = hermes ;

No
?- descendant(gaia,X).

X = _G152 ;

No
```


Complex queries

Answering complex queries requires solving many goals.

Thanks to a complex query, we can find out who is a god *and* a descendant of Cronus:

```
?- god(X), descendant(cronus,X).

X = zeus ;

X = poseidon ;

No
?- descendant(cronus,X), god(X).

X = poseidon ;

X = zeus ;

No
```


Computations (1)

The simplest computation takes place when answering queries like:

```
?- offspring(cronus,zeus).

Yes
?- offspring(cronus,hermes).

No
```

Prolog proceeds by pattern matching. In this case, the query does not match the head of any rule, and matching a fact is equivalent to being the same as this fact.

Prolog tries every clause in the program starting from the first one, and going down the list. The answer is `yes` when a match is found, and `no` otherwise.

Computations (2)

The next level of complexity takes place when answering simple queries containing variables, but that can only be matched against a fact.

A match can be found by matching any variable with any term. If the same variable occurs many times in a goal, it should obviously be bound to the same term.

```
?- offspring(cronus,X).
X = poseidon ;
X = zeus ;
No
?- offspring(X,X).
No
```


Computations (3)

Of course, distinct variables can (but *do not have to*) be matched against distinct terms :

```
?- offspring(X,Y).
X = cronus
Y = poseidon ;
X = cronus
Y = zeus ;
X = zeus
Y = hermes ;
No
```


Computations (4)

Note how anonymous variables can be used to express *there exists*:

```
?- offspring(_,X).
X = poseidon ;
X = zeus ;
X = hermes ;
No
```

Many occurrences of _ denote distinct anonymous variables:

```
?- offspring(_,_).
Yes
```

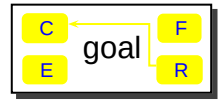

Computations (4)

Things become interesting with simple queries that can be matched against the head of a rule.

Answering the query then amounts to solving a sequence of goals, starting with the query itself.

To trace the computation, we assign four ports to each goal to represent the flow of control though the goal:

call (C), exit (E), fail (F) and redo (R).



Computations (5)

- The goal is called via the call port.
- If the computation succeeds then the goal is exited via the exit port.
- If the computation fails then the goal is exited via the fail port.
- When alternative matches are tried, the redo port is entered. This can happen:
  - 'internally,' before the flow of control has gone through the exit port;
  - 'externally', after the flow of control has gone through the exit port

Unification, backtracking

The process of pattern matching is called unification.

Once a variable X has been unified with a term t_1, it cannot be unified with another term t_2.

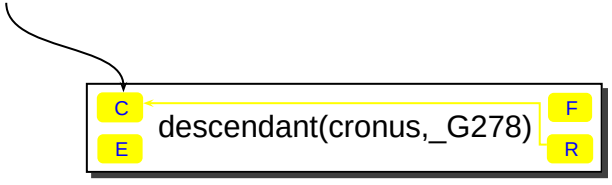
A variable in Prolog cannot be overwritten.

First the bind between X and t_1 has to be removed. Only then can X be unified with t_2.

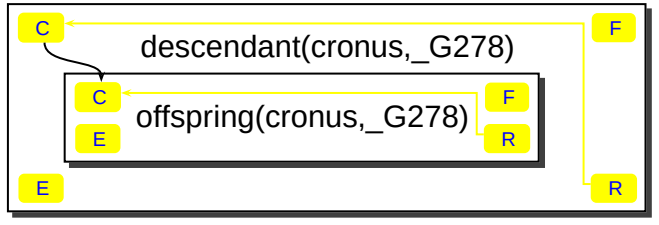
Unbinding happens every time Prolog searches for alternative solutions. This process is called backtracking.

We now illustrate these concepts on an example, tracing the first steps of the computation when solving the query `descendant(cronus,X)`.

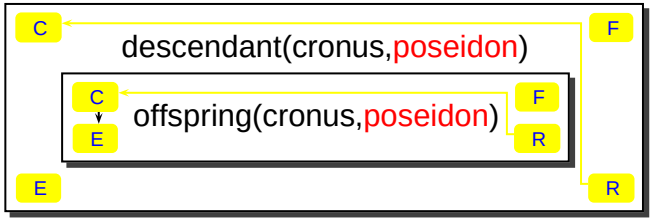
[0] CALL: descendant(cronus,_G278)



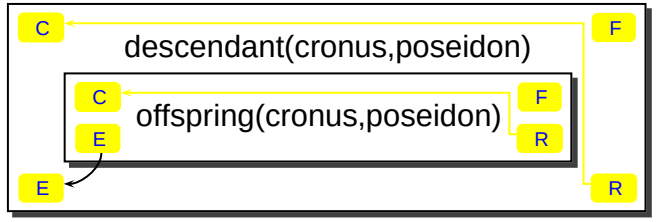
[1] CALL: offspring(cronus,_G278)



[1] EXIT: offspring(cronus,poseidon)

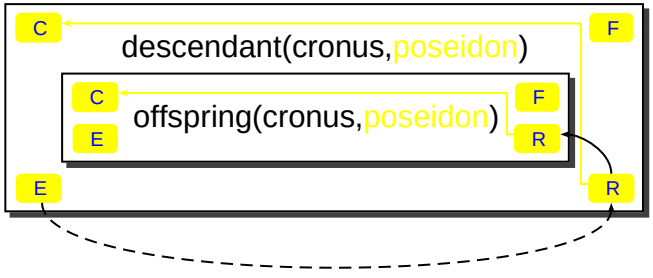


[0] EXIT: descendant(cronus,poseidon)



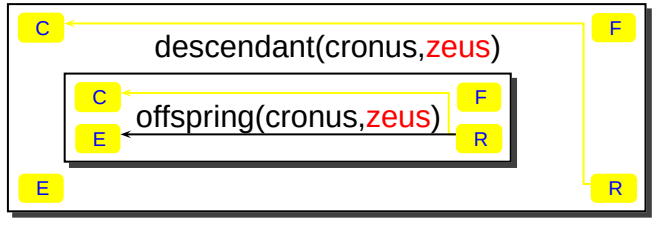
X=poseidon ;

[1] REDO: offspring(cronus,poseidon)



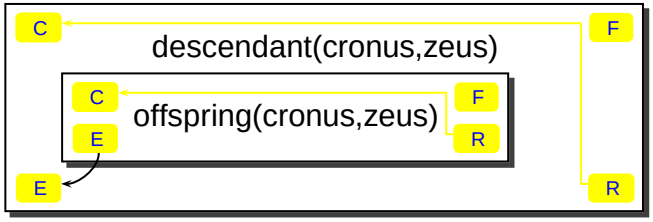
X=poseidon ;

[1] EXIT: offspring(cronus,zeus)



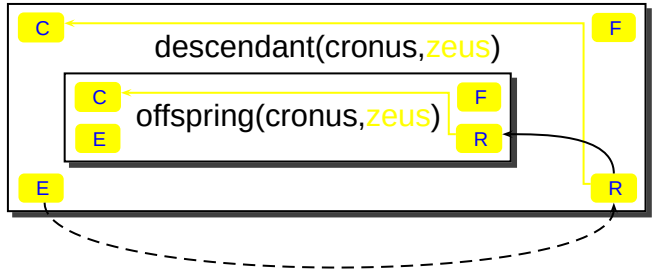
X=poseidon ;

[0] EXIT: descendant(cronus,zeus)



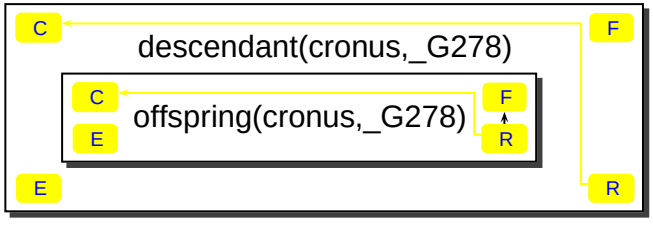
X=poseidon ; X=zeus ;

[1] REDO: offspring(cronus,zeus)



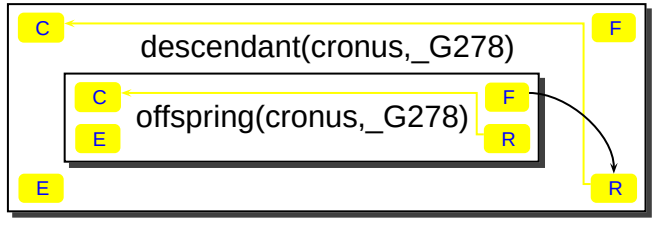
X=poseidon ; X=zeus ;

[1] FAIL: offspring(cronus,_G278)



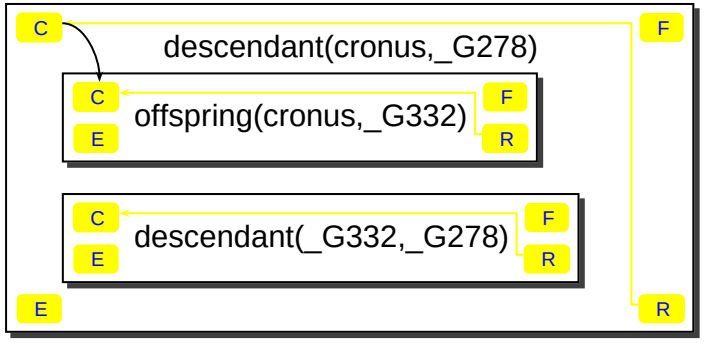
X=poseidon ; X=zeus ;

[0] REDO: descendant(cronus,_G278)



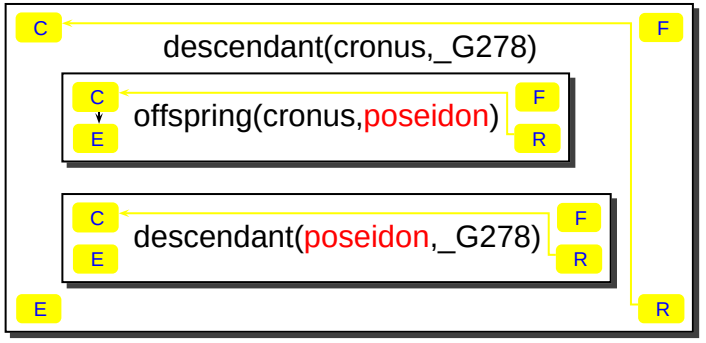
X=poseidon ; X=zeus ;

[1] CALL: offspring(cronus,_G332)



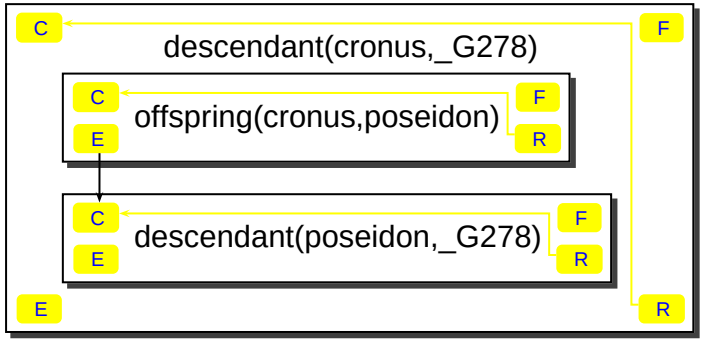
X=poseidon ; X=zeus ;

[1] EXIT: offspring(cronus,poseidon)



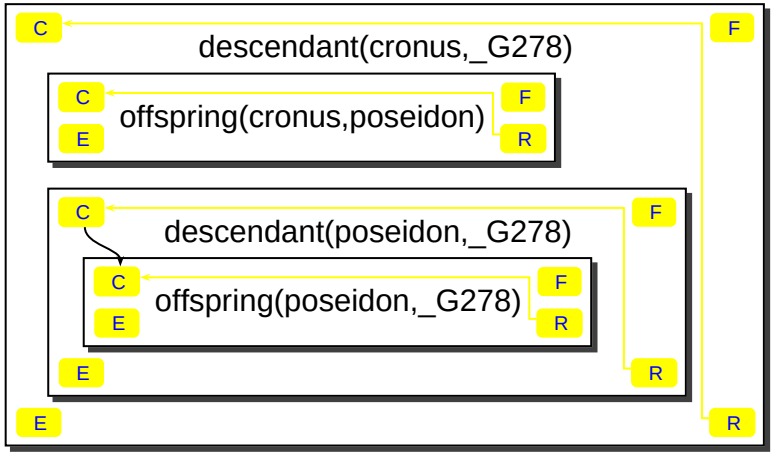
X=poseidon ; X=zeus ;

[1] CALL: descendant(poseidon,_G278)



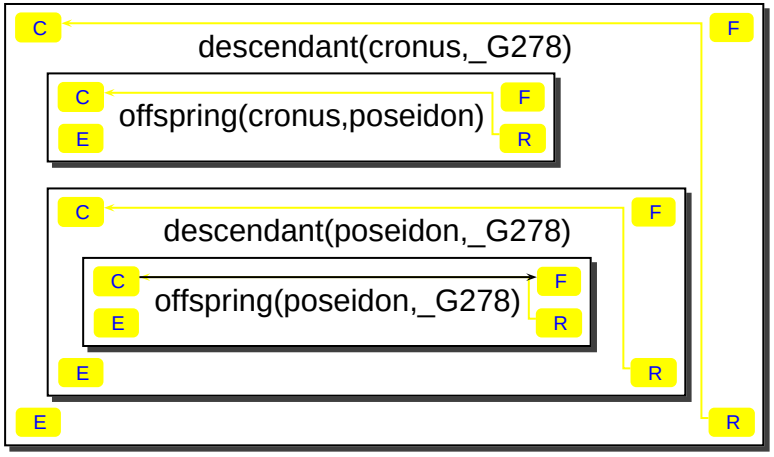
X=poseidon ; X=zeus ;

[2] CALL: offspring(poseidon,_G278)



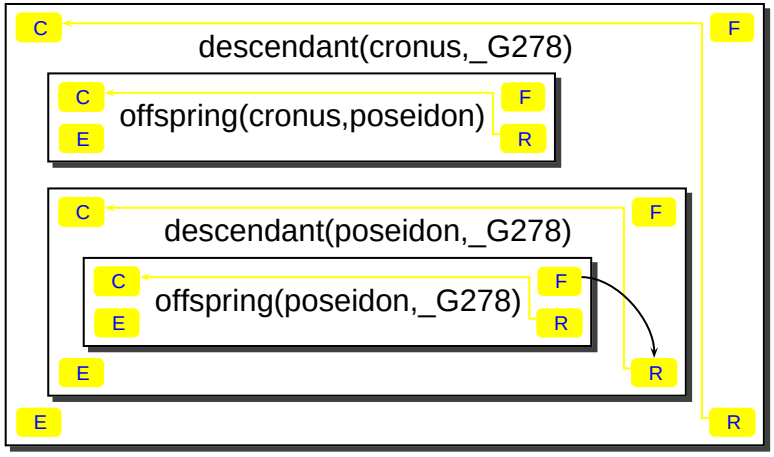
X=poseidon ; X=zeus ;

[2] FAIL: offspring(poseidon,_G278)



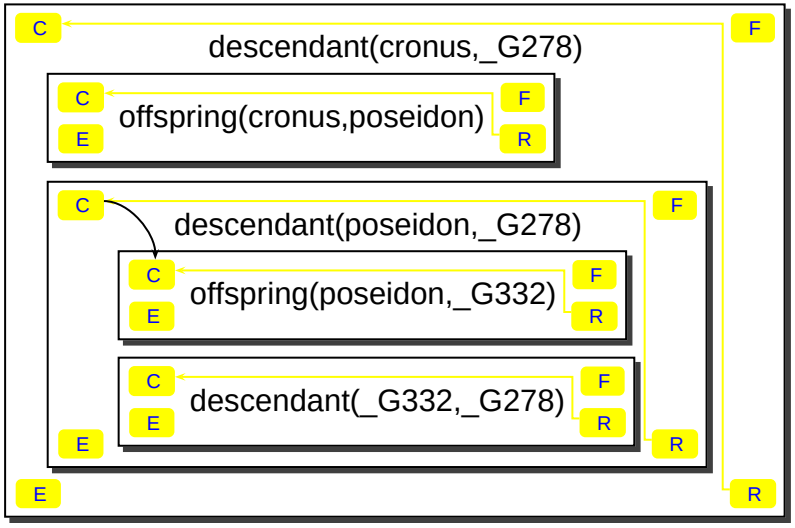
X=poseidon ; X=zeus ;

[1] REDO: descendant(poseidon,_G278)



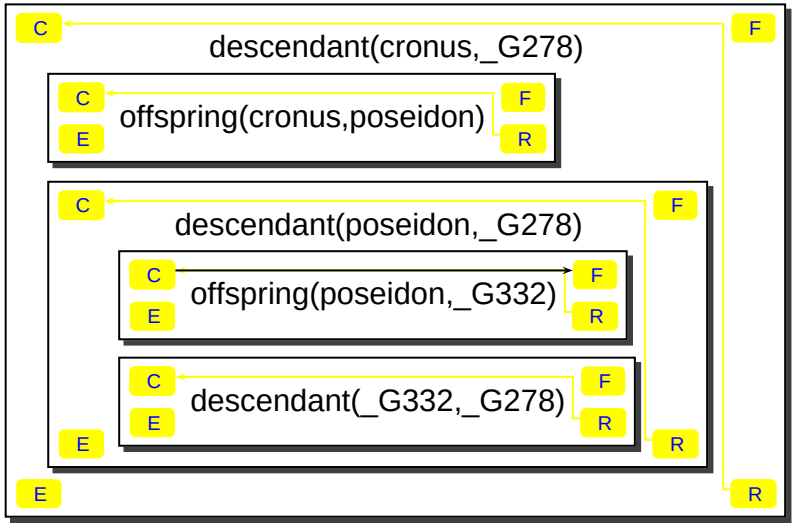
X=poseidon ; X=zeus ;

[2] CALL: offspring(poseidon,_G332)



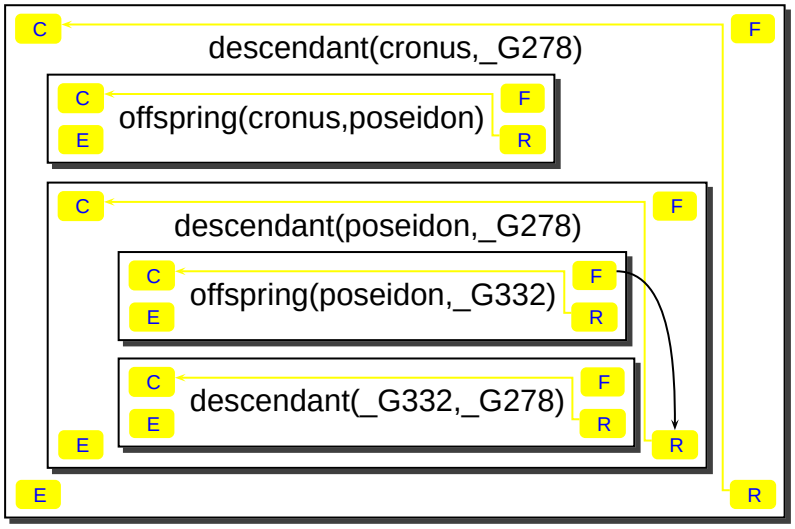
X=poseidon ; X=zeus ;

[2] FAIL: offspring(poseidon,_G332)



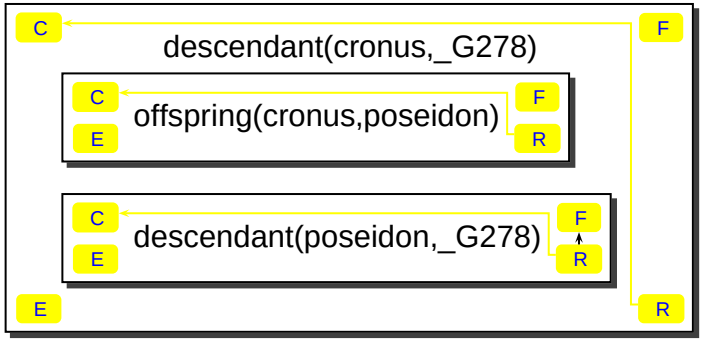
X=poseidon ; X=zeus ;

[1] REDO: descendant(poseidon,_G278)



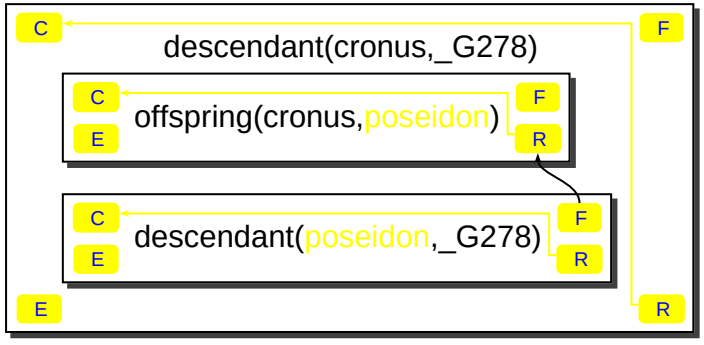
X=poseidon ; X=zeus ;

[1] FAIL: descendant(poseidon,_G278)



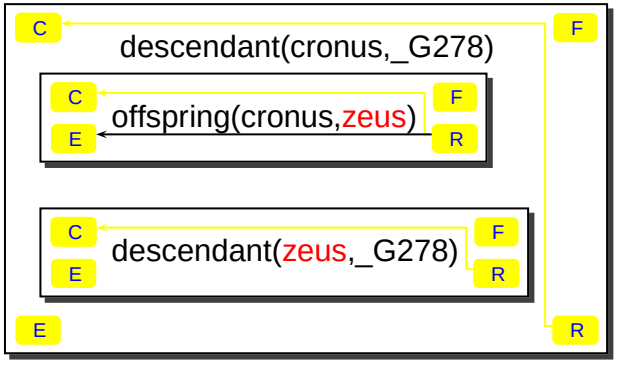
X=poseidon ; X=zeus ;

[1] REDO: offspring(cronus,poseidon)



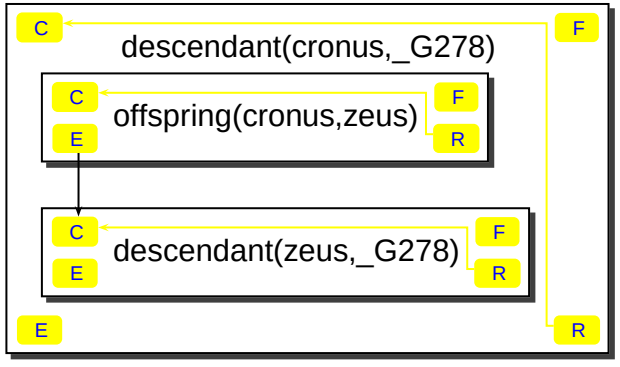
X=poseidon ; X=zeus ;

[1] EXIT: offspring(cronus,zeus)



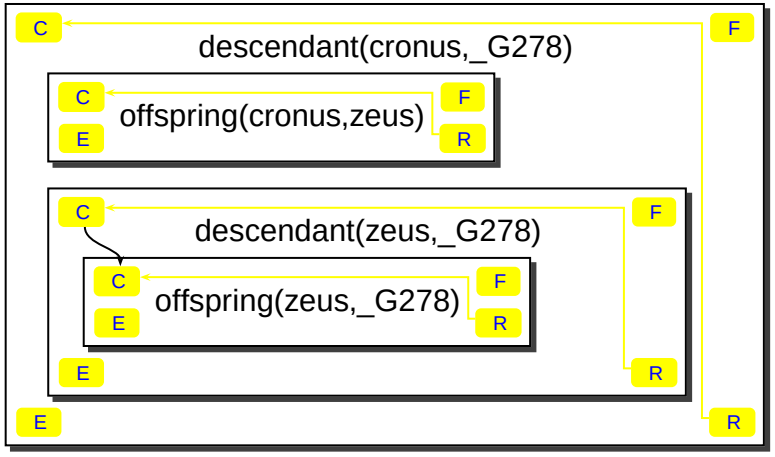
X=poseidon ; X=zeus ;

[1] CALL: descendant(zeus,_G278)



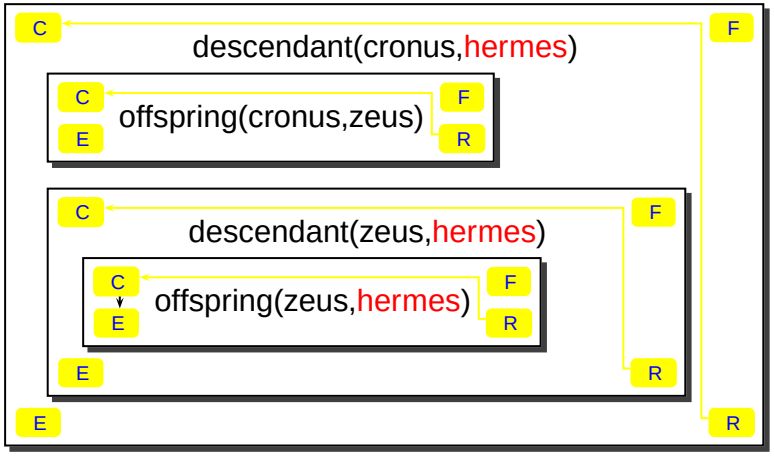
X=poseidon ; X=zeus ;

[2] CALL: offspring(zeus,_G278)



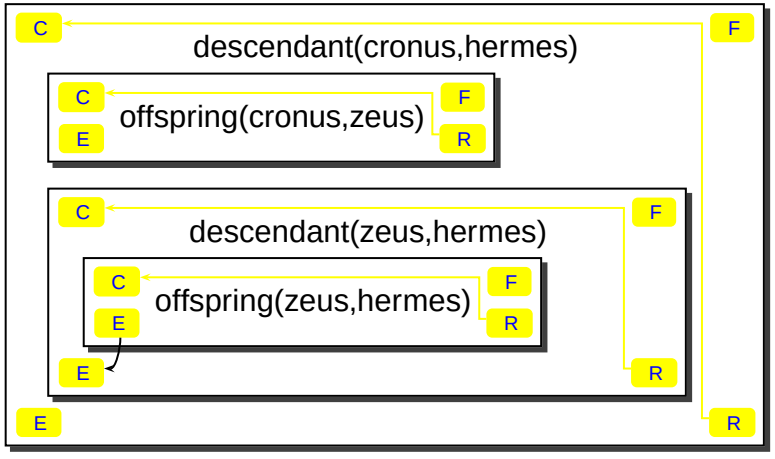
X=poseidon ; X=zeus ;

[2] EXIT: offspring(zeus,hermes)



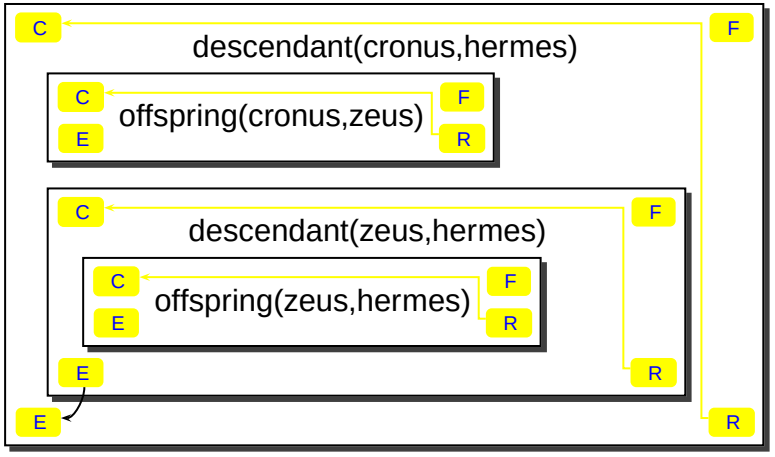
X=poseidon ; X=zeus ;

[1] EXIT: descendant(zeus,hermes)



X=poseidon ; X=zeus ;

[0] EXIT: descendant(cronus,hermes)



X=poseidon ; X=zeus ; hermes .