Introduction to Prolog

Marek Sergot

(Slides and notes by
Chris Hogger, Keith Clark,
Fariba Sadri, Murray
Shanahan)

Prolog = "programming in logic"

Prolog is often described as "a high level declarative programming language based on a subset of first-order predicate logic."

This is quite misleading. Prolog is both a declarative language (sometimes) and a procedural language. Both.

Course aims

To provide an introduction to:

- Prolog – basic features
- Logic programming concepts
- A typical Prolog environment (Sicstus Prolog)
- Prolog programming techniques

This is just a taster.

Procedural and declarative readings

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 --- first b then c

The procedure

a :- b, c.

can be written in logic as

$a \leftarrow b \wedge c$ (equivalently $b \wedge c \rightarrow a$)

and can be read declaratively as

a is true if b is true and c is true

So a Prolog program has both a procedural *and* a declarative reading (sometimes).

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).

Queries are sometimes called *goals*.

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

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

By convention, terms beginning with an upper-case letter are *variables*

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

says "for which X is likes(chris, X) true ?"

or "find X such that likes(chris, X) follows from the program"

Computations

A *computation* is a chain of *derived queries* (or 'goals'), 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 the standard notion of *procedure-calling*

In logic, it is a special case of an inference rule called *resolution*

EXAMPLE

?- a, d, e.   initial query

a :- b, c.   program clause with
head a and body b, c

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

?- b, c, d, e.

Execution then treats the derived query/goal in the same way

Successful computations

A computation succeeds if it derives the *empty* query/goal

EXAMPLE

?- likes(bob, prolog).   query/goal

likes(bob, prolog).   program clause

The call matches the head and is replaced by the clause's (empty) body, and so the derived query/qoal is *empty*

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

Finite failure

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

EXAMPLE

?- likes(bob, haskell).   query/goal

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

ANOTHER EXAMPLE

?- likes(chris, haskell).   query/goal

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

Derived query/goal:

?- nice(haskell).

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

Multiple answers

A query/goal may produce many computations

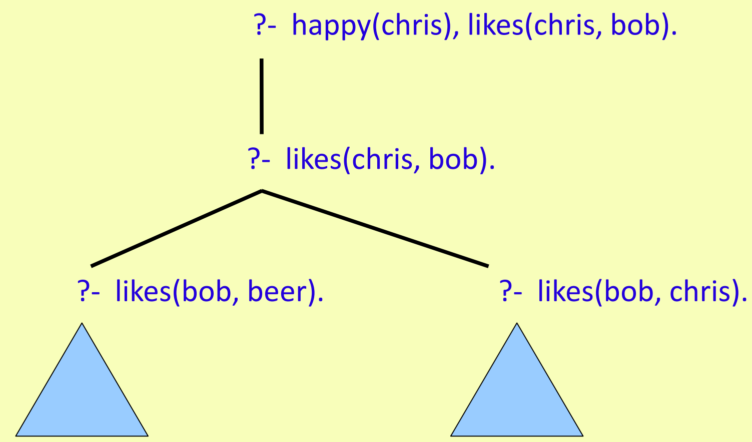
Those, if any, that succeed may yield multiple answers (not necessarily distinct).

EXAMPLE

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

```
happy(chris).

likes(chris, bob)  :-  likes(bob, beer).
likes(chris, bob)  :-  likes(bob, chris).
```


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



these subtrees may or may not succeed, depending on what else is in the program

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 then the query is not a consequence of P.

?- a, d, e.

If this *fails* (finitely) from a program P then we have

not ($P \models a \wedge d \wedge e$)

(which is not the same as $P \models \neg (a \wedge d \wedge e)$)

Variable arguments

Variables in queries are treated as existentially quantified

EXAMPLE

?- likes(X, prolog).

says “is $\exists$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, chris).

expresses the sentence

$\forall$X (likes(chris, X) $\leftarrow$ likes(X, chris))

It follows that the scope of a variable is just the clause or query in which it appears

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

$\forall$X (likes(chris, X) $\leftarrow$ likes(X, chris))

has the same meaning as

likes(chris, P) :- likes(P, chris).

$\forall$P (likes(chris, P) $\leftarrow$ likes(P, chris))

Generalised matching

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

EXAMPLE

?- likes(U, chris).

likes(bob, Y) :- understands(bob, Y).

Here, likes(U, chris) and likes(bob, Y) can be made *identical* (forced to match) by binding

U / bob and Y / chris

This process is called *unification*

The derived query is

?- understands(bob, chris).

EXAMPLE

$\forall S\ (pass\_msc(S) \leftarrow pass\_exams(S) \wedge pass\_cwks(S) \wedge pass\_projs(S)\ )$

```
pass_msc(S) :-
    pass_exams(S),
    pass_cwks(S),
    pass_projs(S).

pass_exams(john).
pass_cwks(john).
pass_projs(john).

pass_cwks(mary).
```


?- pass_msc(john).

Answer: yes P |= pass_msc(john)

?- pass_msc(mary).

Answer: no not (P |= pass_msc(mary))

?- pass_msc(X).

Answer: X = john

EXAMPLE (Trading)

```
sells(usa, grain, mexico).
sells(S, P, R) :-  produces(S, P), needs(R, P).

produces(oman, oil).
produces(iraq, oil).
produces(japan, cameras).
produces(germany, pork).
produces(france, wine).

needs(britain, cars).
needs(japan, cars).
needs(france, pork).
needs(_, cameras).        % _ is a variable
needs(C, oil) :- needs(C, cars).
```

EXAMPLE: Some queries

```
?-  produces(oman, oil).
yes

?-  produces(X, oil).
X = oman ;       %  ';' is request for another answer
X = iraq ;
no               %  'no' means no more answers

?-  produces(japan, X).
X = cameras ;
no
```


EXAMPLE: Some queries, contd

```
?-  produces(X,Y).
X = oman,  Y= oil ;
X = iraq,     Y= oil ;
X = japan,  Y= cameras ;
X = germany,   Y= pork ;
X = france, Y= wine ;
no

?-  produces(X, rice).
no

?-  produces(iraq, Y), needs(britain, Y).
Y = oil ;
no
```


EXERCISE: Write Prolog queries for:

Who sells grain to whom?

Who sells oil to Britain?

Who sells what to Hungary?

Who sells something to Hungary?

Does Britain sell oil to the USA?

Which two countries have mutual trade with one another?

Which two different countries have mutual trade with one another? (X\=Z means X and Z are different from one another.)

Write a Prolog rule for bilateral_traders(X,Z) such that X and Z are two different countries that have mutual trade with one another.

Who produces something that is needed by both Britain and Japan?

EXAMPLE (Work-Manager)

```
worksIn(bill, sales).
worksIn(sally, accounts).

deptManager(sales, joan).
deptManager(accounts, henry).

managerOf(joan, james).
managerOf(henry, james).
managerOf(james, paul).
managerOf(W, M) :-
    worksIn(W, Dept),
    deptManager(Dept, M).
```

Exercise: define colleague/2 such that colleague(W1,W2) holds if W1 and W2 are different colleagues in the same department

EXAMPLE, contd (Recursion)

```
superiorOf(E,S) :-
    managerOf(E,S).
superiorOf(E,S) :-
    managerOf(E,M),
    superiorOf(M,S).
```

superiorOf/2 is a recursive predicate.
The first rule for superiorOf/2 is a base case.
The second rule for superiorOf/2 is a recursive rule.

With earlier facts and rules we get:

```
?- superiorOf(bill,paul).
yes
```

```
?- superiorOf(X,Y).
```
(Try it!)

Answers: logical status

Program P
Query Q with no variables

Prolog	Logic
Answer yes	P |= Q
Answer no	not the case that P |= Q

Answers: logical status

Program P
Query Q with variables $X_1, \ldots, X_m$

Answer displayed by Prolog is θ (e.g. X = john)

Logic: $P \models \forall X_1 \ldots \forall X_m (Q\theta)$

Answer displayed by Prolog is no

Logic: There is no substitution θ such that $P \models Q\theta$

This is the sense in which Prolog is a declarative language

Prolog: procedural programming

Prolog is also a *procedural* programming language.
Some Prolog programs don't have a (meaningful) declarative reading – or maybe just fragments do.

```
example_prog :-
    write('Give me a name  '),
    read(X),
    friend(X, Y),        % a declarative fragment
    write(Y),
    write(' is a friend of '),
    write(X),
    nl.                  % outputs newline
```

Prolog: Syntax

Prolog program is a sequence (set) of *clauses*.

A clause has the form:

	H :- C1, ..., Ck.	*conditional clause*
or	H.	*unconditional clause*

H is the *head* of the clause; C1, ..., Ck is the *body*.

A terminating
'.<space>',
'.<newline>' or
'.<tab>'
is essential after each clause.

Syntax: clauses

H :- C1, ..., Ck.

H and each Ci is an atomic formula of the form:

p(t1, ..., tn) or p

Must be NO SPACE between p *and the* (

p is the *predicate* or *relation name* of p(t1, ..., tn).
t1, ..., tn are terms.

The clause is *about* the predicate of H.
Each Ci in the body is referred to as a *call* or a *condition*.

Syntax: atomic formulas

p(t1, ..., tn) or p

p is the predicate or relation name of p(t1, ..., tn).
t1, ..., tn are terms.

Prolog allows same predicate symbol with different arities:

p, p(t1), p(t1, t2, t3) etc

p/0, p/1, p/3 etc are *different* predicates.

Prolog is *not typed*

Clauses: logical reading

A conditional clause

H :- C1, ..., Ck.

is read as

$$\forall X_1 \ldots \forall X_m \, (C1 \wedge \ldots \wedge Ck \rightarrow H)$$

where $X_1, \ldots, X_m$ are *all* the variables that appear in the clause, or equivalently as

$$\forall X_1 \ldots \forall X_i \, (\exists X_{i+1} \ldots \exists X_m \, (C1 \wedge \ldots \wedge Ck) \rightarrow H)$$

where $X_{i+1}, \ldots, X_m$ are variables that appear *only* in the conditions (body) of the clause.

Clauses: logical reading

An unconditional clause

H.

is read as

$\forall X_1 \ldots \forall X_m (H)$

where $X_1, \ldots, X_m$ are all the variables that appear in H.

Prolog terms

constants (Prolog calls them `atoms') – alphanumeric sequence beginning with a *lower case letter* or in *single quotes*

bill maryJones 'Mary Jones' 'fs@doc' '****'

variable names – alphanumeric sequence beginning with an *upper case letter* or _ (underscore)

X Person _ _46

compound terms – expressions of the form

f(t1, ..., tn)

where f is a *function name* (same syntax as constant) and t1, ..., tn are terms.

There are three other kinds of terms – numbers, strings, and lists (we will come to these later).

'Facts' and 'rules'

If an unconditional clause

H.

contains no variables then the clause is called a *fact*.

```
pass_cwks(john).
father(cain, adam).
```

All other clauses are called *rules*.

```
drinks(john) :- anxious(john).
anxious(X) :- has_Prolog_test(X), lazy(X).
needs(_, water).     % everyone needs water
```


Prolog queries

A *query* is a conjunction of conditions, i.e.

?- C1, ..., Cn. <newline>

Each Ci is a *condition/call* (as in the body of a clause)

?- is a prompt displayed by the Prolog environment

Prolog will report values for all variables in the query except those ('anonymous variables') beginning with _ (underscore).

2.1

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

2.2

Terms containing no variables are said to be *ground*

A special feature of Prolog is that it 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

2.3

SIMPLE TERMS ('constants')

numbers (integers, floating, ...)

3   5.6   –10   –6.31

'atoms' (Prolog terminology)

apple   tom   x2   'Hello there'   [ ]

variables

X   Y31   Chris   Left_Subtree   Person   _35   _

(and some others)

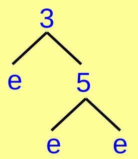
2.4

COMPOUND TERMS

f(t1, …, tn)

mother(chris)

tree(e, 3, tree(e, 5, e)) →

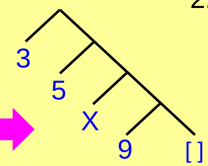


tree(T, N, tree(e, 5, e)) →

a binary tree whose root and left-subtree are unknown *list terms*

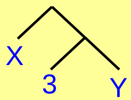
[ ]   [3, 5]   [3, 5, X, 9]

lists form a *subclass of binary trees*

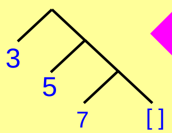


A vertical bar can be used as a separator to present a list in the form [*itemized members* | *residual-list*]

[X, 3 | Y]



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

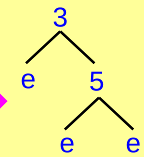


[3, 5, 7]

tuple terms

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

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



Sometimes more efficient when working with *fixed-length* data structures (though not in Sicstus)

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. They are merely shorthand for

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

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

2.9

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


?- 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

2.9a

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


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

fails   (no clause with matching head)

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

2.9b

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


?- all_bs([ b, X, b ]).
?- all_bs([ X, b ]).
?- all_bs([ b ]).            % binds X = b
?- all_bs([ ]).
?- .
```

So here the search tree comprises ONE branch (computation), which happens to succeed with an answer substitution X = b

2.10

ANOTHER 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).
```

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

?- 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).
```



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

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

Another similar step binds Z / [b | Z2] and produces the next derived query

```
?- app( [ ], [ c, d ], Z2).
```

This succeeds by matching the program's first clause, and binds Z2 / [c, d]

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

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 having any mix of argument modes we wish

Introduction to Prolog, contd

3.1

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*

3.2

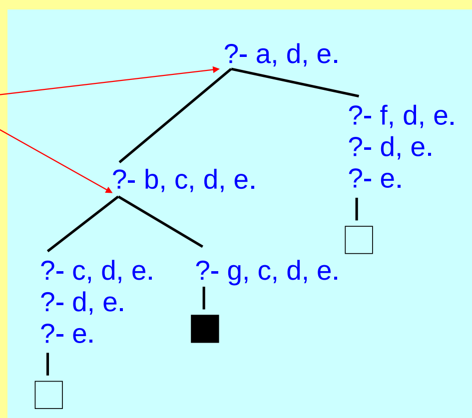
EXAMPLE

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

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

3.3

these nodes
are called
choice points



3.4

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

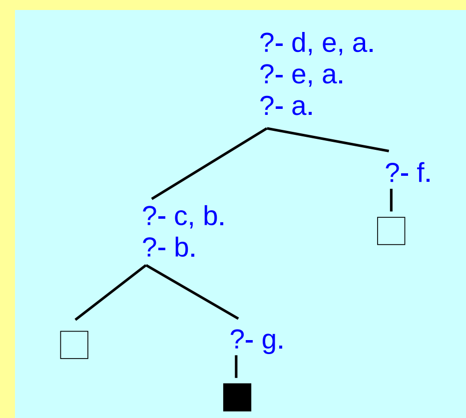
- the way knowledge is represented in the program
- the ordering of calls

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**, whilst 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

UNIFICATION

There is an algorithm for unifying two atomic formulas and producing the most general unifying substitution.

h(X, a, Y) and h(b, Z, W)

unify with

θ = { X / b, Z / a, Y / W} (or W \ Y)

Both become

h(b, a, W)

3.13

UNIFICATION, contd

unify with

p(X, Y, h(Y)) and p(Z, a, Z)

θ = { X / h(a), Y / a, Z / h(a)}

Both become

p(h(a), a, h(a))

Exercise: find out more about unification by entering queries using Prolog's =/2 unification primitive. Try queries:

```
?- p(X, Y, h(Y)) = p(Z, a, Z)    (as above)
?- p(X) = p(t(X))
```

The second one should fail. (But in most Prologs it doesn't. See 'occurs_check' in the Prolog manual.)

3.14

THE GENERAL COMPUTATION STEP

current query ?- P(args1), others.

program clause P(args2) :- body.

If θ exists such that P(args1)θ = P(args2)θ

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 3.15

?- 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 TERMS IN Prolog

[ ] : empty list

[H|T] :a list which has first element H followed by the list T.

H is called the *head* of the list. T is called the *tail* of the list.

[H1,H2|T] a list with head H1 followed by a tail list with head H2 and tail list T. This is equivalent to [H1|[H2|T]]

[[H|T1]|T2] a list with tail T2 and a head that is a list with head H and tail T1.

[1,2,3] is shorthand for [1|[2|[3|[]]]]

EXERCISE

Unifying	with	gives the substitution
[X, Y, Z]	[a, b, c]	
[X|Y]	[a, b, c]	
[X|Y]	[a]	
[X, Y|Z]	[a, b, c]	
[X|Y]	[[1, 2], [3, 4]]	
[[X|Y]|Z]	[[1, 2], 3]	
[X, Y]	[1, [2, 3]]	
[X, Y]	[1]	

You can check your answers by means of Prolog's unification primitive =/2.

4.1

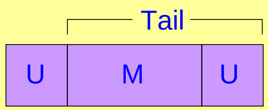
LIST PROCESSING

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).
```



4.2

More abstractly:

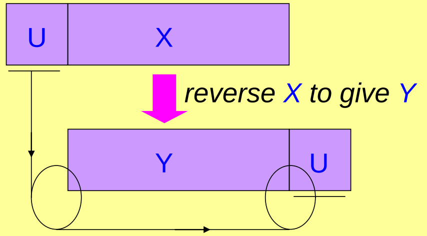
```
palin([ ]).
palin(L) :- first(L, U), last(L, U),
            middle(L, M), palin(M).

first([U | _], U).

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

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


4.3

EXAMPLE

To reverse a list:

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



4.4

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.

List membership: member/2

member(X, L) : X is a member of list L
e.g.

```
member(2,[2,3])
member(3,[2,3])
```

It is provided in the Sicstus list primitives library. Internally, defined like this:

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


Some uses of member/2

Find elements on a given list:

```
?- member(X, [tom, dick, harry]).
   X = tom;
   X = dick;
   X = harry;
   no
```

Check if a given element is on a given list:

```
?- member(1, [9,1,3]).
   yes
```


Some uses of member/2

Insert element on a partially given list:

```
?- member(1, [9,1,U]).
   true;  (equivalent to yes for all U)
   U=1;
   no
```

Generate templates for all lists on which a given element occurs.

```
?- member(1, X).
   X = [1|Xs];
   X = [X1, 1|Xs];
   X = [X1, X2, 1|Xs];
```

(infinitely many answers)

A Combinatorial Puzzle

There are five houses in a line, each with an owner, a pet, a cigarette brand, a drink, and a colour.

The Englishman lives in the red house.
The Spaniard owns the dog.
Coffee is drunk in the green house.
The Ukrainian drinks tea.
The green house is immediately to the right of the ivory house.
The Winston smoker owns snails.
Kools are smoked in the yellow house.
Milk is drunk in the middle house.
The Norwegian lives in the first house on the left.
The man who smokes Chesterfields lives next to the man with the fox.
Kools are smoked in the house next to the house with the horse.
The Lucky Strike smoker drinks orange juice.
The Japanese smokes Parliaments.
The Norwegian lives next to the blue house.

A Combinatorial Puzzle

Who drinks water? Who owns the zebra?

(Find the complete description of the row of houses.)

One Prolog formulation


```
zebra(H,W,Z) :-
  H = [house(norwegian,_,_,_,_),_,
       house(_,_,_,milk,_), _, _],
  member(house(englishman,_,_,_,red), H),
  member(house(spaniard,dog,_,_,_), H),
  member(house(_,_,_,coffee ,green), H),
  member(house(ukrainian,_,_,tea,_), H),
  followedBy(house(_,_,_,_,ivory),
            house(_,_,_,_,green), H),
  member(house(_,snails,winston,_,_), H),
  member(house(_,_,kools,_,yellow), H),
  nextTo(house(_,_,chesterfield,_,_),
         house(_,fox,_,_,_), H),
  nextTo(house(_,_,kools,_,_),
         house(_,horse,_,_,_), H),
  member(house(_,_,lucky_strike, orange_juice,_), H),
  member(house(japanese,_,parliaments,_,_), H),
  nextTo(house(norwegian,_,_,_,_),
         house(_,_,_,_,blue), H),
  member(house(W,_,_,water,_), H),
  member(house(Z,zebra,_,_,_), H).
```

Not very efficient!! But it works. (Try it)

5.1

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

5.2

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 lessl

5.3

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

X is E

EXAMPLES

?- X is (2+2).	succeeds and binds X / 4
?- 4 is (2+2).	succeeds
?- 4 is (2+3).	fails
?- X is (Y+2).	gives an error

5.4

X=Y means “X can be unified with Y”

Do not confuse is with =

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

5.5

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

5.6

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

5.7

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

$$T = S + \sum Ns$$

5.8

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

6. Negation in Prolog

Prolog does not have a connective for classical negation

It has a special operator \+ read as

"fail (finitely) to prove"

The operational meaning of \+ is

\+ P succeeds iff P fails finitely

\+ P fails iff P succeeds

Negation in Prolog

In Prolog's negation \+ is allowed only in queries and in the bodies of rules - not in the heads of rules.
E.g.

```
?- student(X), \+ gets_grant(X).

happy(X) :-
  owns_a_house(X),
  \+ has_mortgage(X).
```


The Negation as Failure (Naf) Rule

- \+ Q is proved if all evaluation paths for the query Q end in failure.
- Proof of \+Q will not generate any bindings for variables in Q.
- If Q contains variables $X_1,..,X_k$ at the time it is evaluated it behaves like:

$$\neg \exists X_1 \ldots \exists X_k\, Q$$

Connected to the '*Closed World Assumption*' --- anything that is not *known* to be true is assumed to be *false*

EXAMPLE

```
student(john).
student(mary).

gets_grant(john).


?- student(X), \+ gets_grant(X).
X = mary
```

Mary is a student and Prolog cannot *prove* that Mary gets a grant.

ANOTHER EXAMPLE

```
dragon(puff).
dragon(macy).
dragon(timothy).

magic(puff).
vegetarian(macy).

lives_forever(X)  :-  magic(X).
lives_forever(X)  :-  vegetarian(X).

        ?-  dragon(X), \+ lives_forever(X).
```

Construct the Prolog evaluation to see how it finds the answers.

ANOTHER EXAMPLE

Assuming a set of male/1 and parent/2 facts:

Who are the males with no children:

```
?- male(P), \+ parent(P,_).
```

P is a male who has no sons:

```
no_sons(P)  :-  male(P),
                \+ (parent(P,C),  male(C)).
```

P is a male who has no daughters

```
no_daughters(P)  :-  male(P),
                     \+ (parent(P,C), \+ male(C)).
```


\+ is not the same as classical negation

EXAMPLE

p $\leftarrow \neg$ p   classically implies p

but

p :- \+ p   cannot solve ?- p
(it has to fail *finitely*)

So p is a logical consequence in the first case, but it is not a computable consequence in the second
(There are other differences)

Negated conditions with unbound variables

Variables in negative conditions can give the wrong answers

```
?-  dragon(X), \+ lives_forever(X).
```

has answers. But

```
?-  \+ lives_forever(X), dragon(X).
```

has no answers. Why?

Apply the Naf inference rule to first condition: what is the result?

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

EXAMPLE

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

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

"X is sad if someone else doesn't like X"

In the example, bob, chris and frank are sad.

ANOTHER EXAMPLE

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

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

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

In the example, bob and chris are very sad.

EXAMPLE

Every person X is happy if all friends of X like logic

In classical logic we could write

$$\text{happy}(X) \leftarrow \text{person}(X) \wedge \forall Y\ (\text{friend}(X,Y) \rightarrow \text{likes}(Y, \text{logic}))$$

or equivalently

$$\text{happy}(X) \leftarrow \text{person}(X) \wedge \neg\exists Y\ (\text{friend}(X,Y) \wedge \neg\ \text{likes}(Y,\text{logic}))$$

EXAMPLE

$$\text{happy}(X) \leftarrow \text{person}(X) \wedge \neg\exists Y\ (\text{friend}(X,Y) \wedge \neg\ \text{likes}(Y,\text{logic}))$$

In Prolog we can write:

```
happy(X) :-
   person(X),
   \+ ( friend(X, Y), \+ likes(Y, logic) )
```

7. CONTROLLING SEARCH (the 'cut')

- Cut, denoted by "!", is a Prolog evaluation control primitive.
- It is "extra-logical": it is used to control the search for solutions and prune the search space.
- The cut can only be understood procedurally, in contrast to the declarative style that logic programming encourages.
- But used carefully, it can significantly improve efficiency without compromising clarity too much.

Example: needless search

```
send(Cust, Balance, Message)  :-
          Balance =< 0,
          warning(Cust, Message).
send(Cust, Balance, Message)  :-
          Balance > 0,
          Balance=< 50000,
          credit_card_info(Cust, Message).
send(Cust, Balance, Message)  :-
          Balance > 50000,
          investment_offer(Cust, Message).
```


For a condition/call:

send(frank, -10, Message)

in a query for which all solutions are being sought, Prolog will try to use second and third clause after an answer has been found using the first clause.

Clearly this search is pointless.

Using the 'cut': !

```
send(Cust, Balance, Message)  :-
          Balance =< 0, !,
          warning(Cust, Message).
send(Cust, Balance, Message)  :-
          Balance=< 50000, !,
          credit_card_info(Cust, Message).
send(Cust, Balance, Message)  :-
         investment_offer(Cust, Message).
```

The Effect of !

```
p(...) :-  T1, ..., Tk, !, B1, ..., Bn.
p(...) :-  …
p(...) :-  …
```

In trying to solve a call:

p(…)

if the first clause is applicable, and T1, ..., Tk all succeed, then on *backtracking*:

* *do not try* to find an alternative solution for T1, ..., Tk and
* *do not try* to use any other clauses for the call p(…).

EXAMPLE

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

This program tests a term X and prints a comment.

The intention is that if X is a number then the comment is yes, and otherwise is no.

Will this program work correctly (assuming X is ground)?

ANOTHER EXAMPLE

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

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

?- least(1, 2, M)   correctly succeeds, with M = 1
?- least(2, 1, M)   correctly succeeds, with M = 1

BUT …

?- least(1, 2, 2)   wrongly succeeds

Exercise: fix this

Sicstus Prolog definition of length/2

?- length(L, 2)

fails if we ask for a *second* solution with L unbound.

But evaluation of len(L, 2) where:

```
len([ ], 0).
len([_|L], N)  :-  len(L, M), N is M+1.
```

goes into infinite loop is this case.

Why the difference?

Sicstus length/2 has a definition that uses !
That definition is equivalent to:

```
length(L, N) :-
        number(N),
        len(L ,N), !.
length(L, N) :-
        len(L, N).
```

The cut ! in the first clause prevents Prolog from backtracking to try to find more solutions to len/2 call and prevents use of the second clause.

Prolog definition of \+

```
\+(P) :- P, !, fail.
\+(_).
```

fail is a Prolog primitive that always fails.

Prolog Conditional

Related to the !, is the Prolog conditional test:

(Test -> P ; Q)

Each of Test, P, Q can be a general Prolog query.

If Test succeeds then evaluate P else evaluate Q --- but don't backtrack for more solutions to Test if P fails

EXAMPLE

```
student_fees(S, F)  :-
    student(S),
    (eu(S)  ->   F=3000  ;  F=19000 ).
```

Equivalent to:

```
student_fees(S, F)  :-
    student(S),
    fees_for(S, F).

fees_for(S, F) :-
    eu(S), !, F = 3000.
fees_for(S, F)  :-  F=19000.
```

EARLIER EXAMPLE

```
send(Cust, Balance, Message)  :-
          (
           Balance =< 0
           ->
               warning(Cust, Message)
           ;
          Balance =< 50000
           ->
                credit_card_info(Cust, Message)
           ;
               % otherwise
                investment_offer(Cust, Message)
           ).
```


EXAMPLE (a common pattern)

We want to print out all the friends of X.

```
print_friends(X)  :-
   write('The friends of '), write(X), write(':'), nl,
   friend(X, Y),
   write('  '), write(Y),
   nl,
   fail.
print_friends(_)  :-
   write('Done'),
   nl.
```


EXAMPLE (just an example)

```
print_all_friends(X)  :-
   person(X),
   friend(X, _), !,
   print_friends(X).    % as above
print_all_friends(X)  :-
   write(X),
   (person(X)
     ->  write(' has no friends!')
     ;     write(' is not a person!')
   ),
   nl.
```

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)
```

List is the list of solutions. It is not sorted and may contain duplicates.

EXAMPLE

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

To find all those whom chris likes:

```
?- findall(X, likes(chris, X), L).
L = [ bob, frank ]
```

Another example:

```
?- findall(X, likes(X, _), L).
L = [ frank, chris, chris ]
```


ANOTHER EXAMPLE

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

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

Another example:

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


ANOTHER (silly) EXAMPLE

```
?- findall(p(X, [X], X) , member(X, [ a, b, c ]), S).
S = [ p(a,[a],a), p(b,[b],b), p(c,[c],c) ]
```

Another (silly) example:

```
?- findall(g , member(X, [ a, b, c ]), S).
S = [ g, g, g ]
```

ANOTHER EXAMPLE

The list of children of a mother M and a father F:

```
children_of(M, F, Children) :-
        findall( C,
                 ( mother_of(M, C), father_of(F, C ) ),
                 Children ).
```

(Children could contain duplicates)

ANOTHER 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

ANOTHER EXAMPLE

Construct a list L of pairs (X, N) where X is a person
and N is the number of friends of X:

```
friend_number_list(L) :-
        findall( (X, N),
                 ( person(X),
                   findall(Y, friend(X, Y), F),
                   remove_duplicates(F, Fx),
                   length(Fx, N)
                 ),
                L ).
```


The setof/3 primitive

setof(Term, Call-term, List)

This is more powerful than findall/3.

It *removes duplicates*. It also *automatically orders* the answer list using the predefined term ordering (=<) -- the normal numeric ordering for numbers and lexical ordering for constants.

There are also some important differences concerning variables in Call-term.

EXAMPLE 8.9

```
admires(jane, peter).     admires(kate, john).
admires(jane, amy).       admires(kate, mary).
admires(jane, bill).
```

```
?-  findall(X, admires(M, X), L)
X = [ peter, john, amy, mary, bill ] ;
no
```

Here M is existentially quantified. Equivalent to:

```
?-  findall(X, admires(_, X), L)
```

BUT

```
?-  setof(X, admires(M, X), L)
M = jane, L = [ amy, bill, peter ] ;
M = kate, L = [ john, mary ]
```


EXAMPLE, contd 8.10

Compare

```
?-  setof(X, admires(M, X), L)
M = jane, L = [ amy, bill, peter ] ;
M = kate, L = [ john, mary ]
```

```
?-  setof(X, admires(_, X), L)
L = [ amy, bill, peter ] ;
L = [ john, mary ]
```

```
?-  setof(X, M^admires(M, X), L)
L = [ amy, bill, john, mary, peter ] ;
no
```

(like findall but sorted)