Comp24412: Symbolic AI
Lecture 1: Prolog I

Ian Pratt-Hartmann

Room KB2.38: email: ipratt@cs.man.ac.uk

2016–17

Outline

- Start SWI Prolog by typing `pl` to Unix/Linux:

```
ipratt@rs0-> pl
Welcome to SWI-Prolog (Version 5.0.10)
Copyright (c) 1990-2002 University of Amsterdam. ...

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

- To leave Prolog, type `halt.` (don't forget the full stop) or hit control-D.
- There is a version for Windows that can easily be downloaded from http://www.swi-prolog.org/download/stable.

- The following is a Prolog 'program'

```
parent(sue,noel).
parent(chris,noel).
parent(noel,ann).
parent(ann,dave).

male(noel).
male(dave).
male(chris).
female(sue).
female(ann).
```

- Pernickety (but important) points:
  - names begin with small letters (capitals have special meaning).
  - Every statement ends with a full stop.

- Once this program is in a file `my_first.pl` it can be loaded into a prolog interpreter using the following command:

  `?-` **[my_first].**

- The `?-` is the Prolog interpreter's prompt. It may vary from system to system.
- The `[ ]` means 'reconsult'. SWI Prolog understands a `.pl` extension.
- **Warning:** Some Prologs require

  `?-` **[-my_first].**

- Don't forget the full stop!

- Once the program has been loaded (or consulted), it can be queried:

```
?- parent(sue, noel).
Yes

?- parent(noel, ann).
Yes

?- parent(denis, mark).
No

?- male(noel).
Yes

?- female(noel).
No
```

- The above Program is a list of facts (in effect, a database)
- It is run by typing in goals (in effect, queries)
- The meaning of the program is separated from the goals that are set
- This sort of programming is called **declarative** programming
- Declarative programming is important in AI.

- Goals may contain variables, signalled by capital letters

```
?-  parent(noel, X).

X = ann  ;

No

?-  parent(X, noel).

X = sue  ;

X = chris

No
```

- The semicolon is typed by the user. It means: “find me any more solutions”.

- You can have more than one variable in a query:

```
?-  parent(X, Y).

X = sue,  Y = noel;

X = chris,  Y = noel;

X = noel,  Y = ann;

X = ann,  Y = dave;

No
```

- But shared variables have to take unitary values:

```
?-  parent(X, X).

No
```

- When Prolog has a goal, it tries to match that goal against something in the program.
- The matching process involves finding values for any variables so that the goal and the fact with which it is matched become identical
- This process is known as **unification**
- It is a relatively routine exercise to write a unification algorithm
- Thus, the query `parent(X,noel)` has the interpretation: "Is there an `X` such that `X` is a parent of Noel?"

- For instance, in answering the query

```
?-  male(X).
```

```
X = noel
```

Prolog used the given substitution to match the query with the fact `male(noel)`

- and in answering the query

```
?-  parent(X, Y).
```

```
X = sue,  Y = noel
```

Prolog used the given substitution to match the query with the fact `parent(ann,noel)`

- Notice that clauses in programs can themselves contain variables
- For example, we could add to the above program the fact

  `person(X).`

  by adding this line to the data base, in this case to my_first.pl. If we now issue the goal `person(noel)` Prolog will find the variable substitution

  `X = noel`

  to match the goal with the fact and answer:

  ```
  ?-  person(noel).
  yes
  ```

- Noel has two parents in the database. Suppose we want to ask who his mother is.
- We could issue the query

```
?- parent(X,noel),female(X).

X = sue  ;

No
```

- Here, ‘,’ has the interpretation “and”. Thus, the whole query reads: “Is there an `X` such that `X` is a parent of Noel and `X` is female?

- Now consider the query

```
?-  parent(noel, X),parent(X, Y).

X = ann,  Y = dave  ;

No
```

- This reads: "Is there an `X` such that there is a `Y` such that Noel is a parent of `X` and `X` is a parent of `Y`?"
- That is, it finds Noel's grandchildren

Outline

- We can encode the relationships between the concepts `mother`, `parent` and `female` in the program itself by adding the **rule**

```
mother(X,Y):-
    parent(X,Y),female(X).
```

- so that we can then issue the query

```
?-  mother(M, noel).

M = sue
```

- Declarative meanings:
  - `:-` means "if"
  - `,` `_` (between tail clauses or goals) means "and"
  - If a variable `X` appears in fact or rule in some program, read "For all `X` ..." before that fact or rule.
  - If a variable `X` appears in a goal, read "There exists `X` such that ..." before the query.

- Suppose we want to define “ancestor” as a Prolog rule
- We rely on the following facts:
  - The parent of someone is that person’s ancestor
  - The ancestor of someone’s parent is that person’s ancestor

- It sometimes helps to reformulate these natural expressions in 'logic-English':
  - For any `X`, for any `Y`, if `X` is a parent of `Y` then `X` is an ancestor of `Y`
  - For any `X`, for any `Y`, for any `Z`, if `Z` is a parent of `Y` and `X` is an ancestor of `Z` then `X` is an ancestor of `Y`

- Using the notation of first-order logic:

$$\forall x \forall y (\text{parent}(x, y) \rightarrow \text{ancestor}(x, y))$$
$$\forall x \forall y \forall z (\text{parent}(z, y) \wedge \text{ancestor}(x, z) \rightarrow \text{ancestor}(x, y)).$$

- We can translate these facts straight into Prolog rules:

```
ancestor(X,Y):- parent(X,Y).

ancestor(X,Y):- parent(Z,Y), ancestor(X,Z).
```

- Now let us add the definition of `ancestor` to our family database:

```
parent(sue,noel).
parent(chris,noel).
parent(noel,ann).
parent(ann,dave).

male(noel).
male(dave).
male(chris).
female(sue).
female(ann).

ancestor(X,Y):- parent(X,Y).
ancestor(X,Y):-
   parent(Z,Y), ancestor(X,Z).
```

- We can then issue queries such as

```
?-  ancestor(sue, dave).

Yes

?-  ancestor(dave, noel).

No
```

and get sensible answers. Magic!

- Note: if we count people as their own ancestors, the following might be more elegant:

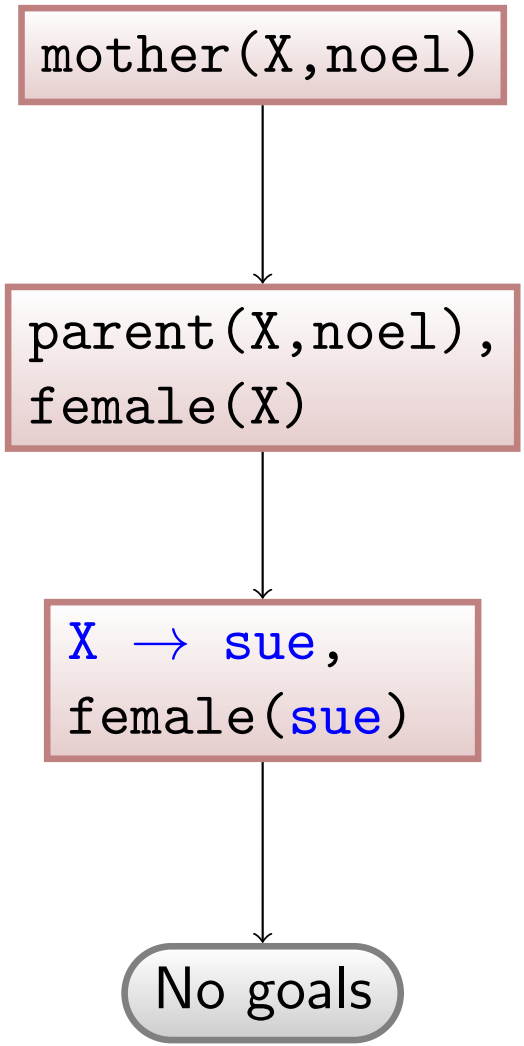
```
ancestor(X,X).

ancestor(X,Y):-
    parent(Z,Y),
    ancestor(X,Z).
```

Outline

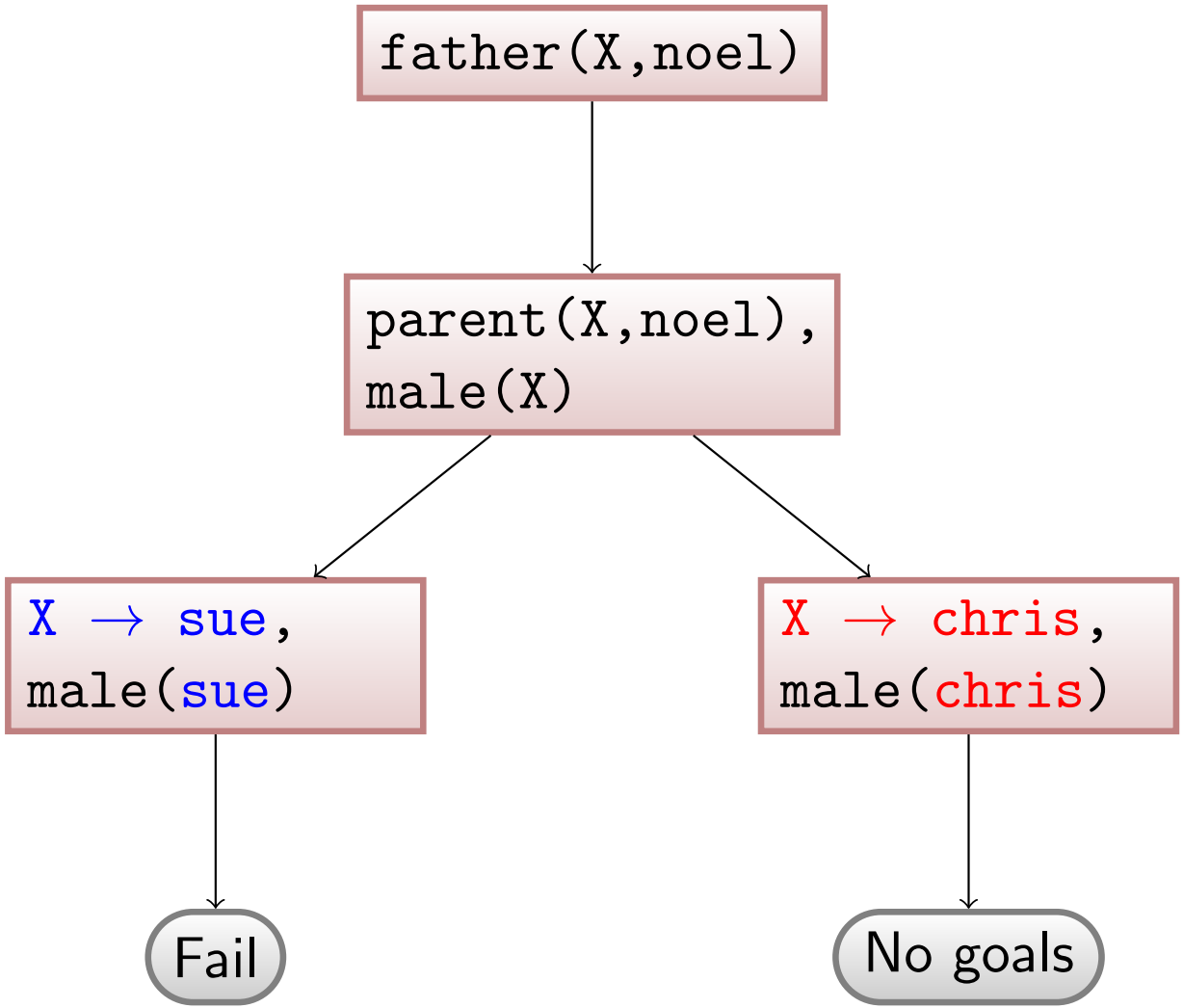
- What is really happening here?
  1. When this goal is typed in, Prolog puts it on a **goal stack**
  2. Prolog then tries to find either (i) a fact or (ii) a rule head in the program which unifies with the top goal on the goal-stack
  3. If (i), then variable bindings are carried through and the goal is popped off the stack
  4. If (ii), then variable bindings are carried through, the goal is popped off the stack and the clauses in the rule tail are pushed onto the top of the goal stack
  5. The process is then repeated from step 2 until the goal stack is empty

- The query `mother(X,noel)`:



- Notice that a given goal may unify with several goals or rule heads
- Such events in the execution of a Prolog program are called **choice points**
- What happens if Prolog does not find a rule that unifies with the top goal on the goal stack?
  - In that case, the goal is said to **fail**
  - Whenever Prolog encounters failure, it backtracks to the most recent `choice point`, takes the next available choice, and proceeds as before

- The query `father(X,noel)`:



- If you want to see what is happening to a program, you can type the goal

  `?- trace.`

  to turn on tracing. Here is a trace of `mother(X,noel)`.

```
call  mother(_884, noel)
UNIFY 1 []
 call  parent(_884, noel)
 UNIFY 1 [_884=sue]
 exit  parent(sue, noel)
 call  female(sue)
 UNIFY 1 []
 exit  female(sue)
exit  mother(sue, noel)
```

- The goal `notrace` turns tracing off again.

- Here is a trace of `father(X,noel).`

```
call  father(_882, noel)
UNIFY 1 []
 call  parent(_882, noel)
 UNIFY 1 [_882=sue]
 exit  parent(sue, noel)
 call  male(sue)
 fail  male(sue)
 redo  parent(sue, noel)
 UNIFY 2 [_882=chris]
 exit  parent(chris, noel)
 call  male(chris)
 UNIFY 3 []
 exit  male(chris)
exit  father(chris, noel)
```

- It sometimes helps to think of the program as implicitly defining an entire execution tree, which the Prolog interpreter searches
- The nodes represent states of the goal stack and the links possible unifications of the top goal with some fact or rule in the program

- An abstract example
- Consider the program

```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

- And the goal

```
?- a.
```

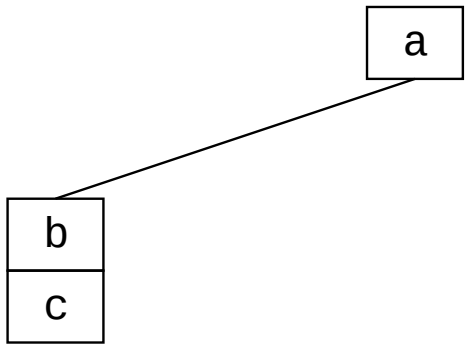
- Start at goal a:

```
a
```

```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Run (trace) Program (database)

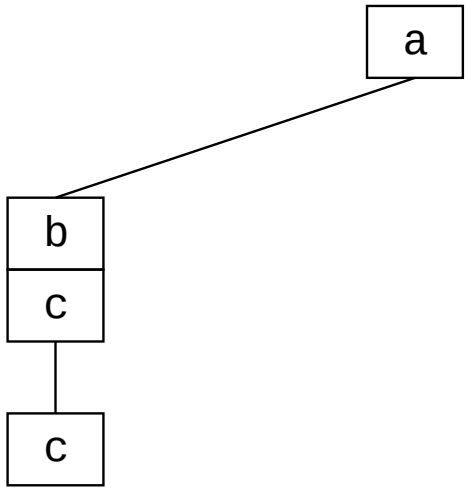
- Process goal a:




```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Run (trace) Program (database)

- Goal `b` succeeds:

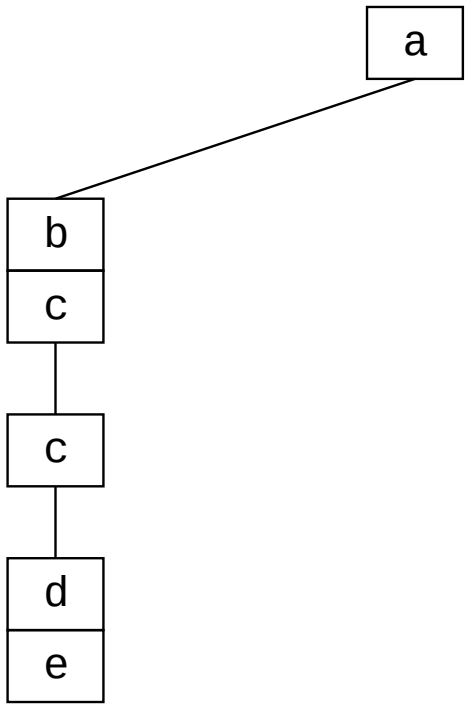



```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Run (trace)

Program (database)

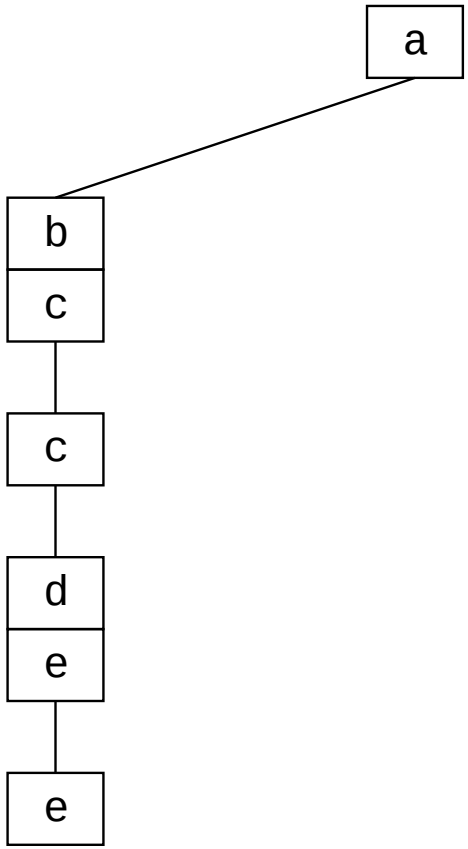
- Process goal c:




```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Run (trace) Program (database)

- Goal `d` succeeds:

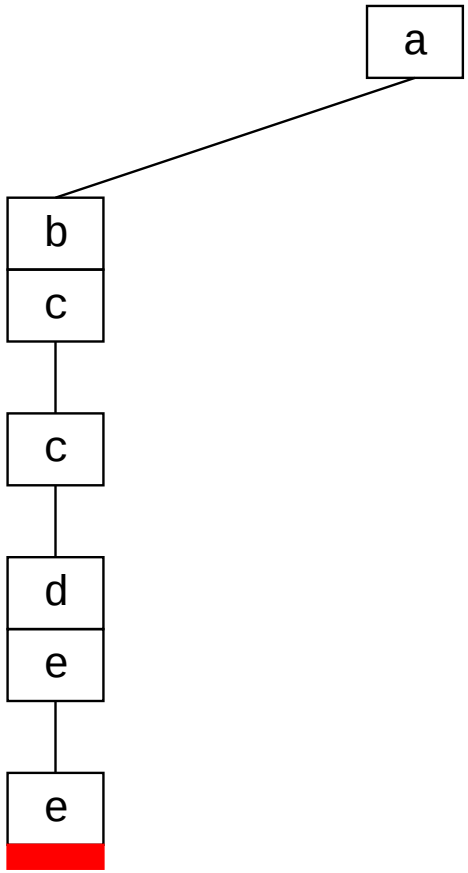


Run (trace)

```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Program (database)

- Backtrack to a:

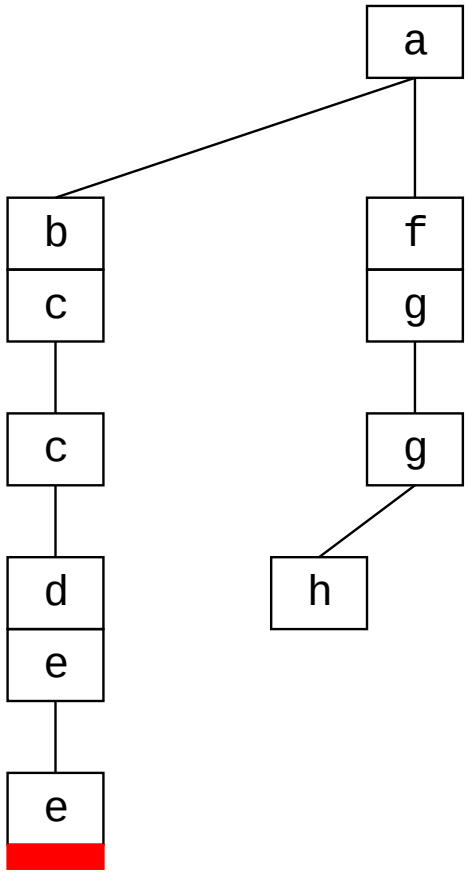


Run (trace)

```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Program (database)

- Proceed as before:

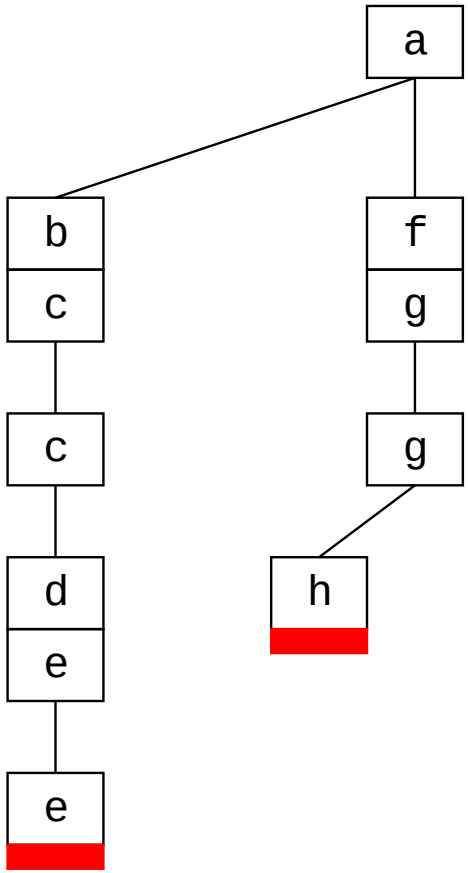


Run (trace)

```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Program (database)

- Backtrack to g:

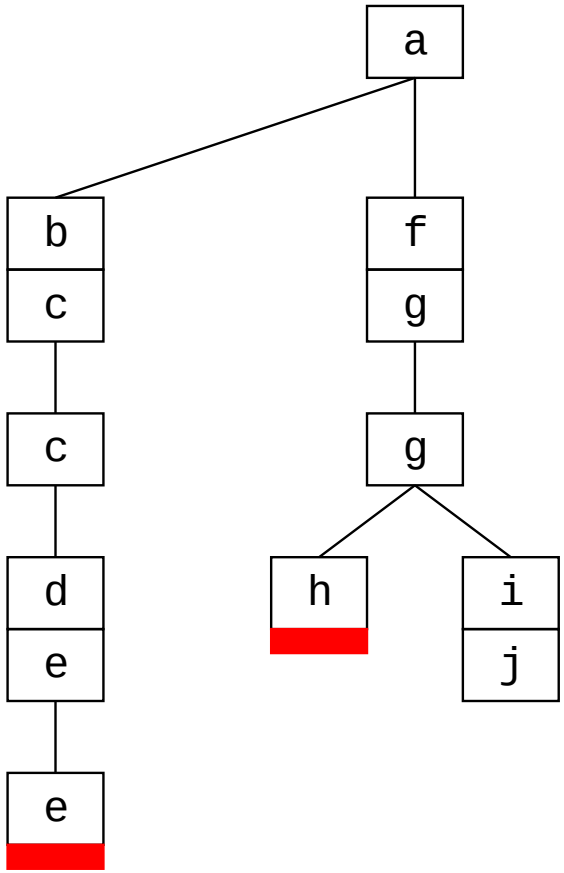



```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Run (trace)

Program (database)

- Proceed as before:

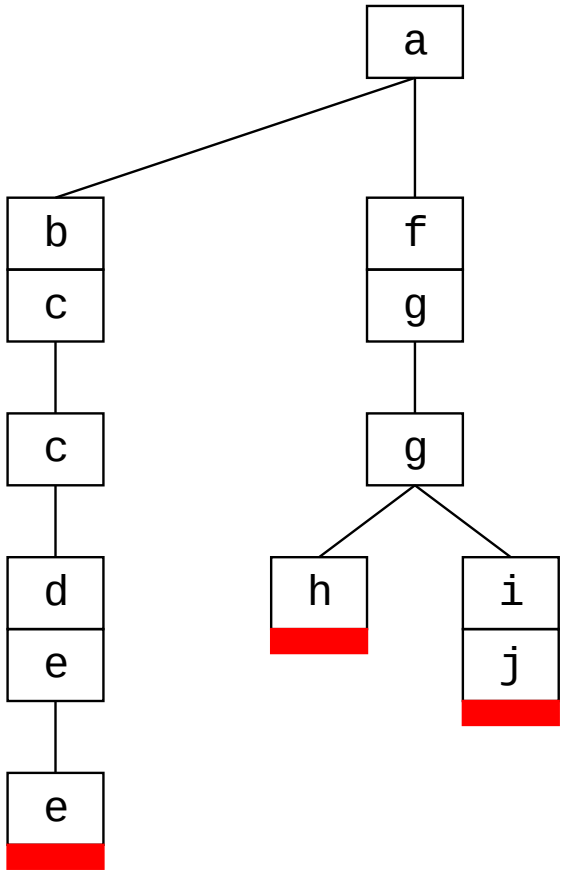



```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Run (trace)

Program (database)

- Backtrack to a:

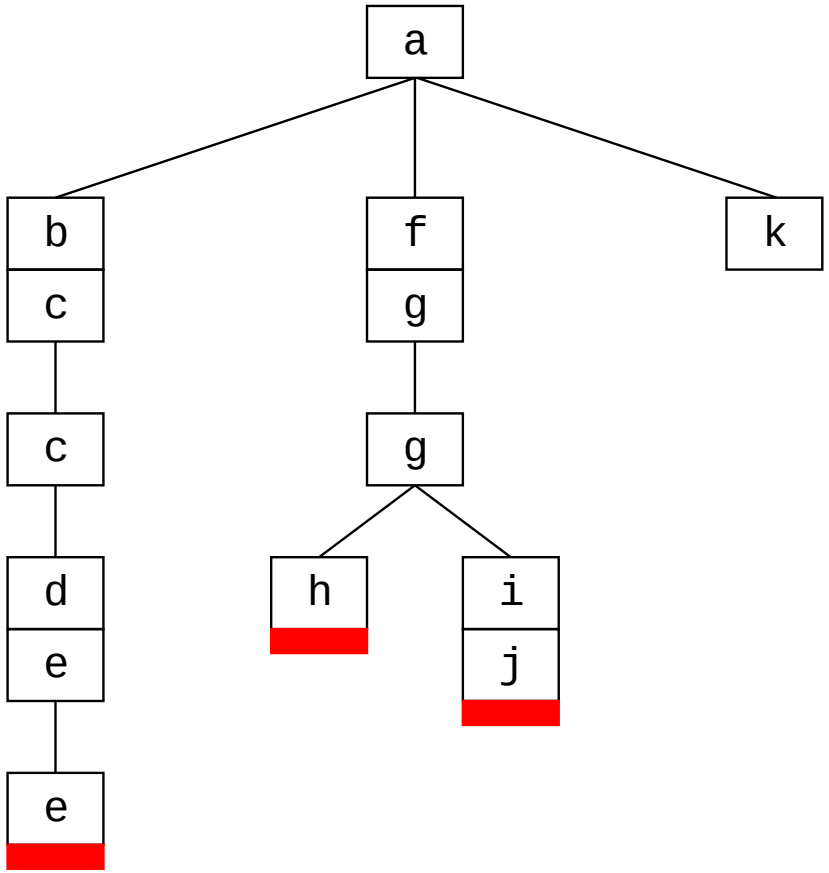



```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Run (trace)

Program (database)

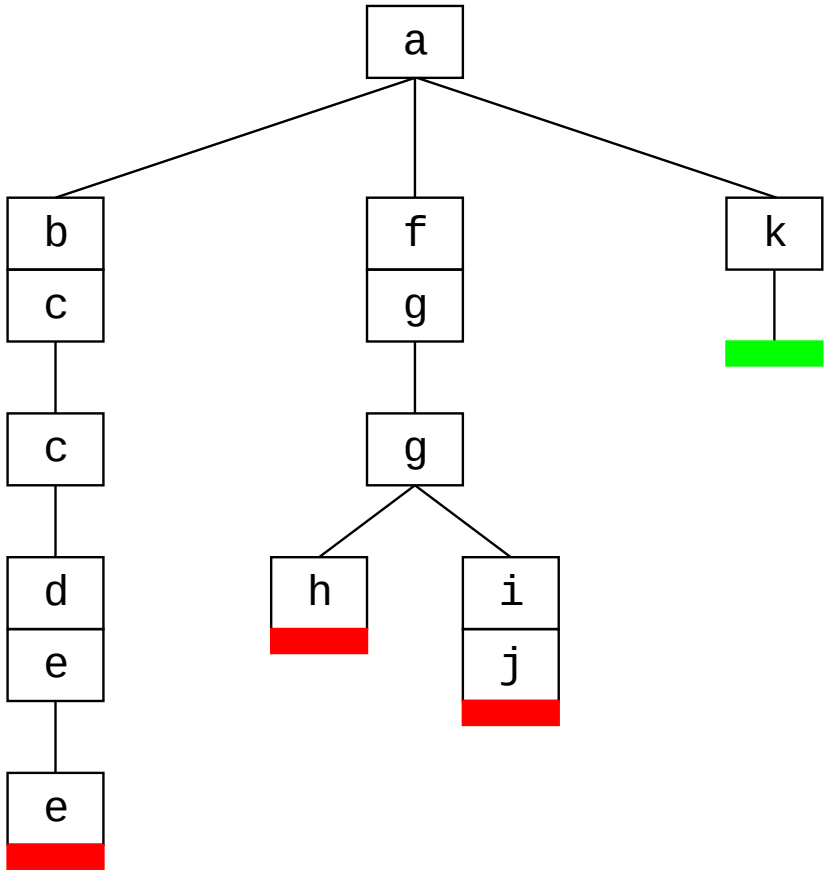
- Proceed as before:




```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Run (trace) Program (database)

- Success at `k` and hence at `a`:




```
a:- b,c.
a:- f,g.
a:- k.
b.
c:- d,e.
d.
f.
g:- h.
g:- i,j.
k.
```

Run (trace) Program (database)

- The tree is search branch-by-branch.
- This search pattern is called **depth-first search** or **chronological backtracking**.

- Thus Prolog is performing two major operations: `unification` and `backtracking`
  - Unification has to do with the matching of goals and facts/rule-heads
  - Backtracking is the process by which all possibilities for matching facts and rules are tried

- Underlying philosophy of Prolog programs
  - It is a good idea to think of them, if possible, as stating facts
  - If you tell Prolog the truth, it will answer your queries truly
  - That is, it is a sort of **theorem prover**

- General appearance of Prolog programs
  - Frequently recursive
  - There is a **base case**, e.g.

    ```
    ancestor(X,X).
    ```

  - And a **recursive case** e.g.

    ```
    ancestor(X,Y):-
        parent(Z,Y),
        ancestor(X,Z).
    ```

  - The recursion may be on all sorts of structures, including numbers

- Principal functions of a Prolog interpreter
  - pattern-matching
  - backtracking
- Important point (wake up)

  Unification is a completely deterministic process. There are no choices available. Choices arise only when several facts and/or rules unify with a given goal.

Outline

- What should I do next?
  - Try out Prolog under Linux.
  - Try running Prolog under Linux from within `emacs`.
  - Try out the built-in editor and graphical debugger.
  - Download SWI Prolog on your PC (if you have one).
  - Start reading *Learn Prolog Now!* (or your favourite Prolog book).
  - Acquire a copy of *Representation and Inference for Natural Language*.