Prolog Programming

Logic Programming Engineering

WS 2011/12

Paola Bruscoli

Prolog = Programming in Logic

Main advantages

- ease of representing knowledge
- natural support of non-determinism
- natural support of pattern-matching
- natural support of meta-programming

Other advantages

- meaning of programs is independent of how they are executed
- simple connection between programs and computed answers and specifications
- no need to distinguish programs from databases

Topics covered

- Preliminary concepts
- Terms; Deterministic evaluations; Input-output non-determinism
- Non-deterministic evaluation; Influencing efficiency; Unification
- List processing; Type checking; Comparing terms; Arithmetic
- Disjunction; Negation; Generation and Test Aggregation
- Controlling search
- Meta-programming

CONCEPT 1 - procedure definitions

Programs consist of *procedure definitions*

A procedure is a resource for evaluating something

EXAMPLE

a :- b, c.

This is read procedurally as a procedure for *evaluating* a by evaluating both b and c

Here “evaluating” something means determining whether or not it is true according to the program as a whole

The procedure

```
a  :-  b, c.
```

can be written in logic as a

$$a \leftarrow b \wedge c$$

and then read declaratively as

a is true if b is true and c is true

CONCEPT 2 - procedure calls

Execution involves evaluating calls, and begins with an *initial query*

EXAMPLES

```
?- a, d, e.
?- likes(chris, X).
?- flight(gatwick, Z), in_poland(Z), flight(Z, beijing).
```

The queries are asking whether the calls in them are true according to the given procedures in the program

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

Convention: terms beginning with an *upper-case letter* or an *underscore* are treated as *variables*

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

Queries and procedures both belong to the class of logic sentences known as *clauses*

CONCEPT 3 - computations

- 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

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

CONCEPT 4 - 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

CONCEPT 5 - finite failure

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

EXAMPLE

?- likes(bob, haskell). query

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

EXAMPLE

?- likes(chris, haskell). query

likes(chris, haskell) :- nice(haskell).

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

CONCEPT 6 - 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

CONCEPT 7 - multiple answers

A query may produce many computations

Those, if any, that succeed may yield multiple answers to the query

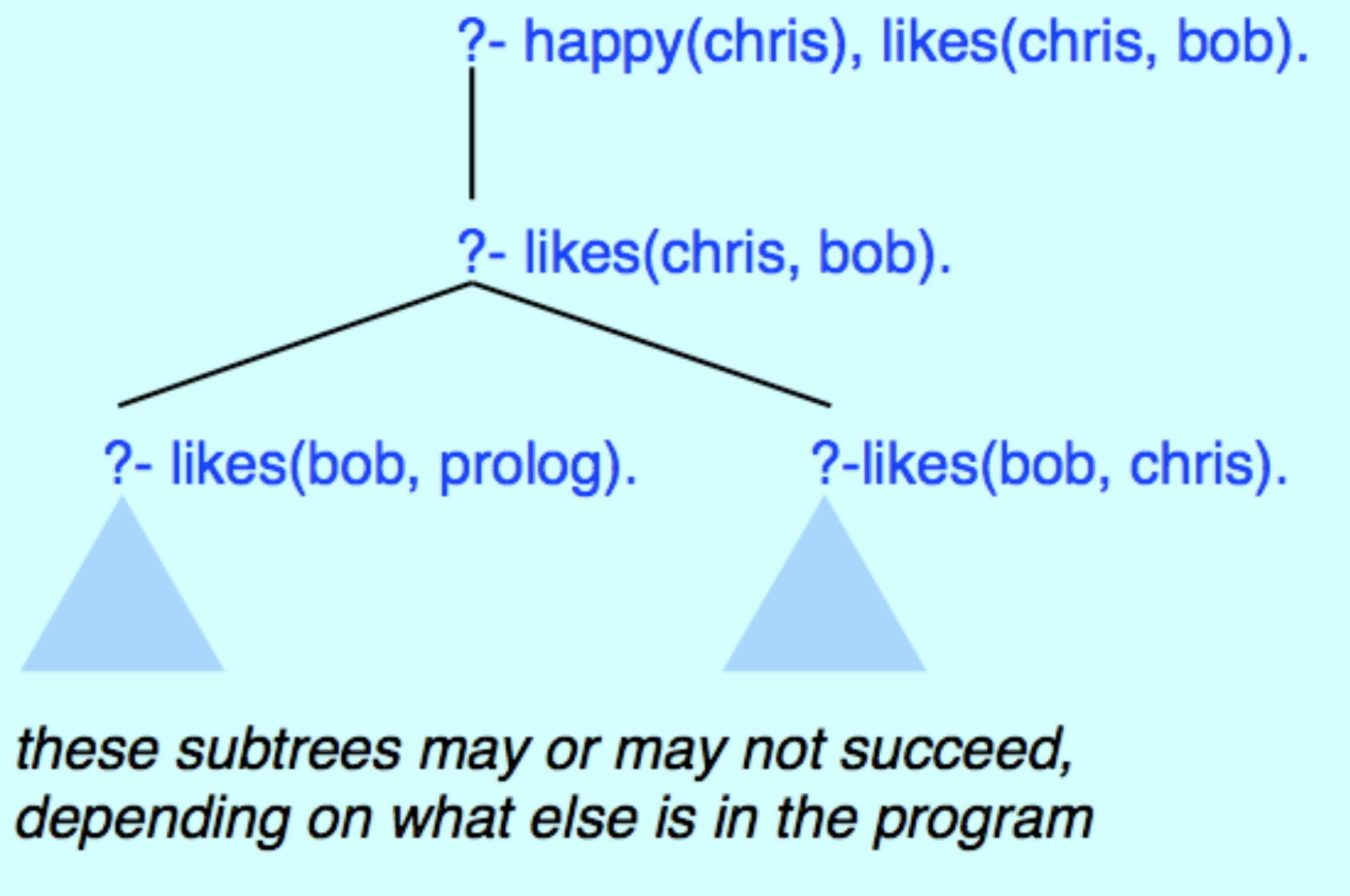
(not necessarily distinct)

EXAMPLE

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

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

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



CONCEPT 8 - answers as 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

CONCEPT 9 - 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) ← likes(X, prolog))

CONCEPT 10 - 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) 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).
```

Prolog Terms

- Terms are the items that can appear as the *arguments* of predicates
- They can be viewed as the *basic data* manipulated during execution
- They may exist statically in the given code of the program and initial query, or they may come into existence dynamically by the process of unification
- Terms containing no variables are said to be *ground*
- Prolog can process both ground and non-ground data
- A Prolog program can do useful things with a data structure even when that structure is partially unknown

SIMPLE TERMS

numbers

3 5.6 -10 -6.31

atoms

apple tom x2 'Hello there' [ ]

variables

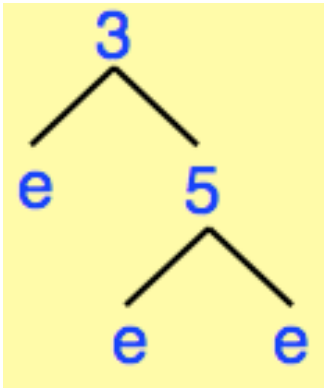
X Y31 Chris Left_Subtree Person _35 _

COMPOUND TERMS

prefix terms

mother(chris)

tree(e, 3, tree(e, 5, e)) i.e.

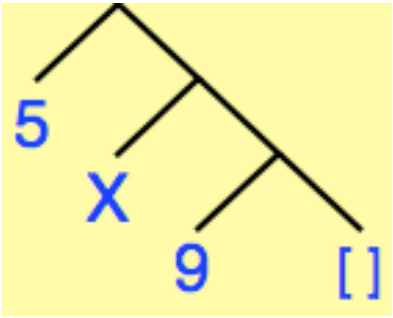


tree(T, N, tree(e, 5, e)) *a binary tree whose root and left subtrees are unknown*

list terms

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

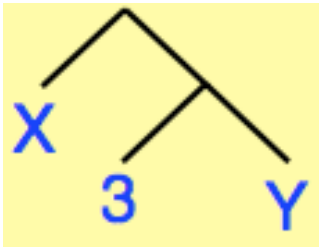
lists form a *subclass of binary trees*



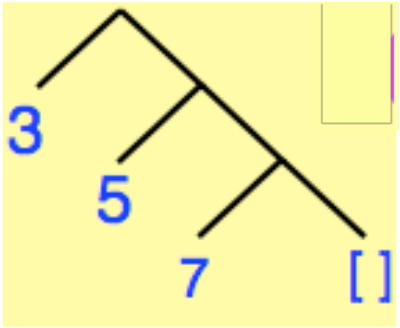
A vertical bar can be used as a separator to present a list in the form

[itemized-members | residual-list]

[X, 3 | Y]



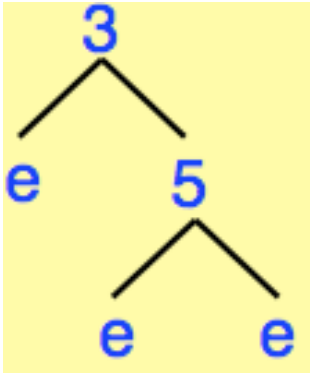
[3 | [5, 7]]
[3, 5, 7]
[3, 5 | [7]]



tuple terms

(bob, chris) (1, 2, 3) ((U, V), (X, Y))

(e, 3, (e, 5, e))



These are preferable (efficiency-wise) when working with fixed-length data structures

arithmetic terms

3*X+5 sin(X+Y) / (cos(X)+cos(Y))

Although these have an arithmetical syntax, they are interpreted arithmetically only by a specific set of calls, presented later on.

DETERMINISTIC EVALUATIONS

- Prolog is non-deterministic in general because the evaluation of a query may generate multiple computations
- 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

EXAMPLE

```
all_bs([ ]).
all_bs([ b | T ])  :-  all_bs(T).
```

This program defines a list in which every member is b

Now consider the query

```
?- all_bs([ b, b, b ]).
```

This will generate a deterministic evaluation

```
?- all_bs([ b, b, b ]).
?- all_bs([ b, b ]).
?- all_bs([ b ]).
?- all_bs([ ]).
?- .
```

So here the search tree comprises ONE branch (computation), which happens to succeed

EXAMPLE

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

```
app([ ], Z, Z).
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 the second program clause by making the bindings U / a X / [b] Y / [c, d] L / [a | Z]

So, we replace the call by the body of the clause, then apply the bindings just made to 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]

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

In the previous example, in each step, the call matched no more than one program clause-head, and so again the evaluation was *deterministic*

Note that, in general, each step in a computation produces bindings which are either propagated to the query variables or are kept on one side in case they contribute to the final answer

In the example, the final output binding is L / [a, b, c, d]

The bindings kept on one side form the so-called *binding environment* of the computation

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

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

SUMMARY

- given a program, we can pose any queries we like, whatever their modes
- some queries will generate just one computation, whereas others will generate many
- multiple successful computations may or may not yield distinct answers
- every answer is a logical consequence of the program

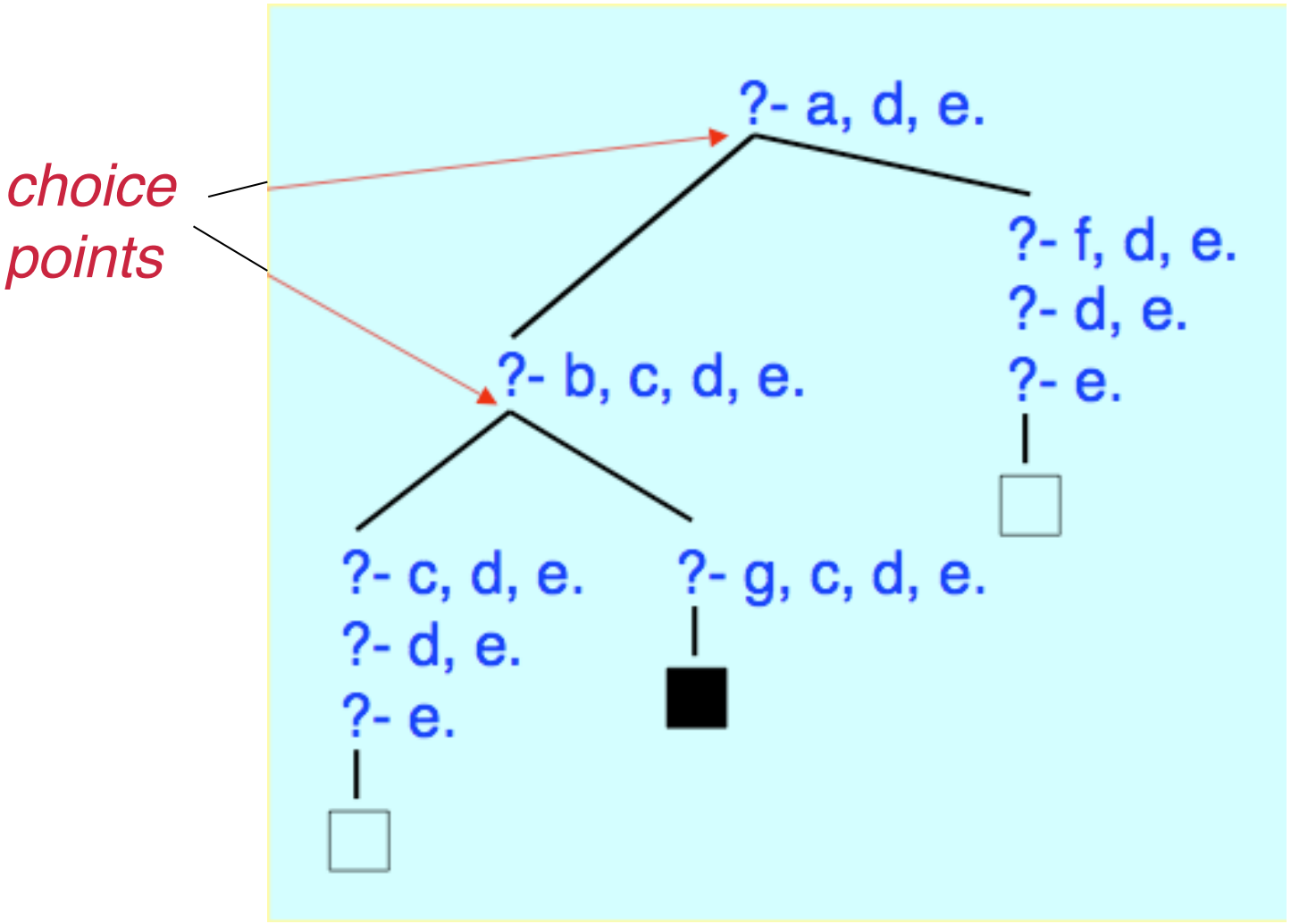
NON-DETERMINISTIC EVALUATIONS

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.
?- f, d, e.
?- d, e.
?- e.
?- b, c, d, e.
?- c, d, e.
?- d, e.
?- e.
?- g, c, d, e.

- Presented with several computations, Prolog generates them *one 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
- This is called **Prolog's search rule**:

  it prioritizes the branches in the search tree

EFFICIENCY

The efficiency with which Prolog solves a problem depends upon

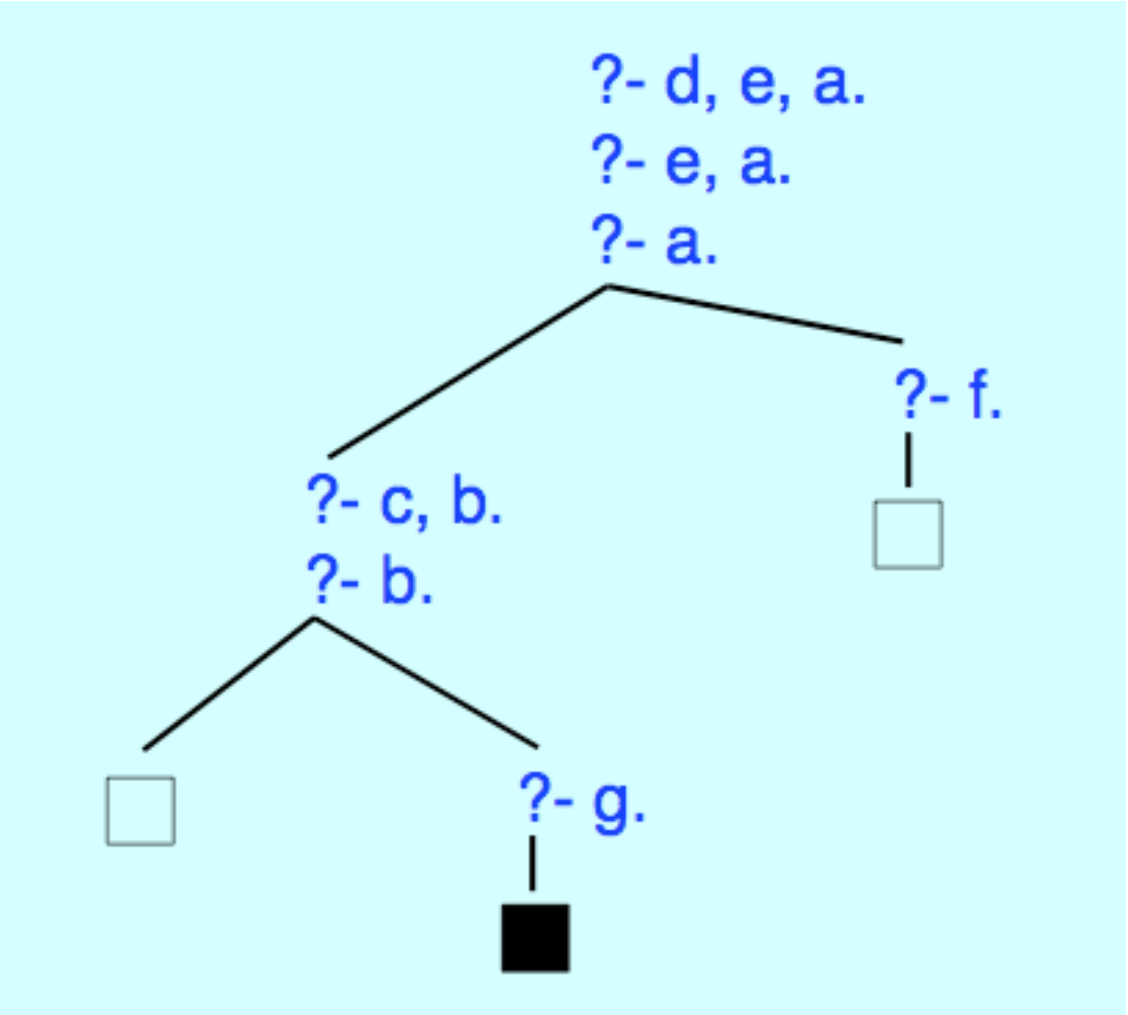
- the way knowledge is represented in the program
- the ordering of calls

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**



The policy for selecting the next call to be processed is called the computation rule and has a major influence upon efficiency

So remember ...

- a **computation rule** decides which call to select next from the query
- a **search rule** decides which program clause to apply to the selected call

and in Prolog these two rules are, respectively,

"choose the **first** call in the current query"

"choose the **first** applicable untried program clause"

UNIFICATION

- This is the process by which Prolog decides that a call can use a program clause
- The call has to be unified with the head
- Two predicates are unifiable if and only if they have a common instance

EXAMPLE

?- likes(Y, chris). likes(bob, X) :- likes(X, logic).

Let θ be the binding set { Y / bob, X / chris }

If E is any logical formula then Eθ denotes the result of applying θ to E, so obtaining an instance of E

likes(Y, chris)θ = likes(bob, chris)

likes(bob, X)θ = likes(bob, chris)

As the two instances are identical, we say that θ is a *unifier* for the original predicates

THE GENERAL COMPUTATION STEP

current query ?- P(args1), others.

program clause P(args2) :- body.

If θ exists such that $P(args1)\theta = P(args2)\theta$

then this clause can be used by this call to produce

derived query ?- bodyθ, othersθ.

Otherwise, this clause cannot be used by this call

EXAMPLE

?- app(X, X, [a, b, a, b]).

Along the successful computation we have

O1 = { X / [a | X1] } *these are the*

O2 = { X1 / [b | X2] } *output bindings*

O3 = { X2 / [ ] } *in the unifiers*

whose *composition* is { X / [a, b], X1 / [b], X2 / [ ] }

The *answer substitution* is then { X / [a, b] }

and applying this to the initial query gives the answer

app([a, b], [a, b], [a, b, a, b])

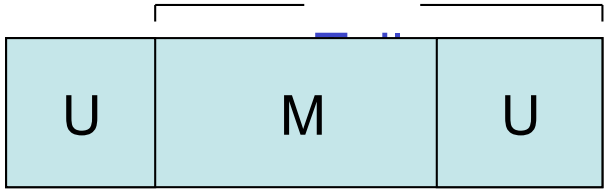
LIST PROCESSING

Lists are the most commonly-used structures in Prolog, and relations on them usually require recursive programs

EXAMPLE

To define a palindrome:

```
palin([ ]).
palin([U | Tail]) :-append(M, [U], Tail),
                    palin(M).
```



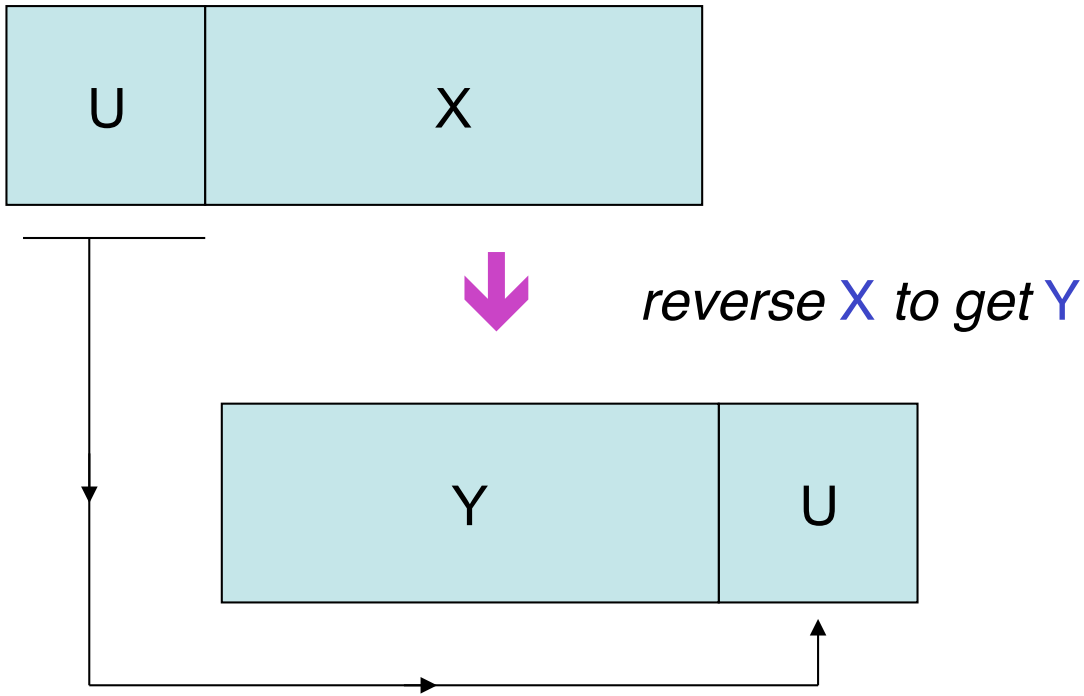
More abstractly:

```
palin([ ]).
palin(L) :-first(L, U), last(L, U),
            middle(L, M), palin(M).

first([U | _], U).

last([U], U).
last([_ | Tail], U) :- last(Tail, U).

middle([ ], [ ]).
middle([_], [ ]).
middle([_ | Tail], M) :- append(M, [_], Tail).
```

EXAMPLE

To reverse a list:

```
reverse([ ], [ ]).
reverse([U | X], R) :- reverse(X, Y),
                       append(Y, [U], R).
```



- Note that the program just seen is *not tail-recursive*
- If we try to force it to be so, by reordering the calls thus:

```
reverse([U | X], R) :- append(Y, [U], R),
                       reverse(X, Y).
```

then the evaluation is likely to go infinite for some modes.

- However, the following is *tail-recursive*:

```
reverse(L, R) :- rev(L, [ ], R).

rev([ ], R, R).
rev([U | Tail], A, R) :- rev(Tail, [U | A], R).
```

- With this program, the time taken to reverse a given list is proportional to the length of that list, and the runtime environment
- It does not generate a stack of pending calls

BUILT-IN LIST PRIMITIVES

Prolog contains its own library of list programs, such as:

append(X, Y, Z)	appending Y onto X gives Z
reverse(X, Y)	reverse of X is Y
length(X, N)	length of X is N
member(U, X)	U is in X
non_member(U, X)	U is not in X
sort(X, Y)	sorting X gives Y

To access these in Sicstus, include in your file

```
?- use_module(library(lists)).
```

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

With these primitives you can then define further type-checking procedures

EXAMPLES

To test whether a term is a list:

```
is_list(X) :- atom(X), X=[ ].
is_list(X) :- compound(X), X=[_ | _].
```

To test whether a term is a binary tree:

```
bintree(X) :- atom(X), X=e.
bintree(X) :- compound(X), X=t(L, _, R),
                    bintree(L), bintree(R).
```

COMPARING TERMS

Prolog has many primitives for comparing terms, including:

X = Y X unifies with Y

e.g. X=a succeeds, binding X / a

X == Y X and Y are identical

e.g. [a, b] == [a, b] succeeds, but

[a, b] == [a, X] fails

X \== Y X and Y are not identical

e.g. [a, b] \== [a, X] succeeds, without binding X

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

REMARK: Different Prolog systems may allow for more/less liberal grammars to compare expressions

COMPARING ARITHMETIC EXPRESSIONS

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

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). gives an error

?- X is (Y+2). gives an error

In SWI it succeeds!
SWI grammar permits a more general test

E1 is E2

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

EXAMPLE

Summing a list of numbers:

```
sumlist([ ], 0).
sumlist([ N | Ns], Total) :- sumlist(Ns, Sumtail),
                             Total  is  N+Sumtail.
```

This is not tail-recursive - the query length will expand in proportion to the length of the input list

Typical non-tail-recursive execution:

```
?- sumlist([ 2, 5, 8 ], T).
?- sumlist([ 5, 8 ]), T is 2+T1.
?- sumlist([ 8 ], T2), T1 is 5+T2, T is 2+T1.
?- sumlist([ ], T3), T2 is 8+T3, T1 is 5+T2, T is 2+T1.
?- T2 is 8+0, T1 is 5+T2, T is 2+T1.
?- T1 is 5+8, T is 2+T1.
?- T is 2+13.
?- .
```

succeeds with the output binding T / 15

EXAMPLE

Doing it tail-recursively:

```
sumlist(Ns, Total) :- tr_sum(Ns, 0, Total).

tr_sum([ ], Total, Total).
tr_sum([ N | Ns ], S, Total) :- Sub is N+S,
                                tr_sum(Ns, Sub, Total).
```

Here, tr_sum(Ns, S, T) means $T = S + \Sigma\ Ns$

Typical tail-recursive execution:

```
?- sumlist([ 2, 5, 8 ], T).
?- tr_sum([ 2, 5, 8 ], 0, T).
?- Sub is 2+0, tr_sum([ 5, 8 ], Sub, T).
?- tr_sum([ 5, 8 ], 2, T).
      :
?- tr_sum([ 8 ], 7, T).
      :
?- tr_sum([ ], 15, T).
?- .
```

and again succeeds with T / 15

Here the query length never exceeds two calls and each derived query can overwrite its predecessor in memory

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

Prolog does not have an explicit connective for classical negation.
It is arguable that we do not need one

EXAMPLE
innocent(X) ← ¬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"

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

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

\+ does not perfectly simulate classical negation

EXAMPLE

p ⇐ ¬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

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 (

Some Prologs (but not Sicstus) require \+P to be *ground* at the instant it is selected for evaluation

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

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

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

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

In classical logic we can express this by

$happy(X) \leftarrow (\forall Y)(friend(X, Y) \rightarrow likes(Y, logic))$

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

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

- Some Prologs (but not Sicstus) supply forall as a primitive

- If necessary we can define it ourselves:

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

"no way of solving P fails to solve Q"

- Note that forall does not perfectly simulate ∀

(∀...)(P → P) is true in classical logic

but

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

CALL-TERMS

A call-term is anything that the Prolog interpreter can be asked to evaluate logically, such as

an atomic call
a call \+ P where P is a call-term
a conjunction of call-terms
a disjunction of call-terms
besides others ...

In a call forall(P, Q) the arguments P and Q may be any call-terms, however complex - but if they are not atomic then they need to be parenthesized

AGGREGATION

- 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]

EXAMPLE

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]]

EXAMPLE

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

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

EXAMPLE

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

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

CONTROLLING SEARCH

- The extent to which a search tree is generated can be controlled by use of the “cut” primitive, denoted by !
- When executed, a cut prunes some parts of the search tree
- It is motivated by a wish to suppress unwanted computations
- It can be placed anywhere in a query or program where one might otherwise place an ordinary call
- Any number of cuts can be used

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 the clause containing the cut**

and

- **prunes all untried ways of evaluating the calls in this clause which precede the cut**

EXAMPLE

```
s :- p, b.
s :- a.

p :- c.
p :- q, !, r.
p :- d.

q.
q :- e.

etc ...
```

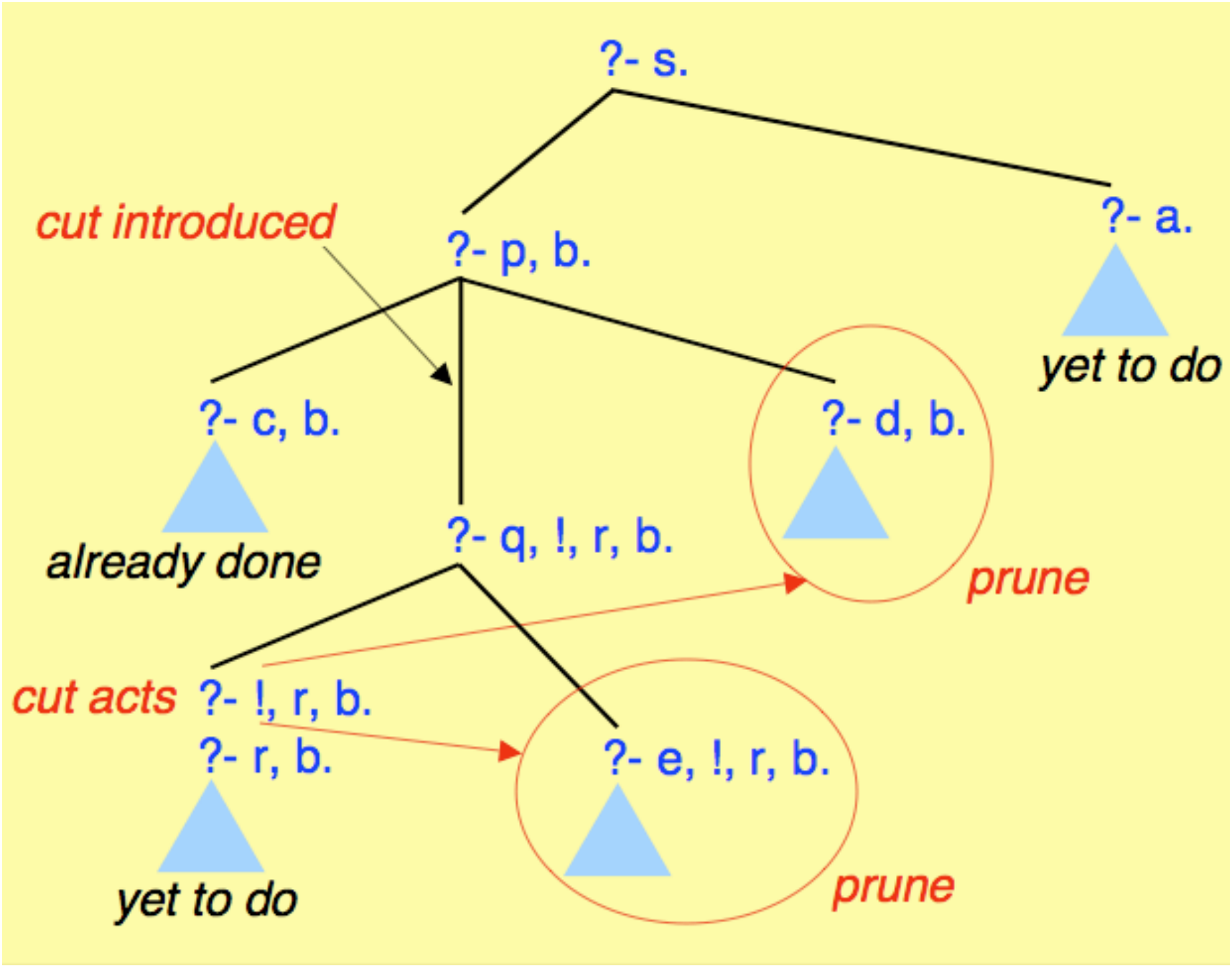
If selected, the cut will prune

all untried ways of evaluating p

and

all untried ways of evaluating q

All other branches in the tree waiting to be constructed will survive this cut's action


?- s.
cut introduced
?- p, b.
?- a.
yet to do
?- c, b.
?- d, b.
already done
?- q, !, r, b.
prune
cut acts
?- !, r, b.
?- r, b.
?- e, !, r, b.
yet to do
prune

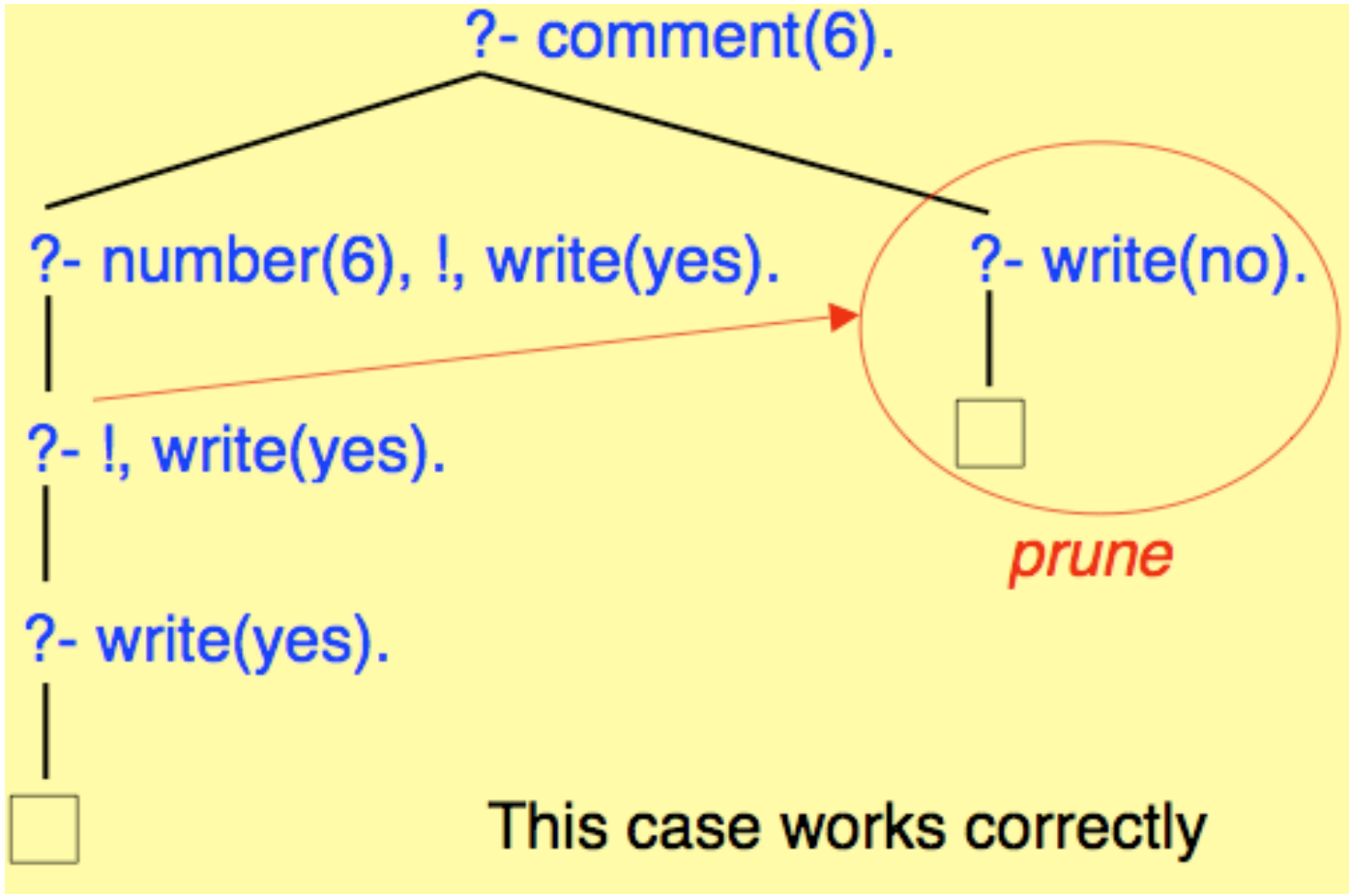
EXAMPLE

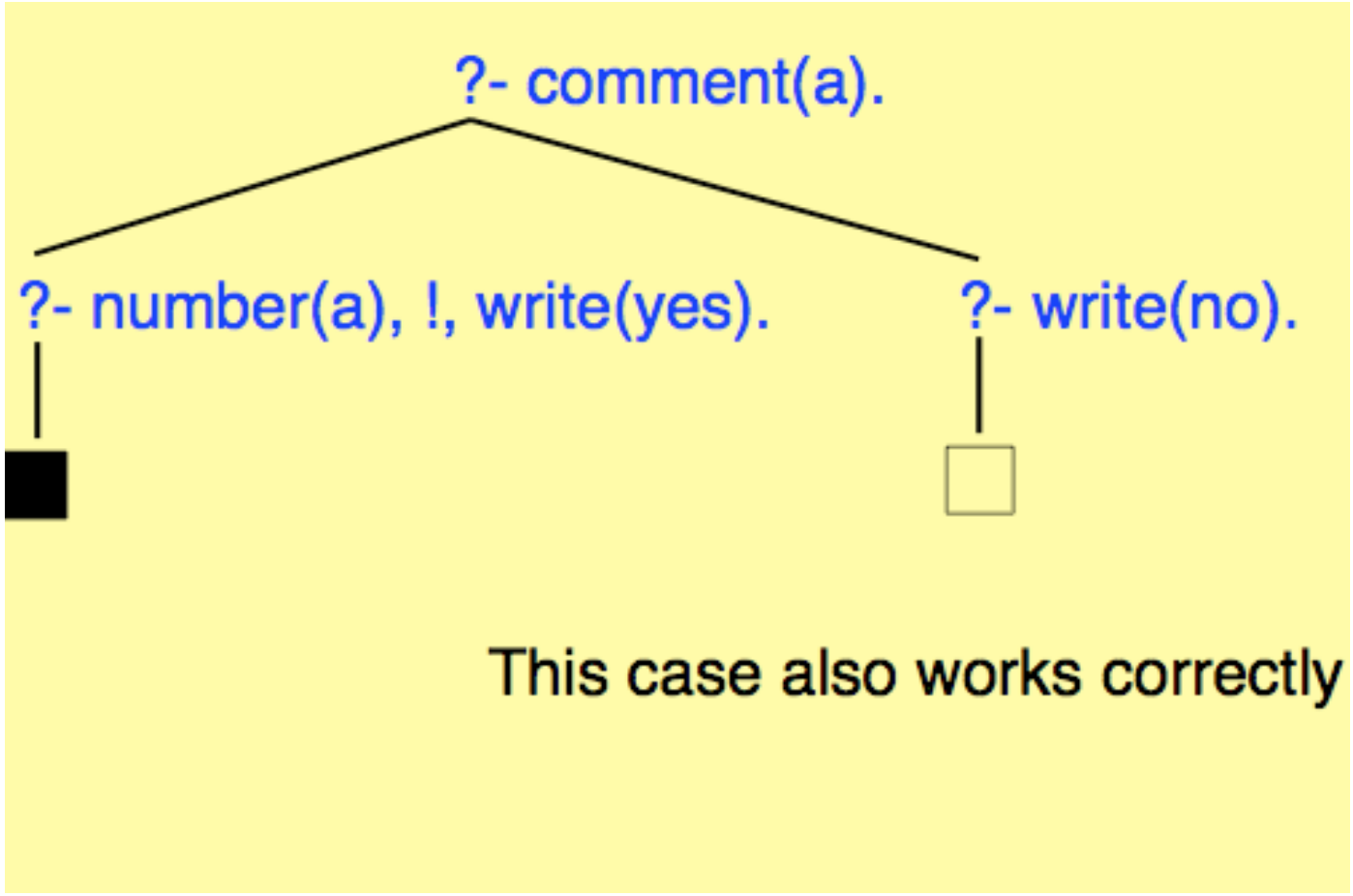
This program tests a term X and prints a comment

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).
```

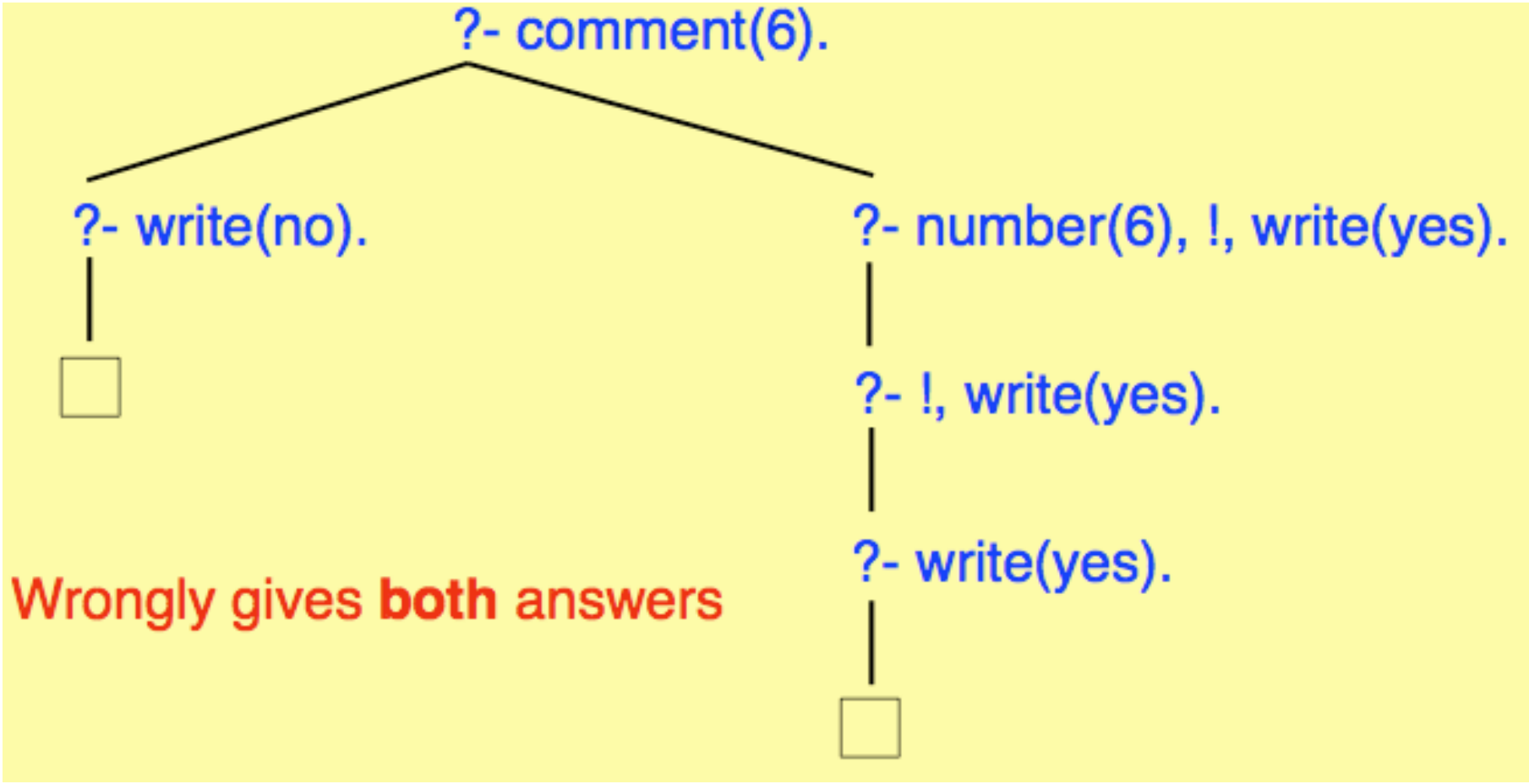
Will it work (assuming X is ground)?


?- comment(6).
?- number(6), !, write(yes).
?- write(no).
?- !, write(yes).
prune
?- write(yes).
This case works correctly


?- comment(a).
?- number(a), !, write(yes).
?- write(no).
This case also works correctly

BUT - suppose we reorder the clauses as:

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



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

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

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

```
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

EXAMPLES

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 (HARDER)

- 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).
```

META-VARIABLES

These are ordinary variables but are expected to become bound to terms that will then be treated as call-terms

EXAMPLE

Here is a program that simulates \+X

```
our_not(X) :- X, !, fail.
our_not(X).
```

(Here, X acts as a meta-variable)

note - "fail" always fails finitely

The query ?- our_not(happy(chris)).

binds X / happy(chris) in the first clause, so that
X will be a call-term at the instant it is selected for evaluation
The above query behaves exactly the same as ?- \+happy(chris).

EXAMPLE

```
aspect(logical).
aspect(strict).
aspect(fair).
aspect(crazy).
```

```
person(chris).
person(susan).
person(marek).
person(mike).
```

```
logical(chris).
strict(susan).
fair(susan).
fair(marek).
```

```
tell_us_about(X, Y) :-
        person(X), aspect(Y), Test=..[Y, X],
        Test.
```

?- tell_us_about(susan, Y). returns Y / strict or Y / fair
?- tell_us_about(X, logical). returns X / chris

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
  assert - creates a clause
  retract - deletes a clause

THE "CLAUSE" PRIMITIVE

A call to this has the form

clause(H, B) where H is any predicate in
which at least the relation name is given

It succeeds if and only if

H unifies by θ with the head of an existing
dynamic clause Head :- Body. whereupon
B is returned as Bodyθ

EXAMPLE

```
likes(chris, X) :- likes(X, prolog).
likes(chris, X) :- honest(X), praises(X, chris).
likes(frank, prolog).
```

?- clause(likes(chris, frank), B).

returns two alternative values for B

B / likes(frank, prolog)

B / (honest(frank), praises(frank, chris))

?- clause(likes(frank, X), B).

returns X / prolog, B / true

THE "ASSERT" PRIMITIVE

This has the form assert(Clause)

EXAMPLES

```
?- 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))) )
```

META-INTERPRETERS

These are programs which express ways of executing queries using other programs treated as data

EXAMPLE

```
solve([ ]).
solve(Query) :-
        select(Call, Query,Othercalls), clause(Call, Body),
        callterm_to_list(Body, Bodycalls),
        combine(Bodycalls, Othercalls, DQ),
        solve(DQ).
```

This expresses the behaviour of a *sequential*, *depth-first* interpreter asked to evaluate a list of calls given as Query

The computation rule used depends upon how select and combine are defined

To express the Prolog computation rule:

```
select(Call, [Call | Othercalls], Othercalls].

combine(A, B, C) :- append(A, B, C).
```

The result is then an interpreter, written in Prolog, which simulates Prolog's own behaviour

EXAMPLE

A computation-rule that is often superior to Prolog's is the *procrastination principle* (a standard heuristic in AI):

"select whichever call can invoke
the fewest number of clauses"

To obtain this behaviour we have to write an appropriate definition of select

Defining select for the procrastination principle:

```
select(Call, Query, Othercalls) :-
        findall((N, C),
                (member(C, Query),
                findall(_, clause(C, _), Cs),
                length(Cs, N)), NCs),
        sort(NCs, [(_, Call) | _]),
        del(Call, Query, Othercalls).

  del(_, [ ], [ ]).
  del(U, [V | X], X) :- U==V, !.
  del(U, [V | X], [V | Y]) :- U\==V, del(U, X, Y).
```

and, as before, define combine as

```
  combine(A, B, C) :- append(A, B, C).
```