Prolog Programming

Prolog programming is a special case of **logic programming** — the use of logic as a programming language combined with the use of deduction as a computational mechanism. Prolog is commonly described as a language best suited to artificial intelligence applications, but it is powerful enough to serve also as a general-purpose language. It was first implemented at the University of Marseilles in 1972.

The name "Prolog" stands for "programming in logic", and its central core is indeed just a subset of pure predicate logic from which computations are elicited using a simple deductive inference rule called **resolution**. However, Prolog also supports various non-logical extensions which, although not technically necessary, make certain programming tasks easier to formulate.

Programming in Prolog is both easy and difficult, according to one's perspective. It is made easy in one sense by the fact that the Prolog interpreter which executes our programs contains powerful built-in mechanisms for pattern-matching and search, the existence of which spares us from much tedious coding that would be demanded by more conventional formalisms. On the other hand, its high-level nature requires us to formulate our computational objectives with greater logical clarity and precision than we might need to with those other formalisms.

Supporting Texts

1. Bratko: *Prolog Programming for AI*

2. Sterling and Shapiro: *The Art of Prolog*

3. O'Keefe: *The Craft of Prolog*

The first two of these are probably the best known Prolog texts used at IC and both should be available in the College library and the book shop. Note, however, that these course notes are intended to be sufficient.

Course Structure

The following is the approximate course structure:

Lecture 1	Primary concepts Introductory examples
Lecture 2	Terms Relationship of Prolog to clausal-form logic Deterministic evaluations
Lecture 3	Non-deterministic evaluations Influencing efficiency Unification Answer extraction
Lecture 4	List processing Type checking Comparing terms Arithmetic
Lecture 5	Disjunction Negation Generate and test Aggregation
Lecture 6	Pruning the search tree
Lecture 7	Meta-programming

Primary Concepts

Concept 1 — procedure definitions

Prolog is traditionally introduced as a declarative formalism, but here we shall view it first as a procedural one. In this view, programs consist of procedure definitions and computations are the evaluations of calls made to the defined procedures.

Example: a :- b, c.

This Prolog statement is a procedure definition. Read procedurally, it says that we may evaluate a by evaluating both b and c. "Evaluating" something in this context means determining whether or not it is true. The statement can also be read logically (or "declaratively") as

a if (b and c)

So, the connectives "if" and "and" are represented in Prolog by :- and comma, respectively. In the syntax of standard logic we might write the above as

$a \leftarrow (b \wedge c)$ or as $(b \wedge c) \rightarrow a$

Read logically, Prolog statements are taken to express what we believe to be true about the world or, at least, about the particular problem domains that we want our programs to describe.

More examples:

```
likes(chris, bob) :- likes(bob, prolog).
mother(X, Y) :- parent(X, Y), female(X).
happy(U) :- student(U), teaches(chris, U), understand(U, chris).
```

Note that the basic logical units connected in all these statements are first-order **predicates**, exactly as you have already met in your logic course.

Concept 2 — procedure calls

Prolog programs come to life by responding to procedure calls, which are just predicates. Activity is initiated by posing an initial **query** comprising a conjunction of such calls.

Examples:
```
?- a, d, e.
?- likes(chris, X).
?- happy(U), likes(U, prolog), mother(X, U).
```

The logical reading of such a query is "is this conjunction true according to the defined procedures?". Operationally, it is treated by the Prolog interpreter as a collection of calls to be evaluated (executed) in sequence, in much the same way as it would be treated by most other sequential procedural languages.

Syntax notes

Prolog statements are referred to as **clauses** by logicians. Those that express procedure definitions are called **program clauses**. Queries are also clauses. Later we will look in more detail at the correspondence between Prolog and clausal-form logic.

Program clauses and queries are the two principal kinds of statement we have in Prolog. Both kinds must be terminated by **full stops**, as shown in all the examples above.

Predicate symbols begin with lower-case letters. The arguments of predicates may be constants, variables or compound terms. Variable symbols begin with upper-case letters or underscores. Constants and compound terms take various forms and will be described later on.

Concept 3 — computations

Given any program clause, the left-most predicate is called the **head**, whilst a conjunction on the right of :- is called the **body**. If there is no :- then the body is empty.

Example: a :- b, c. the head is a
the body is b, c

Suppose now that we pose a query

?- a, d, e.

in which the first call to be executed is a. At this point it is useful to bear in mind the standard view of what it means for a call to invoke a procedure—abstractly, it is seen as the act of substituting the procedure's body for the call. If we apply this notion then it will transform our query to

?- b, c, d, e.

This is exactly how Prolog works. Given some current query, it selects from it the **first** call, then seeks a program clause whose head can be **matched** with the call, and finally replaces the call by that clause's body. It thus derives a new query and treats this one in just the same fashion. By this means the interpreter generates a chain of derived queries, and this chain is what we call a **computation**. Each step in the computation is a single procedure-calling operation enabled solely by the ability of a call to be matched with a clause head.

Concept 4 — successful computations

A computation succeeds if and only if it derives the **empty query**. This happens only when the penultimate query contains just one call and this can be matched with the head of a program clause having an empty body.

Example: ?- likes(bob, prolog). penultimate query

likes(bob, prolog). program clause

Clearly, replacing the query's call by the clause's (empty) body will produce a query containing no calls at all—that is, the empty query. Operationally this signifies that nothing remains to be evaluated, equivalently, that any earlier calls have been successfully dealt with. The computation has **succeeded**.

Concept 5 — finitely-failed computations

A computation fails finitely if and only if the call selected from the current query cannot be matched with the head of any program clause.

Example: ?- likes(bob, haskell). current query

where no program clause has a head that can be matched with likes(bob, haskell)

Concept 6 — infinitely-failed computations

A computation fails infinitely if and only if every query within it is followed by a non-empty query.

Example: ?- a. initial query

a :- a, b. program clause

Syntax notes

Program clauses having non-empty bodies are called **rules** in Prolog, whilst those having empty bodies are called **facts**. Since neither can be regarded as being more "factual" than the other in the ordinary sense, this is a rather irritating piece of terminology—but we are now stuck with it.

Note that the :- connective is omitted when we have an empty body—a fact is just a predicate followed by a full stop.

A Prolog program is therefore just some collection of facts and rules.

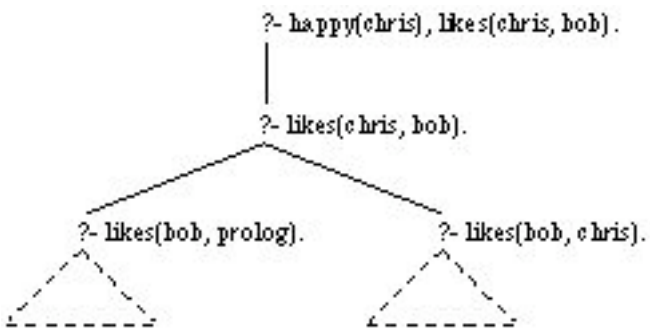
Concept 7 — multiple answers

A query may spawn multiple computations. This situation arises when some call can be matched with the heads of **several** program clauses.

Example: ?- happy(chris), likes(chris, bob).

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

The evaluation of this query can be depicted as below:



We call this sort of diagram a **search tree**, each of whose branches depicts some computation derivable from the initial query. We will discuss search trees in more detail later on. In the meantime, just note that the complete evaluation of a query is represented by some such tree, whose branches may variously succeed, fail finitely or fail infinitely—depending solely upon the pattern-matching opportunities afforded by the program clauses.

Note also that in such a tree there may be several successful branches yielding (possibly different) answers to the initial query. So, a single program may yield **multiple answers** to a single query, and the Prolog interpreter will automatically seek all of them by exploring the entire tree.

Concept 8 — answers as logical consequences

A successful computation confirms that the conjunction in the initial query is a **logical consequence** of the program. For instance, if the query ?- a, d, e. is solved by some computation using a program P then this establishes, in logical terms, the implication

$P \models a \wedge d \wedge e$ and this conjunction is taken to be a **computed answer**.

Concept 9 — variable arguments

The predicates used to construct queries and program clauses may have any arguments, including ones which consist of, or contain, variables. Variables play exactly the same role in Prolog as they do in logic—they are slots capable of being **instantiated** by terms.

A call selected from a query for evaluation may, at that instant, contain variables in its arguments. This is a feature of logic programming that is absent from most other procedure-calling formalisms, including functional programming. Variables in queries are treated as existentially quantified.

Example: ?- likes(X, prolog).

this is read logically as "is (∃X)likes(X, prolog) true?"
and is read operationally as "find X for which likes(X, prolog) is true"

Variables in program clauses, by contrast, are treated as universally quantified.

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

this is read logically as (∀X)(likes(chris, X) ← likes(X, prolog))
equivalently, as "chris likes anyone who likes prolog"

Concept 10 — generalized matching

We noted earlier that Prolog evaluations depend on the ability of query calls to be matched with the heads of program clauses. Two predicates match if and only if they are syntactically identical. Otherwise, if they contain variables then we may be able to find matching *instances* of them by **binding** their variables to particular terms.

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

the call and the head yield matching instances if we make
the bindings U/bob and Y/chris. The derived query will then be

?- understands(bob, chris).

Prolog automatically seeks suitable bindings by a process called **unification**. We will say more about this later on.

Example program 1 (Family Database)

Some Prolog programs have the character of a "deductive database"—a combination of data made explicit in the form of variable-free facts (analogous to the tuples in conventional database tables) and various rules enabling the deduction of implicit data.

```
male(peter).    male(john).    male(mark).    male(tom).
female(ann).    female(joan).  female(mary).
child_of(peter, mark).
child_of(peter, ann).
child_of(mark, john).
child_of(mark, mary).
child_of(joan, john).
child_of(joan, mary).
child_of(ann, tom).
```

data in the form of variable-free facts

```
father_of(F, C) :- child_of(C, F), male(F).
mother_of(M, C) :- child_of(C, M), female(M).
parent_of(P, C) :- child_of(C, P).
grandparent_of(G, C) :- parent_of(G, P), parent_of(P, C).
ancestor_of(A, C) :- parent_of(A, C).
ancestor_of(A, C) :- parent_of(P, C), ancestor_of(A, P).
sibling_of(S, C) :- child_of(S, P), child_of(C, P), S\==C.
brother_of(B, C) :- sibling_of(B, C), male(B).
sex_of(P, male) :- male(P).
sex_of(P, female) :- female(P).
```

rules

Some queries and answers for this database

the main point to note here — a query may have more than one answer,
— and Prolog will search for all of them.

1. ?- male(john). % is john male?
yes

2. ?- male(X). % who is male?
X / peter
X / john
X / mark
X / tom

3. ?- child_of(X, mary). % who are mary's children?
X / mark
X / joan

4. ?- child_of(X, tom), female(X). % who are tom's daughters?
X / ann

5. ?- sibling_of(X, joan), father_of(X, Y). % who is a sibling of joan *and*
% of whom is that sibling a father?
X / mark, Y / peter
X / mark, Y / peter (*twice, because mark is proved to be a sibling of joan in two ways—via his father and via his mother*)

6. ?- ancestor_of(A, peter), sex_of(A, S). % who are the ancestors of peter *and*
% of what sex are they?
A / mark, S / male
A / ann, S / female
A / john, S / male
A / mary, S / female
A / tom, S / male

7. ?- ancestor_of(mary, D), sex_of(D, S). % who are the descendants of mary *and*
% of what sex are they?
D / mark, S / male
D / joan, S / female
D / peter, S / male

Example program 2 (DoC Examinations Database)

Most aspects of this Department's examinations were for many years managed by a Prolog system.
Some tiny fragments of this are shown below as they stood in 1994-95.

```
% degree codes and degree titles
degree_titles(be, 'BEng in Computing').
degree_titles(me, 'MEng in Computing').
degree_titles(me_ai, 'MEng in Computing (Artificial Intelligence and Knowledge Engineering)').
…
% classes in the various degrees
class_in_degree(be, Class) :- on(Class, [b1, b2, b3]).
class_in_degree(me, Class) :- on(Class, [m1, m2, m3, m4]).
class_in_degree(me_ai, Class) :-  on(Class, [m1, m2, m3, m4]).
…
% exams taken by each class, with their titles, examiners and durations
exam(b1, '1.1', 'Logic', [kb], 90).
exam(b1, '1.2', 'Reasoning about Programs', [imh, cjh], 90).
exam(b1, '1.3', 'Discrete Mathematics', [mjs, iccp], 90).
exam(b1, '1.4', 'Architecture I', [nd], 90).
exam(b1, '1.5', 'Operating Systems I', [jnm], 90).
exam(b1, '1.6', 'Hardware', [dfg, pb], 90).
exam(b1, '1.7', 'Modula Programming and Data Structures', [se, mrc], 120).
exam(b1, '1.8', 'Miranda and Prolog Programming', [hk, cjh], 90).
exam(b1, '1.9', 'Mathematical Methods and Graphics', [dfg, ajm], 120).
…
% candidates in each class and the exams they will take
cands(b1, [ 3403, 'Balraj, B. (resit)', C,
            941001, 'Bengston, J.R.', C,
            941002, 'Chavrimootoo, S.C.', C,
            941003, 'Choo, H.J.', C,
            941004, 'Conroy, G.R.', C,
            …
            941027, 'Tam, D.', C,
            941028, 'Thomson, D.G.', C]) :-
            C=['1.1', '1.2', '1.3', '1.4', '1.5', '1.6', '1.7', '1.8', '1.9'].
cands(m1, [ ... etc.
…
```

Like the previous example, this one contains lots of facts defining what are, essentially, just the tuples in various relations within a relational database. Unlike the previous one, this one has tuples some of whose components are not simple constants but compound terms, such as [se, mrc]. One use of these is to make the data more compact, for instance allowing us to write

```
exam(b1, b1_7, 'Modula Programming and Data Structures', [se, mrc], 120).
```

instead of the more cumbersome representation

```
exam(b1, b1_7, 'Modula Programming and Data Structures', se, 120).
exam(b1, b1_7, 'Modula Programming and Data Structures', mrc, 120).
```

The complete database also contains numerous constraints such as:

```
%_____check that no paper lasting more than two hours is scheduled in a morning.
check(7, ['long paper ', Code, ' scheduled on morning of ', Date]) :-
        exam(_, Code, _, _, Dur),
        paper(Code, _, Date, am, Start, End, _),
        duration(Start, End, Duration),
        one_of(Dur>120, Duration>120).
```

You will notice that this looks more like a conventional procedure than a segment of a database.

Prolog Terms

Terms can be viewed as the basic items of data manipulated by Prolog programs. They exist statically as arguments in the predicates we use to write our queries and program clauses, and they also come into existence dynamically by the unification process mentioned earlier.

To determine the full range of allowed terms you would need to consult the Prolog reference manual, but the following covers the most common kinds.

Number 3, 5.6, –10, –6.31

Atom apple, tom, x2, 'Hello there'

atoms and numbers are all constants and in Prolog are jointly referred to as "atomic"

Either a quote-free sequence of non-blank characters, starting with a lower-case letter, or a quoted sequence of any characters (other than single quotes).
There is also a special atom [] denoting the empty list.

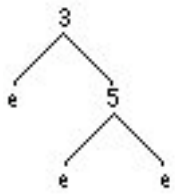
Variable X, Y31, Chris, Left_Subtree, Person, _35, _

A sequence of alphanumeric characters, starting either with an upper-case letter or with an underscore.

A variable consisting of a single underscore is called an "anonymous variable". If a clause contains several of these then they are treated as being distinct from each other and from all other variables in the clause.

Compound A term built from a functor and a tuple of arguments each of which is any term. A functor has the same syntax as an atom. Examples are shown below.

mother(chris)	here mother is a functor of arity 1
t(e, 3, t(e, 5, e))	here t is a functor of arity 3, and the term as a whole might be viewed as representing the tree



t(T, N, t(e, 5, e))	here the root node and the left-subtree are undetermined
[3, 5]	syntactic sugar for the list .(3, .(5, [])) where the functor '.' is the 2-ary list-constructor
[3, 5, X, 9]	here, the third member is undetermined
[X | Y]	syntactic sugar for the list .(X, Y) where X is the head of the list and Y is its tail, both of which are here undetermined
[3, 5 | [7]]	a legal but pointless alternative for [3, 5, 7]
[X, 3 | Y]	here, only the second member is determined
3 * X + 5	syntactic sugar for the arithmetic expression +(*(3, X), 5) since X is undetermined, so is the expression's value
(1, 2, 3)	a tuple of three members—the functor is an implicit 2-ary tuple-constructor note—this tuple is *not* the same as the *list* [1, 2, 3]

Relationship of Prolog to Clausal-Form Logic

Clausal-form logic consists of just those sentences that are disjunctions of **literals**.
Such sentences are called **clauses**.

A literal is either a predicate (atomic formula) or a negated predicate.

Here on the left are two examples of clauses in propositional logic, and on their right are the corresponding ways they are written in Prolog:

a	a.	(a fact)
a ¬b ¬c	a :- b, c.	(a rule)

The correspondence in the second example can be clarified as follows:

a ¬b ¬c	≡	a ¬ (b ∨ c)	
	≡	(b ∨ c) ⊃ a	
	≡	a ⊃ (b ∨ c)	
	≡	a :- b, c.	read as "a if b and c"

In the last step the final Prolog form has been obtained by

replacing ⊃ by :-
replacing ∨ by comma
omitting the brackets
putting a full-stop at the end

The definition of clausal-form logic also allows clauses all of whose literals are negative, such as

¬a ¬d ¬e ≡ ¬(a ∨ d ∨ e)

In Prolog this corresponds to a query and is written simply as

?- a, d, e.

by replacing ∨ by comma
omitting the ¬ and the brackets
putting ?- at the beginning
putting a full-stop at the end

The definition of clausal-form logic even allows the special case of a clause having no literals at all !
In logic we would write this as the symbol □. In Prolog it corresponds to the empty query.

The basic evaluation step of Prolog replaces a query call by the body of a program clause whose head it can match. In logic this corresponds to an application of the inference rule called **resolution** which, given two clauses presented in disjunctive format, eliminates from them a pair of complementary literals and constructs a new clause (the **resolvent**) from the literals that remain. Two literals are **complementary** if and only if one is the negation of the other. The correspondence is indicated in the following example.

in Prolog		**in logic**	
query	?- a, d, e.	¬a ¬d ¬e	¬a and a are eliminated
program clause	a :- b, c.	a ¬b ¬c	
derived query	?- b, c, d, e.	¬b ¬c ¬d ¬e	to give the resolvent

and the logicalrelationship between these statements is

{ query, program clause } ⊨ derived query

The identification of the two complementary literals corresponds to the matching of the selected call and the head of the program clause. Remember that, in the case where these literals contain variables, Prolog will automatically seek bindings of those variables that make the literals complementary (i.e. yield matching instances of their predicates), though it may not be able to find any.

Deterministic Evaluations

Prolog is a **non-deterministic** formalism in that it offers the possibility of multiple computations arising from a single query and program. We briefly noted this earlier in Concept 7. Presently we will have to discuss how Prolog manages such situations. Before then, however, we will look at some simple **deterministic** evaluations in which just **one** computation arises.

Example:

```
all_bs([]).                    this program defines the property "all_bs" of a list
all_bs([b | T]) :- all_bs(T).  which holds exactly when every member is b

?- all_bs([b, b, b]).          this query asks whether [b, b, b] has this property
```

To find out what happens we consider **only** the possibilities of matching.

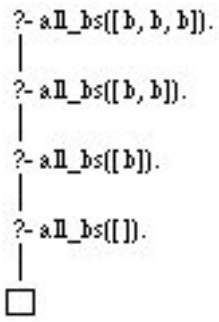
- select the first (left-most) call in the query—this is what Prolog **always** does
- there is only one such call, and its argument is [b, b, b]
- look to see whether this argument can be matched with that in any clause head
- it can be matched in the case of the second clause, by binding T/[b, b]
- replace the selected call by that clause's body and apply the binding to the result, yielding the derived query

?- all_bs([b, b]).

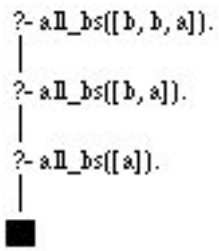
- continuing in the same way develops the computation further as follows:

```
?- all_bs([b]).    again, using just the second clause
?- all_bs([]).     again, using just the second clause
?-.                here, using just the first clause
```

Now we have the empty query and so the computation has succeeded. The search tree representing the entire evaluation consists of a single branch:



Example: ?- all_bs([b, b, a]). a different query using the same program, which again gives just one computation—but this one fails finitely



The last state of the query cannot be solved because the argument [a] cannot be matched with the argument of any clause head. To indicate this we put a solid square at the end of the computation.

Example:

app(X, Y, Z) expresses that the result of appending list Y to list X gives list Z.

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

```
?- app([a, b], [c, d], [a, b, c, d]).      does [a, b, c, d] result from appending
                                           [c, d] to [a, b] ?
```

Again, the evaluation is deterministic and relies only upon matching.

- only the second clause offers the opportunity to match the call
- matching [a, b] to [U | X] requires binding U/a and X/[b]
- matching Y to [c, d] requires binding Y/[c, d]
- matching [a, b, c, d] to [U | Z] requires binding U/a and Z/[b, c, d]

- we have therefore found a way of matching all three argument pairs **compatibly**

- replace the call by the body and apply the bindings to give

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

- continuing in the same way we derive

```
?- app([], [c, d], [c, d]).      again, using the second clause
?-.                              here, using the first clause
```

Example:

```
?- app([a, b], [c, d], L).       what is the result L of appending
                                 [c, d] to [a, b] ?
```

- again, only the second clause offers the opportunity to match the call
- matching [a, b] to [U | X] requires binding U/a and X/[b]
- matching Y to [c, d] requires binding Y/[c, d]
- matching L to [U | Z] requires binding L/[a | Z], since we have already committed to U/a

- the derived query is thus

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

- continuing in the same way will again result in success
- we can extract a final **output binding** L/[a, b, c, d] for our original query variable L
- the precise way in which such output bindings are extracted will be explained later on

The "app" examples above illustrate a second way in which Prolog is a non-deterministic formalism, in that it offers the possibility of **multiple input-output modes** for queries.

The "app" program does not determine the input-output mode in which it will be used—that mode is determined only by whatever query is posed. In the first "app" example above the query mode was

```
?- app(input, input, input).
```

whereas in the second example the query mode was

```
?- app(input, input, output).
```

In fact, the "app" program **works for any mode**. For instance, it will solve

```
?- app(X, [b | L], [a, E, c, d]).      to give the output bindings X/[a], L/[c, d], E/b
```

where the query mode is

```
?- app(output, mixed, mixed).
```

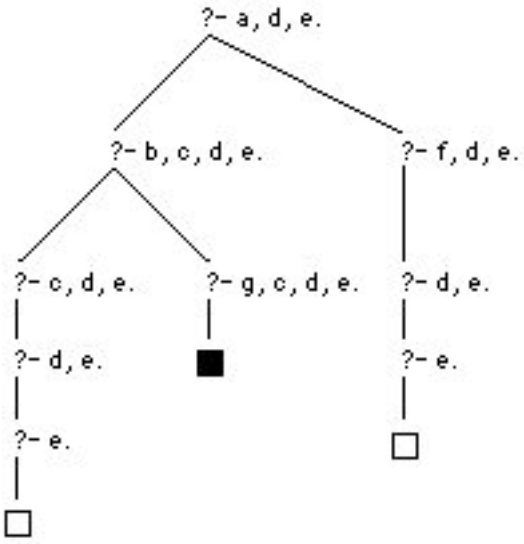
Non-deterministic Evaluations

By now you should have a moderately clear idea of how Prolog contructs an individual computation. It is really quite trivial, involving only procedure-calling governed by pattern-matching, inducing bindings as necessary.

In the case where several computations are possible, Prolog applies a simple strategy to deal with them.
To explain this we will consider the following example:

query	?- a, d, e.	*program*	a :- b, c.	*note* —two clauses defining a
			a :- f.	
			b.	*note* —two clauses defining b
			b :- g.	
			c.	
			d.	
			e.	
			f.	

There are *three* possible computations, so the entire evaluation now takes the form of a tree:



The sole reason why the tree has several computations is that several program clauses apply to certain query calls (a and b). If this were not the case then the tree would consist of just one computation.

A node which has more than one immediate descendant is called a **choice-point**. In this tree there are two choice-points, these being the queries ?- a, d, e. and ?- b, c, d, e. Each of these presents a choice as to which program clause (among just those that match) to use next in response to the selected call.

The Prolog **search strategy** is to explore **one computation at a time**. It will not explore other computations until and unless the current one has terminated, whether successfully or otherwise. This policy for sequential tree-search is called **depth-first search**.

When any such termination occurs, Prolog **backtracks** to the most recently encountered choice-point in the tree offering some so-far-unexplored branch. From here, it then proceeds in the same manner with the new computation. The overall evaluation of the initial query ends only when there remain no choice-points offering unexplored branches.

We have so far said nothing about how Prolog decides *which* branch to explore at a choice-point. This is determined solely by the text-order of the clauses. In the example, the clauses defining a appear in the order

```
a :- b, c.          followed by …
a :- f.
```

Consequently, when the call a is selected, the first computation explored from this choice-point is the one arising from use of the first clause, whilst the second computation arises from the use of the second clause.

If we altered the text-order of these clauses it would make no difference at all to the search tree. Exactly the same computations would be developed, leading to exactly the same answers with exactly the same efficiency. For the programmer, the only significance in deciding on a particular order is that Prolog will report some answers sooner than others, but the total time it takes to report them all is unaffected.

The text-order of clauses thus prioritizes computations. The chosen prioritization is called the **search rule**.

Influencing Efficiency

The efficiency with which a query is solved by Prolog is governed by two main factors:

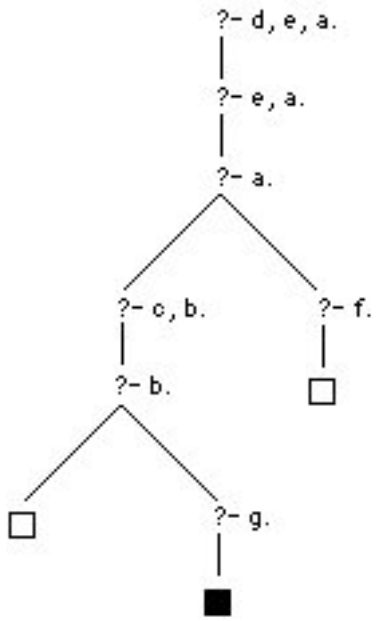
what the program clauses say about the relations involved
and *the text-order of the calls* in those clauses and in the query.

Let us look at the effect of re-ordering some calls in the earlier program, as follows:

query ?- d, e, a. *program*

```
a :- c, b.      c.
a :- f.         d.
b.              e.
b :- g.         f.
```

Here we have re-ordered the calls in both the query and in the first clause defining a. This is the result:



This is arguably a more efficient overall evaluation than the earlier one. The total number of computation steps in this search tree is 8, whereas the earlier tree required 10. The latter suffered the inefficiency of having to evaluate twice (once in each of two different computations) the derived query ?- d, e.

The text-order in which the programmer writes calls effectively prioritizes their selection. This prioritization is called the **computation rule** and, in general, has a dramatic influence over efficiency. Just as dramatic an influence, however, is the choice of statements that one writes about the relations of interest.

Unification

Unification is the process of making two predicates match, if necessary by binding variables in them to particular terms. Its name derives from the Latin for "making into one thing".

When predicates are merely propositional (arity 0), unification requires only that the two predicate symbols be identical.

When predicates possess arguments, however, more work is needed to determine whether they are unifiable. To see what is involved, consider this example:

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

In trying to use the first clause, Prolog must find a way of unifying likes(Y, chris) and likes(bob, X). Here it is a simple matter to see that they match once we bind Y/bob and X/chris, for then the two predicates become identically likes(bob, chris). We say that {Y/bob, X/chris} is a **unifier** for them. Much of the work of a Prolog interpreter is entailed in seeking unifiers, using an efficient built-in algorithm.

The logical justification in using unifiers to enable interaction between queries and program clauses can be seen by looking at the above example in the following way.

The query is asking "who likes chris?", whilst the program is saying that

bob likes anyone who likes logic
and that chris likes logic

When the query call likes(Y, chris) is selected, Prolog has to decide which clauses, if any, are "about" anyone liking chris. Since clause variables are universally quantified, the first clause is about the liking of anyone provided bob is doing the liking. In particular, it implies the particular **clause instance** obtained by binding X/chris, namely

likes(bob, chris) :- likes(chris, logic).

So in response to "who likes chris" this instance replies "bob does, provided that chris likes logic." Thus we can extract the answer Y/bob *provided* we can solve the derived query

?- likes(chris, logic).

The **general computation step** of Prolog can thus be summarized as follows:

given	the current query	?- **P**(args1), others.
and	a program clause	**P**(args2) :- body.
such that	**P**(args1) and **P**(args2) have a unifier θ —i.e., so that **P**(args1) θ = **P**(args2) θ	
derive	the next query as	?- body θ, others θ.

Note—an expression E θ denotes the result of applying to E all the relevant bindings in θ.

Answer Extraction

When a query such as ?- likes(Y, chris). succeeds via some successful computation, we want to know the answer thereby discovered. If Prolog gave us no access to the bindings it had made, the most we could conclude would be

(∃Y) likes(Y, chris) "yes, someone does like chris"

But usually this is not enough for us—we want to know also the output bindings made to the query variables, enabling us to reach more specific conclusions such as

likes(Y, chris) {Y/bob} "yes, Y likes chris in the case that Y is bob"

In many cases the final value of a query variable is determined not by a single binding in one computation step, but rather by the *incremental effect of bindings made in several steps*.

We can illustrate this with a slightly more ambitious example dealing with the predicate app(X, Y, Z), which we have already met.
The query asks whether there is a list X which can be appended to itself to give [a, b, a, b].

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

program

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

As we look into the evaluation we will deal with a few points concerning the administration of variables and unifiers. Here is the first such matter:

Using clause variants

Whenever we express on paper the use of a program clause, we must not confuse its variables with any others previously involved in the current computation. In the example, the variable named X in the initial query must not be confused with that named X in the second program clause—they are entirely independent variables which just happen to share the same name in the given text.

All possible confusions are avoided by the simple convention that each time we use a program clause we first *make a virtual copy of it with new names for its variables*, and then use this copy instead. Such copies are called **clause variants**.

The Prolog interpreter automatically ensures that there is no confusion of variables.

Starting with the initial query, which of the clauses applies to the call app(X, X, [a, b, a, b]) ?
Certainly not the first one—any attempt to unify the call with the head app([], Z, Z) must fail because their first arguments [] and [a, b, a, b] cannot be matched.

So let's try the second clause after making this variant of it:

app([U1 | X1], Y1, [U1 | Z1]) :- app(X1, Y1, Z1).

Our call app(X, X, [a, b, a, b]) does unify with the head of this variant via the unifier

θ_1 = {X/[a | X1], U1/a, Y1/[a | X1], Z1/[b, a, b]}

At this stage we have a second point to deal with.

Choosing most-general unifiers

When two predicates can be unified there may be many (perhaps infinitely many) unifiers for them. It is standard practice always to choose the **most-general unifier**, which will always exist and will always be unique (*modulo* the possibility of renaming its variables).

Choosing the most-general unifier ("mgu") is not logically necessary. But failing to do so when we restrict, as in Prolog, the way in which queries and program clauses interact can have the result that some answers to the query may not be found.

For the example above, another unifier is θ = {X/[a], U1/a, Y1/[a], Z1/[b, a, b]}.
This is less general than θ_1 because it can be obtained by specializing θ_1 to the case where X1=[].

θ_1 is in fact the mgu. If we chose θ instead we would find that, under Prolog's strategy, the query could not be solved.

The Prolog interpreter automatically seeks the mgu in each step.

At this stage we also have a third point to deal with.

Distinguishing input from output

When two predicates are unified it is generally the case that bindings are made to variables in each of them. More specifically, in the unification of a call with a clause head,

output bindings are those that bind variables in the call, whilst
input bindings are those that bind variables in the head.

An mgu can always be partitioned into two such binding sets **O** (output) and **I** (input).

In the example, θ_1 = {X/[a | X1], U1/a, Y1/[a | X1], Z1/[b, a, b]} is partitionable as

O = {X/[a | X1]}
I = {U1/a, Y1/[a | X1], Z1/[b, a, b]}

More generally, when any computation step arises from the

query ?- **A**(args1), others.
and the clause **A**(args2) :- body.

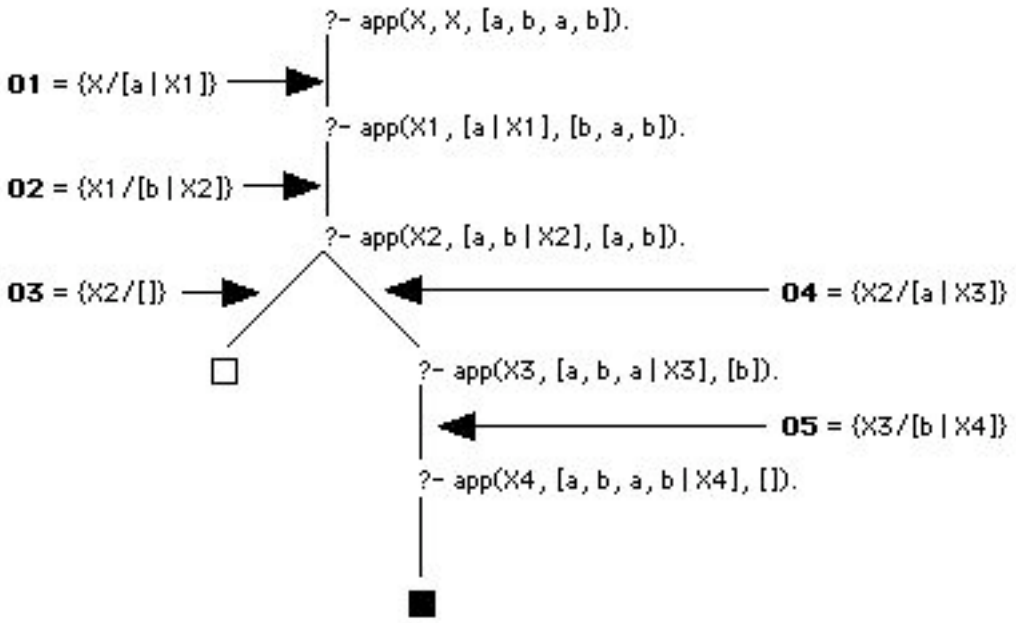
with the mgu partitioned into **O** and **I**, we obtain the

derived query ?- body **I**, others **O**.

and we meanwhile *keep a record* of **O** for the later purpose of *answer extraction.*

The Prolog interpreter automatically records **O** in this computation's binding environment.

Since the output bindings are what we are most interested in, it is common practice to include them in our search tree representing the evaluation. Below is the complete search tree for the example together with the output bindings. Note that no variable is ever bound more than once in any particular computation (branch).



There is one successful computation, so we know at least that there is *some* X for which the initial query is true, that is, is implied by the program. Finding out that X is part of the process of answer extraction.

Extracting an answer

To extract the final answer from some successful computation, we first have to recover the output bindings recorded during the computation and then *compose* them.

Informally, the **composition** S1 ∘ S2 of two binding sets S1 and S2 is a set comprising

- the bindings in S1 as *modified* by the effects of the bindings in S2, together with
- the bindings in S2 of any variables that are not bound in S1.

For instance: {U/[V], Z/T} ∘ {V/3, W/4} is {U/[3], Z/T, V/3, W/4}.

The next step is to restrict the composition to just the bindings of variables in the initial query. This gives the so-called **computed answer substitution**.

This substitution is applied to the initial query conjunction, giving the final answer.

The Prolog interpreter automatically extracts and reports this answer.

In the "app" example above, we therefore form the composition **O1** ∘ **O2** ∘ **O3** and then restrict this to {X/[a, b]}, which is the computed answer substitution for this computation. The final answer is then app([a, b], [a, b], [a, b, a, b]), which is necessarily logically implied by the program.

List Processing

The most commonly used compound data structure in Prolog is the list. In fact, all other data structures can be represented, one way or another, by lists, and so one actually needs no functors of arity>0 in the language other than the list-constructor. Any use of other functors is motivated solely by aesthetic or pragmatic preferences.

Prolog systems come equipped with a considerable range of useful built-in programs (primitives) for list-processing, and in many cases it is easy to define further programs based upon them.

Since lists are recursively-defined structures, most list-processing programs involve recursion.

Some common requirements	**Programs**
defining the first member of a list	first([F | _], F).
defining the second member	second([_, S | _], S).
defining the tail	tail([_ | Tail], Tail).
defining membership	member(U, [U | _]). member(U, [_ | L]) :- member(U, L).
defining non-membership	non_member(_, []). non_member(U, [V | L]) :- U\==V, non_member(U, L).

Note— this employs Prolog's primitive \== which means "not identical to"
"member" and "non-member" are primitives in most Prologs

defining concatenation	append([], Z, Z). append([U | X], Y, [U | Z]) :- append(X, Y, Z).

Note— this is the same as our earlier relation "app"
"append" is a primitive in most Prologs

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

Note— "reverse" is a primitive in most Prologs

defining the deletion of all members that also belong to another list Es	del_all(_, [], []). del_all(Es, [E | T], L) :- member(E, Es), del_all(Es, T, L). del_all(Es, [E | T], [E | L]) :- non_member(E, Es), del_all(Es, T, L).
defining a prefix	prefix([], _]). prefix([U | X], [U | Y]) :- prefix(X, Y).
defining a sublist	sublist(X, Y) :- prefix(X, Y). sublist(X, [_ | Y]) :- sublist(X, Y).
alternatively ...	sublist(X, Y) :- append(Z, _, Y), append(_, X, Z).
quick-sorting	sort([], []). sort(X, Y) :- split(X, X1, X2), sort(X1, Y1), sort(X2, Y2), append(Y1, Y2, Y).

Note— "sort" is a primitive in most Prologs.

Type Checking

Most Prologs do not recognize type declarations or perform automatic type-checking, though some do.
If we want to protect our programs with type checks then we usually have to enter these checks explicitly.

All Prologs provide primitives for checking certain simple types, but for more elaborate types—and especially for polymorphic types—we have to write our own code if we want to check them.

Here are some examples of Prolog's own type primitives, where **P**/n denotes that **P** has arity n.

atom/1 tests whether its argument is an atom

query	?- atom(cheese).	*query*	?- atom(2.35).
answer	yes	*answer*	no

number/1 tests whether its argument is a number (real or integer)

query	?- number(123.005).	*query*	?- number(cheese).
answer	yes	*answer*	no

Note also that we have **integer**/1 and **float**/1 to test for the types integer and floating-point.

atomic/1 tests whether its argument is an atom or a number

compound/1 tests whether its argument is a compound term

query	?- compound(f(2)).	*query*	?- compound([X | Y]).
answer	yes	*answer*	yes
query	?- compound(X).	*query*	?- compound(interest).
answer	no	*answer*	no

var/1 tests whether its argument is an unbound variable

query	?- var(X).	*query*	?- X=3, var(X).
answer	yes	*answer*	no

We can use these various primitives to define more elaborate types of our own.

For example, this program defines a list of atoms:

```
list_of_atoms(X) :- atom(X), X=[].
list_of_atoms(X) :- compound(X), members_are_atoms(X).

members_are_atoms([]).
members_are_atoms([X | Y]) :- atom(X), members_are_atoms(Y).
```

What could go wrong if we used instead the simpler program below?

```
list_of_atoms([]).
list_of_atoms([X | Y]) :- atom(X), list_of_atoms(Y).
```

Here is another example—this program defines certain terms to be of type "binary tree":

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

Comparing Terms

Prolog provides a number of primitives for comparing terms. A common example is

=	"unifies with"	?- f(X, chris) = f(mum(bob), Y). succeeds and binds X/mum(bob), Y/chris

It is also possible to test terms for identicality *without binding their variables* :

==	"identical to"	?- [a, b] == [a, b].	succeeds
		?- [a, b] == [a, X].	fails
\==	"not identical to"	?- [a, b] \== [a, X].	succeeds

Note— each of the last two primitives contains **two** adjacent "equals" symbols.

Arithmetic

Prolog recognizes roughly the same range of arithmetic expressions as do other languages.
The elementary *arithmetic operators* are

+	addition
–	subtraction
*	multiplication
/	division

all of which may be used in infix mode.

Usually there are also numerous *built-in functions* such as sqrt, abs, ln, sin, cos, tan, etc. accessible from special libraries.

An example of an arithmetic expression is thus (7 + 89 * sin(Y+1)) / (cos(X) + 2.43)

The main contexts in which such expressions are used are:

- *evaluating two expressions in order to compare their values*

Expression1 =:= Expression2	test whether their values are equal
Expression1 =\= Expression2	test whether their values are unequal
Expression1 < Expression2	test whether the value of the first is less than the value of the second

(similarly we can use >, >= or =<)

So,	?- X = 3, (2+2) =:= (X+1).	succeeds
but	?- (2+2) =:= (X+1), X=3.	error— (X+1) cannot be evaluated.

- *evaluating one expression in order to bind some variable to its value*

	Variable is Expression	bind Variable to the value of Expression
So,	?- X is 2+2.	succeeds and binds X/4
but	?- X is 2+Y.	error— 2+Y cannot be evaluated

Note— Prolog's "is" primitive is specifically for evaluating an arithmetic expression. Do **not** try to use it for any other purpose. It is **not** the same as "=".

Example: to test whether three given numbers can be the side-lengths of a triangle:

```
triangle(X, Y, Z) :-  X+Y > Z,
                      X+Z > Y,
                      Y+Z > X.
```

Example: to add up a given list of numbers:

first way

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

Note that this is *not tail-recursive,* and so its use will generate an activation-record stack whose size is proportional to the length of the input list.

Trying to make it tail-recursive by re-ordering it thus:

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

will **not** work, since N+Sumtail cannot be evaluated when the "is" call is selected.

Example of computation from the program as first given:

```
?- sumlist([2, 5, 8], T).
?- sumlist([5, 8], T1), 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 T/15

second way

```
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).
```

This program *is* capable of tail-recursive execution.
The predicate tr_sum(Ns, S, T) is defined by the program to mean T = S + ∑ Ns.
Therefore tr_sum(Ns, 0, T) means the same as sumlist(Ns, T).

Example of computation from the tail-recursive program:

```
?- 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).
```

succeeds with T/15

Disjunction

Calls can be separated by a semicolon to represent their disjunction.

Example: test whether a given number X is out of range:

first way

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

second way

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

Disjunctions may be parenthesized when convenient or when necessary to avoid ambiguity.

Suppose we also want to check that X, Low and High are all numbers before we do the range test.
The program could then be written:

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

Here, the parentheses around the disjunction ensure there is no ambiguity.

Negation

Prolog does not have an explicit connective for classical negation (¬) within queries or program clauses.
It is arguable that we do not need one.

Example: innocent(X) ⇐ ¬ guilty(X) (in pure classical logic)

In practice we do not establish the innocence of X by proving ¬ guilty(X). Instead, we establish it by finitely failing to prove guilty(X).

Prolog provides a special operator \+ (commonly pronounced "not") denoting "finitely fail to prove".
So in Prolog we might write

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

The operational meaning of \+ is as follows:

\+ P	succeeds	*iff*	the evaluation of P fails finitely
\+ P	fails finitely	*iff*	the evaluation of P succeeds

Example: "X is sad if someone else does not like X"

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

query

```
?- sad(X).
```

data

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

The query gives answers bob, chris and frank—in each case, someone does not like them.

Note that \+ is not a perfect approximation to classical negation, as shown by the next example.

Example:

p ⇐ ¬ p classically implies p

but

p :- \+ p. cannot solve ?- p. (it will loop infinitely)

Example: "X is very sad if no one else likes X"

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

query

```
?- very_sad(X).
```

data

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

The answers are now just bob and chris—in each case, no one likes them.

The safest context in which to evaluate \+P is when P is **ground** (variable-free) at the instant it is selected. Some Prologs insist upon this, whilst others are more permissive. If P is not ground then the evaluation of \+P may give a result different from what was intended.

In the last example, Y will be unbound at the instant \+ is selected. A safer formulation is therefore

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

Note that \+ partially compensates for the head of a clause being required to be a single atom.
If we want to use the knowledge that, say,

$$A \vee B \leftarrow C$$

we can write it either as

```
A :- \+B, C.
```

or as

```
B :- \+A, C.
```

Often the use of \+ can be avoided by casting the problem in terms of testing for non-identicality (\==).

Example: non-membership in a list

To ask whether 4 does not belong to [1, 2, 3] we could simply ask ?- \+ member(4, [1, 2, 3]).
Alternatively we could ask ?- non_member(4, [1, 2, 3]). and define non-member by

```
non_member(_, []).
non_member(U, [V | L]) :- U \== V, non_member(U, L).
```

(In Sicstus, of course, we would not need to define it, since it is supplied as a primitive).

Generate and Test

Some Prologs (but **not** Sicstus) support a "forall" primitive having the form

forall(P, Q)

which succeeds if and only if **every** way of solving the call-term P also solves the call-term Q.

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

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

This provides an approximation to what we might otherwise write in full predicate logic:

happy(X) ⇐ (∀Y)(friend(X, Y) ⇒ likes(Y, logic))

If one's Prolog does not provide "forall" as a primitive, it can instead be defined by the single clause

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

that is "no way of solving P fails to solve Q"

The forall(P, Q) predicate is a convenient way to express **generate-and-test** procedures, in which P serves as the generator and Q as the tester. Here is another example, where we want to perform a type-check for a list comprising only positive numbers by generating each member and testing it:

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

Note— forall(P, Q) succeeds even when there are no solutions to P
this is why we must include another type-check test "is_list" in this example, otherwise we would have, for instance, ?- all_pos_nums(chris). succeeding even though chris is not a list.

Note also that forall(P, Q) is not a perfect approximation to (∃...)(P ⇒ Q), as shown by the next example.

Example:

(∃...)(P ⇒ P) is true in logic, no matter how P is defined

but

forall(P, P) succeeds only if P is defined in such a way that the complete evaluation of P terminates

Call-Terms

When we write something like forall(P, Q) it is understood that P and Q are any **call-terms**. The simplest kind of call-term is a single predicate. Besides this, a call-term may also take such forms as

- \+P where P is a call-term
- a conjunction of call-terms
- a disjunction of call-terms

Loosely, a call-term is anything that the Prolog interpreter can be asked to evaluate.

Aggregation

Prolog supplies a primitive "findall" which is very convenient for perfoming **aggregation**—it constructs a list of **all** the terms for which some call-term can be solved. Every (error-free) use of it is bound to succeed.

The general form is: findall(Term, Call-term, List)

Example: "find all those whom chris likes"

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

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

Prolog finds every way of solving likes(chris, X) and, in each case, constructs the corresponding value of the given term—in this case just X. The values found for X here are just logic and frank.
The output binding for the query is thus L/[logic, frank].

Note that the list takes the order in which solutions are found, and duplicates (if any) are *not* removed.

Example: "find all sublists of [a, b, c] that have just two members"

query ?- findall([X, Y], sublist([X, Y], [a, b, c]), Sublists).

This returns the output binding Sublists/[[a, b], [b, c]], given a suitable "sublist" program.

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]
and ?- 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 X's friends:

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

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).
```

Example: given a list L of classes test whether each one has more than 50 students:

```
large_classes(L) :- forall( (member(C, L),
                             findall(S, member(S, C), Students),
                             length(Students, N)), N>50).
```

Pruning the Search Tree

Prolog supplies a primitive ! (pronounced "cut") whose purpose is to **prune unwanted computations** from the search tree. A cut can be placed anywhere within queries or clause bodies as though it were just another call. Thus a clause containing a cut has the general form

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

When the cut is selected for execution it will *immediately*

- prune all so-far-untried ways of evaluating the call that invoked this clause,

and

- prune all so-far-untried ways of evaluating the calls in this clause that precede this cut.

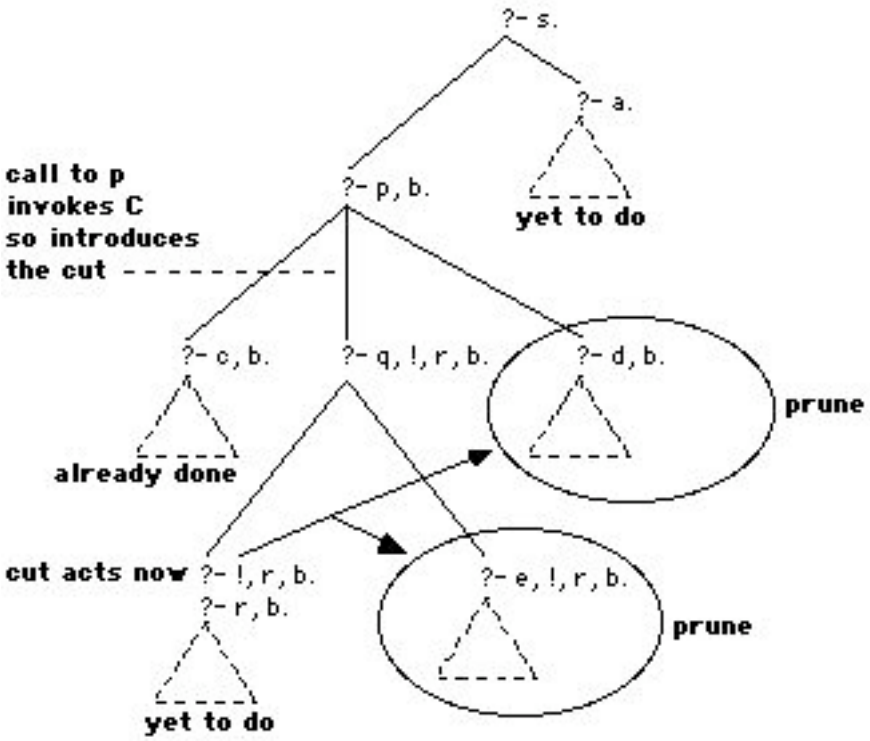
Example: p :- q, !, r. refer to this clause as "C"

If execution reaches a point at which C's cut is the call selected, then some call to p must have invoked clause C. There may be other clauses that p could invoke that remain untried—the cut now prunes those possibilities from the tree. In addition, it prunes from the tree any untried ways of evaluating C's call to q that have arisen since p invoked C.

Let's fill out this example a bit more by assuming the program contains at least the following clauses:

s :- p, b.	p :- c.	q.
s :- a.	p :- q, !, r.	q :- e.
	p :- d.	

Now consider a call to s. The figure below shows which parts of the tree are pruned and which survive following the action of C's cut at the moment it is selected.



Note that the scope of the cut's action is fairly limited. The cut does not prune from the tree any possibilities existing above the point where it is introduced, nor any possibilities existing below the point where it acts. Equivalently, the regions marked "yet to do" in the tree shown are not pruned by this cut.

Example —"print a comment on whether or not X is a number"

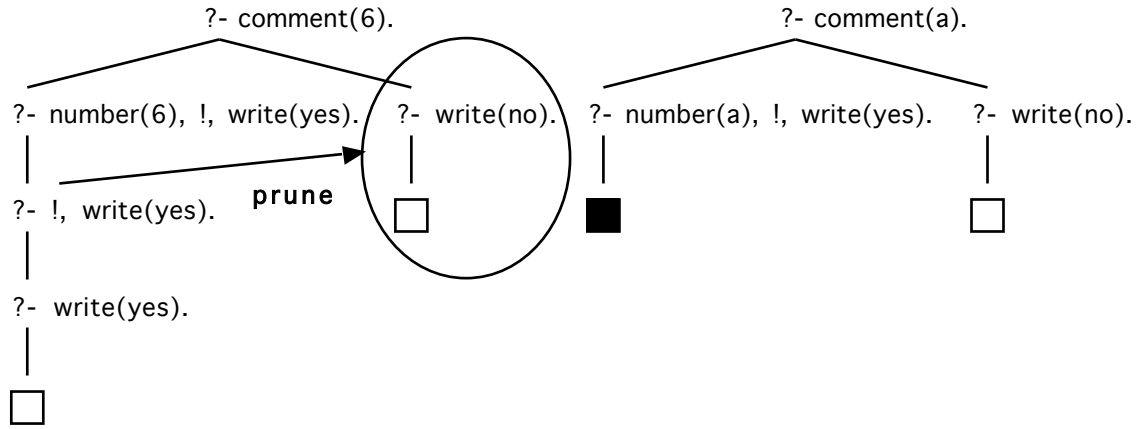
```
comment(X) :- number(X), write(yes).      Note— "write" is a 1-ary Prolog primitive
comment(X) :- \+ number(X), write(no).    which just prints its argument
```

This suffers the (admittedly minor) inefficiency of testing number(X) twice, once in each clause. Using the cut, however, we could instead write

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

so that number(X) is tested only once.

This is what would happen with two sample queries:



In the left-hand example the first computation eventually selects the cut and this immediately prunes out the opportunity of pursuing the second (so-far-untried) computation. Thus the only comment printed is "yes", and this is the desired outcome.

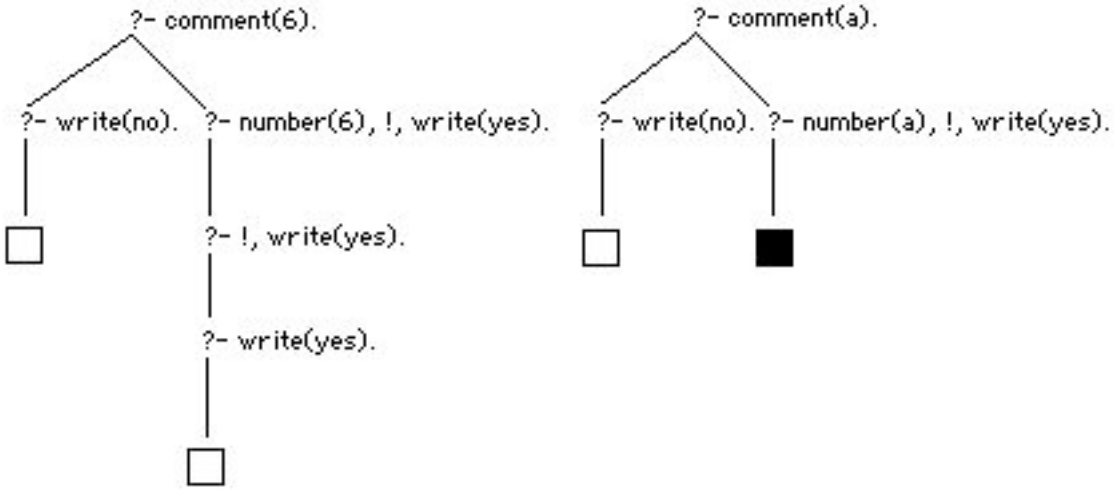
In the right-hand example, the first computation correctly fails *before the cut is selected*, and thus the second computation survives and prints "no", which is again the desired outcome.

Earlier in the course we remarked that, in a finite tree, the branches that will be generated are fixed no matter how the program's clauses are ordered. **This does not necessarily hold in the presence of a cut**.

Suppose we reordered the clauses as

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

In this case the search tree for one of our queries (the left-hand one) differs structurally from what it was previously, whilst the search tree for the other query remains unaffected:



The left-hand example now works incorrectly — it prints both "yes" *and* "no". The cut is executed too late for it to eliminate the "no" outcome.

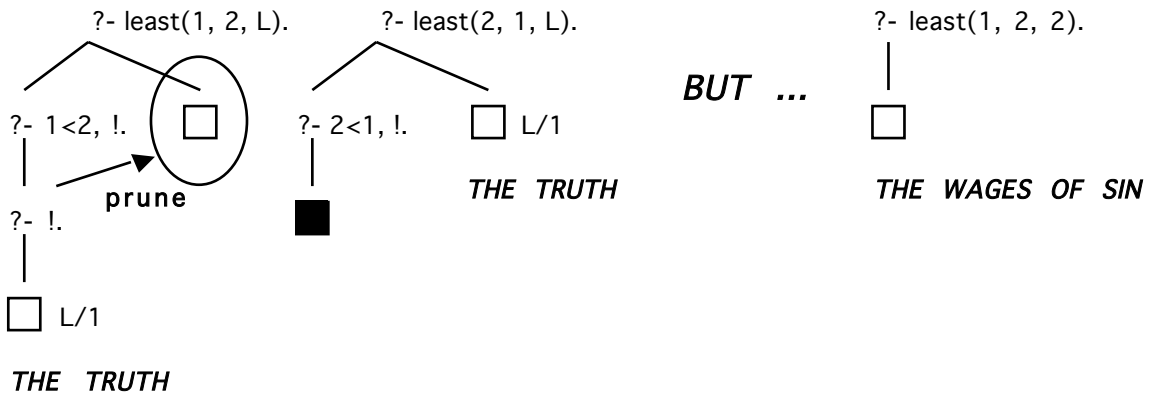
The Great Moral

1. If you can reasonably avoid using the cut, then do so.
2. If you can't, then take great care with the way you order the program's clauses.
3. In either case your duty is to **compute only the truth**.
   Thus, ideally, you should write only clauses that are logically true
   in what they say about the problem domain, whether they include cuts or not.

Here is a more dramatic example of the price you can pay for writing untrue clauses. The following program is intended to define the least number L of any two given numbers X and Y:

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

where least(X, Y, L) ≡ "L is the least of X and Y"



The "sin" lies in the declaration made by the program's second clause that the least of any two numbers X and Y is Y. The result in the right-hand example will be the same whichever way the clauses are ordered.

Another Example

One of the simplest, though least efficient, algorithms for sorting a list is "random interchange-sort":

given an unsorted list X,
- randomly select a disordered pair U, V of consecutive members in X,
- interchange them to produce an improved list Z,
- then repeat the process on Z.

Eventually this algorithm is bound to eliminate all disordered pairs and deliver a sorted list Y.

Here is a preliminary way of expressing this in Prolog, for the case where we want to sort a list X of numbers into *ascending* order:

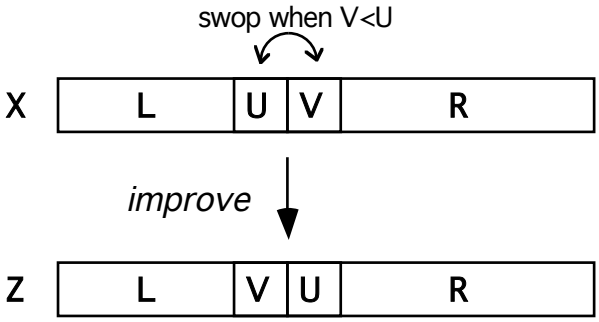
```
rsort(X, X) :- ordered(X).
rsort(X, Y) :- improve(X, Z), rsort(Z, Y).

improve(X, Z) :-  triple_append(L, [U, V], R, X), V<U,
                  triple_append(L, [V, U], R, Z).

triple_append(A, B, C, ABC) :- append(A, B, AB), append(AB, C, ABC).

ordered([]).
ordered([_]).
ordered([U, V | T]) :- U=<V, ordered([V | T]).
```

The operation of "improve(X, Z)" is depicted below:



The program has several defects. Imagine a context in which *all* solutions are required, for instance:

```
p(Ys) :- findall(Y, (inputlist(X), rsort(X, Y)), Ys).
```

Here, for each X delivered by the call inputlist(X), Prolog will seek *every* Y for which rsort(X, Y) holds. We know that there can be only one such Y in each case. But the program will, in fact, produce many identical answers Y for each X.

Take the case X = [3, 1, 4, 2] and suppose the program obtains its first solution by this sequence of swops:

```
swop [3, 1]      to give   [1, 3, 4, 2]
swop [4, 2]      to give   [1, 3, 2, 4]
swop [3, 2]      to give   [1, 2, 3, 4] = first solution for Y
```

However, "findall" works by exhaustively backtracking over *all* ways of solving its call-term, and so it will next do another sequence of swops:

```
swop [4, 2]      to give   [1, 3, 2, 4]
swop [3, 2]      to give   [1, 2, 3, 4] = second (identical) solution for Y
```

Thus, in general the program will needlessly find lots of identical answers. There are various ways of dealing with this by exploiting the cut.

First possibility

Define a new relation as follows:

```
once_only_sort(X, Y) :- rsort(X, Y), !.
```

Then, wherever we would otherwise write rsort(X, Y) we instead write once_only_sort(X, Y), as in

```
p(Ys) :- findall(Y, (inputlist(X), once_only_sort(X, Y)), Ys).
```

Second possibility

Use a cut to prevent the "rsort" program from backtracking over calls to "improve", as in:

```
rsort(X, X) :- ordered(X).
rsort(X, Y) :- improve(X, Z), !, rsort(Z, Y).
```

Note —putting a cut immediately before the **last** body call also helps Prolog to implement *tail-recursion*.

Third possibility

Use a cut to prevent the "improve" program from backtracking over the discovery of disordered pairs, as in:

```
rsort(X, X) :- ordered(X).
rsort(X, Y) :- improve(X, Z), rsort(Z, Y).

improve(X, Z) :-  triple_append(L, [U, V], R, X), V<U, !,
                  triple_append(L, [V, U], R, Z).
```

Another, independent, defect of the original program is that when no more improvements can be made it performs a final, potentially-expensive check for ordererness before concluding that the latest value of Y is indeed the answer. We know that it *has* to be the answer, but Prolog does not have this knowledge. Again, there are various ways of dealing with this.

We might simply drop the "ordered" call from the first clause to leave

```
rsort(X, X).
```

However, we could not then let this remain as the first clause, because we would then have, for instance, sort([3, 1, 4, 2], Y) giving [3, 1, 4, 2] as an answer. Suppose we make it the second clause instead:

```
rsort(X, Y) :- improve(X, Z), rsort(Z, Y).
rsort(X, X).
```

Now, the second clause will be used when the first one cannot make any more improvements. But there is a snag to this—a query such as ?- rsort(chris, chris). will succeed, which is unlikely to be what we want. So we would have to protect this version with a type-check:

```
rsort(X, Y) :- improve(X, Z), rsort(Z, Y).
rsort(X, X) :- list_of_numbers(X).
```

In addition, we would also have to ensure one way or another that *no more than one solution* was sought.

In general, approaches of this kind are unsatisfactory because of the high risk entailed in using *untrue statements* (recall the Wages of Sin).

Altogether it might be better to write a true (if more complicated) program—perhaps employing an additional argument as a flag to indicate whether the latest attempt to improve the list had succeeded or failed.

Meta-programming

"Meta" means "about", and so by "meta-programming" we mean the writing of programs that refer to programs.

Most programming languages in popular use have little or no meta features and are consequently weakened in their expressive qualities. By contrast, most declarative languages possess strong meta features, and this is especially the case with Prolog.

The key to Prolog's meta capabilities lies in the recognition that

terms and predicates have identical syntactic structure.

For instance, the term user_of(X, prolog) can be employed either as a predicate:

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

or as an argument of a predicate:

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

Thus, because we can represent predicates by terms, clauses by lists of predicates and programs by lists of clauses, we do not need to extend our syntax further in order to accommodate meta-programming. Moreover, the computational manipulation of such terms by unification operates without regard to what the terms are intended to denote , so that we do not need to extend our interpreter either.

Built-in meta-predicates

Prolog provides lots of these, and some of them we have already met:

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

Here, P and Q are object-level arguments in the predicates "\+", "forall" and "findall", but are treated at run-time as call-terms at the meta-level (the level at which they are interpreted).

Another particularly useful meta-predicate is the 2-ary "=.." primitive.
This relates a term to a list comprising that term's 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: Given any term, construct a list of all its functors with their arities.
For instance, from p(a, f(X, g(b)), Y) we want the list [(p, 3), (a, 0), (f, 2), (g, 1), (b, 0)].

```
functors(Term, []) :- var(Term), !.
functors(Term, [(F, Arity) | Functors]) :-
        Term =.. [F | Args], length(Args, Arity),  % "length" is a Prolog primitive
        findall(E, ( member(Arg, Args),
                     functors(Arg, Es),
                     member(E, Es) ), Functors).
```

The query for the example above is then ?- functors(p(a, f(X, g(b)), Y), Functors).

Observe the highly complex form of the recursion in this example—overall we have something like

```
q(..., X) :- ..., findall(..., (..., q(...), ...), X).
```

where each call to q spawns, in a single step, an arbitrary number of subtly-related further calls to q.

Meta-variables

A meta-variable is no different, fundamentally, from any other variable, but we give it this title when it we anticipate that is going to become bound to a call-term.

Example —we could write our own program to simulate \+X as follows:

```
our_not(X) :- X, !, fail.      "fail" is a Prolog primitive which always fails finitely
our_not(X).
```

Here, X plays the role of a meta-variable. When we pose a query such as

```
?- our_not(happy(chris)).
```

then unification will bind X/happy(chris), so that X has become a known call-term by the time it is selected for evaluation. If this were not so, then selecting X (unbound) for evaluation would give a run-time error.

Example —we can access predicate symbols from one source and arguments from another, then glue them together using =.. to produce testable calls:

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

and the (entirely fictitious) database

```
aspect(logical).    person(chris).    logical(chris).
aspect(strict).     person(susan).    strict(susan).
aspect(fair).       person(marek).    fair(susan).
aspect(crazy).      person(mike)      fair(marek).
```

we can then ask for aspects of susan by the query ?- tell_us_about(susan, Y).
We will find that she is strict but fair.

Clauses as terms

Entire Prolog clauses are treated by Prolog as terms, a feature which has much significance for the way in which Prolog interpreters are implemented.

The significance of being able to treat clauses as terms is that they can be passed to meta-variables in programs (such as interpreters, transformers and editors) whose role is to manipulate other programs.

Run-time manipulation of clauses

Most Prologs allow clauses to be dynamically consulted, created or deleted during execution. Some (though not Sicstus) require that the relation defined by the clause first be declared "dynamic" (meaning "allowed to vary at run-time").

Example —the 2-ary predicate "likes" can be made dynamic by the declaration

```
:- dynamic likes/2.   Note— declarations begin with :-
```

A dynamic clause may be created by the 1-ary **assert** primitive, or deleted by the 1-ary **retract** primitive.

Examples:
```
?- assert(likes(chris, prolog)).
?- retract((likes(X, haskell) :- crazy(X))).
```

Assertion and deletion of clauses enable one to write in Prolog such things as program editors, transformation tools, tutoring systems, expert systems, volatile databases and much else besides.

Caution —chaotic behaviour will arise if execution is asked to delete clauses currently being executed.

The 2-ary **clause** primitive consults the current dynamic clauses. A call clause(H, B), in which H is a given predicate, succeeds if and only if H unifies with the head of an existing dynamic clause Head :- Body. by some unifier [], whereupon B is returned as the call-term Body[]. In the case that Body is empty, B is returned as the constant *true,* which is Prolog's special way of representing an empty call-term.

Example —the query

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

posed to the set of (dynamic) clauses

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

will give the (alternative) output bindings

```
B/likes(frank, prolog)        B/(honest(frank), praises(frank, chris))
```

these comprising all those instances of our clause-bodies whose successful evaluations would each establish the truth of likes(chris, frank). The third "likes" clause has no contribution to make to this query, since it cannot unify with likes(chris, frank). However, the query ?- clause(likes(frank, X), B). can use that clause to return the output bindings X/prolog and B/true (since the body of the clause was empty).

Example —the query

```
?- depends_on(likes(_, _), honest).
```

posed to the program above together with the clauses

```
depends_on(H, P) :- clause(H, B), callterm_to_list(B, L), member(Call, L), Call =.. [P | _].

callterm_to_list(true, []).                    % this procedure converts a call-term to a list
callterm_to_list((X), [X]) :- \+X=(_, _).
callterm_to_list((X, Y), [X | Ys]) :- !, callterm_to_list(Y, Ys).
```

will succeed, because at least one "likes" clause has a body containing an "honest" call.

Meta-interpreters

These are meta-programs which express ways of interpreting programs and queries supplied as data. They are especially useful for expressing problem-solving strategies more exotic than Prolog's own strategy.

Here is an example of how we might formulate a generic sequential depth-first interpreter:

```
to solve a non-empty query,
        select one of its calls,
        unify this with the head of some program clause,
        combine the other query calls with that clause's body to derive a new query,
        then solve that derived query.
```

```
solve([]).
solve(Query) :-  select(Call, Query, Othercalls),
                 clause(Call, Body),   % seeks a clause whose head unifies with Call
                 callterm_to_list(Body, Bodycalls),
                 combine(Bodycalls, Othercalls, DQ),  % DQ is the derived query
                 solve(DQ).
```

Supporting programs about "select" and "combine" can then be written to specify particular computation rules. We could, for instance, merely simulate Prolog's own computation rule:

```
select(Call, [Call | Othercalls], Othercalls).
combine(A, B, C) :- append(A, B, C).
```

But since we already have Prolog at our disposal, there would not be much point in this!

Another example — the "*procrastination principle*" is a particular computation rule which is often superior to Prolog's, in that it can give much more compact search trees (though at the price of some overhead). It says:

> select whichever call can invoke
> the *fewest* number of clauses

To obtain this behaviour we can write:

```
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(_, [ ], [ ]).              % this procedure deletes a member of a list to extract the others
del(U, [V | X], X) :- U==V, !.
del(U, [V | X], [V | Y]) :- U\==V, del(U, X, Y).

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

The "select" procedure works as follows. Suppose the current query Query is [p, q, r] and that there are two (dynamic) clauses for p, one for q and three for r. Then the list NCs is computed as [(2, p), (1, q), (3, r)]. Sorting this and extracting the first resulting pair (_, Call) will make Call=q, so q is the call selected by the computation rule. The "del" call then extracts Othercalls as [p, r].

Another example —here is how we might simulate a *breadth-first* search strategy:

```
solve_breadth_first(Queries) :- member([], Queries), write(yes), nl.
solve_breadth_first(Queries) :-        % Queries is the current frontier
        findall(DQ, (   member(Q, Queries),
                select(Call, Q, Othercalls),
                clause(Call, Body), callterm_to_list(Body, Bcalls),
        combine(Bcalls, Othercalls, DQ) ), DQs),
        write(DQs), nl, !,             % DQs is the next frontier
        solve_breadth_first(DQs).
```

In each recursive cycle, a breadth-first evaluator has a **list of queries**, each being a leaf on the current frontier of the search tree. To proceed further, it has to find all these queries' immediate descendants, so pushing the search uniformly down one level in the tree. These descendants make up the list DQs of (derived) queries forming the next frontier. A solution is found whenever the current frontier contains an empty query [], as expressed by the first solve_breadth_first clause.

Suppose we supply, as data, the following simple object-level program:

```
a :- b.          d :- e.          f :- g.
a :- d.          d :- f.          f :- h.
b.               e.               g.
                                  h.
```

and that we want to evaluate the query ?- a. Expressed as a list of calls this query is [a]. Thus, our initial *list* of queries (the frontier comprising just the root node) is [[a]]. So, we start the ball rolling with

```
?- write([[a]]), nl, solve_breadth_first([[a]]).
```

Here is the trace of the resulting evaluation, as revealed by the "write" statements:

```
[ [a] ]
[ [b], [d] ]
[ [ ], [e], [f] ]
yes                    finds 1st solution in the 3rd frontier
[ [ ], [g], [h] ]
yes                    finds 2nd solution in the 4th frontier
[ [ ], [ ] ]
yes                    finds 3rd solution in the 5th frontier
yes                    finds 4th solution in the 5th frontier
```

This is exactly what we would expect for the uniform breadth-first traversal of the object-level search tree: