

Introduction to Prolog

Dalal Alrajeh

www.doc.ic.ac.uk/~da04

dalal.alrajeh@ic.ac.uk

Spring 2018

Material collated with notes and slides from
Marek Sergot, Chris Hogger, Keith Clark, Fariba Sadri and 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” called *logic programs*.

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

A Prolog Example

Natural language processing (e.g., IBM Watson)

Machine learning (e.g., inductive logic programming)

Expert systems (e.g., decision systems for medical diagnoses)

Planning (e.g., autonomous agents)

AI problem solving (e.g., [Sudoku](#))

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.

What will be covered

Lectures (10 hours)

- Main concepts (declarative/procedural meaning, procedures, arguments, computations, answers, matching ...)
- Prolog constructs (atomic formulas, clauses, terms, queries)
- Prolog computation (evaluation, efficiency, unification)
- List processing
- Arithmetic operations
- Negation
- Controlling search (cut, conditionals)
- Aggregation (findall/3, setof/3)

What will be covered

Labs (2 sessions)

- Running prolog
- Databases & recursions
- Lists & structures

Coursework (1)

- Issue date: Jan 11th, 2018
- Deadline: Jan 18th, 2018
- Work in pairs permitted.

Resources

Main

- Lecture slides on CATE

Additional

- Online:
 - Sicstus manual
<https://sicstus.sics.se/documentation.html>
 - Additional resources (programs)
www.doc.ic.ac.uk/~da04/teaching/prolog276
- Book:
 - *Prolog Programming for Artificial Intelligence*
By Ivan Bratko, 3rd edition available in the library

Main Concepts

Procedural and declarative readings

Programs consist of procedure definitions.

A *procedure* is a resource for evaluating something, i.e., determining whether or not it is true according to the program.

EXAMPLE

$$\underbrace{a}_{\text{head}} \text{ :- } \underbrace{b, c}_{\text{body}}.$$

This is read *procedurally*: evaluating *a* (the head) is achieved by evaluating both *b* and *c* (the body) --- first *b* then *c*.

Procedural and declarative readings

The procedure

$a \text{ :- } b, c.$

can be written as the logic clause

$a \leftarrow b \wedge c$

or equivalently as: $b \wedge c \rightarrow a$, $\neg b \vee \neg c \vee a$

and can be read *declaratively* as

a is true if b is true and c is true

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

Procedure calls

Execution involves evaluating *procedure calls*, or *queries*, starting 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*.

Procedure calls

Prolog evaluates calls in queries sequentially, in the left-to-right order as written, e.g.,

?- 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”

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

Variable arguments

Variables in *queries* are treated as *existentially* quantified.

EXAMPLE

Query: ?- likes(X, prolog).

says “is $\exists X$ likes(X, prolog) true?”

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

Variable arguments

Variables in *program clauses* are treated as *universally* quantified.

EXAMPLE

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

expresses the sentence

$\forall X (\text{likes}(\text{chris}, X) \leftarrow \text{likes}(X, \text{chris}))$

Variable arguments

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

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

Computations

A *computation* is a chain of *derived queries/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*.

Computations

EXAMPLE

Initial query/goal: ?- a, d, e.

Program: a :- b, c. (clause with head a and body b, c)

The first call in the initial query/goal 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.

see: pt1_s17.pl

Successful computations

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

EXAMPLE

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

Program: `likes(bob, prolog).` (clause with *empty body*)

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

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

see: pt1_s18.pl

Finite failure of computations

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

EXAMPLE

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

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

see: pt1_s17.pl

Finite failure of computations

EXAMPLE

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

Program: 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.

see: pt1_s18.pl

Multiple answers

A query/goal may produce many computations.

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

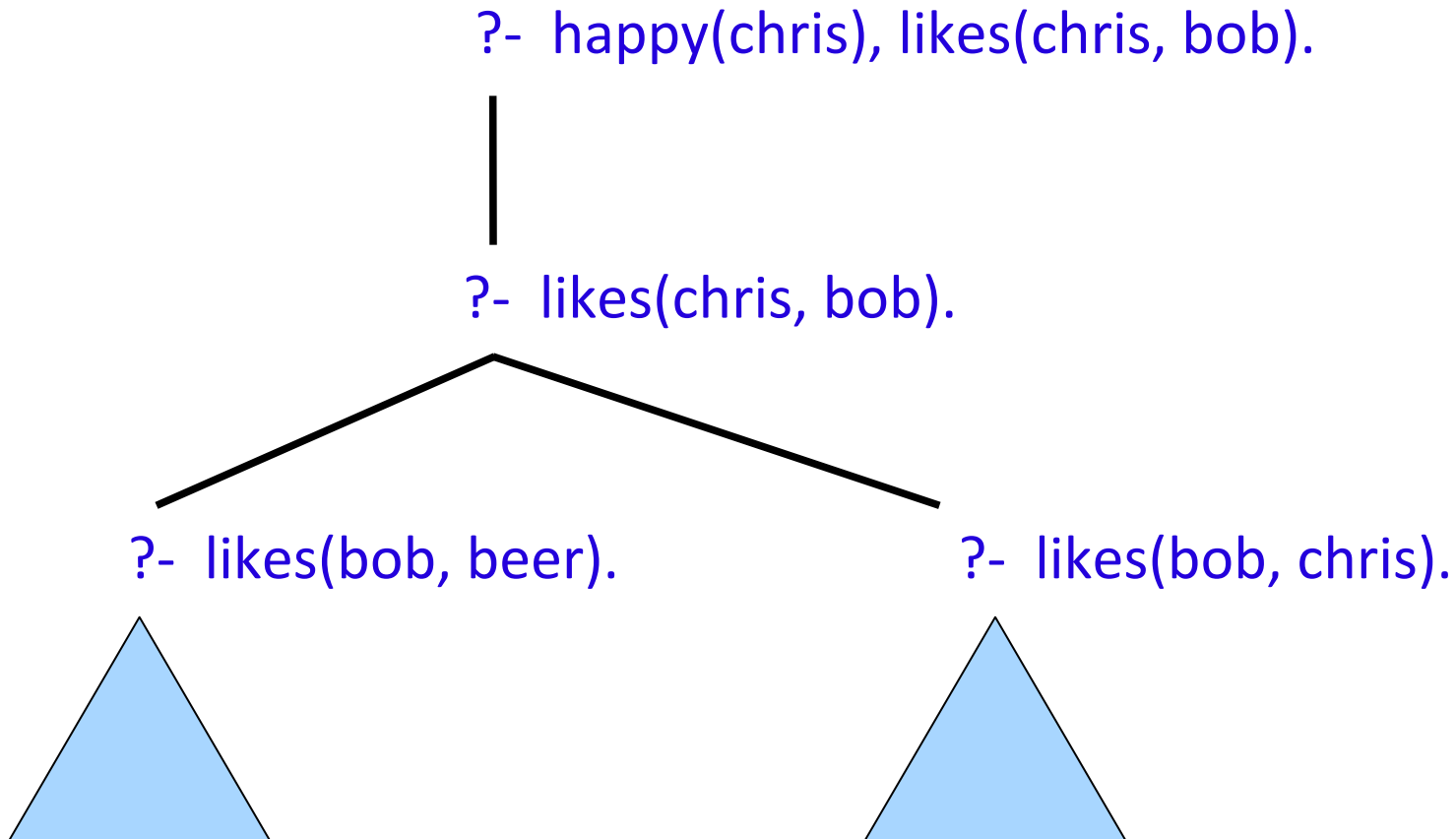
EXAMPLE

Initial query/goal: ?- happy(chris), likes(chris, bob).

Program: {
 happy(chris).
 likes(chris, bob) :- likes(bob, beer).
 likes(chris, bob) :- likes(bob, chris).

see: pt1_s18.pl

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



These sub-trees 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

Initial query/goal: $?- a, d, e.$

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

$$a \wedge d \wedge e$$

and we have $P \models a \wedge d \wedge e.$

Answers as consequences

Conversely

If the program P does not offer any successful computation then the query is not a consequence of P .

Initial query/goal: $? a, d, e.$

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 $P \models \neg (a \wedge d \wedge e)$

Answers: logical status

Program: P

Query: Q (no variables)

Prolog answer

yes

no

Logic

$P \models Q$

not ($P \models Q$) *read as not the case
that $P \models Q$*

Answers: logical status

EXAMPLE

Query: ?- pass_year(mary).

Program: pass_year(S) :-
 pass_exams(S), pass_cwks(S),
 pass_projs(S).

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

pass_cwks(mary).

see: pt1_s26.pl

Answer: no

Answers: logical status

Program: P

Query: Q with variables X_1, \dots, X_n

Prolog answer

Substitution θ

no

Logic

$P \models \forall X_1, \dots, \forall X_n (Q\theta)$

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

EXAMPLE

Query: `pass_year(X).`

Answer: `X/john`

*This is the sense in which
Prolog is declarative*

Logic: $P \models \text{pass_year}(\text{john})$

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.

Matching

EXAMPLE

Query: `?- likes(U, chris).`

Program: `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).`

see: pt1_s18.pl

Matching

EXAMPLE

$$\forall S \text{ (pass_year}(S) \leftarrow$$

$$\text{pass_exams}(S) \wedge \text{pass_cwks}(S) \wedge \text{pass_projs}(S))$$

Program: `pass_year(S) :-
pass_exams(S), pass_cwks(S), pass_projs(S).`

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

```
pass_cwks(mary).
```

see: pt1 s26.pl

Matching

EXAMPLE

Query: ?- pass_year(john).

Answer: yes which is to say $P \models \text{pass_year}(\text{john})$

Query: ?- pass_year(mary).

Answer: no which is to say not $P \models \text{pass_year}(\text{mary})$

Query: ?- pass_year(X).

Answer: $X = \text{john}$

see: pt1_s26.pl

Trading Example: Program

```
sells(usa, grain, mexico).
```

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

```
produces(oman, oil).
```

% This is a comment

```
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).
```

see: pt1_s32.pl

Trading Example: Queries

Query: ?- produces(oman, oil).

Answer: yes

Query: ?- produces(X, oil).

Answer: X = oman ;

X = iraq ;

no

‘;’ is a request for another answer.

‘no’ after ‘;’ means no more answers.

Query: ?- produces(japan, X).

Answer: X = cameras ;

no

see: pt1_s32.pl

Trading Example: Queries

Query: ?- produces(X,Y).

Answer: X = oman, Y= oil ;
 X = iraq, Y= oil ;
 X = japan, Y= cameras ;
 X = germany, Y= pork ;
 X = france, Y= wine ;
 no

Query: ?- produces(X, rice).

Answer: no

Query: ?- produces(iraq, Y), needs(britain, Y).

Answer: Y = oil ;
 no

see: pt1_s32.pl

Trading Example: Exercises

Write Prolog queries for:

1. Who sells grain to whom?
2. Who sells oil to Britain?
3. Who sells what to Hungary?
4. Does Britain sell oil to the USA?
5. Which two countries have mutual trade with one another?
6. Which two different countries have mutual trade with one another? ($X \neq Z$ means X and Z are different from one another.)
7. Who produces something that is needed by both Britain and Japan?

Write a Prolog clause for:

9. `bilateral_traders(X,Z)` such that X and Z are two different countries that have mutual trade with one another. see: pt1_s32.pl

Work Manager Example

Program: worksIn(bill, sales).
 worksIn(sally, accounts).

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

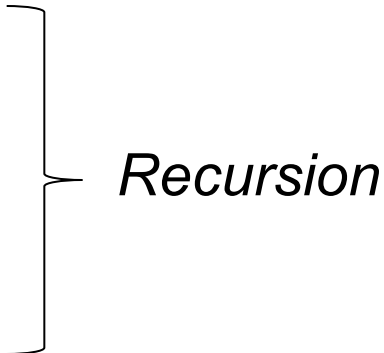
 managerOf(joan, james).
 managerOf(henry, james).
 managerOf(james, paul).
 managerOf(W, M) :-
 worksIn(W, Dept),
 deptManager(Dept, M).

EXERCISE

Define `colleague/2` such that `colleague(W1,W2)` holds see: pt1_s36.pl
if `W1` and `W2` are different colleagues in the same department.

Work Manager Example

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



Recursion

`superiorOf/2` is a recursive predicate.

The first clause for `superiorOf/2` is a *base case*.

The second clause for `superiorOf/2` is *recursive*.

With earlier clauses we get:

Query: `?- superiorOf(bill,paul).`

Answer: `yes`

see: `pt1_s36.pl`

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

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
```

see: pt1_s38.pl

Prolog Constructs

Prolog Syntax: atomic formulas

$p(t_1, \dots, t_n)$ or p

p is the *predicate* or *relation name* of $p(t_1, \dots, t_n)$.

t_1, \dots, t_n are terms.

There must be **NO SPACE** between “ p ” and the “ $($ ”.

Prolog allows the same predicate symbol with different arities:

$p, p(t_1), p(t_1, t_2, t_3)$ etc.

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

Prolog is *not typed*.

Prolog Syntax: clauses

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

A clause has the form:

	$H \text{ :- } C_1, \dots, C_k.$	<i>conditional clause</i>
or	$H.$	<i>unconditional clause</i>

H is the *head* of the clause; C_1, \dots, C_k is the *body*.
($H.$ has an empty body)

A terminating

‘.<space>’,

‘.<newline>’ or

‘.<tab>’

is essential after each clause.

Prolog Syntax: clauses

$H \text{ :- } C_1, \dots, C_k.$
or
 $H.$

H and each C_i is an atomic formula of the form $p(t_1, \dots, t_n)$ or p .

The clause is *about* the predicate of H .

Each C_i in the body is referred to as a *call* or a *condition*.

Prolog Syntax: clauses

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 Syntax: clauses: logical reading

A conditional clause

$$H \text{ :- } C_1, \dots, C_k.$$

is read as

$$\forall X_1 \dots \forall X_m (C_1 \wedge \dots \wedge C_k \rightarrow H)$$

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

$$\forall X_1 \dots \forall X_i (\exists X_{i+1} \dots \exists X_m (C_1 \wedge \dots \wedge C_k) \rightarrow H)$$

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

Prolog Syntax: clauses: logical reading

An unconditional clause

$H.$

is read as

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

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

Prolog Syntax: terms

A term may be:

- Simple (e.g., constants, variables)
- Compound (e.g., functors)

Terms are the items that can appear as the *arguments* of predicates.

Terms containing no variables are said to be *ground*.

They can be viewed as the *basic data* manipulated during execution.

Prolog systems recognise the type of a term from its syntactic form.

Prolog Syntax: simple terms

Constants

- *numbers* (integers, floating)
e.g., 3 5.6 -10 -6.31
- *'atoms'* (Prolog terminology)
 - String of letters, digits and '_', beginning with a *lower case letter* (e.g., apple x2 t_1);
 - String of characters in single quotes (e.g., 'Hello there');
 - String of special characters (e.g., [] <--->).

Prolog Syntax: simple terms

Variables

- Identified as a string of letters, digits and `'_'`, beginning with an *upper case letter* or `'_'`.

e.g., `X Y31 Chris Left_Subtree _35 _person _`

- An anonymous variable is a variable whose value we are not interested in.

`likes(X):- likes(X,_).`

Prolog Syntax: compound terms

Consist of a *functor* (same syntax as atoms) followed by a sequence of one or more sub-terms called *arguments*.

$$f(t_1, \dots, t_n)$$

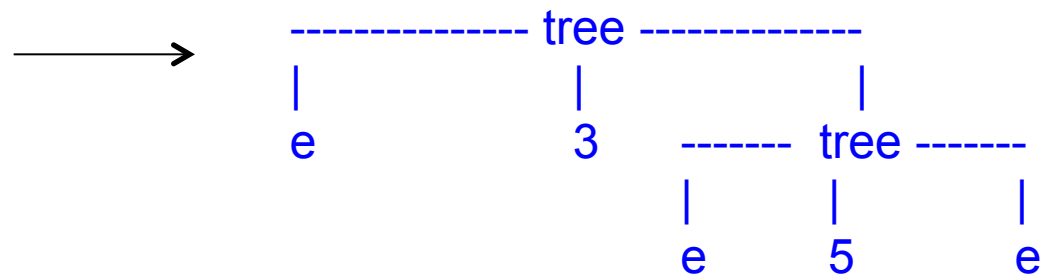
Compound terms can be represented as trees. The functor at the root is called the *principal* functor.

EXAMPLES

`mother(chris)`

`tree(e, 3, tree(e, 5, e))`

`tree(L, R, tree(e, 5, e))`



Compound terms: lists

A sequence of *arbitrary* number of sub-terms, i.e., zero or more.

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

They are represented in Prolog as either:

- The atom `[]` (i.e., the empty list).
- A compound term with functor `'.'` and two arguments (*head* and *tail*). The tail argument must, itself be a list (possibly empty), e.g., `.(3, .(5, []))`.

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

e.g., `[3|[5,X,9]]`

Compound terms: lists

$[]$: empty list

$[H|T]$: a list with first element H followed by the list T .

H is called the *head* and 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 .

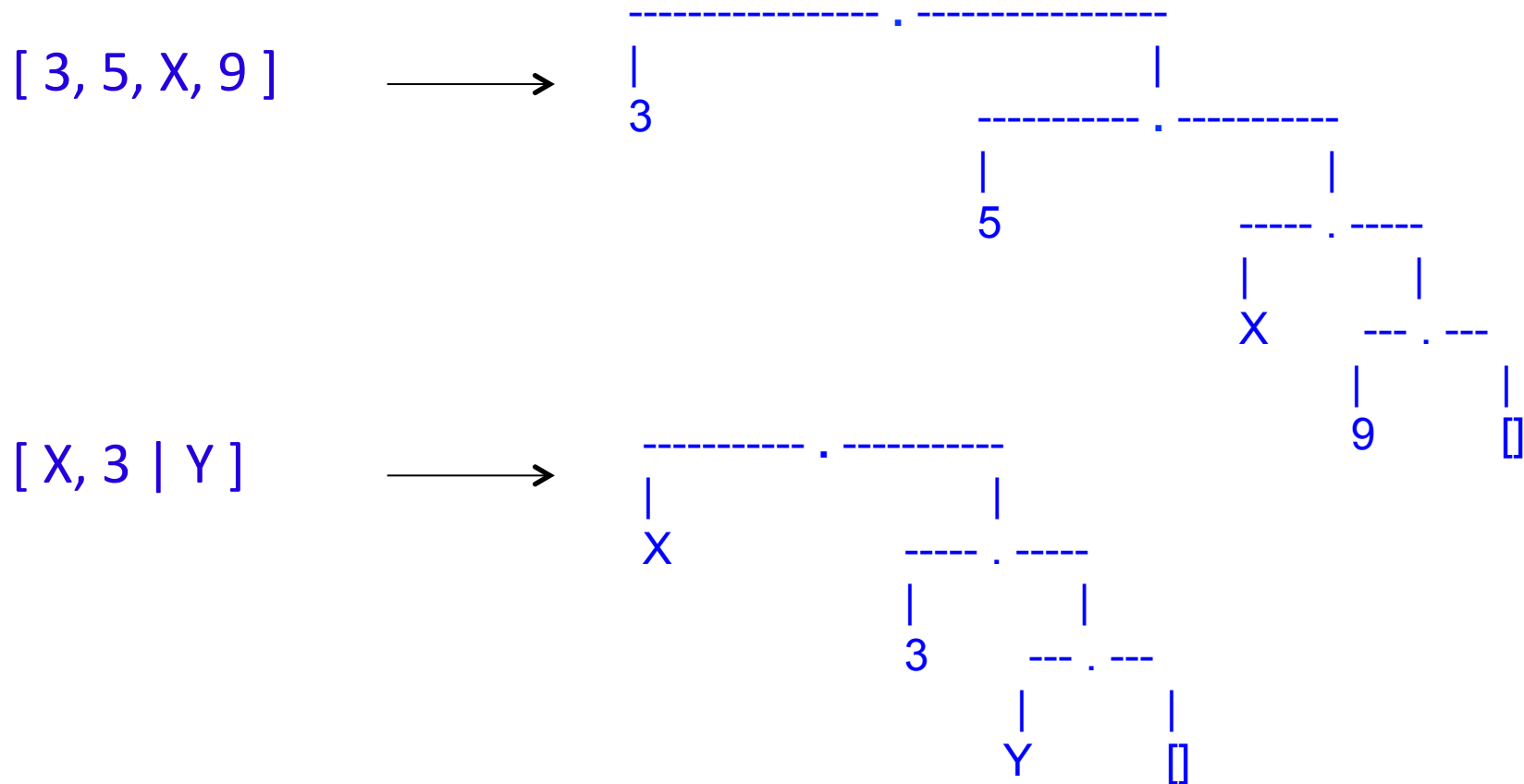
This is equivalent to $[H1|[H2|T]]$

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

$[l_1,l_2,l_3]$ is a shorthand for $[l_1|[l_2|[l_3|[]]]]$.

Compound terms: lists

Lists form a subclass of *binary trees* (right-branching binary trees).



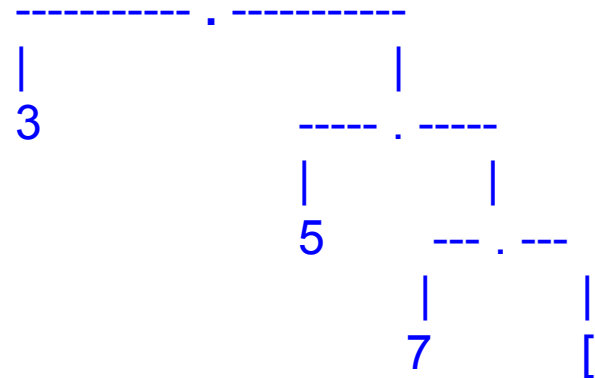
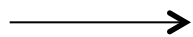
Compound terms: lists

Lists form a subclass of *binary trees* (right-branching binary trees).

$[3 \mid [5, 7]]$

$= [3, 5 \mid [7]]$

$= [3, 5, 7]$

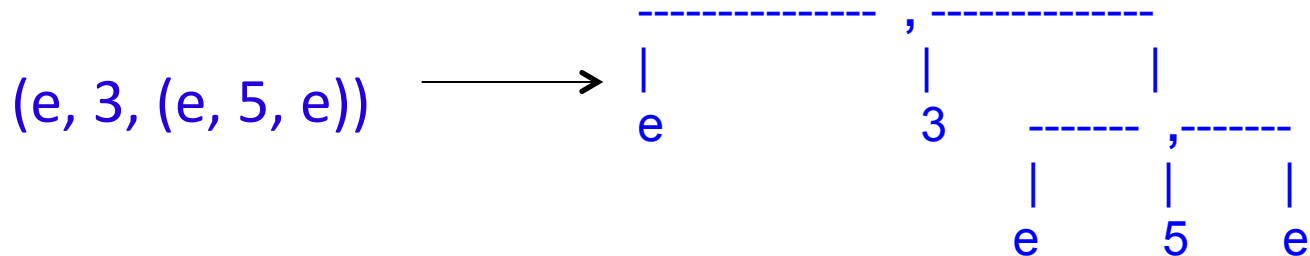


Compound terms: tuples

A sequence of *two* or *more* sub-terms.

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

Represented in Prolog as a compound term with functor *'* and two arguments.



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

Consider prolog representation of (2,3) and [2,3].

Compound terms: arithmetic

$3 * X + 5$ $\sin(X + Y) / (\cos(X) + \cos(Y))$ $20 > 7$

Treated as a functor with arguments.

$*$, $+$, etc. are called *infix* operators and are treated as functors with arguments.

The operators with highest precedence is the principal functor.

The above is a shorthand for

$*(3, +(X, 5))$ $/(\sin(+(X, Y)), +(\cos(X), \cos(Y))$ $>(20, 7)$

NOTE that terms representing numerical expressions are *not* automatically evaluated, like in a functional language.

Prolog Syntax: 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).

Prolog Computation

Prolog Evaluation

To evaluate a query means to try to satisfy it.

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.

Deterministic Evaluations: Example

Program: `all_bs([]).`

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

This program defines a list in which every member is `b`.

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

This will generate a deterministic evaluation:

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

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

`?- all_bs([b]).`

`?- all_bs([]).`

`?-.`

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

Answer: `yes`

see: `pt1_s59.pl`

Deterministic Evaluations: Example

Program: `all_bs([]).`

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

Now consider the *query*:

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

This will generate a deterministic evaluation:

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

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

which fails (no clause with matching head).

Answer: `no`

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

see: `pt1_s59.pl`

Deterministic Evaluations: Example

Program: `all_bs([]).`

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

Query: `?- all_bs([b, X, b]).`

Evaluation:

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

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

`?- all_bs([]).`

see: `pt1_s59.pl`

`?- .` % success

So here the search tree comprises *one* branch (computation), which happens to succeed with an answer substitution `X/b`.

Deterministic Evaluations: Example 2

Prolog supplies the list-concatenation built-in predicate `append(X, Y, Z)`.

If it did not, then we could define our own:

Program: `app([], Z, Z).`

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

Now consider the *query*

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

The call matches the head of the second program clause by making the bindings

see: `pt1_s62.pl`

`U / a X / [b] Y / [c, d] L / [a | Z]`

Deterministic Evaluations: Example 2

So, we replace the call by the body of the clause, then apply the bindings just made to produce the *derived query*:

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

Another similar step binds `Z / [b | Z2]` and 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]`.

see: pt1_s62.pl

Deterministic Evaluations: Example 2

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

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 binding $Y / [c, d]$ was propagated, and the final output binding is $L / [a, b, c, d]$.

Deterministic Evaluations

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.

Non-Deterministic Evaluations

A Prolog evaluation is *non-deterministic* (i.e., contains more than one computation) when some call unifies with several clause-heads.

When this is so, the search tree will have *several branches*.

Non-Deterministic Evaluations: Example

Program:

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

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

see: pt1_s67.pl

Non-Deterministic Evaluations: Example

*These points are called
choice points.*

Program:

a :- b, c.

a :- f.

b.

b :- g.

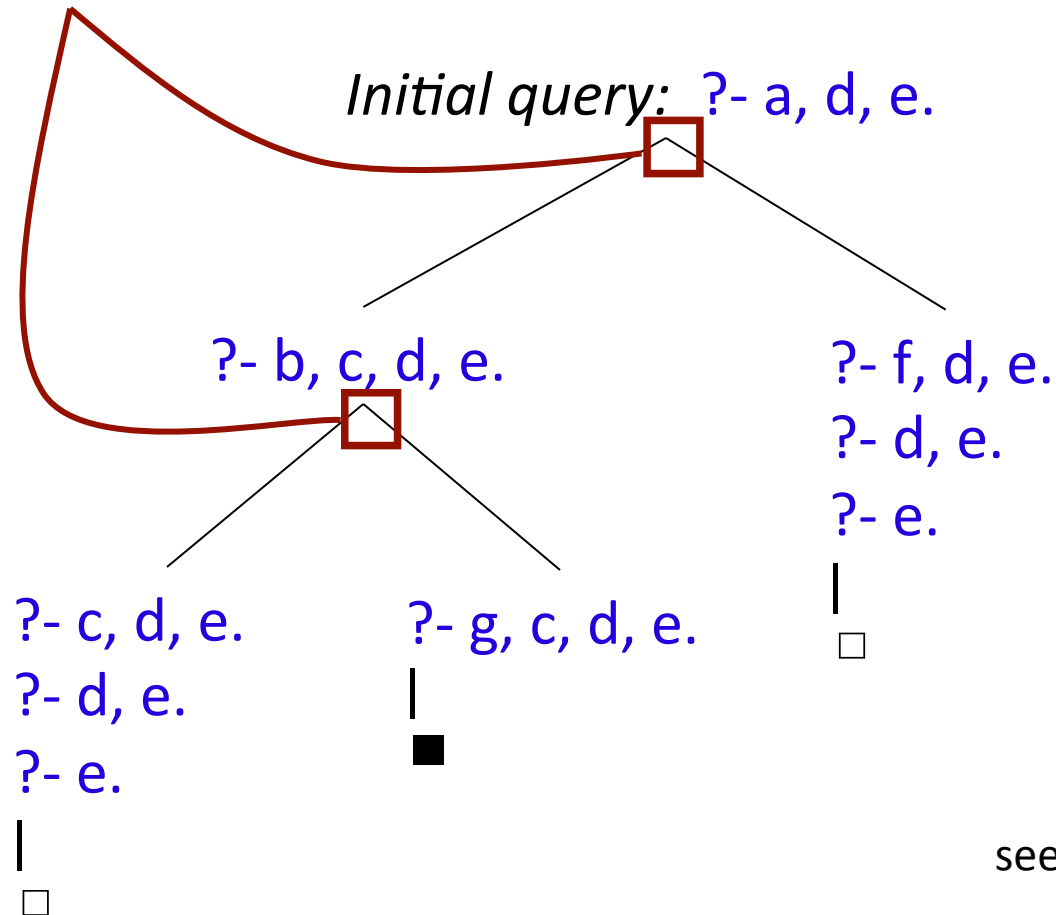
c.

d.

e.

f.

Initial query: ?- a, d, e.



see: pt1_s67.pl

Non-Deterministic Evaluations

Prolog generates computations *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.

Non-Deterministic Evaluations

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*.

Computation Efficiency

Prolog computation of a query Q is a complete depth-first traversal of the specific search tree of Q obtained by always choosing the left most goal.

The efficiency with which Prolog solves a problem depends upon

- the order of the clauses in the program
- the ordering of calls

When the search tree for a query has an infinite branch, the order of the clauses can determine whether a solution is given at all.

Computation Efficiency: Example

Change the earlier query and program to:

Program:

$a :- c, b.$ % originally $a :- b, c.$

a :- f.

b.

b :- g.

C.

d.

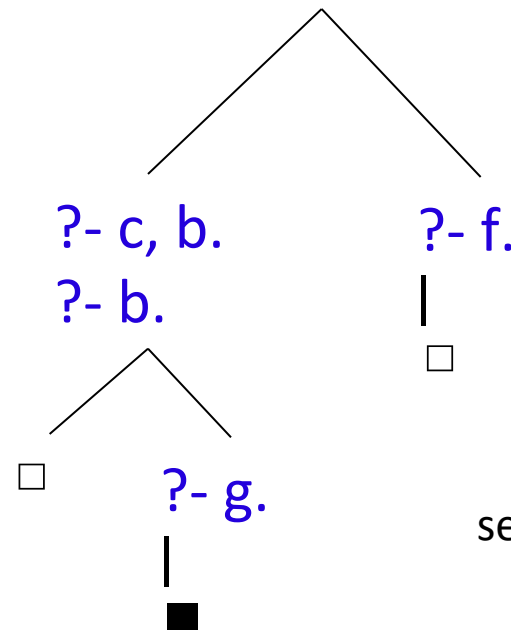
e.

f.

Initial query: ?- d, e, a.

?- e, a.

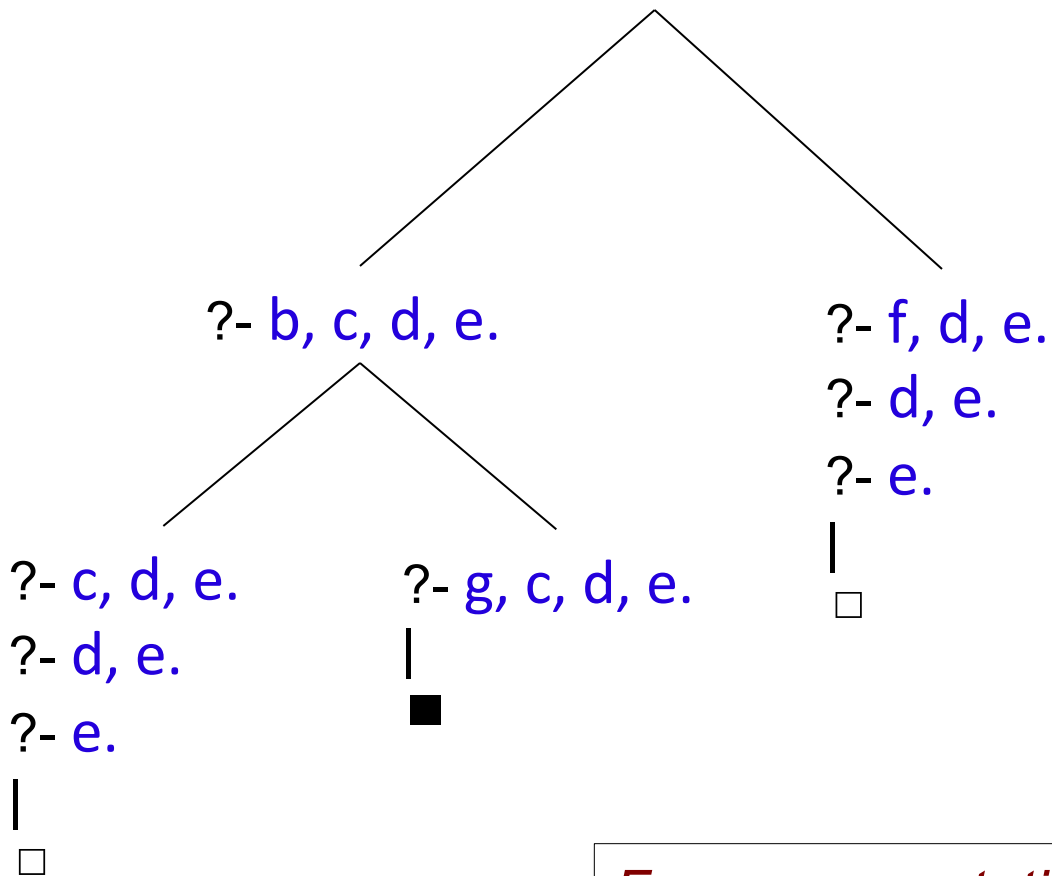
?- a.



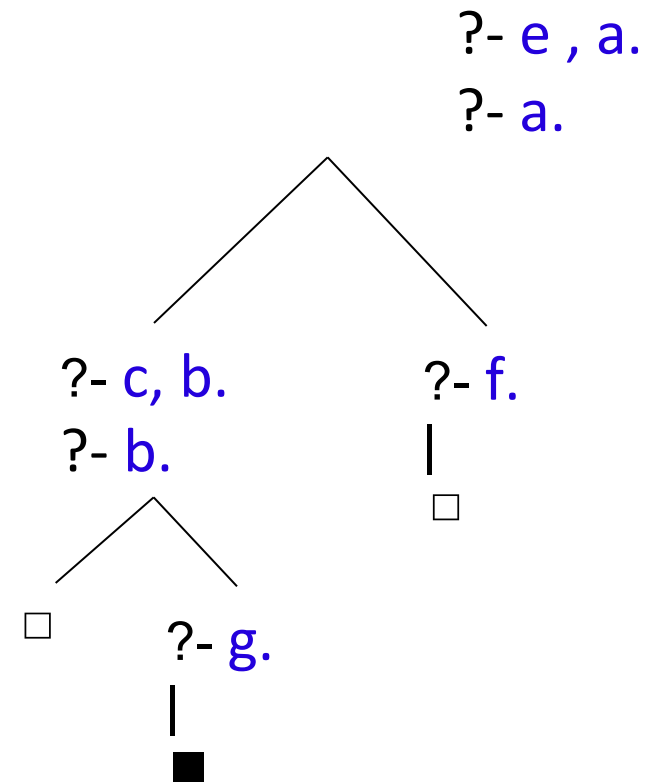
see: pt1_s72.pl

Computation Efficiency: Example

Original query: ?- a, d, e.



Modified query: ?- d, e, a.



Fewer computation steps!

Computation Efficiency

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

A *search rule*, on the other hand, decides which program clause to apply to the selected call.

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 (i.e., matching).

The call has to be *unified* with the head.

Two predicates are *unifiable* if and only if they have a *common instance*.

A solution to a unification problem is a *substitution*.

Unification: Example

Program: likes(bob, X) :- likes(X, logic).

Query: ?- likes(Y, chris).

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

If E is any atomic formula, then $E\theta$ 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.

see: pt1_s17.pl

Unification

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

In other words, one that commits the variables to the least possible extent.

EXAMPLE

$h(X, a, Y)$ and $h(b, Z, W)$ unify via

$$\theta = \{ X / b, Z / a, Y / W \} \quad (\text{or } W / Y)$$

Both become $h(b, a, W)$.

$p(X, Y, h(Y))$ and $p(Z, a, Z)$ unify via

$$\theta = \{ X / h(a), Y / a, Z / h(a) \}$$

Both become $p(h(a), a, h(a))$.

Unification

Prolog provides built-in predicate `=/2`, which succeeds if its two arguments unify and fails if they do not.

Unification is a symmetric operation ($X=Y$ is the same as $Y=X$), and is not the same as assignment.

EXERCISE

Find out more about unification by entering queries using `=/2`.

Try:

?- `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.)

Unification during computation

Query: ?- p(args1), others.

Program: p(args2) :- body.

If θ exists such that $p(\text{args1})\theta = p(\text{args2})\theta$ then this clause can be used by this call to produce.

Derived query: ?- body θ , others θ .

otherwise, this clause cannot be used by this call.

Unification during computation: Exercise

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

Along the successful computation we have

$O1 = \{ X / [a \mid X1] \}$	<i>these are the</i>
$O2 = \{ X1 / [b \mid X2] \}$	<i>output bindings</i>
$O3 = \{ X2 / [] \}$	<i>in the unifiers</i>

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

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

Applying this to the initial query gives the *answer*

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

see: pt1_s62.pl

Unification during computation: Exercise

Unifying with gives the substitution

$[X, Y, Z]$ $[a, b, c]$

$[X|Y]$ $[a, b, c]$

$[X|Y]$ $[a]$

$[X, Y|Z]$ $[a, b, c]$

$[X|Y]$ $[[1, 2], [3, 4]]$

$[[X|Y]|Z]$ $[[1, 2], 3]$

$[X, Y]$ $[1, [2, 3]]$

$[X, Y]$ $[1]$

(Check your answers using Prolog's unification primitive $=/2$)

AI Problem Solving – Sudoku

	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

[Back to slides](#)