UNIVERSITÉ JEAN MONNET SAINT-ÉTIENNE

Introduction to AI

(Part 3 : Prolog Programming)

About the practical sessions/project

- The practical grade is half of the AI grade
- Use GNU or SWI Prolog (install it on your machine): http://gprolog.univ-paris1.fr/ or http://www.swi-prolog.org/Download.html
- Assignments can be found here (they are also given in Leuven):

http://labh-curien.univ-st-etienne.fr/~fromont/Prolog/

- Session i is dedicated to discuss/correct the assignment i : make sure to organise yourself and complete them at home before the session
- The project (game programming) can be started in parallel. The subject is on in the *Claroline* repository for the AI class
- There will be a MCQ exam the 21/12 on the assignments (**1/3** of the practical grade)
- The project (code, report with the code (about heuristics), tests (the relevance of the tests is very important for the grade) and comments) have to be sent before 10/01/2014 (**2/3** of the practical grade).
- + theory exam in the 19/12 (will also contain some PROLOG)

Outline

- Introduction
  - Syntax of terms
  - Some simple programs
  - Terms as data structures, unification
  - Call, computation, search tree, mode
  - Arithmetic in Prolog
- Recursion and Lists
- Cut
- Built-in meta predicates
  - Findall, forall, …

What is Prolog?

- Prolog is the most widely used language to have been inspired by logic programming research.
- Some features:
  - Prolog uses logical variables. These are not the same as variables in other languages. Programmers can use them as ‘holes’ in data structures that are gradually filled in as computation proceeds.

…More Features

- Unification is a built-in term-manipulation method that passes parameters, returns results, selects and constructs data structures.
- Basic control flow model is backtracking.
- Program clauses and data have the same form.
- The relational form of procedures makes it possible to define ‘reversible’ procedures.

History of Prolog

- Developed in 1970s by Alain Colmerauer, Phillip Roussel (University of Marseilles, France)
- David H. D. Warren provided foundations of modern implementation in the Warren Abstract Machine for DEC PDP-10 (University of Edinburgh)
- Prolog is a basis for numerous other languages such as CLP(R), Prolog III, etc.

Not for general purpose programming

- More restricted computation model of proving assertions from collections of facts and rules
- Think of queries working on a database of facts with rules that permit inferring new facts

- A query is just a theorem to be proven

Why restrict applicability of a language?

- Prolog provides better built-in support for the algorithms and tasks especially useful in search problems
  - Theorem proving is "just" a search problem
- Search problems are incredibly important
  - Exponential complexity
  - But efficient techniques and heuristics help solve practical programs in a timely fashion

Example applications

- Medical patient diagnosis
- Theorem proving
- Solving Rubik's cube
- Type checking
  - Type inference in ML and Haskell is done in this way
- Database querying
- …

What a program looks like

```
/* At the Zoo */

elephant(george).
elephant(mary).

panda(chi_chi).
panda(ming_ming).

dangerous(X) :- big_teeth(X).
dangerous(X) :- venomous(X).

guess(X, tiger) :- stripey(X), big_teeth(X), isaCat(X).
guess(X, koala) :- arboreal(X), sleepy(X).
guess(X, zebra) :- stripey(X), isaHorse(X).
```

Prolog is a ‘declarative’ language

- Clauses are statements about what is true about a problem, instead of instructions how to accomplish the solution.
- The Prolog system uses the clauses to work out how to accomplish the solution by searching through the space of possible solutions.
- Not all problems have pure declarative specifications. Sometimes extra-logical statements are needed.

Example: Concatenate lists a and b

In an imperative language

```
list procedure cat(list a, list b)
{
   list t = list u = copylist(a);
   while (t.tail != nil) t = t.tail;
   t.tail = b;
   return u;
}
```

In a functional language

```
cat(a,b) ≡
 if b = nil then a
else cons(head(a), cat(tail(a),b))
```

In a declarative/ relational language

```
cat([], Z, Z).
cat([H|T], L, [H|Z]) :- cat(T, L, Z).
```

Complete Syntax of Terms

Term

Constant

Names an individual

Atom

alpha17
gross_pay
john_smith
dyspepsia
+
=/=
’12Q&A’

Number

0
1
57
1.618
2.04e-27
-13.6

Compound Term

Names an individual that has parts

likes(john, mary)
book(dickens, Z, cricket)
f(x)
[1, 3, g(a), 7, 9]
-(+(15, 17), t)
15 + 17 - t

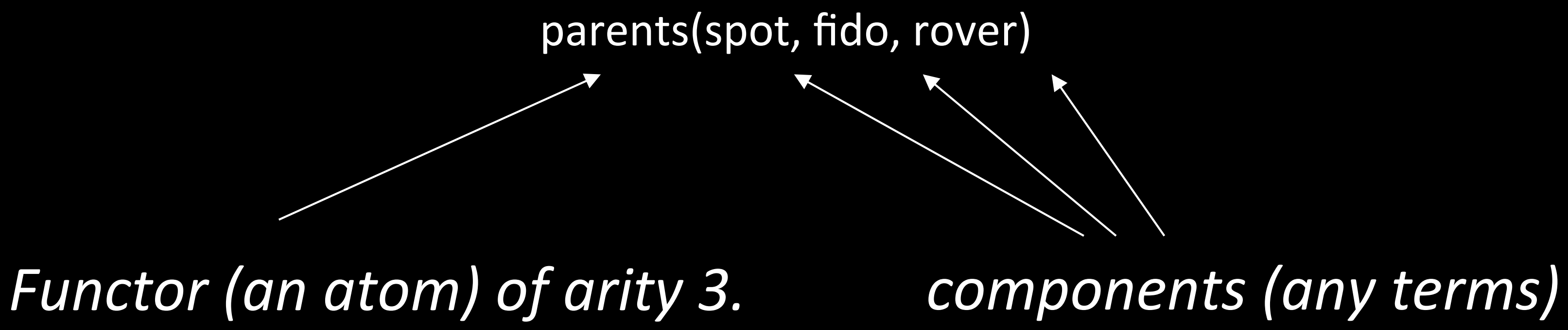
Variable

Stands for an individual unable to be named when program is written

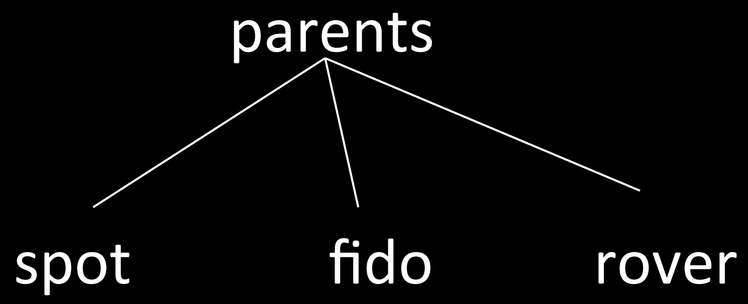
X
Gross_pay
Diagnosis
_257
_

Compound Terms

The parents of Spot are Fido and Rover.



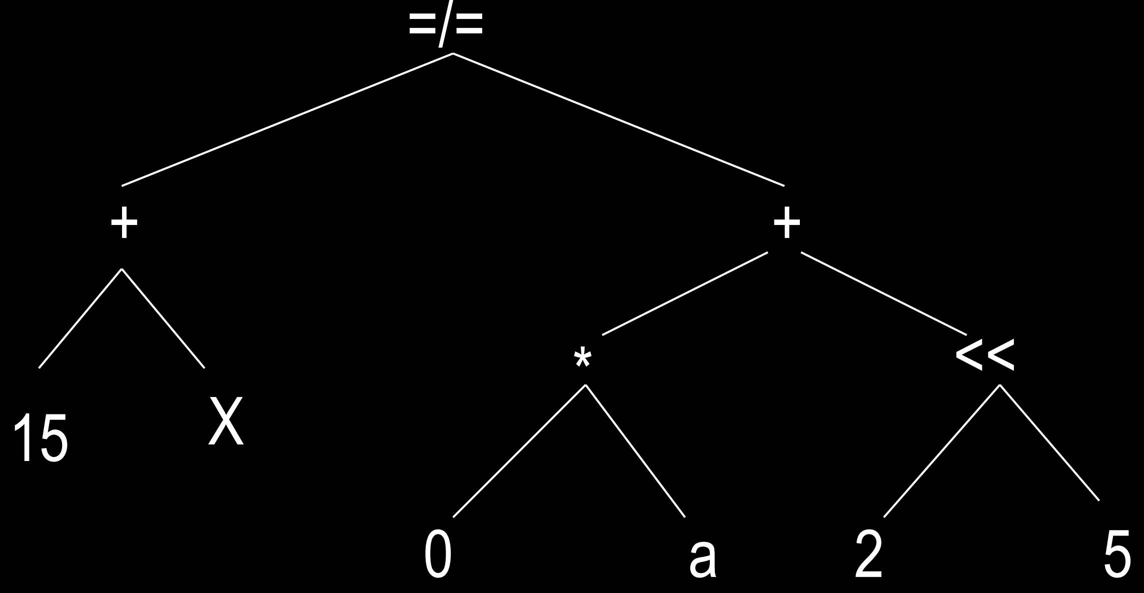
It is possible to depict the term as a tree:



Compound Terms

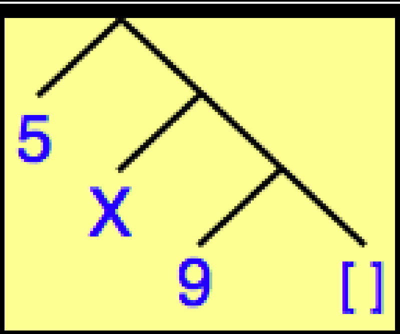
Some atoms have built-in operator declarations so they may be written in a syntactically convenient form. The meaning is not affected. This example looks like an arithmetic expression, but might not be. It is just a term.

```
=/=(15+X, (0*a)+(2<<5))
```



A particular compound term: the list

[ ] [3, 5] [5, X, 9]

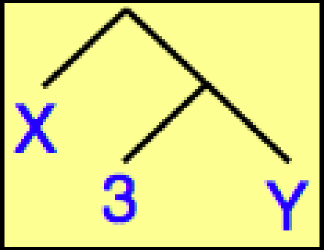


lists form a *subclass of binary trees*

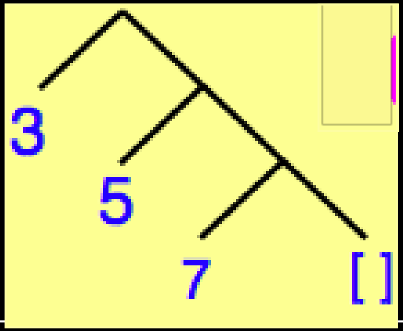
A vertical bar can be used as a separator to present a list in the form

[itemized-members | residual-list]

[X, 3 | Y]



[3 | [5, 7]]
[3, 5, 7]
[3, 5 | [7]]



The last point about Compound Terms…

Constants are simply compound terms of arity 0.

badger

means the same as

badger()

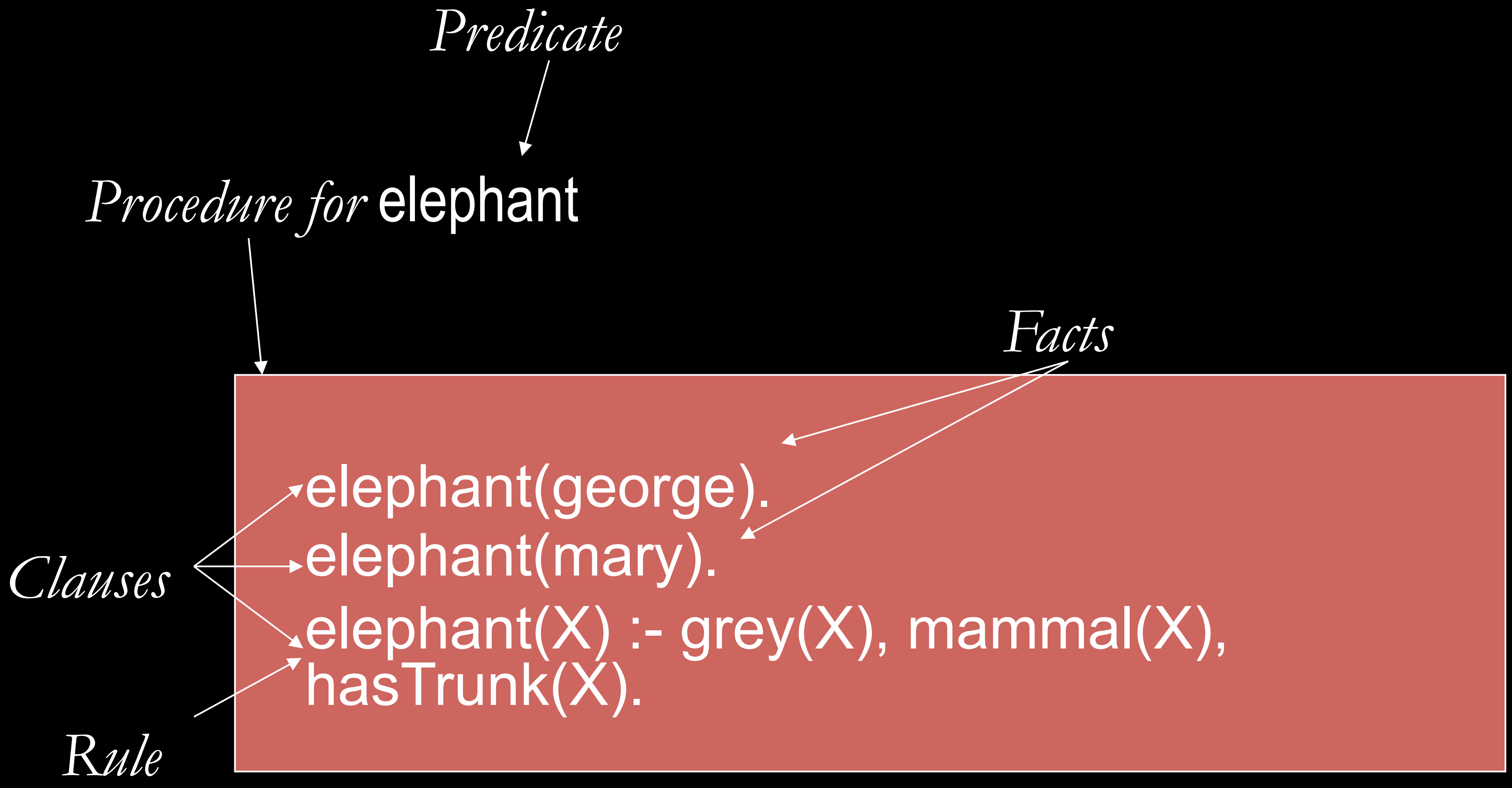
Structure of Programs

- Programs consist of procedures.
- Procedures consist of clauses.

- Each clause is a fact or a rule.
- Programs are executed by posing queries.

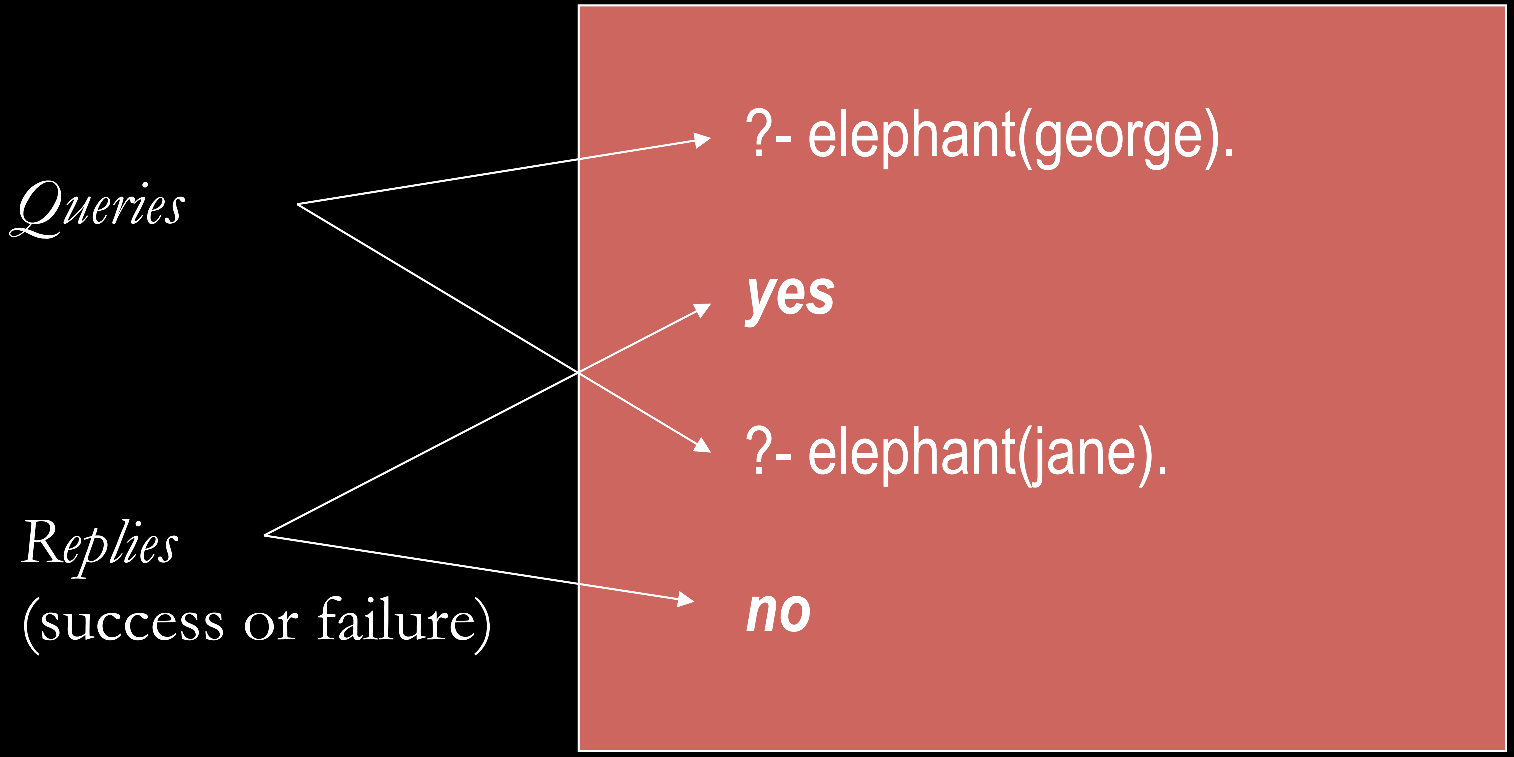
An example…

Example



```
elephant(george).
elephant(mary).
elephant(X) :- grey(X), mammal(X),
hasTrunk(X).
```

Example



```
?- elephant(george).
yes
?- elephant(jane).
no
```

Clauses: Facts and Rules

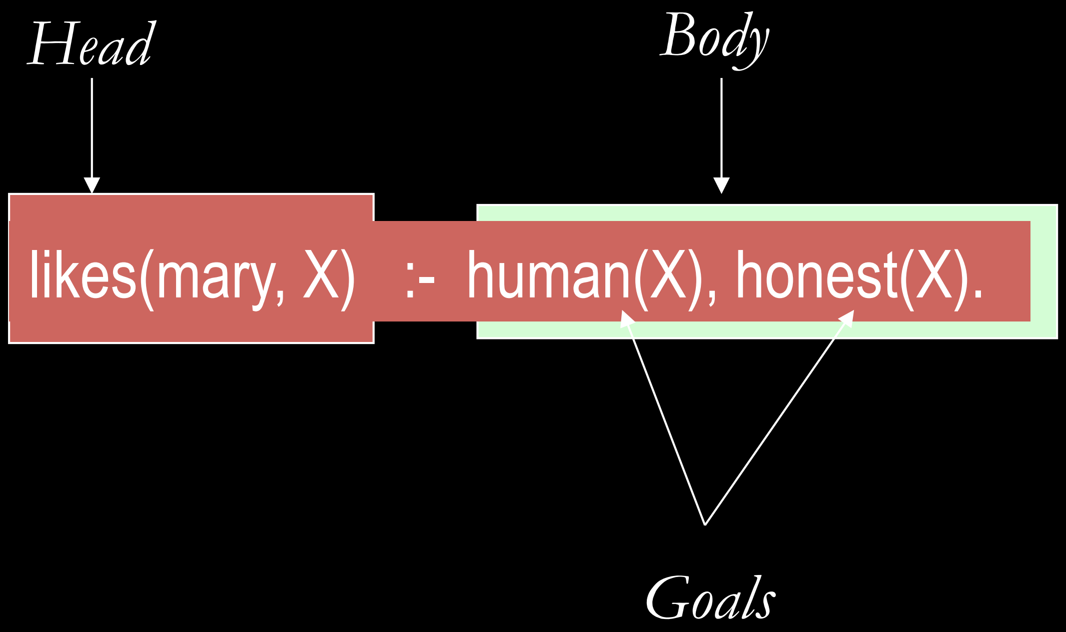
'if'
'provided that'
'turnstile'

Head **:-** *Body*. This is a rule.

Head. This is a fact.

Full stop at the end.

Body of a (rule) clause contains goals.



Variables

- Begin with an uppercase letter
- Either “instantiated” or “uninstantiated”
- X is instantiated means X stands for a particular value (similar to binding)
- Variables instantiations can be undone
  - Used to produce multiple answers during search
- Multiple uses of the same variable in same scope must refer to same value

Variables are scoped within a query

These two uses of X must represent same value

```
?- person(X), female(X).
```

X=karen

Read the comma as “and”

Variable arguments

Variables in queries are treated as *existentially quantified*

EXAMPLE

```
?- likes(X, prolog).
```

says “is ($X) likes(X, prolog) true?”

or “find X for which likes(X, prolog) is true”

Variables in program clauses are treated as *universally quantified*

EXAMPLE

```
likes(chris, X) :- likes(X, prolog).
```

expresses the sentence ("X) (likes(chris, X) $\leftarrow$ likes(X, prolog))

Interpretation of Clauses

Clauses can be given a declarative reading or a procedural reading.

Form of clause:	H :- G_1, G_2, …, G_n.
Declarative reading:	"That H is provable follows from goals G_1, G_2, …, G_n being provable."
Procedural reading:	"To execute procedure H, the procedures called by goals G_1, G_2, …, G_n are executed first."

Procedure calls

Execution involves evaluating calls, and begins with an *initial query (= initial resolvent)*

Prolog evaluates the calls in a query *sequentially*, in the *left-to-right order*, as written

```
?- a, d, e.                     evaluate a, then d, then e
```

(don't forget) **Convention:** terms beginning with an *upper-case letter* or an *underscore* are treated as *variables*

```
?- likes(chris, X).             here, X is a variable
```

Computations (1/2)

- A *computation* is a chain of derived queries, starting with the *initial query*
- Prolog selects the *first call in the current query* and seeks a program clause whose head matches the call
- If there is such a clause, the call is replaced by the clause body, giving the next derived query
- This is just applying the standard notion of procedure-calling in any formalism

Computations (2/2)

EXAMPLE

?- a, d, e.	initial query
a :- b, c.	program clause with head a and body b, c

Starting with the initial query, the first call in it matches the head of the clause shown, so the derived query is

```
?- b, c, d, e.
```

Execution then treats the derived query in the same way

Successful computations

A computation succeeds if it derives the empty query

EXAMPLE

```
?- likes(bob, prolog).        query
likes(bob, prolog).           program clause
```

The call matches the head and is replaced by the clause’s (empty) body, and so the derived query is empty.

So the query has succeeded, i.e. has been solved

Finite failure

A computation ***fails finitely*** if the call selected from the query does not match the head of any clause

```
?- likes(bob, haskell).                    Query
```

This fails finitely if there is no program clause whose head matches likes(bob, haskell).

```
?- likes(chris, haskell).          Query
likes(chris, haskell) :- nice(haskell). Program clause
```

If there is no clause head matching nice(haskell) then the computation will fail after the first step

Infinite failure

A computation ***fails infinitely*** if every query in it is followed by a non-empty query

EXAMPLE

`?- a.`	query
`a :- a, b.`	clause

This gives the infinite computation

```
?- a.
?- a, b.
?- a, b, b.
  .....
```

This may be useful for driving some perpetual process

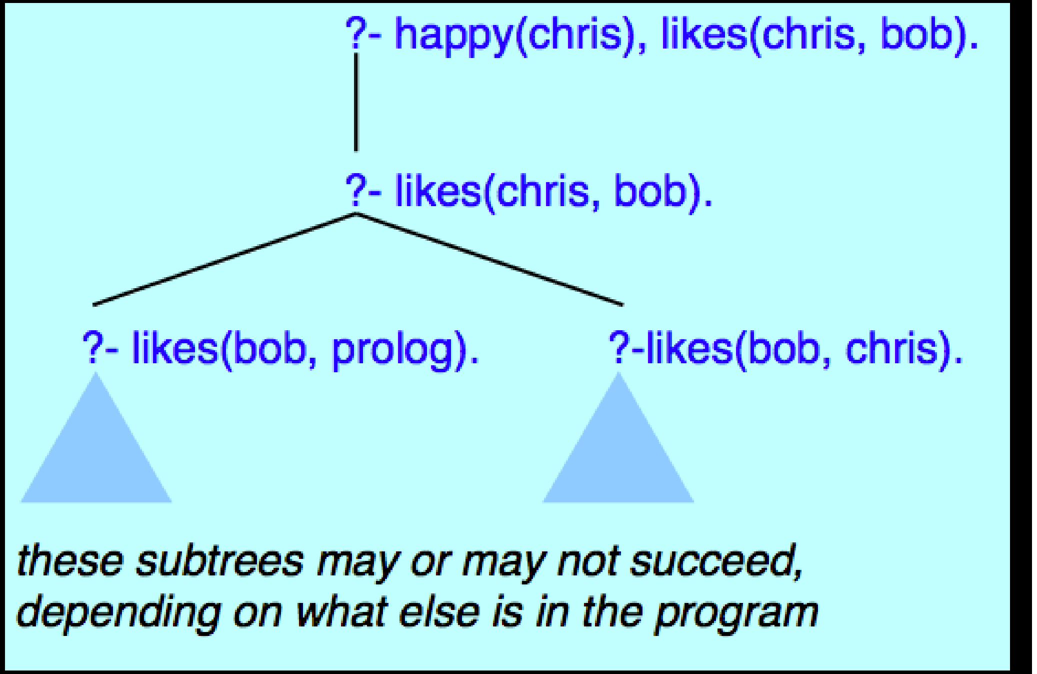
Multiple answers

A query may produce many computations

Those, if any, that succeed may yield multiple answers to the query (not necessarily distinct)

EXAMPLE

```
?- happy(chris), likes(chris, bob).

happy(chris).
likes(chris, bob) :- likes(bob, prolog).
likes(chris, bob) :- likes(bob, chris).
<…etc…>
```



?- happy(chris), likes(chris, bob).
?- likes(chris, bob).
?- likes(bob, prolog).
?-likes(bob, chris).
these subtrees may or may not succeed, depending on what else is in the program

We then have a *search tree* in which each branch is a separate computation:

Answers as logical consequences

A ***successful computation*** confirms that the conjunction in the initial query is a logical consequence of the program.

EXAMPLE

?- a, d, e.

If this succeeds from a program P then the *computed answer* is

$$a \wedge d \wedge e$$

and we have

$$P \models a \wedge d \wedge e$$

Conversely: if the program P does not offer any successful computation from the query, then the query conjunction is not a consequence of P

Generalized matching (unification)

Matching a call to a clause head requires them to be

either already identical

or able to be made identical, if necessary by instantiating (binding/unifying) their variables (unification)

EXAMPLE

```
?- likes(U, chris).
likes(bob, Y) :- understands(bob, Y).
```

Here, likes(U, chris) and likes(bob, Y) can be made identical (unify) by binding U / bob and Y / chris

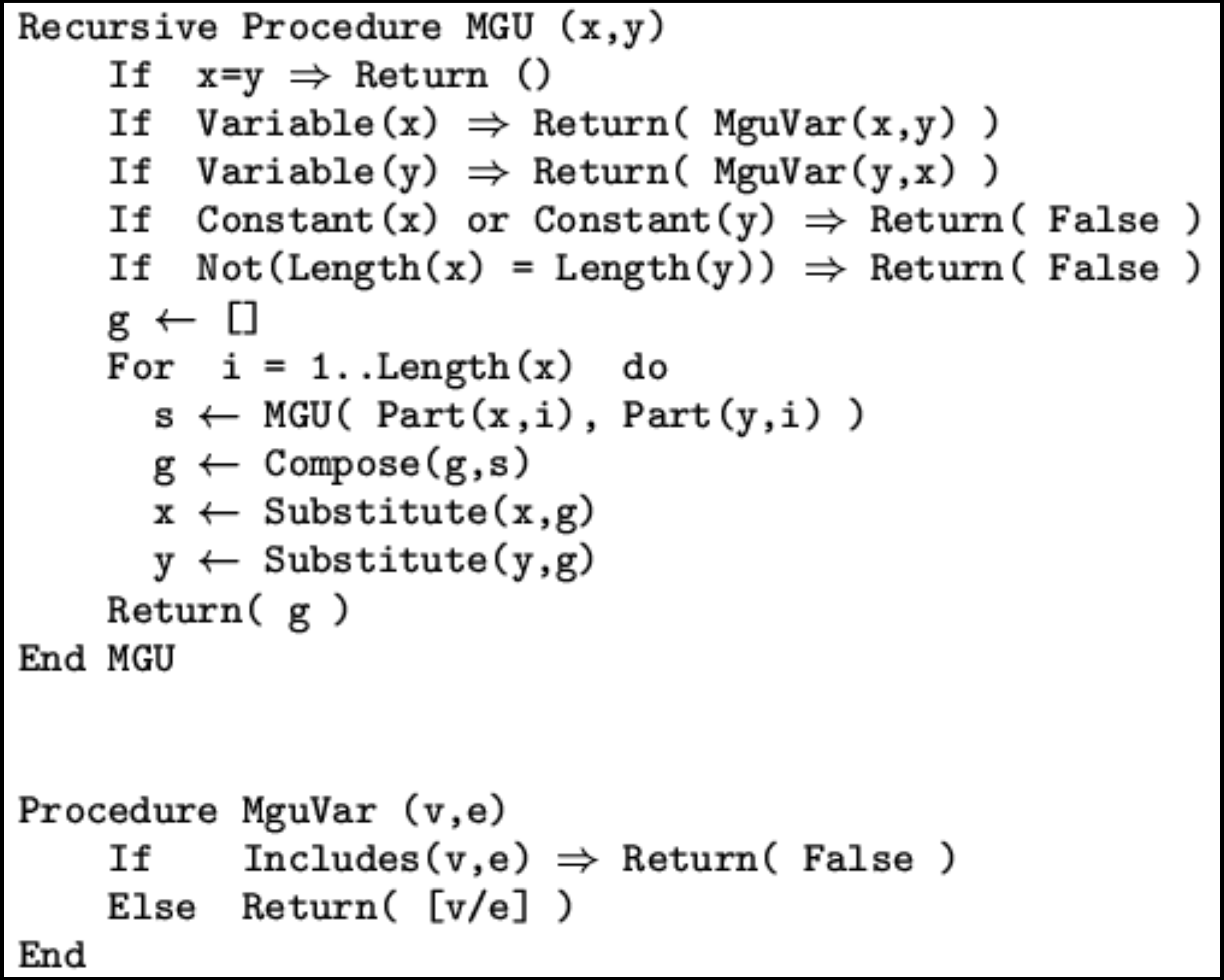
The derived query is

```
?- understands(bob, chris).
```

Unification (recall from logic)

- Two terms unify if substitutions can be made for any variables in the terms so that the terms are made identical. If no such substitution exists, the terms do not unify.
- The Unification Algorithm proceeds by recursive descent of the two terms.
  - Constants unify if they are identical
  - Variables unify with any term, including other variables
  - Compound terms unify if their functors and components unify.

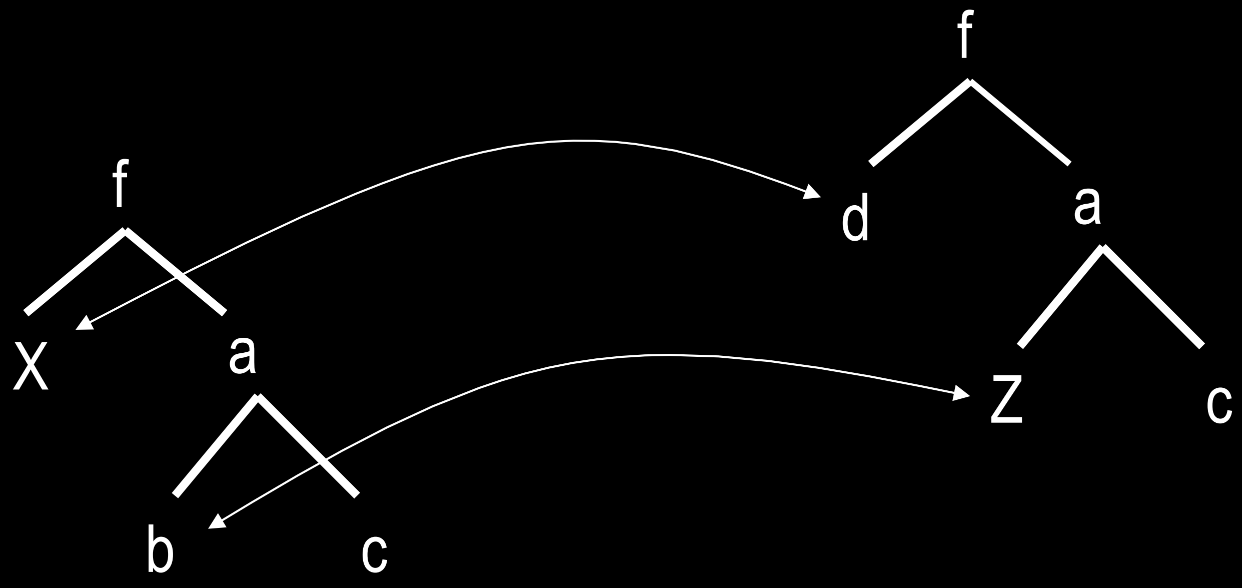
Herbrand Unification algorithm (already seen in Logic)



```
Recursive Procedure MGU (x,y)
    If  x=y ⇒ Return ()
    If  Variable(x) ⇒ Return( MguVar(x,y) )
    If  Variable(y) ⇒ Return( MguVar(y,x) )
    If  Constant(x) or Constant(y) ⇒ Return( False )
    If  Not(Length(x) = Length(y)) ⇒ Return( False )
    g ← []
    For  i = 1..Length(x)  do
      s ← MGU( Part(x,i), Part(y,i) )
      g ← Compose(g,s)
      x ← Substitute(x,g)
      y ← Substitute(y,g)
    Return( g )
End MGU


Procedure MguVar (v,e)
    If    Includes(v,e) ⇒ Return( False )
    Else  Return( [v/e] )
End
```

Examples

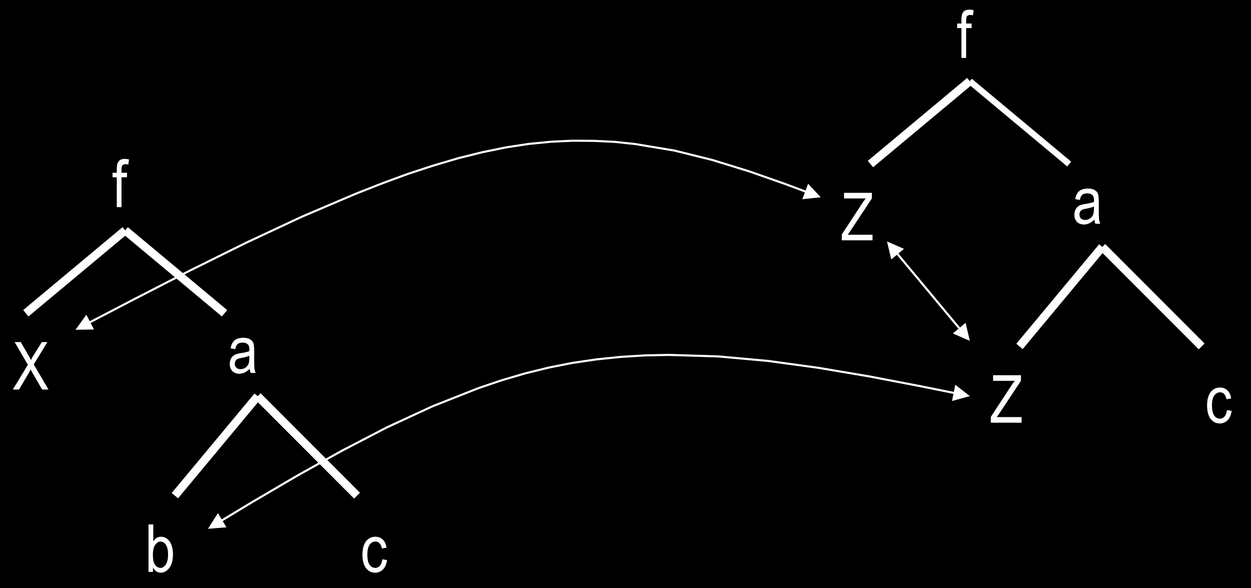
The terms f(X, a(b,c)) and f(d, a(Z, c)) unify.



The terms are made equal if d is substituted for X, and b is substituted for Z. We also say X is instantiated to d and Z is instantiated to b, or X/d, Z/b.

Examples

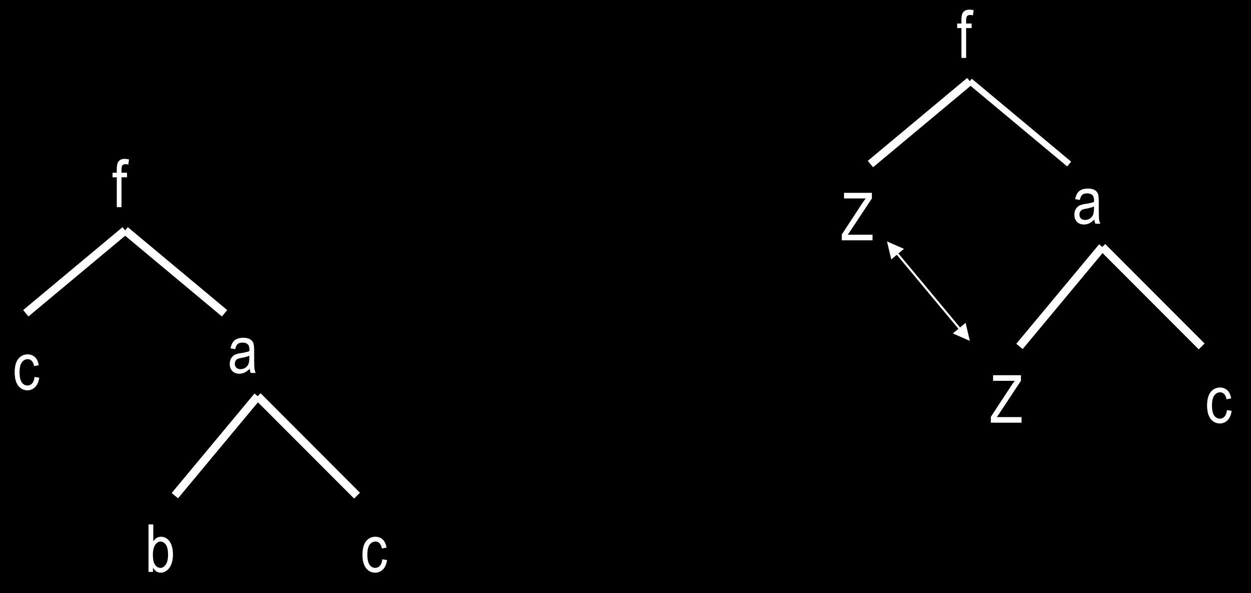
The terms f(X, a(b,c)) and f(Z, a(Z, c)) unify.



Note that Z co-refers within the term.
Here, X/b, Z/b.

Examples

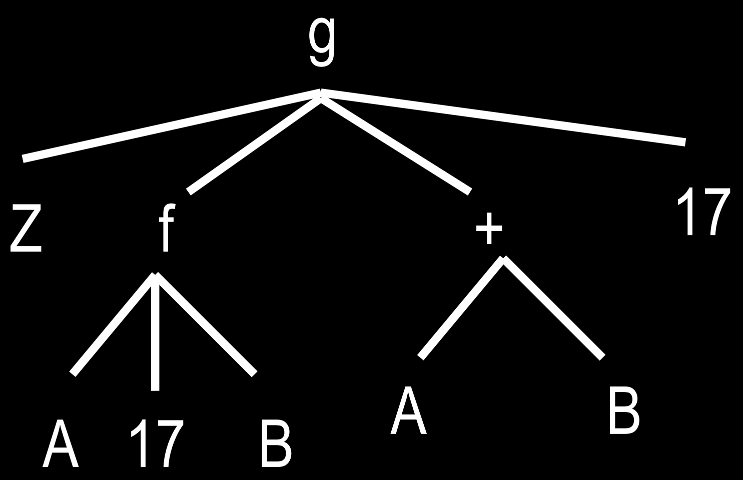
The terms f(c, a(b,c)) and f(Z, a(Z, c)) do not unify.

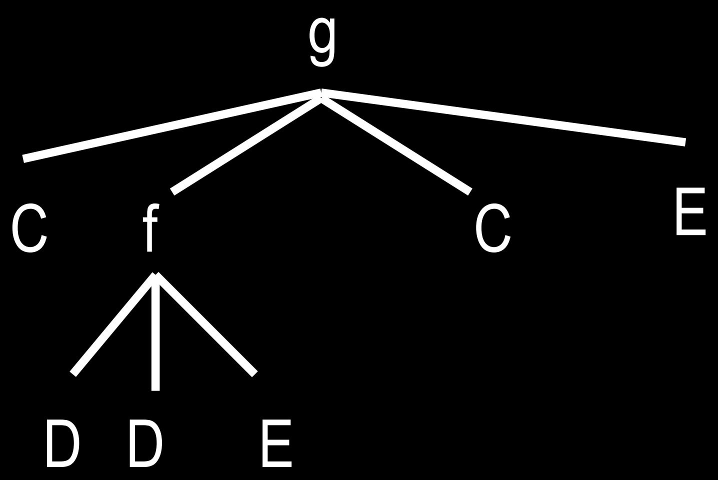


No matter how hard you try, these two terms cannot be made identical by substituting terms for variables.

Exercise

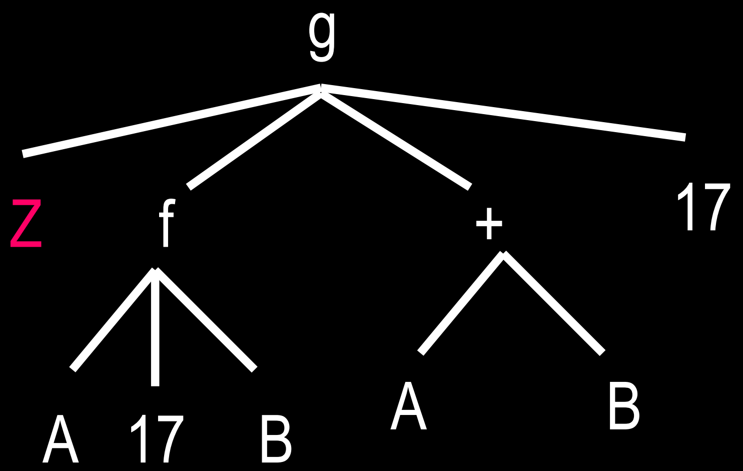
Do terms g(Z, f(A, 17, B), A+B, 17) and g(C, f(D, D, E), C, E) unify?

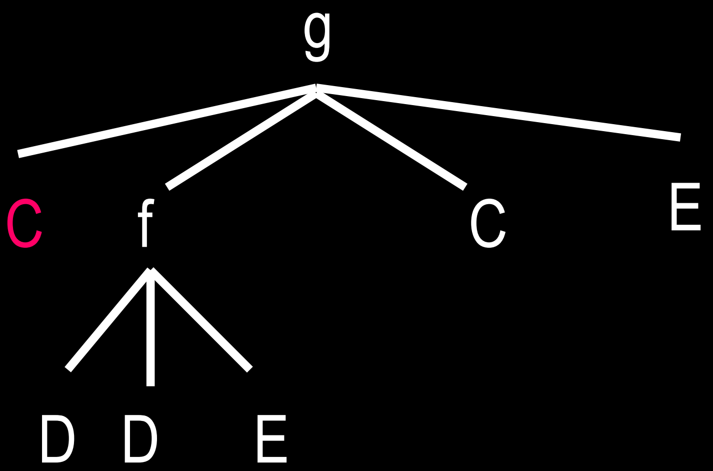




Exercise – Sol

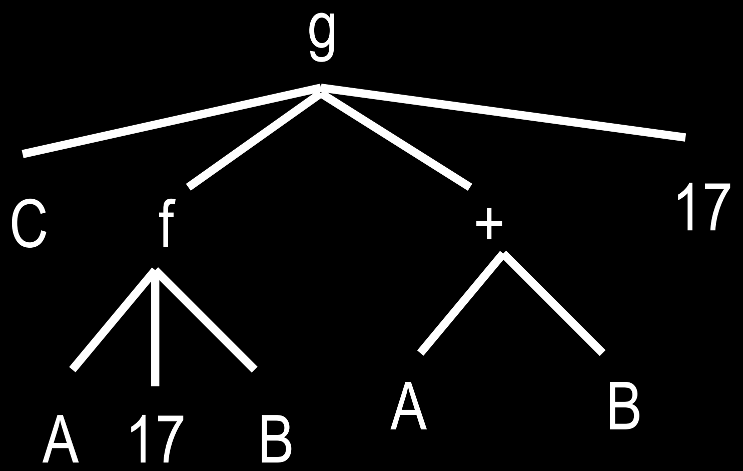
Z/C

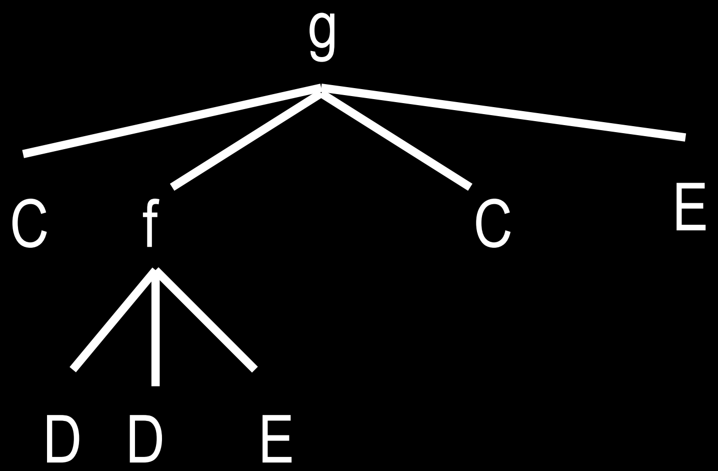




Exercise – Sol

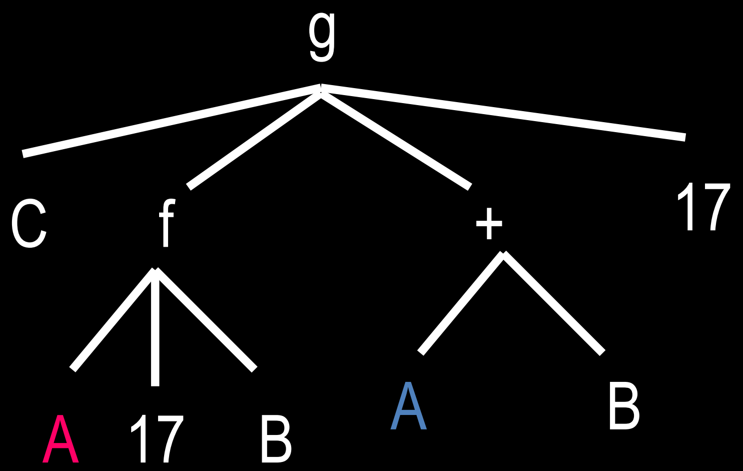
Z/C

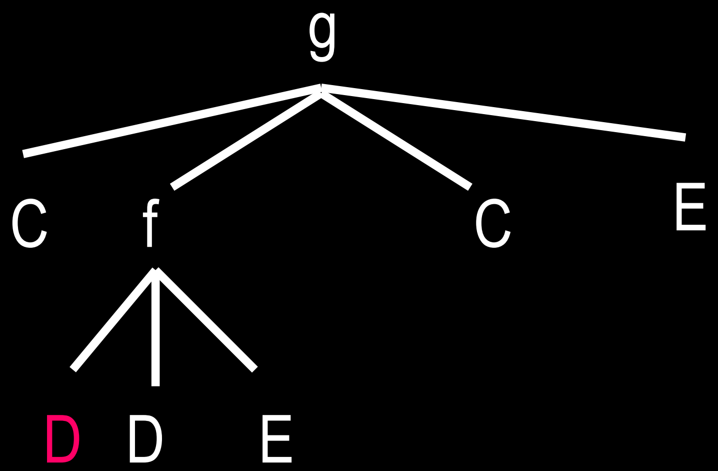




Exercise – Sol

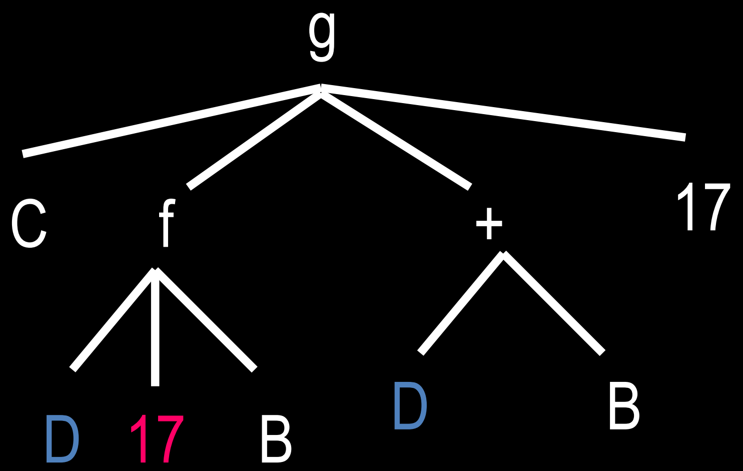
A/D, Z/C

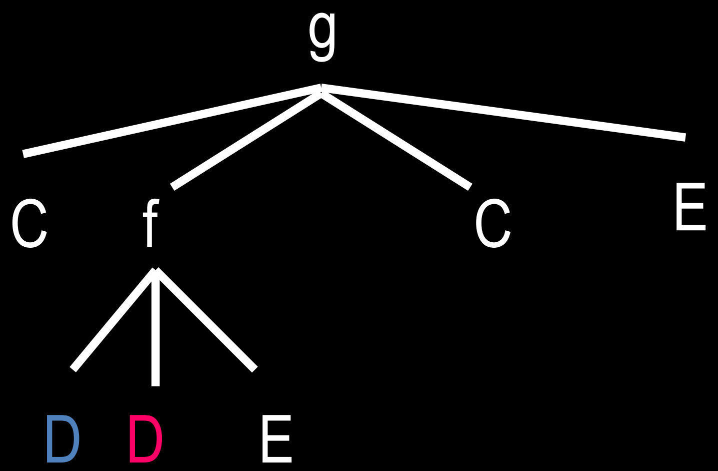




Exercise – Sol

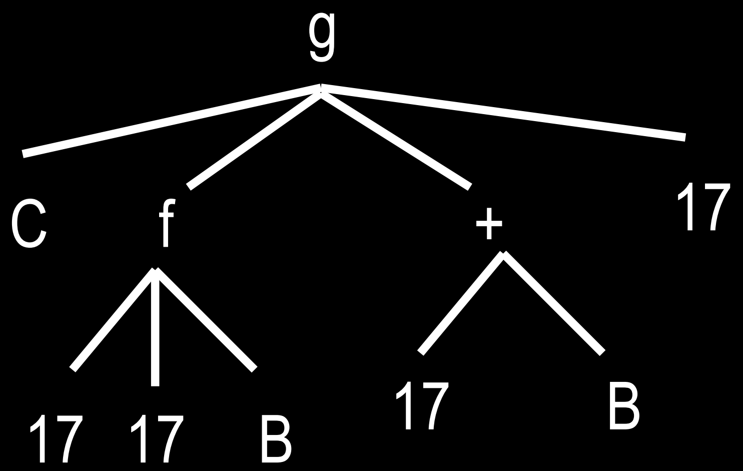
D/17, A/D, Z/C

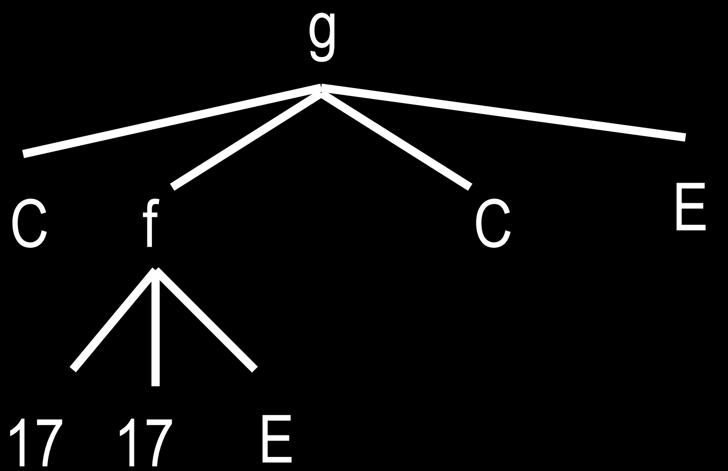




Exercise – Sol

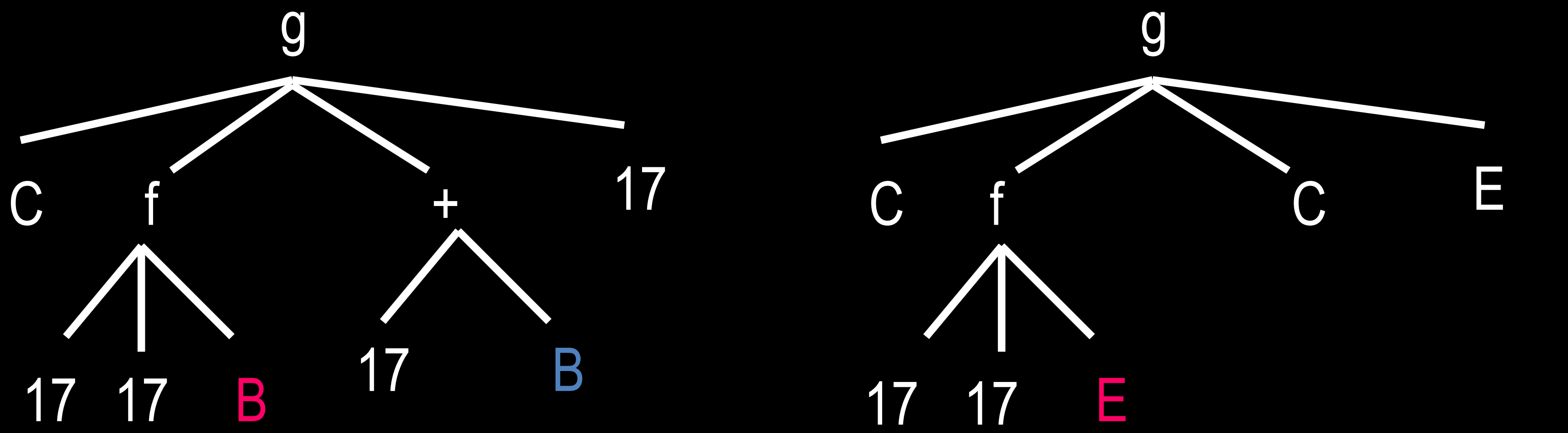
D/17, A/17, Z/C





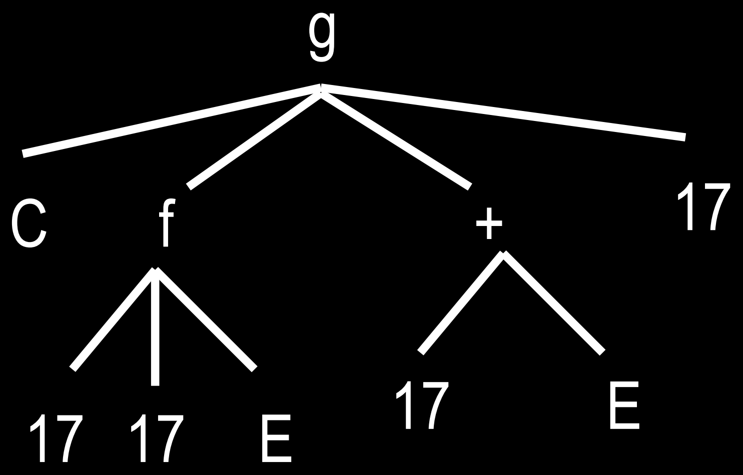
Exercise – Sol

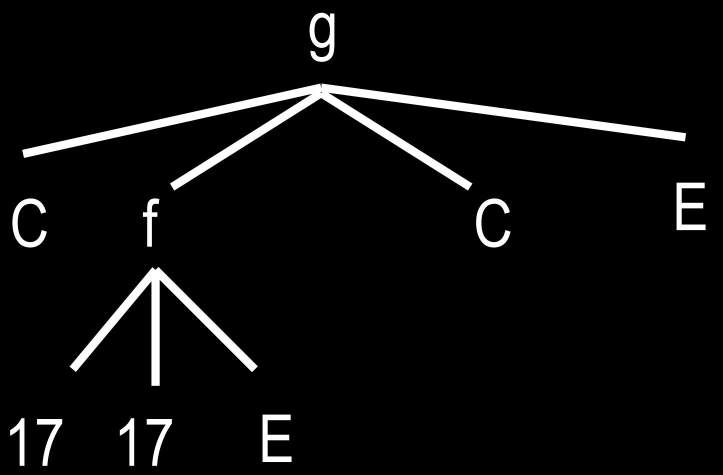
B/E, D/17, A/17, Z/C



Exercise – Sol

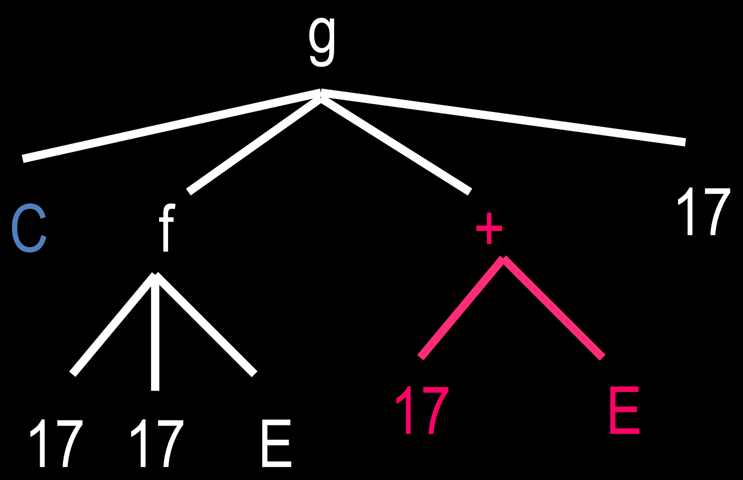
B/E, D/17, A/17, Z/C

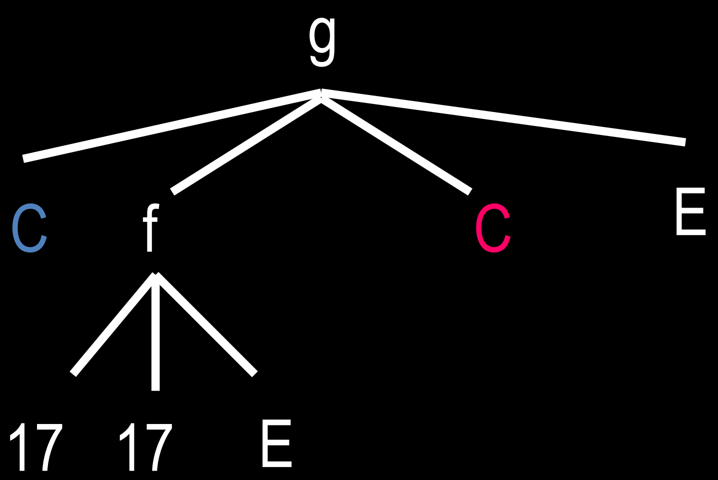




Exercise – Sol

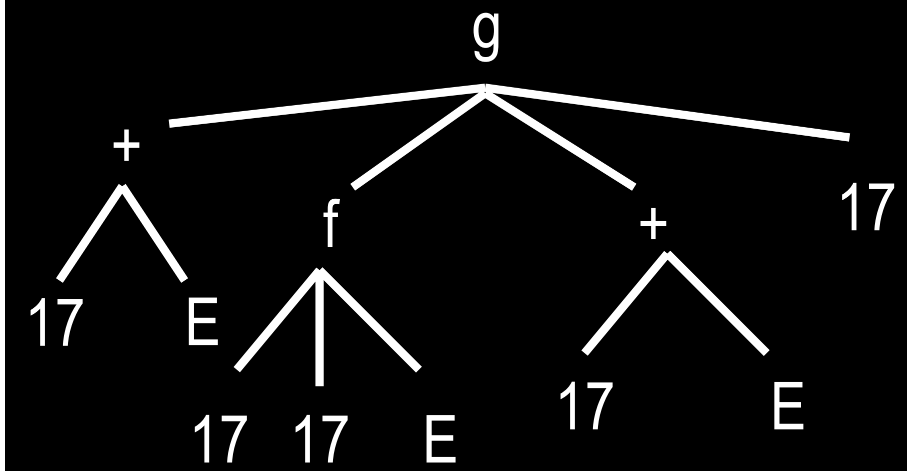
C/17+E, B/E, D/17, A/17, Z/C

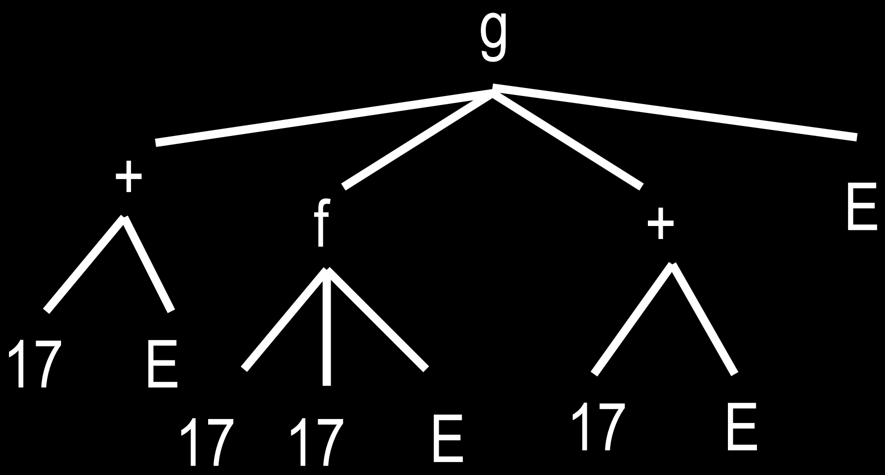




Exercise – Sol

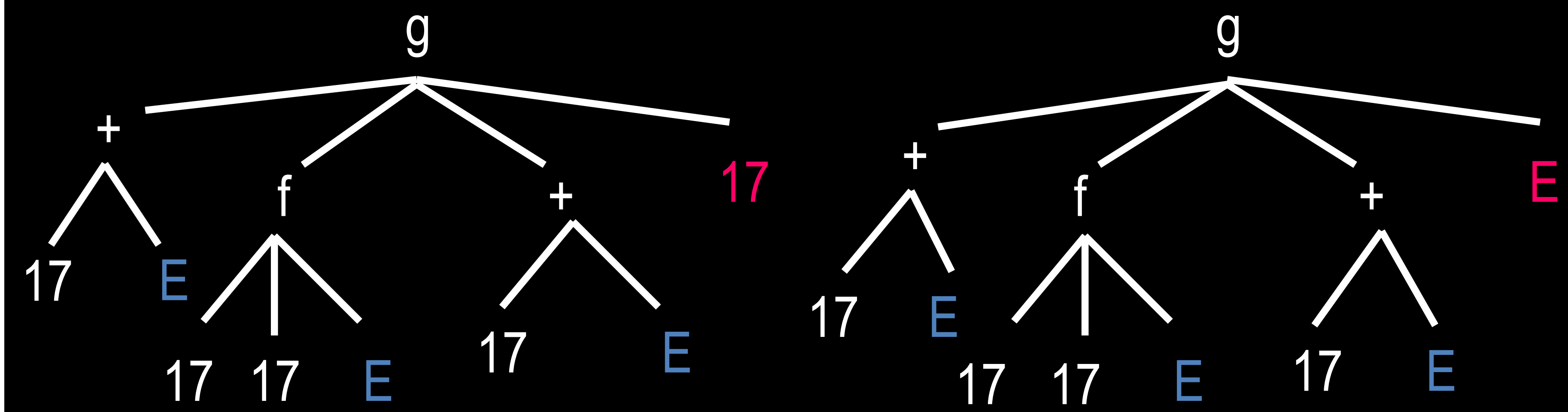
C/17+E, B/E, D/17, A/17, Z/17+E





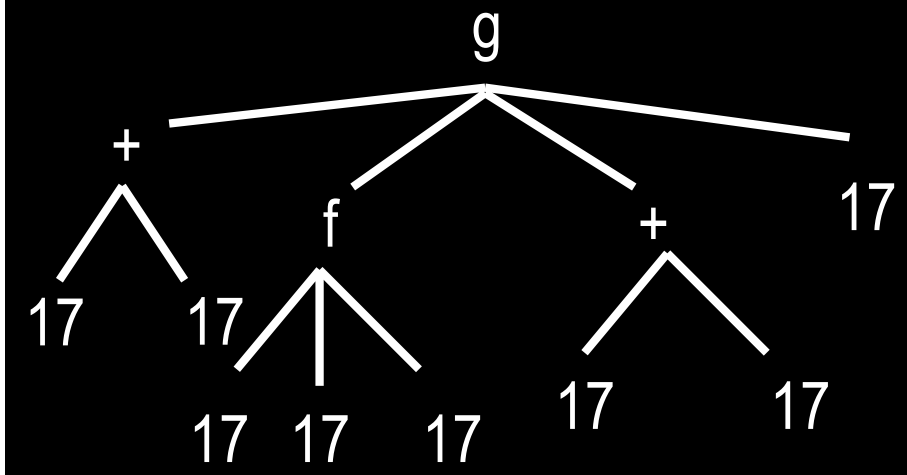
Exercise – Sol

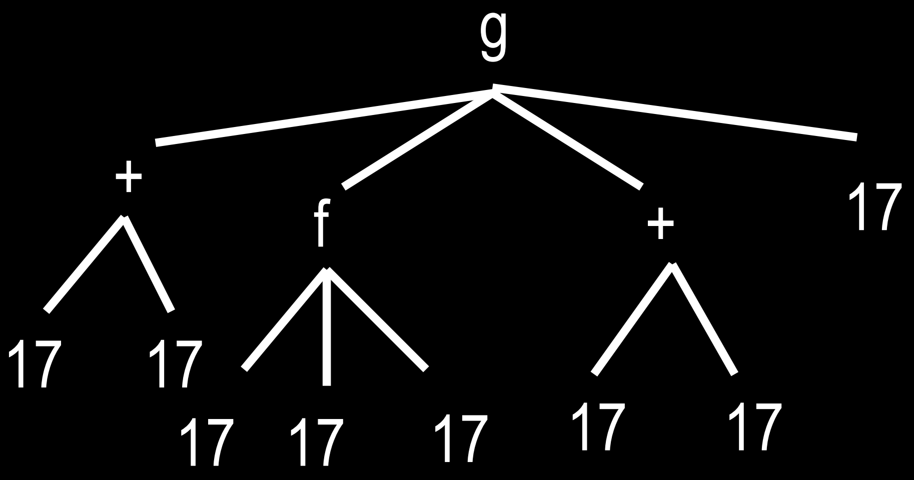
E/17, C/17+E, B/E, D/17, A/17, Z/C



Exercise – Sol

E/17, C/17+17, B/17, D/17, A/17, Z/C





Deterministic evaluations

- If only ONE computation is generated (whether it succeeds or fails), the evaluation is said to be deterministic
  - The search tree then consists of a single branch
- Prolog is non-deterministic in general because the evaluation of a query may generate multiple computations

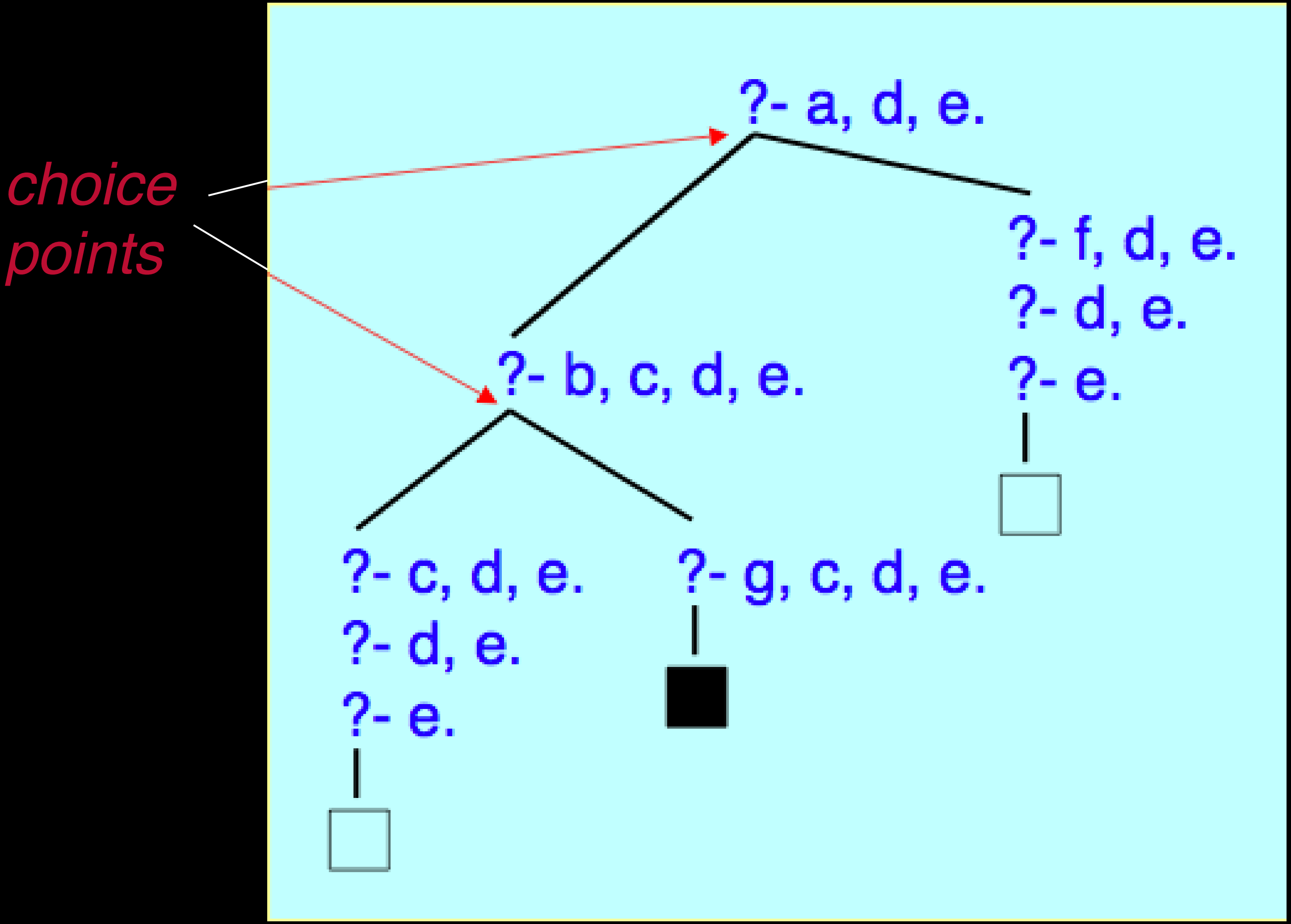
Example of non-determinism

A Prolog evaluation is non-deterministic (contains more than one computation) when some call unifies with several clause-heads
 When this is so, the search tree will have *several branches*

EXAMPLE

```
a :- b, c.   (two clause-heads unify with a)
a :- f.
b.           (two clause-heads unify with b)
b :- g.
c.
d.
e.
f.
```

A query from which calls to a or b are selected must therefore give several computations


choice points
?- a, d, e.
?- b, c, d, e.
?- f, d, e.
?- d, e.
?- e.
?- c, d, e.
?- d, e.
?- e.
?- g, c, d, e.

Non determinism: mode 1/3

Prolog supplies the list-concatenation primitive append(X, Y, Z) but if it did not, then we could define our own:

```
C1: app([ ], Z, Z).
C2: app([ U | X ], Y, [ U | Z ]) :- app(X, Y, Z).
```

Now consider the query ?- app([a, b], [c, d], L).

The call matches the head of C2 by making the bindings
{U / a, X / [b], Y / [c, d], L / [a | Z] }
which produce the derived query: ?- app([b], [c, d], Z).
Another similar step binds Z / [b | Z2] and gives the next derived query
?- app([ ], [c, d], Z2).
This succeeds by matching the first clause, and binds Z2 / [c, d]

Internal process

The computed answer is therefore L / [a, b, c, d]

Non determinism: mode 2/3

The ***mode*** of the query in the previous example was

?- app(input, input, output).

where the first two arguments were *wholly-known* input, whilst the third argument was *wholly-unknown output*

However, we can pose queries with any mix of argument modes we wish

So there is a second way in which Prolog is non-deterministic: **a program does not determine the mode of the queries posed to it**

EXAMPLE

Using the same program we can pose a query having the mode

?- app(input, input, input). such as

```
?- app([ a, b ], [ c, d ], [ a, b, c, d ]).
```

This gives just one computation, which succeeds, but returns no output bindings.

Take a query having mode *?- app(output, mixed, mixed).* such as

```
?- app(X, [ b | L ], [ a, E, c, d ]).
```

This succeeds deterministically to give the output bindings

X / [a], L / [c, d], E / b

Non determinism: mode 3/3

This second kind of non-determinism is called *input-output non-determinism*, and distinguishes Prolog from most other programming formalisms

With a single Prolog program, we may pose an infinite variety of queries, but with other formalisms we have to change the program whenever we want to solve a new kind of problem

This does not mean that a single Prolog program deals with all queries with equal efficiency

Often, in the interest of efficiency alone, we may well change a Prolog program to deal with a new species of query

SLD Resolution

- Prolog generates *one computation at a time*
- Whichever computation it is currently generating, Prolog remains totally committed to it until it either succeeds or fails finitely
- This strategy is called *depth-first search*
- It is an *unfair* strategy, in that it is not guaranteed to generate all computations, unless they are all finite
- When a computation terminates, Prolog backtracks to the most recent choice-point offering untried branches
- The evaluation as a whole terminates only when no such choice-points remain
- The order in which branches are tried corresponds to the text-order of the associated clauses in the program
- Prolog's strategy is called SLD **resolution**

  - S *selection rule* (in Prolog from left to right)
  - L *linear resolution* (rule)
  - D *definite clauses* (one atom in the head)

Efficiency

The efficiency with which Prolog solves a problem depends upon

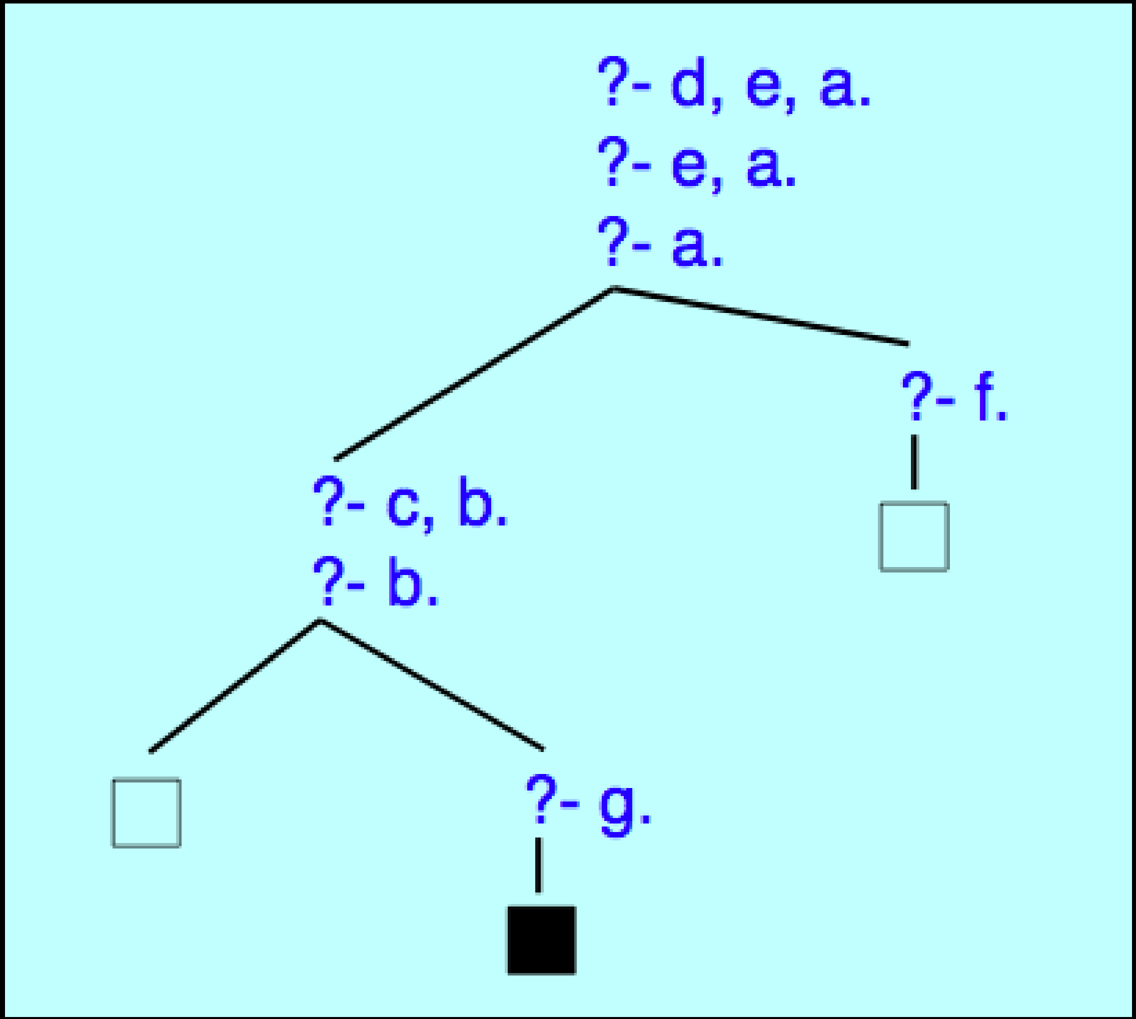
- the way knowledge is represented in the program
- the ordering of calls

EXAMPLE

Change the earlier query and program to

```
?- d, e, a.              different call-order
   a :- c, b.            different call-order
   a :- f.
   b.
   b :- g.
   c.
   d.
   e.
   f.
```

This evaluation has only **8 steps**, whereas the previous one had **10 steps**



Exercise/Example

```
male(bertram).
male(percival).

female(lucinda).
female(camilla).

pair(X, Y) :- male(X),

          female(Y).
```

1. ?- pair(percival, X).
2. ?- pair(bertram, lucinda).
3. ?- pair(X, daphne).
4. ?- pair(apollo, daphne).
5. ?- pair(X, Y).
6. ?- pair(camilla, X).
7. ?- pair(X, lucinda).
8. ?- pair(X, X).

Example 2

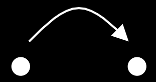
```
drinks(john, martini).
drinks(mary, gin).
drinks(susan, vodka).
drinks(john, gin).
drinks(fred, gin).

pair(X, Y, Z) :-
        drinks(X, Z),
        drinks(Y, Z).
```

1. ?- pair(X, john, martini).
2. ?- pair(mary, susan, gin).
3. ?- pair(john, mary, gin).
4. ?- pair(john, john, gin).
5. ?- pair(X, Y, gin).
6. ?- pair(bertram, lucinda).
7. ?- pair(bertram, lucinda, vodka).
8. ?- pair(X, Y, Z).

This definition forces X and Y to be distinct:

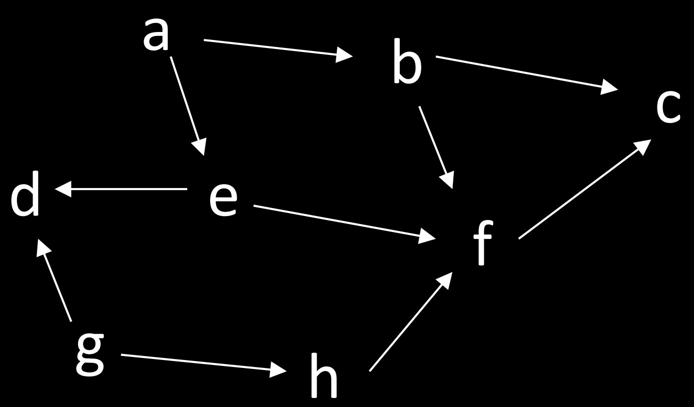
```
pair(X, Y, Z) :- drinks(X, Z), drinks(Y, Z), X \== Y.
```

Ex 3 : a graph

arc 

```
a(g, h).
a(g, d).
a(e, d).
a(h, f).
a(e, f).
a(a, e).
a(a, b).
a(b, f).
a(b, c).
a(f, c).
```

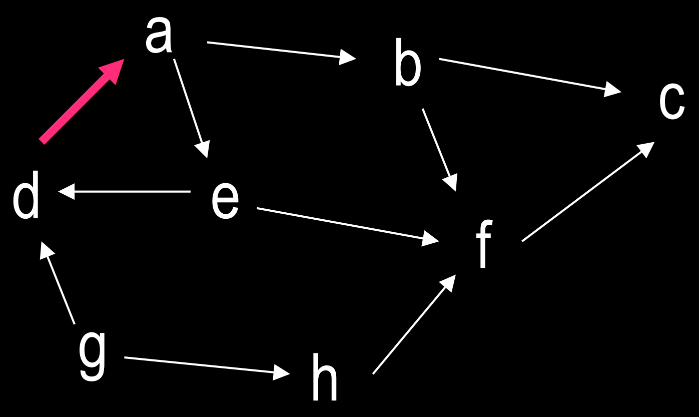
Note that Prolog can distinguish between the 0-ary constant a *(the name of a node) and the 2-ary functor* a *(the name of a relation).*



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

```
?- path(f, f).
?- path(a, c).
?- path(g, e).
?- path(g, X).
?- path(X, h).
```

But what happens if…


$$\begin{tikzcd}
 & a \arrow[r] \arrow[d] & b \arrow[r] \arrow[d] & c \\
d \arrow[ur] & e \arrow[l] \arrow[r] & f \arrow[ur] & \\
g \arrow[u] \arrow[r] & h \arrow[ur] & &
\end{tikzcd}$$

```
a(g, h).
a(g, d).
a(e, d).
a(h, f).
a(e, f).
a(a, e).
a(a, b).
a(b, f).
a(b, c).
a(f, c).
a(d, a).
```

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

This program works only for acyclic graphs. The program may infinitely loop given a cyclic graph. We need to leave a ‘trail’ of visited nodes. This is accomplished with a data structure (to be seen later).

Exercise 1

- Let rel(aa,bb) means « aa is in a (oriented) relation with bb ». Express the queries:

1. Is there a relation btw alpha and beta?
2. Is there a symmetrical relation btw alpha and beta?
3. What is in relation with alpha?
4. What is in a symmetrical relation with alpha?
5. What is in relation with both alpha and beta?
6. What are the couples which share a symmetrical relation?
7. What are the objects for which there exist a reflexive relation?
8. What are the triplets with in transitive relation?

Exercise 2: choose your menu

Database = Prolog Facts

```
%these are the starters
starter( 'Prawn Cocktail' ).
starter( 'Pate Maison' ).
starter( 'Avocado Vinaigrette' ).
starter( 'Stuffed Mushrooms' ).
starter( 'Parma Ham With Melon' ).
starter( 'Asparagus Soup' ).
% and here are the main courses
(vegetables are included!)
fish( 'Dover Sole' ).
meat( 'Fillet Steak' ).
meat( 'Calves Liver' ).
meat( 'Chicken Kiev' ).
meat( 'Ragout Of Lamb' ).
fish( 'Poached Salmon' ).
```

```
% the desserts follow here
dessert( 'Chocolate Fudge Cake' ).
dessert( 'Vanilla Ice Cream' ).
dessert( 'Peach Melba' ). dessert( 'Waffles
With Maple Syrup' ). dessert( 'Fresh Fruit
Salad' ). dessert( 'Apple And Blackberry
Pie' ).
% and these are the wines wine( 'Chablis' ).
wine( 'Muscadet Sur Lie' ). wine( 'Beaujolais
Nouveau' ). wine( 'Nuits Saint George' ).
wine( 'Gewurztraminer' ). wine( 'Cabernet
Shiraz' ).
```

Exercise 2: express in Prolog

1. Is the « Asparagus Soup » a starter?
2. What are all the starters?
3. A main dish is either a fish dish or a meat
4. Any meal consists of a starter, a main dish, a dessert and a wine.
5. We would like to choose a meal with meat (give two possible queries, which one is the most efficient? → draw the call tree)

Arithmetic

Arithmetic expressions use the standard operators such as + - * / (besides others)

Operands are simple terms or arithmetic expressions

EXAMPLE

(7 + 89 * sin(Y+1)) / (cos(X) + 2.43)

Arithmetic expressions must be *ground* at the instant Prolog is required to evaluate them

Comparing arithmetic expressions (1/3)

E1 =:= E2	tests whether the values of E1 and E2 are equal
E1 =\= E2	tests whether their values of E1 and E2 are unequal
E1 < E2	tests whether the value of E1 is less than the value of E2

Likewise we have

> for greater

>= for greater or equal

=< for equal or less

EXAMPLES

```
?- X=3, (2+2) =:= (X+1).          succeeds
?- (2+2) =:= (X+1), X=3.          gives an error
?- (2+2) > X.                     gives an error
```

Comparing arithmetic expressions (2/3)

The value of an arithmetic expression E may be computed and assigned to a variable X by the call

```
X  is  E
```

EXAMPLES

?- X is (2+2).	succeeds and binds X / 4
?- 4 is (2+2).	succeeds
?- 5 is (2+2).	gives an error
?- X is (Y+2).	gives an error

Comparing arithmetic expressions (3/3)

Do not confuse is **with** =

X=Y means “X can be unified with Y” and is rarely needed

EXAMPLES

```
?- X = (2+2).        succeeds and binds X / (2+2)
?- 4 = (2+2).        does not give an error, but fails
?- X = (Y+2).        succeeds and binds X / (Y+2)
```

The ”is” predicate is used **only** for the very specific purpose

variable **is** arithmetic-expression-to-be-evaluated

Exercise 3: birthday database

```
person(paul, 10,5, 1992)
person(pierre, 29, 2, 1992)
person(arnaud, 10, 3, 2011)
person(luc, 14, 1, 2005)
person(michele, 21, 2, 2012)
person(laurence, 29,2, 1924)
person(elodie, 10, 5, 1934)
person(alain, 17, 11, 1977)
person(veronique, 22, 5, 1956)
person(lucie, 10, 12, 1940)
person(sophie, 30, 11, 1994)
person(victor,1,12, 1994)
person(philippe, 2, 12, 1994)
person(andre, 21, 4, 1947)
```

Exercise 3

1. Write the predicate “curious(N)” which prints the name of the persons born the 29/02.
2. Write the predicate “baby(N)” which prints the name of the babies born in the last 2 years.
3. Write the predicate “age(N,V)” which prints the name and the age of a person.
4. Write the predicate “active(N)” which prints the name of the persons which are (strictly) more than 23 and (strictly) less than 63
5. Write the predicate “lucky(N)” which prints the name of the persons which are either less than 23 or more than 63.
6. Write the predicate “adult(N)” which prints the name of the persons above 18.

You can use the predicate date(X) which is always true and unify X with the data structure **datime(Year, Month, Day, Hour, Minute, Second).**

Outline

- Introduction
- Recursion and Lists
- Cut
- Built-in predicates

NB: Some slides are taken from © Patrick Blackburn, Johan Bos & Kristina Striegnitz

Recursion?

- We want to compute **the sum of all digits in a given number**.
  - E.g. if the number is 26, we must return 8 (=2 + 6); if the number is 13758, we must return 24.
- First, some remarks on *iteration* and *recursion* for people who know other programming languages such as C/C++, Java, Pascal, ...
  - (*don't* worry if you do not know any of these languages !).

The C/C++, Java, ... -style

- In most programming languages one can iterate an instruction with for, while or repeat (this repeats the instruction until some condition holds). A typical 'iterative' solution for our example-problem is the next program (this is **not** prolog):

```
total = 0
while number>9 do
        total = total + (number mod 10)
        number = number / 10
total = total + number
```

The prolog-style

- No for, while or repeat. In prolog iteration is done through **recursion**: define the solution of the complete problem in terms of solving a smaller variant of the same problem.

- More precisely, we can solve our example-problem in prolog as follows:
  - we express the solution as either the solution of the 'simple case' (the number only has one digit) or
  - we reduce the problem to a simpler variant of the problem, namely for a number with one digit less.

Count

```
% the simple case: if the number has only one digit
% then this number is the sum of all digits
count(Number,Number):- Number < 10.

/* the recursive case: cut away the first digit, let someone (=recursive call) add the rest,
and add your digit to that sum. */
count(Number,Sum):-
        Number >= 10,
        Digit is mod(Number,10),
        NewNumber is Number // 10,
        count(NewNumber,TmpSum), Sum is TmpSum + Digit.
```

'//' stands for integer division (X is 555 / 10 results in X being 55.5, but X is 555 // 10 results in X being 55).

Question: Draw a call tree for a call of count(437,Result).

Exercise 4

- Suppose a group of students gets the following instructions:
  - if you receive from your left neighbour a piece of paper with a single letter on it, go to the blackboard and write that letter on it, then return to your seat
  - if you receive from your left neighbour a piece of paper with more than one letter on it, tear the first letter off, give the rest of the paper to your right neighbour, and when he/she is finished with 'processing' that piece of paper, go to the blackboard and write your letter on it (to the right of what has been written on it already)

Exercise 4 cont.

1. What does this 'program' do?
2. What would appear on the blackboard if the leftmost student is given a piece of paper with the word 'hello'?
3. Try to write a prolog program that solves this task.
   – You may assume that there is a predicate split_string that splits a word into the first letter and the rest of the word. E.g. ?-split_string('hello',First,Rest) results in First being 'h' and Rest being 'ello'. Of course, you can also use the built-in predicate « write »

Exercise 5

1. Write a predicate all_even/1 taking as input a number. The predicate should succeed if all digits in that number are even, otherwise it should fail.
   - Draw a call tree for all_even(2496).
2. Write a predicate count_even/2 taking as input parameter a number. The output parameter should be the number of even digits in the input-number. E.g. count_even(24,A) should result in A being 2; count_even(25,A) should result in A being 1.
   - Draw a call tree for these two examples.

Exercise 6:
Try to understand the following program

```
even(N):- A is mod(N,2),
          A = 0.
odd(N):- A is mod(N,2),
         A = 1.
funny(1):- write('I will quit'),nl.
funny(N):-
        even(N),
        write(N), nl, % <-this
        NewN is N // 2,
        funny(NewN).
funny(N):-
        odd(N),
        write(N), nl, % <-this
        N>1,
        NewN is 3 * N + 1,
        funny(NewN).
```

Exercise 6 cont.

1. What is the result of the query funny(6) (if you only ask for the first answer of Prolog)?
2. Write down what prolog would write to the screen.
3. Do the same for funny(7)?
4. What happens when you put the two lines with write(N), nl (marked with % <-this) before the even(N) and odd(N) in the definition of funny/1?
5. What would be the second answer of Prolog if you ask for another solution ?

Lists?

An important recursive data structure often used in Prolog programming

- A list is a finite sequence of elements
- Examples of lists in Prolog:

```
[mia, vincent, jules, yolanda]
[mia, robber(honeybunny), X, 2, mia]
[ ]
[mia, [vincent, jules], [butch, friend(butch)]]
[[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]
```

Definitions

- [] is the empty list
- . predicate is like Scheme’s cons:
```
?- A = .(1, .(2, .(3, []))).          A=[1, 2, 3]
```
- […] shorthand syntax:
```
?- A = [1,2,3]                        A=[1, 2, 3]
```
- [E1...|Tail] notation
```
?- A = [1,2|3].                        A=[1, 2|3]
?- A = [1,2|[3]].                     A=[1, 2, 3]
```

Exercise 7

Can you guess the result of the following queries?

1. ?- [A,B] = .(a,.(.(b,.(c,[])),.(d,[]))).

2. ?- [A,B,C] = .(a, .(.(b,.(c,[])),.(d,[]))).

3. ?- [A,B,C,D] = .(a,.(.(b,.(c,[])),.(d,[]))).

Head and Tail

- A non-empty list can be thought of as consisting of two parts
  - The head
  - The tail
- 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
  - <u>The tail of a list is always a list</u>

Head and Tail example 1

- [mia, vincent, jules, yolanda]

  Head:
  Tail:

Head and Tail example 2

- [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

  Head:
  Tail:

Head and Tail example 3

- [dead(z)]

  Head:
  Tail:

Head and tail of empty list

- The empty list has neither a head nor a tail
- For Prolog, [ ] is a special simple list without any internal structure

- The empty list plays an important role in recursive predicates for list processing in Prolog

The built-in operator |

- Prolog has a special built-in operator | which can be used to decompose a list into its head and tail
- The | operator is a key tool for writing Prolog list manipulation predicates

The built-in operator |

```
?- [Head|Tail] = [mia, vincent, jules, yolanda].

Head = mia
Tail = [vincent,jules,yolanda]
yes

?-
```

The built-in operator |

```
?- [X|Y] = [mia, vincent, jules, yolanda].

X = mia
Y = [vincent,jules,yolanda]
yes

?-
```

The built-in operator |

```
?- [X|Y] = [ ].

no

?-
```

The built-in operator |

```
?- [X,Y|Tail] = [[ ], dead(z), [2, [b,c]], [], Z, [2, [b,c]]] .

X = [ ]
Y = dead(z)
Z = _4543
Tail = [[2, [b,c]], [ ], Z, [2, [b,c]]]
yes

?-
```

Notice: Lists need not be homogeneous

```
?- A = "Hi".                                A=[72,105]

?- A = [1,"Hi",greg].                       A=[1,[72,105],greg]

?- A = [1,g], B=[A,A].                      A=[1,g]
                                            B=[[1,g],[1,g]]

?- A = [1,g], B=[A|A].                      A=[1,g]
                                            B=[[1,g],1,g]
```

Exercise 8

- Will these queries succeed? If so, what will be the variable bindings? If not, why does it fail? If you are not sure, draw the tree representations of the lists!!

1. ?- .(a, .(b, .(c,.(d,[])))) = [A,B|C].
2. ?- [A|B] = .(a, .(.(b,.(c,[])),.(d,[]))).
3. ?- [A,B|C] = .(a, .(.(b,.(c,[])),.(d,[]))).
4. ?- [a, b | []] = [H | T].
5. ?- [a, b, c] = [E1, E2 | T].
6. ?- X = a, Y = [1, [2,3]], Z = [X | Y].
7. ?- X = a, Y = [1, [2,3]], Z = [X,Y].
8. ?- X = [a], Y = [1, [2,3]], Z = [X | Y].

Anonymous variable

- Suppose we are interested in the second and fourth element of a list

```
?- [X1,X2,X3,X4|Tail] = [mia, vincent, marsellus, jody, yolanda].
X1 = mia
X2 = vincent
X3 = marsellus
X4 = jody
Tail = [yolanda]
yes

?-
```

Anonymous variables

- There is a simpler way of obtaining only the information we want:

```
?- [ _,X2, _,X4|_ ] = [mia, vincent, marsellus, jody, yolanda].
X2 = vincent
X4 = jody
yes

?-
```

The underscore is the anonymous variable

The anonymous variable

- Is used when you need to use a variable, but you are not interested in what Prolog instantiates it to
- Each occurrence of the anonymous variable is independent, i.e. can be bound to something different

Member

- One of the most basic things we would like to know is whether something is an element of a list or not
- So let’s write a predicate that when given a term X and a list L, tells us whether or not X belongs to L
- This predicate is usually called member/2

member/2

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

```
?- member(yolanda,[yolanda,trudy,vincent,jules]).
```

member/2

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

```
?- member(yolanda,[yolanda,trudy,vincent,jules]).
yes
?-
```

member/2

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

```
?- member(vincent,[yolanda,trudy,vincent,jules]).
```

member/2

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

```
?- member(vincent,[yolanda,trudy,vincent,jules]).
yes
?-
```

member/2

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

```
?- member(zed,[yolanda,trudy,vincent,jules]).
```

member/2

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

```
?- member(zed,[yolanda,trudy,vincent,jules]).
no
?-
```

member/2

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

```
?- member(X,[yolanda,trudy,vincent,jules]).
```

member/2

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

```
?- member(X,[yolanda,trudy,vincent,jules]).
X = yolanda;
X = trudy;
X = vincent;
X = jules;
no
```

Rewriting member/2

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

Recursing down lists

- The member/2 predicate works by recursively working its way down a list
  - doing something to the head, and then
  - recursively doing the same thing to the tail
- This technique is very common in Prolog and therefore very important that you master it
- So let’s look at another example!

Example: a2b/2

- The predicate a2b/2 takes two lists as arguments and succeeds
  - if the first argument is a list of as, and
  - the second argument is a list of bs of exactly the same length

```
?- a2b([a,a,a,a],[b,b,b,b]).
yes
?- a2b([a,a,a,a],[b,b,b]).
no
?- a2b([a,c,a,a],[b,b,b,t]).
no
```

Defining a2b/2: step 1

```
a2b([],[]).
```

- Often the best way to solve such problems is to think about the simplest possible case
- Here it means: the empty list

Defining a2b/2: step 2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

- Now think recursively!
- When should a2b/2 decide that two non-empty lists are a list of as and a list of bs of exactly the same length?

Testing a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,a,a],[b,b,b]).
yes
?-
```

Testing a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,a,a,a],[b,b,b]).
no
?-
```

Testing a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,t,a,a],[b,b,b,c]).
no
?-
```

Further investigating a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,a,a,a,a], X).
X = [b,b,b,b,b]
yes
?-
```

Further investigating a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b(X,[b,b,b,b,b,b,b]).
X = [a,a,a,a,a,a,a]
yes
?-
```

Exercise/Example

- Let L = [a,2,c,4]. Write a prolog predicate elemodd/2 that creates a list with only the element at an odd position in the input list so [a,c].

- Think about how you can use the recursion
- Think about how you can go through your list
- Think about how to stop going through your list
- Think about the different possible cases
- Think about how to give the result back

Exercise 9

1. Write a predicate listlength(+,?) that, given a list, counts the number of elements in that list (we call this the length of the list).
   - Think about the basic clause. What is the most simple list for which we can solve this problem? Some people will think of a list that only contains a single element, but there exists an even simpler list: the empty list!
   - Think about the recursive clause: if I can solve a smaller problem, how can I solve this problem? A smaller problem is counting the number of elements in a shorter list. Suppose I split the given list into its head and tail and I can solve the problem for the tail, how to solve the problem for the complete list?

2. What will be the result of the following queries

?- listlength([1,[2,3]],L).

?- listlength([1|[2,3]],L).

Does this predicate work like you had expected?

Exercise 10

Do the following queries succeed or fail? In case of success, what are the variable bindings?

1. ?- length(List,3).
2. ?- length(List,3), nth_element(1,List,a).
3. ?- length(List,3), nth_element(1,List,a), nth_element(2,List,b), nth_element(3,List,c).

The definitions of length(+,?) and nth_element/3 are the usual ones:

```
length([],0).
length([_|T],N) :-
   length(T,N2),
   N is N2+1.
```

```
nth_element(1,[H|_],H).
nth_element(N,[_|T],E) :-
N>1,
N1 is N-1,
          nth_element(N1,T,E).
```

Append

- Write the predicate append/3 such that append(L1,L2, L3) succeeds if L3 is the concatenation of L1 and L2.

```
?-append([a,c],[b],L).
L = [a,c,b]
yes
```

We can NOT use L3 = [L1IL2] to concatenate (or append) two lists…..why?

Accumulating Parameters

- An often used technique in Prolog is that of using an accumulating parameter: we gradually build up a 'result' in a parameter, and we return this result when we arrive in the base case.
- An advantage of accumulating parameters is that they can be used to make a program 'tail-recursive' (roughly speaking, a program is tail-recursive if the recursive call is at the very end in the body of the recursive clause).
- This is good because a tail-recursive program **uses less memory** than its non-tail-recursive counterpart.

Efficiency issues

Sometimes using an accumulating parameter also makes a program **faster**. Suppose we want to reverse a list. Without accumulating parameter we can write this as follows:

```
reverse([], []).
reverse([Head|Tail], Reversed):-
          reverse(Tail, TailReversed),
          append(TailReversed, [Head], Reversed).
```

This program has quadratic time-complexity (i.e. the time needed to reverse a list with N elements using this program is proportional to N^2)!

Reduced complexity

- Using an accumulating parameter, we can get a program with linear time-complexity (time proportional to N).

```
reverse(List, Reversed):-
        reverse_acc(List, Reversed, []).
% the accumulater is initially empty
reverse_acc([], Result, Result).

% Return the result (=the accumulated data)
reverse_acc([Head|Tail], Result, Acc):- reverse_acc(Tail, Result, [Head|
Acc]).
```

Built-in predicates for list processing (in GNU Prolog)

- append(List1, List2, List12): succeeds if the concatenation of the list List1 and the list List2 is the list List12. This predicate is re-executable on backtracking (e.g. if List12 is instantiated and both List1 and List2 are variable).
- member(Element, List): succeeds if Element belongs to the List.
- memberchk/2 is similar to member/2 but only succeeds once.
- reverse(List1, List2): succeeds if List2 unifies with the list List1 in reverse order.
- delete(List1, Element, List2): removes all occurrences of Element in List1 to provide List2. A strict term equality is required, cf. (==)/2
- select/3 , subtract/3 , permutation/2 , prefix/2, suffix/2 , sublist/2 , last/2 , length/2 , nth/3 ,max_list/2, min_list/2, sum_list/2 , maplist/2-8, sort/2, msort/2, keysort/2 sort/1, msort/1, keysort/1 …

Outline

- Introduction
- Lists
- Cut
- Built-in predicates

Backtracking is not always what we want

- Patterns may match where we do not intend
- Backtracking is expensive—we may know more about our problem and can help the algorithm be “smarter”
- We may want to specify a situation that we know definitively results in failure

delete_all example

```
/*  delete_all(List,E,Result) succeeds if Result is a list just
    like List except that all elements E are missing. */
delete_all([], E, []).

delete_all([E|Tail], E, Res) :-
        delete_all(Tail, E, Res).

delete_all([Head|Tail], E, [Head|Res]) :-
        delete_all(Tail, E, Res).
```

A query for delete_all

```
?- delete_all([1,2,3],2,R).                 R=[1,3]   ;
                                          R=[1,2,3]   ;
                                                 No
```

Why this solution?

delete_all can succeed in any of three ways...

```
delete_all([], E, []).

delete_all([E|Tail], E, Res) :-
        delete_all(Tail, E, Res).

delete_all([Head|Tail], E, [Head|Res]) :-
        delete_all(Tail, E, Res).
```

- Order in file only tells which rules are attempted first—later matching rules can be used after backtracking!

delete_all has multiple matching rules

```
delete_all([E|Tail], E, Res) :-
        delete_all(Tail, E, Res).


delete_all([Head|Tail], E, [Head|Res]) :-
        delete_all(Tail, E, Res).
```

```
delete_all([2,3],2,R).
```

- Can be proven using either of the above!
  R=[3], or R=[2,3]

Third rule contained implicit assumption

```
delete_all([Head|Tail], E, [Head|Res]) :-
        delete_all(Tail, E, Res).
```

- Want above rule to apply only when Head is not E
- That is exactly the complement of rule 2
- So we can make the algorithm only try rule 3 if rule 2 did not succeed

Use a “cut” — !

- We can make rule 2 prevent backtracking with the “cut” operator, written !

```
delete_all([E|Tail], E, Res) :-
        delete_all(Tail, E, Res), !.
```

- Now the search algorithm will not try any other rules for delete_all after the above rule succeeds
- ! succeeds and stops further backtracking for more results

The query again

```
?- delete_all([1,2,3],2,R).                    R=[1,3]  ;
                                                    No
```

- Now we get only the single correct solution!

Cut effect

A program clause having a cut looks like:

head :- preceding-calls, !, other-calls.

The cut acts *only* when it is selected as the next call to be evaluated, and it then

- prunes all untried ways of evaluating

  whichever call invoked in the clause containing the cut

  (all clauses below the one with the cut are discarded, as if they did not exist for this particular call) ex: another clause with the same head

and

- prunes all untried ways of evaluating

  the calls in this clause which precede (left of) the cut

NB: The goals at the right of the cut are executed normally (i.e., they can backtrack).

Cut divides problem into backtracking regions

```
foo :- a, b, c, !, d, e, f.
```

- Search may try various ways to prove a, b, and c, backtracking freely while solving those sub-goals
- Once a, b, and c are proved, that sub-answer is frozen, and d, e, f must be proved without changing a, b, or c

Controversy over cut

- Prolog is meant to be declarative
- cut operator alters the behavior of the built-in searching algorithm
- No declarative interpretation for cut—you must think about resolution to understand its effects

cut and not

- We can write the not predicate using a cut operator:
  ```
  not(P) :- P, !, fail.
  not(P).
  ```
- Uses built-in fail predicate that always fails
- Cut operator prevents the search algorithm from backtracking to use the second rule to satisfy P when the first rule already failed
- 2nd rule applies only if P cannot be proven

!, fail combination

- Another common use of the cut is with fail
- Use to force failure in special cases that are easy to rule out immediately

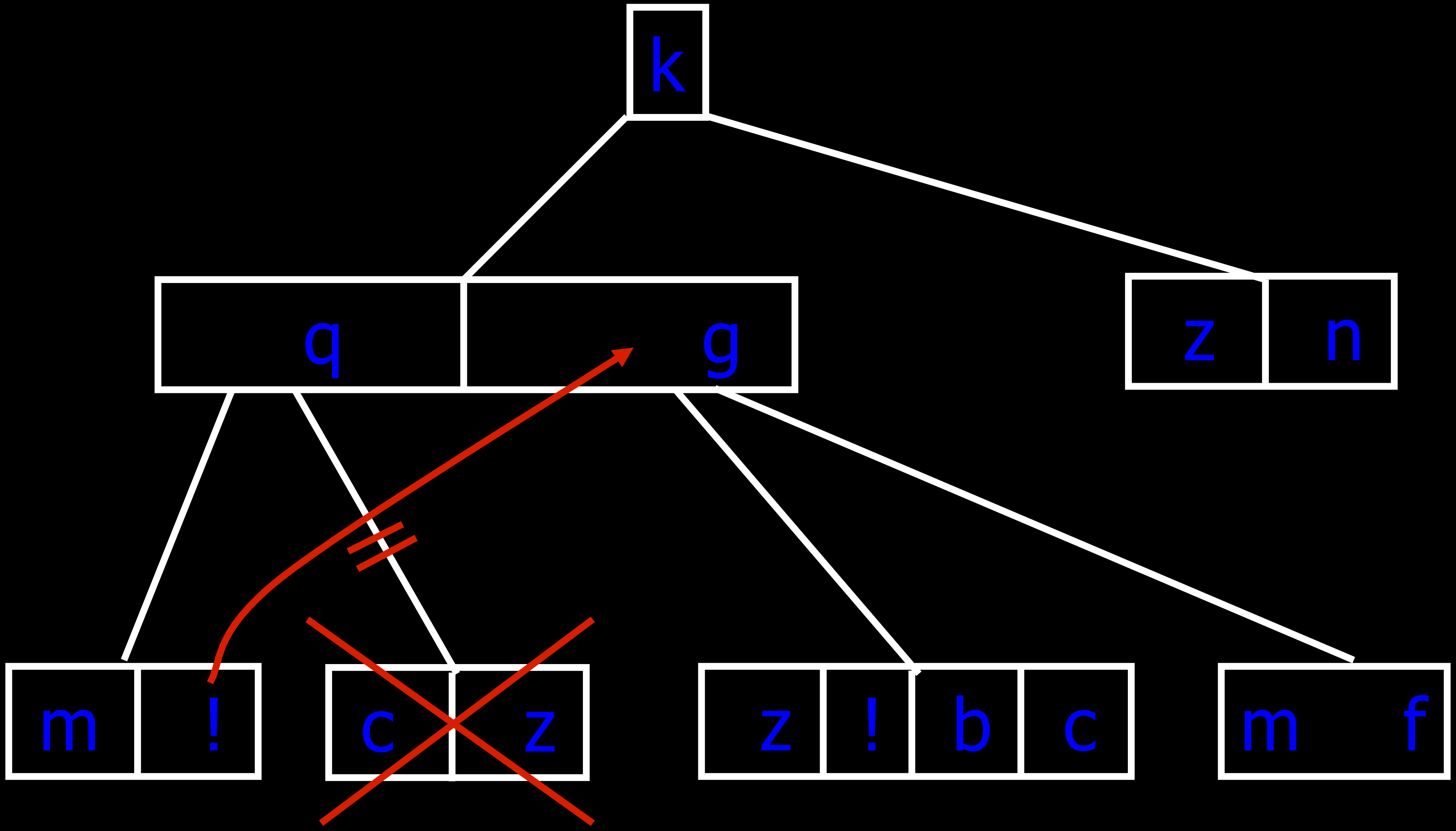
```
average_taxpayer(X) :-
   lives_in_bermuda(X), !, fail.

average_taxpayer(X) :-
   /* complicated rules here… */
```

IF THEN ELSE

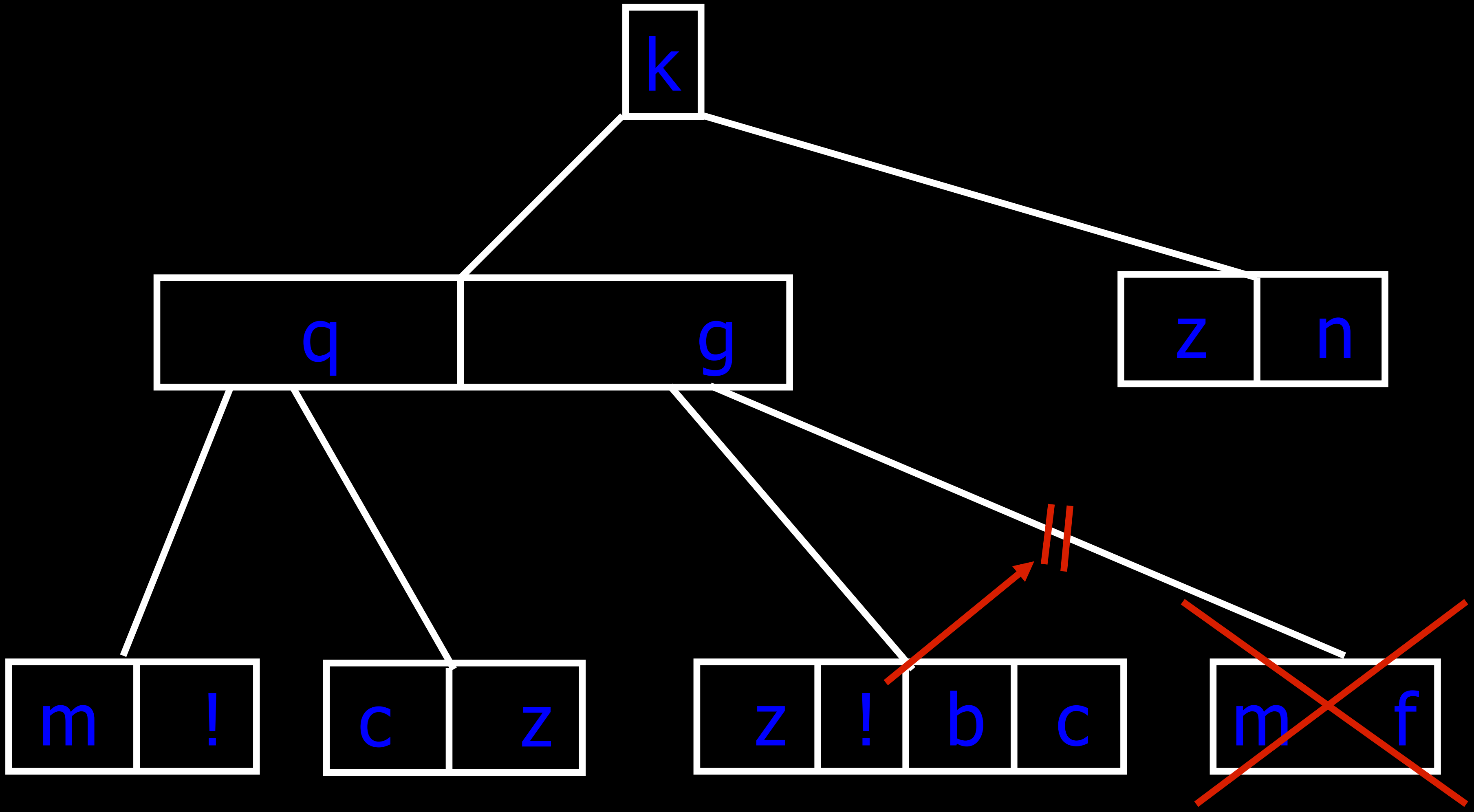
% = max(X, Y, Z)
% true if Z is the max btw X and Y.

```
max(X,Y,Z) :-
      X >= Y,
      !,
      Z = X.

max(X,Y,Y).
```

Tree view of a cut



Tree view of a cut



EXAMPLE

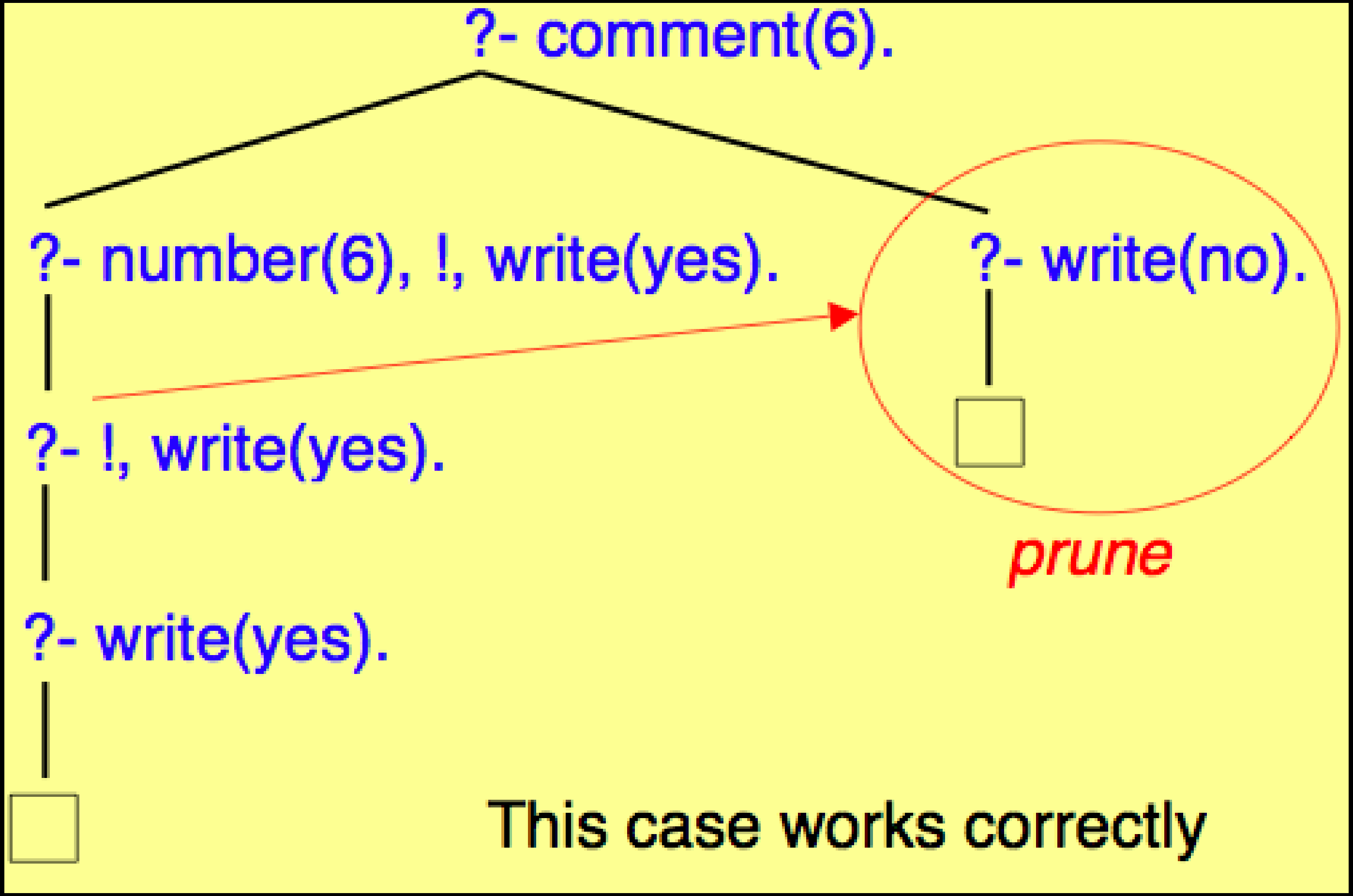
This program tests a term X and prints a comment

The intention is that if X is a number then
the comment is yes but is otherwise no

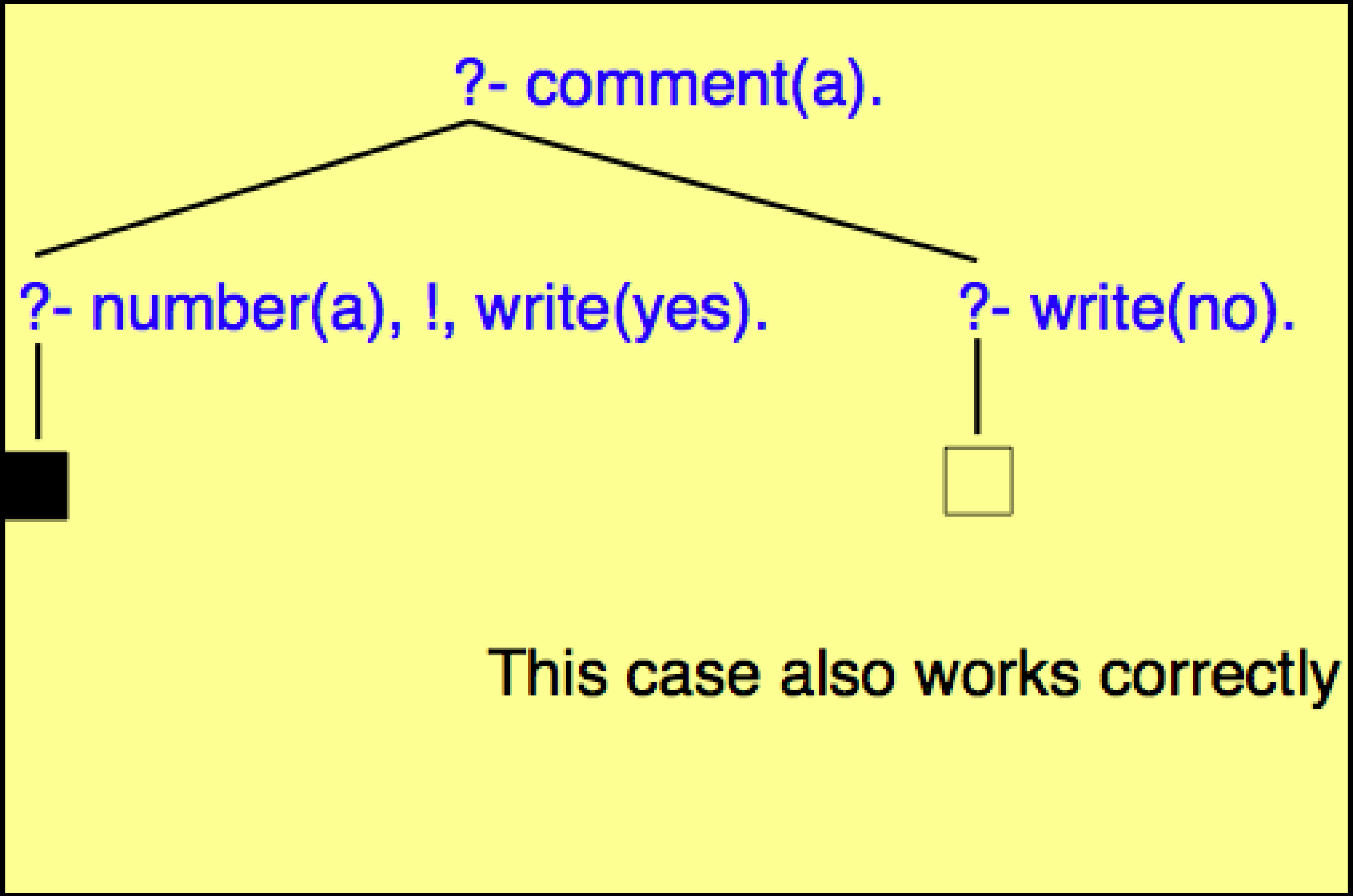
```
comment(X) :- number(X), !, write(yes).
 comment(X) :- write(no).
```

Will it work (assuming X is ground)?

EXAMPLE - call tree (1/2)



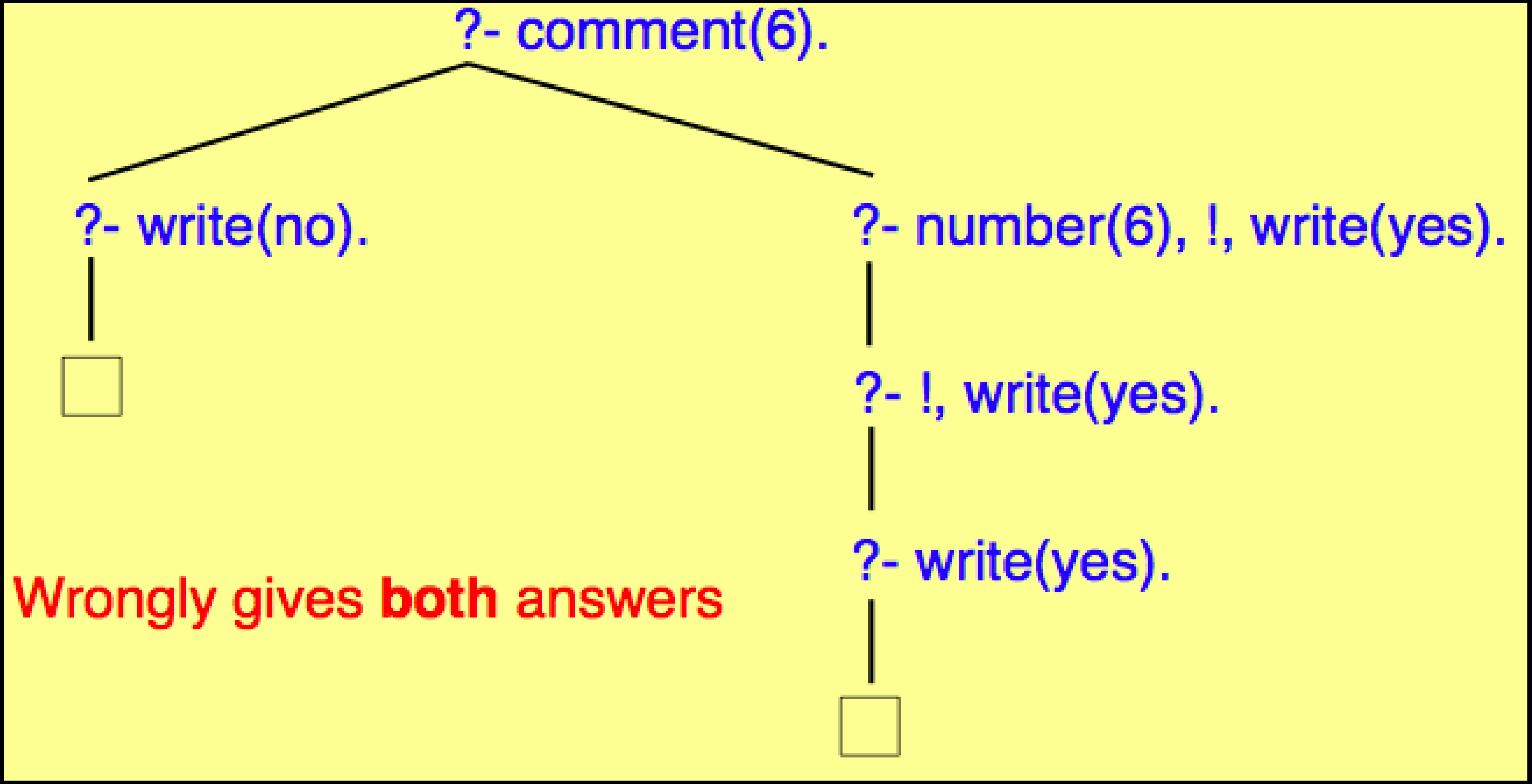
EXAMPLE - call tree (2/2)



This case also works correctly

BUT - suppose we reorder the clauses as:

```
comment(X) :- write(no).
comment(X) :- number(X), !, write(yes).
```



Exercise 11

```
pp0(X,Y):- qq(X), qq(Y).
pp0(0,1).
pp1(X,Y):- qq(X), qq(Y), !.
pp1(0,1).
pp2(X,Y):- qq(X),!, qq(Y).
pp2(0,1).
qq(1).
qq(2).
```

Draw the call tree for the calls

```
?-pp0(X,Y).
?-pp1(X,Y).
?-pp2(X,Y).
```

Exercise 12: back to the menu

- What is the answer to the following calls:
  1. ?-meal(A,B,C,D), !.
  2. ?-meal(A,B,C,D), meat(B), !.
  3. ?-meal(A,B,C,D), !, meat(B).

Exercise 13: remove/add

- Write (with and without CUT) the predicate remove(+P, +E1, -E2) which succeeds if the list E2 = the list E1 – {P} (fail if P is not in E1). E2 and E1 do not contain duplicates.

- Write (with and without CUT) the predicate add(+P, +E1, -E2) which succeeds if the list E2 = the list E1 + {P} (if P is already in E1, E2 = E1).
You can use memberset(X,E) which succeeds if an element X is in a set E and the predicate \+(predicate) which mean ¬predicate.

Exercise 14: Women and children first

- Consider that the predicates male/1, female/1 and age/2 exist.

Write the predicate save(+X) which succeeds if X can be saved (if it is a man under 14 or a woman).

  – Try without the cut first
  – Write a version with the cut (be careful with the position of the cut)

Another example

Define least(X, Y, L) to mean “L is the least of X and Y”

```
least(X, Y, X) :- X<Y, !.
least(X, Y, Y).
```

```
?- least(1, 2, L).          correctly succeeds, binding L / 1
?- least(2, 1, L).          correctly succeeds, binding L / 1
```

BUT ...

```
?- least(1, 2, 2).          wrongly succeeds
?- least(a, b, b).          wrongly succeeds
```

and this happens however the clauses are ordered

Cut: THE GREAT MORAL

1. If you can reasonably avoid using cut, do so
2. If you must use it, take great care with clause order
3. In any event, compute only the **TRUTH**

EXAMPLE

```
comment(X) :-    number(X), write(yes).
comment(X) :- \+number(X), write(no).
```

This program, having no cut, potentially evaluates number(X) twice, depending on the query - a small overhead

Outline

- Introduction
- Lists
- Cut
- Built-in predicates (most of the time extra-logical predicates such as =/= or “is” or \==)

TYPE-CHECKING

To check argument types you can make use of the following, which are supplied as primitives:

atom(X)	X is an atom
number(X)	X is a number
integer(X)	X is an integer
var(X)	X is (an unbound) variable
nonvar(X)	X is not a variable
compound(X)	X is a compound term

Prolog INPUT/OUTPUT

- Use built-in predicates (ex: write, read)
- The logical meaning of those predicates is “true” but we only care about the side effects (here to read or write a term)
- http://gprolog.univ-paris1.fr/manual/html_node/gprolog038.html

```
/* write_liste(Liste): write the Liste in the terminal with one element on each line*/


write_liste([]).
write_liste([X|Xs]) :-
      write(X),nl,
      write_liste(Xs).
```

Example INPUT/OUTPUT

- The following program:
  - Read the name of the Input and Ouput files
    - The name should finish by “.”
  - Collect the current « stdin » and « stdout » and redirect it.
  - For each input, call the predicate « work(X) »
  - Put everything as it was before at the end.

```
example :-
        write('Input file: '),
        read(FileIn),
        write('Output file: '),
        read(FileOut),
        open(FileIn, read, In),
        open(FileOut, write, Out),
        current_input(Stdin),
        set_input(In),
        current_output(Stdout),
        set_output(Out),
        repeat,
        read(X),
        ( X == end_of_file ->
                !;
                work(X), nl, fail
        ),
        set_input(Stdin),
        close(In),
        set_output(Stdout),
        close(Out).

work(X) :- write(X). /%this is just a stupid program
```

Input/ Output: format

- format(+stream_or_alias, +character_code_list_or_atom, +list)
- format(+character_code_list_or_atom, +list)

format(SorA, Format, Arguments) writes the Format string replacing each format control sequence F by the corresponding element of Arguments (formatted according to F) to the stream associated with the stream-term or alias SorA.

Format examples

```
format(' ~w', [Terme])       % ~w = write
format(' ~w~n', [Terme])    % ~n = nl
format('  - ~w~n'  [Terme])

      =
          write(' - '),
          write(Terme),
          nl
```

```
% = write_liste(Liste)
% ecrit Liste vers le terminal, avec
% chaque element sur une ligne.

write_liste([]).
write_liste([X|Xs]) :-
      format('  - ~w~n', [X]),
      write_liste(Xs).
```

Disjunction

- Disjunction between calls can always be expressed using procedures offering alternative clauses

  EXAMPLE

  ```
  out_of_range(X, Low, High) :- X<Low.
  out_of_range(X, Low, High) :- X>High.
  ```

- Equivalently, use Prolog's disjunction connective, the semi-colon

  EXAMPLE

  ```
  out_of_range(X, Low, High) :- X<Low ; X>High.
  ```

- With mixtures of conjunctions and disjunctions, use parentheses to avoid ambiguity:

  EXAMPLE

  ```
  a :- b, (c ; (d, e)).
  ```

Negation (1/7)

Prolog does not have an explicit connective for classical negation.

It is arguable that we do not need one

EXAMPLE

$innocent(X) \leftarrow \neg guilty(X)$ in classical logic

In practice we do not establish the innocence of X by *proving the negation* of “X is guilty”

Instead, we establish it by *finitely failing to prove* “X is guilty”

Negation (2/7)

Prolog provides a special operator \+ read as
“finitely fail to prove”

So in Prolog we would write

```
innocent(X) :-  \+guilty(X).
```

The operational meaning of \+ is

\+P succeeds iff P fails finitely

\+P fails finitely iff P succeeds

Negation (3/7)

EXAMPLE

```
person(bob).          likes(bob, frank).
person(chris).
person(frank).

sad(X) :-
        person(X),
        person(Y), X \== Y, \+likes(Y, X).
```

"X is sad if someone else fails to like X"

Using the data, bob, chris and frank are sad,
because in each case someone else fails to like them

Negation (4/7)

\+ does not perfectly simulate classical negation

EXAMPLE

$p \leftarrow \neg p$	classically implies p but
`p :- \+p.`	cannot solve `?- p.`
	(it will fail infinitely, not finitely)

So, p is a logical consequence in the first case, but is not a computable consequence in the second

Negation (5/7)

EXAMPLE

```
person(bob).          likes(bob, frank).
person(chris).
person(frank).

very_sad(X) :-
        person(X),
        \+ (person(Y), X \== Y, likes(Y, X)).
```

"X is very sad if no one else likes X"
Here, just bob and chris are very sad,
because in each case no one else likes them

Syntax Note - essential to put a space between \+ and (

Negation (6/7)

We can reformulate the previous example as

```
very_sad(X) :- person(X), \+liked(X).
liked(X) :- person(Y), Y \== X, likes(Y, X).
```

This is the *safe* option:
if our Prolog does not reject non-ground \+ calls then it may compute intuitively wrong answers when it evaluates them

The above \+liked(X) call is *ground* when it is selected, because the person(X) call has already grounded X

Rqs Negation (7/7)

The \+ operator partially compensates for the head of a clause

being restricted to a single predicate

If we want to use the knowledge that, say,

$A \vee B \leftarrow C$ we can approximate it

by A :- C, \+B.

or by B :- C, \+A.

or by both of them together

GENERATE-AND-TEST: forall

Generate-and-test is a feature of many algorithms

It can be formulated as

generate items satisfying property P,

test whether they satisfy property Q

P acts as a *generator*   Q acts as a *tester*

Example 1

"X is happy if all friends of X like logic"

In classical logic we can express this by

happy(X) ← ("Y)(friend(X, Y) → likes(Y, logic))

In Prolog we can rewrite this as

```
happy(X) :- forall(friend(X, Y), likes(Y, logic)).
```

in which the forall will

generate each friend Y of X

test whether Y likes logic

Example 2

Show that L is a list of positive numbers

```
all_pos_nums(L) :-    is_list(L),
                      forall(member(U, L), (number(U), U>0)).
```

and some appropriate is_list procedure

Remarks about Forall

- Could be defined manually as:

```
forall(P, Q) :- \+ (P, \+ Q).
```

  “no way of solving P fails to solve Q”

- Note that forall does not perfectly simulate "

  ("...)(P $\rightarrow$ P) is true in classical logic

but

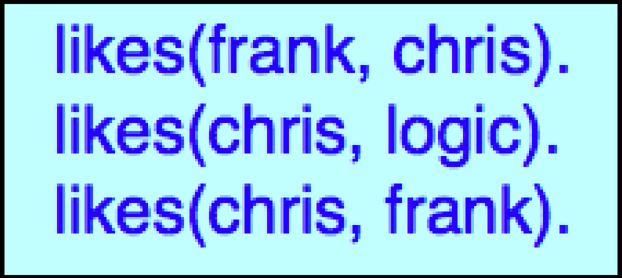
  forall(P, P) succeeds only if the number of ways of solving P is finite

AGGREGATION: findall

- Often we want to collect into a single list all those items satisfying some property
- Prolog supplies a convenient primitive for this:

findall(Term, Call-term, List)

EXAMPLE



```
likes(frank, chris).
likes(chris, logic).
likes(chris, frank).
```

To find all those whom chris likes:

```
?- findall(X, likes(chris, X), L).
```

this returns L / [logic, frank]

Findall: Example 1

- To find all sublists of [a, b, c] having length 2:

```
?- findall([ X, Y ], sublist([ X, Y ], [ a, b, c ]), S).
```

this returns S / [[a, b], [b, c]]

- Given any list X, construct the list Y obtained by replacing each member of X by E:

```
replace(X, E, Y) :- findall(E, member(_, X), Y).
```

Then,

```
?- replace([ a, b, c ], e, Y).
```

returns Y / [e, e, e]

```
?- replace([ a, b, c ], [ 0 ], Y).
```

returns Y / [[0], [0], [0]]

Example 2

- Construct a list L of pairs (X, F) where X is a person and F is a list of all the friends of X:

```
friend_list(L) :-
      findall( (X, F),
              (person(X), findall(Y, friend(X, Y), F)),                    L).
```

So here we have a findall inside a findall

- Construct a list L of persons each of whom does whatever

chris does:

```
clones_of_chris(L) :-
       findall(X,(person(X),
             forall(does(chris, Y), does(X, Y)) )
                    , L).
```

So here we have a forall inside a findall

Example 3

Given a list L of classes, test whether all of them contain more females than males:

```
mostly_female_classes(L) :-
        forall(
                (member(C, L),
        findall(F, (member(F, C), female(F)), Fs),
        findall(M, (member(M, C), male(M)),  Ms),
        length(Fs, NF),
        length(Ms, NM)
                ),
        NF > NM).
```

So here we have findalls inside a forall

META-PROGRAMMING

- This concerns programs that variously access, control or analyse other programs or their components
- It is a feature of many declarative formalisms and gives them a high degree of expressiveness
- It is approximately comparable to the use of higher-order functions in a functional programming language
- In Prolog, most meta-programming exploits the fact that

  terms and predicates have
  identical syntactic structure

EXAMPLE of meta-predicates

```
overcome_with_joy(X) :- user_of(X, prolog).
```

In the above, user_of(X, prolog) is a *predicate*

```
overcome_with_joy(X) :- true_that(user_of(X, prolog)).
```

In the above, user_of(X, prolog) is an *argument (term)*

BUILT-IN META-PREDICATES

We have already met some of these:

```
\+P          forall(P, Q)          findall(Term, Q, List)
```

Here, P and Q are *object-level arguments*,
but are interpreted as *call-terms at the meta-level*

Their run-time manipulation can use the same unification mechanism as used for ordinary object-level terms

EXAMPLE

```
choose(X, wants(chris, X)).
?- choose(Y, Q), forall(nice(Y), Q).
```

From this query we get the derived query

```
?- forall(nice(Y), wants(chris, Y)).
```

by binding X / Y, Q / wants(chris, Y)

The =.. primitive

This is another built-in meta-predicate
It relates a term to a list comprising that term’s principal functor and arguments

chris =.. L	binds L / [chris]
happy(chris) =.. L	binds L / [happy, chris]
likes(X, prolog) =.. L	binds L / [likes, X, prolog]
T =.. [append, X, Y, Z]	binds T / append(X, Y, Z)
T =.. [s, s(0)]	binds T / s(s(0))

Example

- From any given non-variable term, extract a list L of all that term’s functors with their arities
- For instance, we want the query

  ?- functors(p(a, f(X, g(b)), Y), L).

  to return L / [(p, 3), (a, 0), (f, 2), (g, 1), (b, 0)]
- **Syntax Note**: Prolog atoms are just functors whose arity is 0

Here is the program (make sure you understand it)

```
functors(Term, [ ]) :- var(Term), !.

functors(Term, [(F, Arity) | Functors]) :-
        Term =.. [F | Args],
        length(Args, Arity),
        findall(E, (  member(Arg, Args),
                      functors(Arg, Es),
                      member(E, Es)), Functors).
```

DYNAMIC CLAUSES

- Clauses can be created, consulted or deleted dynamically

- Their head relations can be declared as “dynamic”, but Sicstus does not insist upon this, unless those relations are additionally defined by explicit procedures

  e.g. `:- dynamic likes/2.` forces likes to be dynamic

- The most common primitives acting on dynamic clauses are:
  - clause - finds a clause body, given the head relation
  - asserta or assertz - creates a clause
  - retract - deletes a clause

THE "ASSERT(a/z)" PRIMITIVE

This has the form assert(Clause)

```
?- assert(likes(chris, prolog)).
```

adds to the dynamic-clause-base the clause likes(chris, prolog).

```
?- assert((likes(X, prolog) :- wise(X))).
```

adds to the dynamic-clause-base the clause
likes(X, prolog) :- wise(X).

THE “RETRACT” PRIMITIVE

This has the form retract(Clause)

EXAMPLE

```
?- retract((likes(X, haskell) :- crazy(X))).
```

deletes from the dynamic-clause-base the clause
likes(X, haskell) :- crazy(X).

Additional note

To retract *all* current dynamic clauses for a relation P, execute the call retractall(P(...)) in which each argument of P is an underscore, as in retractall(likes(_, _))

Example

Simulating destructive assignment

Suppose that a 2-dimensional array “a” of numbers is represented by a set of assertions which have already been set up using assert:

a(I, J, V) *represents* a[I, J] = V

Suppose now we want to update “a” so that any element previously <0 is altered to become, say, 10. We can do this by evaluating the call-term

```
forall( (a(I, J, V), V<0),
        (retract(a(I, J, V)), assert(a(I, J, 10))) )
```

Controlling the flow of computation

- Control how queries are proved.

(we limit ourself to their use in normal Prolog program but they can also be used to create Meta-Interpreters.)

- The main predicate of this type is **call/1**. : takes one argument in the form of a goal (i.e. a single term) and checks whether the goal succeeds.

```
|?- call(write( 'Hello?' )).
Hello?
yes
```

- Mostly used to call goals constructed using =.. , functor/3 and arg/3.

A Conjunction of Goals

- A conjunction of goals (P∧Q) can be called by collecting the goals together in round brackets.

```
| ?- X = ( Y=[a,b,f,g], member(f,Y) ), call(X).

X = [a,b,f,g]=[a,b,f,g], member(f,[a,b,f,g]),
Y = [a,b,f,g] ?
yes
```

- The two goals Y=[a,b,f,g] and member(f,Y) are conjoined as one term and instantiated with X.
- call(X) then calls them in order and will only succeed if all the goals contained within X succeed (hence, it is checking if the conjunction of the two goals is true).

A Conjunction of Goals (2)

- The actual job of conjoining goals is performed by the ‘,’ operator.

  (‘,’ = the logical $\wedge$)

  ```
  ?- (3,4) = ','(3,4).
  yes
  ```
- This is a right-associative operator:
  - You can see this using **?- current_op(1000, xfy, ’ ,’).**
  - When used in a series of operators with the same precedence the comma associates with a single term to the left and groups the rest of the operators and arguments to the right.
  - *(works in a similar way to Head a Tail list notation).

```
| ?- (3,4,5,6,7,8) = (3,(4,(5,(6,(7,8))))).
yes
| ?- (3,4,5,6,7,8) = (((((3,4),5),6),7),8).
no
```

A Conjunction of Goals (2)

- Because of this associativity, groups of conjoined goals can be stripped apart by making them equal to (FirstGoal,OtherGoals).
  - **FirstGoal** is a single Prolog goal
  - **OtherGoals** may be a single goal or another pair consisting of another goal and remaining goals (grouped around ‘,’).

```
| ?- (3,4,5,6,7,8) = (H,T).
H = 3,
T = 4,5,6,7,8 ? ;
no
```

- This allows us to recursively manipulate sequences of goals just as we previously manipulated lists.

```
| ?- (3,4,5,6,7,8) = (A,B), B = (C,D), D = (E,F), ....
A = 3,              B = 4,5,6,7,8,
C = 4, D = 5,6,7,8,
E = 5, F = 6,7,8, .......
```

Repeated use of same test = recursion

Why use call?

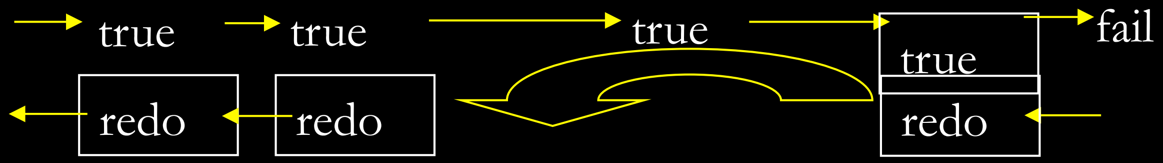
- But, why would we use call(X) as it seems to have the same function as just placing the variable X as a goal in your code:

  e.g. **X = (Y=[a,b,f,g], member(f,Y)), call(X).**

  X = (Y=[a,b,f,g], member(f,Y)), X.

- The main reason is because it keeps the solution of X isolated from the rest of the program within which call(X) resides.
  - Specifically, any cuts (!) within the conjoined set of goals X only stop backtracking within X.
  - It does not stop backtracking outside of call(X).

|?- goal1, goal2, call((goal3, !, goal4, goal5)).



A Disjunction of Goals (;)

- As well as ‘,’ = the logical AND ($\wedge$)
- We also have an operator that represents the logical OR ($\vee$).
  - **Goal1 ; Goal2** = A disjunction of Goal1 and Goal2.
  - This will succeed if either Goal1 or Goal2 are true.

```
| ?- 5<4;3<4.
yes
```

- Semicolon is an operator (**current_op(1100, xfy, ;)**) so it can be used in prefix position as well:

```
| ?- ;(5<4, 3<4).
yes
```

- This operator is right associative like ‘,’ :

```
| ?- (3;4;5;6;7;8) = (A;B), B = (C;D), D = (E;F), ....
A = 3, B = 4;5;6;7;8,
C = 4, D = 5;6;7;8,
E = 5, F = 6;7;8, .....
```

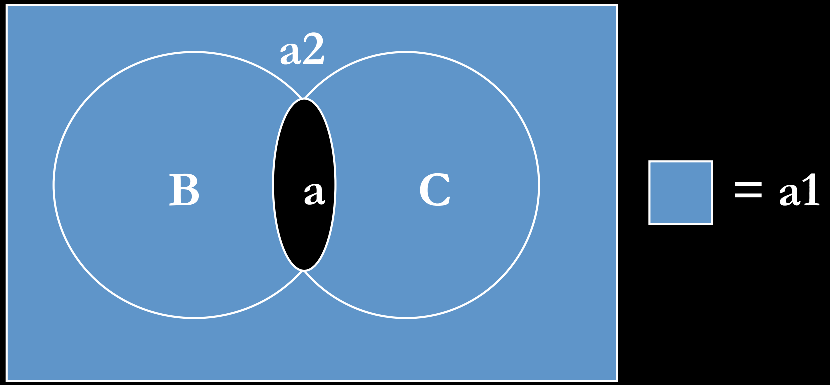
Creating a Conjoined 'not'

- Now that we can conjoin goals we can also check for their negation i.e. ¬ (P∧Q).
- Usually we are checking if a conjunction of terms in the body or a clause is true **e.g. a(X):- b(X), c(X).**
- But sometimes we want a predicate to succeed only if a conjunction of terms is false

  e.g a1(X):- \+ (b(X),c(X)).
- * The space before the prefix operator \+ and the brackets is important. If there was no space the interpreter would look for \+/2.
- This is distinct from:

  a2(X):- \+b(X), \+c(X).

  Which is equal to the space outside of both b and c.



Creating a Conjoined 'not' (2)

- But when would you use a conjoined not?
  - "*X is true if it is less than 4 or greater than 8.*"
    - For example, we want X to be true if it is 3 or 9.
- We could represent this using a disjunction:
  - (X<4 ; 8<X).
- Or we could represent it as a conjoined not:
  - \+ (4=<X, X=<8).
- This is possible as logic permits this transformation:
  - $\neg P \lor \neg Q = \neg (P \land Q)$
- Sometimes it might be easier to prove a goal (4=<X) rather than its opposite (X<4) so we would need to use a conjoined not: $\neg (P \land Q)$

If… then…else

- In Prolog there is a built-in operator (**->/2**) that allows you to make similar constructions:
  - "if X then Y" = X **->** Y.
  - "if X then Y else Z" = X **->** Y**;** Z.
  - n.b. the **;** is part of the "if..then…else…" construction so its scope is limited to the if.. construction.
- These can be used at the command line or within your predicate definitions.
- However, whenever we are writing Prolog rules we are already representing an "if….then…." relationship.
  - This rule a:- b, c, d, e -> f**;** g.
  - Is equal to a:- b, c, d, aux(X). aux(f):- e. aux(g).