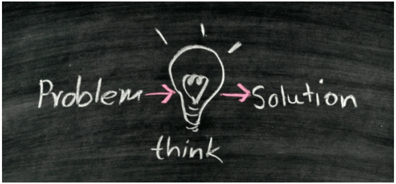
Programming in Prolog

Search Space, Unification, Recursion

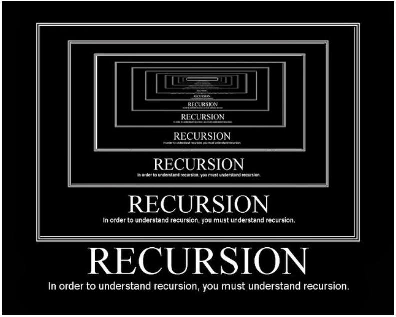
Claudia Schulz

Logic and AI Programming
(Course 518)

What you will learn in this lecture



How does Prolog generate answers?



How does recursion work in Prolog?

How does Prolog generate answers?

A query with various calls

```
?- person(ada), quit(bob), quit(B), B \= bob
```

- try to prove `person(ada)`, then try to prove `quit(bob)`, ...
  $\Rightarrow$ try to prove every call **from left to right**
  $\Rightarrow$ depth-first search
- try to match (**unify**) current call with head of a clause, then replace it with body of chosen clause
  $\Rightarrow$ try clauses **from top to bottom**
- call succeeds if it is unified with a fact or it evaluates to true (`5 = 5`); fails otherwise
- query succeeds ("yes") if every call succeeds; fails otherwise ("no")

Unification

match a call in a query to the head of a clause

- easy for constants, e.g. `sunny`
- variables can be matched with other variables or constants, e.g. `X` matches `sunny`
- what about compound terms?
  e.g. `father(person(X), john)`

Prolog terms

- constant: starts with lower case letter – `sunny`
- number: 5
- variable: starts with upper case letter – `Sunny`
- compound term: function name(term 1, ..., term n) –
  `father(person(X), john)`
  note: this has nothing to do with functions in mathematics!

Unification Rules

two terms unify (=) if and only if

1. two constants/numbers: if and only if they are the same
   - `bill = bill`
   - `7 = 7`
   - `bill \= 7`
   - note: `'bill'= bill`
   - note: `'7'\= 7`

Unification Rules

two terms unify (=) if and only if

2. a constant/number and a variable: always unify
   ⇒ variable is instantiated with constant/number
   - `X = 7` X instantiated with 7
   - `bill = X` X instantiated with `bill`
3. two variables: always unify
   ⇒ variables are considered the same, i.e. have same value
   - `X = Y`
     X and Y are the same: `X = _154`, `Y = _154`

⇒ instantiation of variables is sometimes called **variable binding**

Unification Rules

two terms unify (=) if and only if

4. two compound terms: if and only if
   1. same function name
   2. same number of arguments
   3. all corresponding arguments unify
   4. variable instantiations are compatible

   ⇒ variables are instantiated with unified constants/numbers

   - `k(X,p) = k(Y,Mp)`
     instantiation: `X = Y`, `Mp = p`
   - `k(X,p) = k(f(l,p),Mp)`
     instantiation: `X = f(l,p)`, `Mp = p`
   - `k(X,p) \= k(f(l,p),l)`
   - `k(X,p) \= k(f(l,p),Mp,Y)`
   - `k(X,p,m(Y)) \= k(t(Z), Z, X)`
   - `k(X,p,t(Y)) = k(t(Z), Z, X)`

Unification - Try it yourself

Do these terms unify? If so, what is the instantiation of the variables?

- `m(X,Y)` and `p(Y,X)`
- `mia(X)` and `'mia'(f(a))`
- `t(X,Y)` and `t(Y,Z)`
- `p(1,Y,f(a))` and `p(X,m,Z)`
- `k(X,m(Y))` and `k(p,X)`
- `k(X,m(Y))` and `k(m(5),X)`
- `s(X,Y)` and `s(Y,f(X))`

= versus ==

- = Do two terms unify?
- == Are two terms identical?

- `a = 'a'` `a == 'a'`
- `X = a` `X \== a`
- `X = Y` `X \== Y`
- `X = Y, X == Y`

⇒ if `term1 == term2` then `term1 = term2`

Note the difference to the `is` and the `=:=` (`=\=`) predicates

- `X is 5+7` `X =\= 5+7` `X = 5+7` `X \== 5+7`
- `12 is 5+7` `12 =:= 5+7` `12 \= 5+7` `12 \== 5+7`

How Prolog Searches

$cl_1 : p(bob).$
$cl_2 : p(X).$
$cl_3 : q(X) :- u(X), p(X).$
$cl_4 : q(bob).$
$cl_5 : u(ada).$
$cl_6 : u(sam)$

Query:
$p(ada), q(bob), q(B),$
$B \setminus= bob$

Solution:

$? - p(ada), q(bob), q(B), B \setminus= bob$

cl_2

$(ada = \_123)$

$? - q(bob), q(B), B \setminus= bob$ Choice Point

cl_3

$(bob = \_436)$

$? - u(bob), p(bob), q(B), B \setminus= bob$

no applicable clause

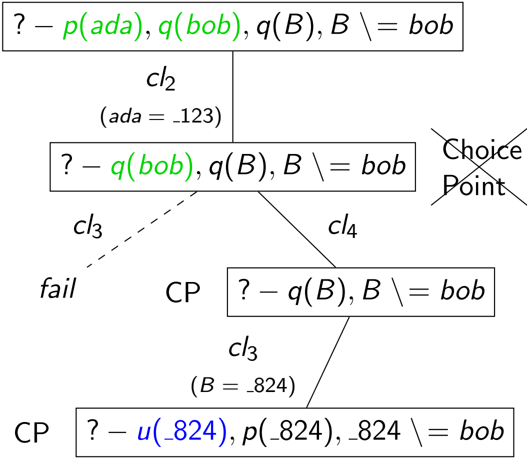
fail

How Prolog Searches

$cl_1 : p(bob).$
$cl_2 : p(X).$
$cl_3 : q(X) :- u(X), p(X).$
$cl_4 : q(bob).$
$cl_5 : u(ada).$
$cl_6 : u(sam)$

Query:
$p(ada), q(bob), q(B),$
$B \backslash = bob$

Solution:

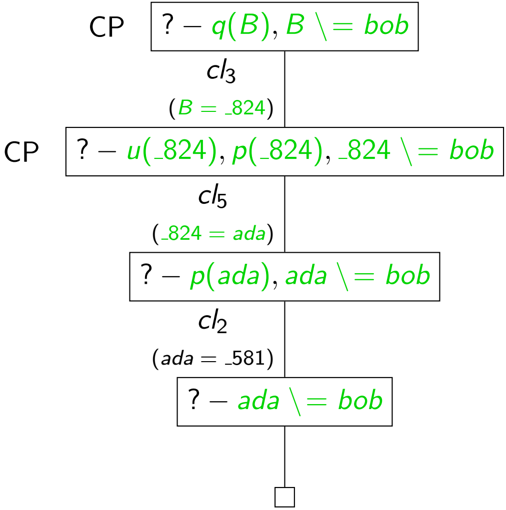


How Prolog Searches

$cl_1 : p(bob).$
$cl_2 : p(X).$
$cl_3 : q(X) :- u(X), p(X).$
$cl_4 : q(bob).$
$cl_5 : u(ada).$
$cl_6 : u(sam)$

Query:
$p(ada), q(bob), q(B),$
$B \setminus= bob$

Solution:

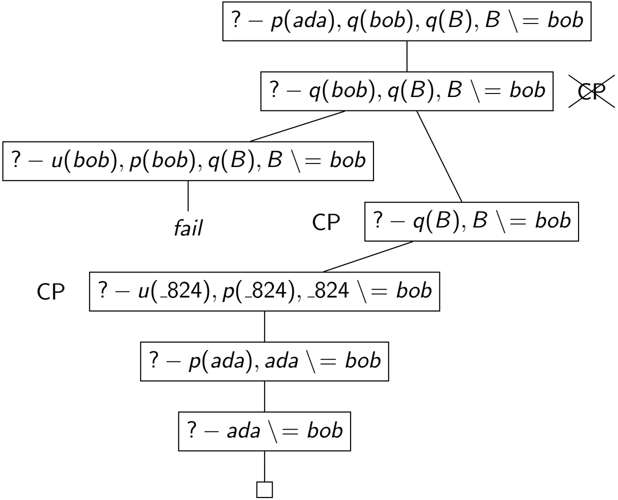


The whole search tree

$cl_1 : p(bob).$
$cl_2 : p(X).$
$cl_3 : q(X) :$
$-u(X), p(X).$
$cl_4 : q(bob).$
$cl_5 : u(ada).$
$cl_6 : u(sam)$

Query:
$p(ada), q(bob),$
$q(B), B \backslash = bob$

Solution:
$\mathbf{B = ada}$



Choice Points

Are there any other solutions to the query?

Prolog: Are there any choice points left that might lead to further solutions?

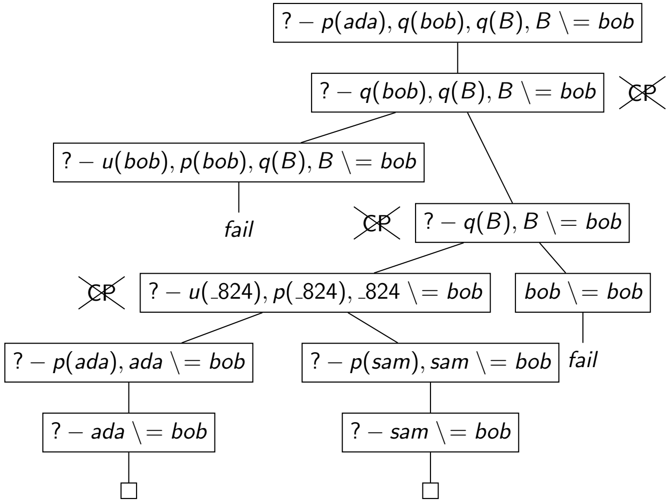
- starting from the last choice point

Choice Points

$cl_1 : p(bob).$
$cl_2 : p(X).$
$cl_3 : q(X) :$
$-u(X), p(X).$
$cl_4 : q(bob).$
$cl_5 : u(ada).$
$cl_6 : u(sam)$

Query:
$p(ada), q(bob),$
$q(B), B \backslash = bob$

Solution:
B = ada
B = sam
No more
solutions



trace

To see how this looks in Prolog:

```
trace.
  ⋮
notrace.
```

Summary – How does Prolog generate answers?

A general query Q

```
?- C1 ..., Cn
```

1. try to prove first call `C1`
   - by unifying it with next matching head of a clause (top to bottom)
2. if unification fails - go back to last choice point and retry
3. else substitute C_1 in Q with conditions in body of the clause
4. repeat 1.) and 2.) until
   - no goals left in query – solution found
     $\Rightarrow$ return all variable bindings
   - goals left in query & no choice points left to retry unification – query fails/cannot be proven
     $\Rightarrow$ return `no`

Summary – How does Prolog generate answers?

Remember: **order matters**

- Order of calls in a query – first one first
- Order of clauses in a program – top one first
- Order of conditions in a clause – first one first

What is recursion?

Most powerful technique for programming in Prolog!
(in particular for working with lists – more on that later)

$\Rightarrow$ If you understand recursion, you can program in Prolog

The **idea** of recursion:

- similar to while/for loop: repeat a procedure until some lower/upper bound is reached
- in contrast to while/for loop: a whole function calls itself repeatedly
- in Prolog: a predicate calls itself
  $\Rightarrow$ one of the conditions in a clause refers to the same predicate as the head of this clause

A recursive predicate

```
my_predicate_name(X,Y) :-
    check_first(X),
    do_second(Y,Z),
    my_predicate_name(X,Z).
```

Ancestor Example

```
is_ancestor_of(Ancestor, Person) :-
    human(Person),
    human(Ancestor),
    is_parent_of(Parent, Person),
    is_ancestor_of(Ancestor, Parent).
```

The Ancestor Example corrected

Ancestor Example

```
is_ancestor_of(Parent, Person) :-
    is_parent_of(Parent, Person).

is_ancestor_of(Ancestor, Person) :-
    is_parent_of(Parent, Person),
    human(Person),
    is_ancestor_of(Ancestor, Parent),
    human(Ancestor).
```

Base case & recursive definition

Don't forget the **base case**
$\Rightarrow$ Prolog won't find the correct solutions or even loop forever

Defining a recursive predicate

1. **base case** – most basic case (most basic arguments) which is not recursive; terminates the recursion
2. **recursive definition** – one of the conditions is the predicate itself

Order matters – especially for recursion

Natural Number Example

Tail1:

```
natural_no(0).
natural_no(X) :-
    Y is X-1,
    natural_no(Y).
```

NoTail1:

```
natural_no(0).
natural_no(X) :-
    natural_no(Y),
    Y is X-1.
```

Tail2:

```
natural_no(0).
natural_no(X) :-
    X is Y+1,
    natural_no(Y).
```

NoTail2:

```
natural_no(0).
natural_no(X) :-
    natural_no(Y),
    X is Y+1.
```

Order matters – especially for recursion

	test	generate	reversed, test 0
Tail1	✓	✗	loop
Tail2	✗	✗	no
NoTail1	✗	✗	loop
NoTail2	✓	✓	loop

- base case (usually) before recursive definition
- think about whether you want to test or generate (or both)
- **tail recursion** is more efficient!
  $\Rightarrow$ but it's not always possible to use it

Prolog versus Logic

The previous examples have the same **declarative (logical) meaning** but a different **procedural meaning**
⇒ different behaviour

```
p :- p
```

- declarative meaning: "If `p` holds then `p` holds"
- in logic: $p \rightarrow p$
  ⇒ a tautology – it's always true
- procedural meaning: "To prove `p` you must prove `p`"
  ⇒ `?- p` loops forever

So: Prolog is not a full logic programming language! (not fully declarative)

What you should know now

- How does Prolog generate answers?
  - What is unification and how does it work?
  - Why does the order of Prolog clauses matter?
  - What is a choice point?
  - How can `trace` be used for debugging?

- How does recursion work in Prolog?
  - Why is the base case important?
  - What is tail recursion?
  - What is the difference between the declarative and the procedural meaning of a recursive program?

Useful reading/resources

Introductory Book:

- “Learn Prolog Now!” Blackburn, Bos, Striegnitz
  $\Rightarrow$ A free online version is also available

Prolog Manual

- HTML: `https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/`
- PDF: `https://sicstus.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf`