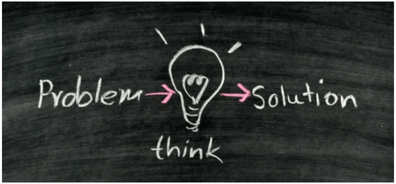
Programming in Prolog

Search Space, Unification, Recursion

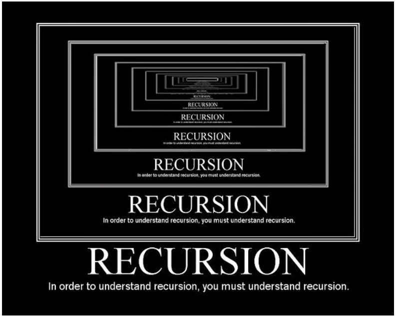
Claudia Schulz

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 (conditions)

$?{-}p(5), q(3), q(B), B \setminus= 3.$

1. prove every call in a query
   $\Rightarrow$ **from left to right**
   $\Rightarrow$ depth-first search
2. **unify** current call and first matching (head of a) clause and replace current call with conditions of chosen clause
   $\Rightarrow$ **from top to bottom**
3. succeeds if the whole query can be proven
4. fails if all possible choices of clauses and variable bindings fail (for at least one goal in a query)

Unification

?- $C_1, \ldots, C_n$.<newline>

If it contains variables, the query is a request for a substitution (a set of term values) θ for the variables of the query such each of the conditions:

$C_1\theta, \ldots, C_n\theta$

is a logical consequence of the program clauses, or for a confirmation that there is no such θ.

$C_i\theta$ is C_i with any variable in C_i (given a value in θ) replaced by its assigned value.

If there are no vars in query, then the query is a request for a report on whether or not the query, as given, is a logical consequence of the program clauses.

unification = **substitution** of variables or **match** of same term

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

$\Rightarrow$ 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

   $\Rightarrow$ 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/2 predicate

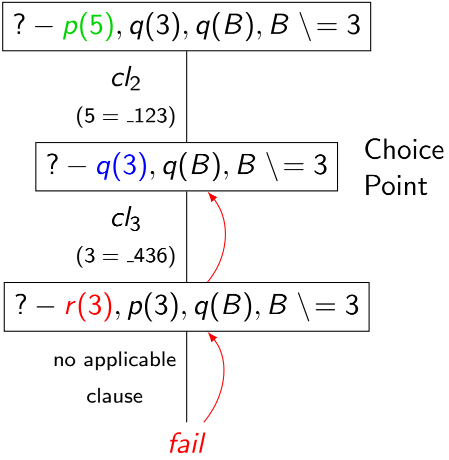
- `X is 5+7`   `X = 5+7`   `X \== 5+7`
- `12 is 5+7`   `12 \= 5+7`   `12 \== 5+7`

How Prolog Searches

$cl_1 : p(3).$
$cl_2 : p(X).$
$cl_3 : q(X) :- r(X), p(X).$
$cl_4 : q(3).$
$cl_5 : r(5).$
$cl_6 : r(2)$

Query:
$p(5), q(3), q(B), B \backslash= 3$

Solution:

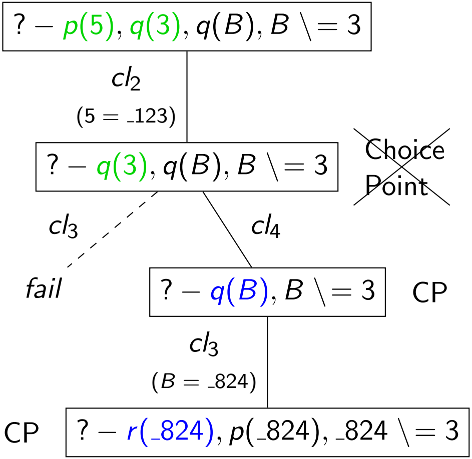


How Prolog Searches

$cl_1 : p(3).$
$cl_2 : p(X).$
$cl_3 : q(X) :- r(X), p(X).$
$cl_4 : q(3).$
$cl_5 : r(5).$
$cl_6 : r(2)$

Query:
$p(5), q(3), q(B), B \backslash= 3$

Solution:

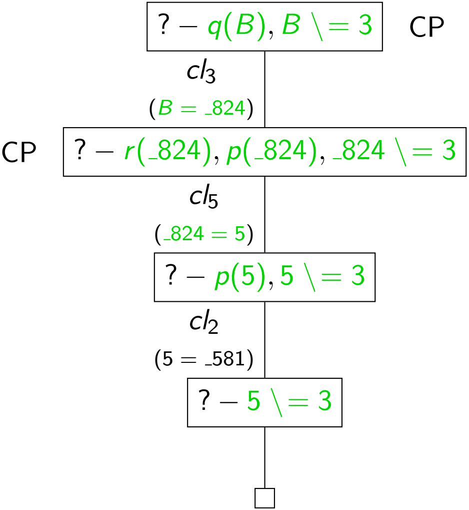


How Prolog Searches

$cl_1 : p(3).$
$cl_2 : p(X).$
$cl_3 : q(X) :- r(X), p(X).$
$cl_4 : q(3).$
$cl_5 : r(5).$
$cl_6 : r(2)$

Query:
$p(5), q(3), q(B), B \backslash= 3$

Solution:

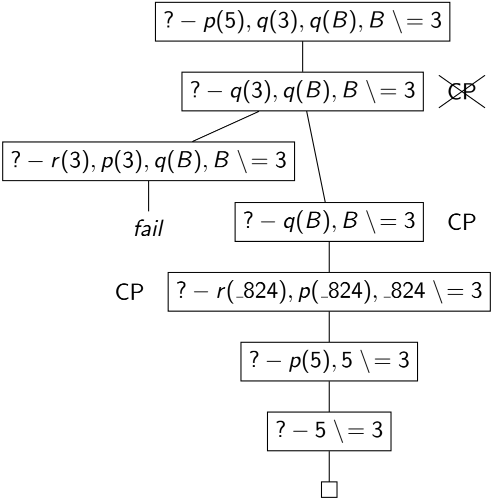


The whole search tree

$cl_1 : p(3).$
$cl_2 : p(X).$
$cl_3 : q(X) :- r(X), p(X).$
$cl_4 : q(3).$
$cl_5 : r(5).$
$cl_6 : r(2)$

Query:
$p(5), q(3), q(B), B \backslash= 3$

Solution:
$\mathbf{B = 5}$



Choice Points

Are there any other solutions to the query?

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

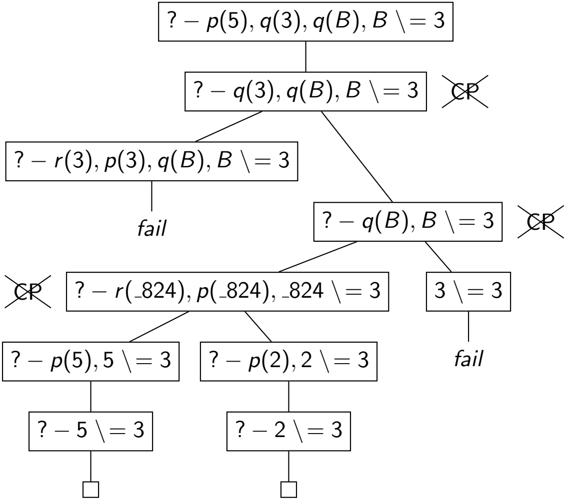
- starting from the last choice point

Choice Points

$cl_1 : p(3).$
$cl_2 : p(X).$
$cl_3 : q(X) :- r(X), p(X).$
$cl_4 : q(3).$
$cl_5 : r(5).$
$cl_6 : r(2)$

Query:
$p(5), q(3), q(B), B \setminus= 3$

Solution:
$\mathbf{B = 5}$
$\mathbf{B = 2}$
No more solutions



`trace`

To see how this looks in Prolog:

```
trace.
  ⋮
notrace.
```

Summary – How does Prolog generate answers?

A general query Q

$?- C_1, \ldots, C_n$

1. unify first goal C_1 with next matching head of a clause *cl*
   - from top to bottom
   - if unification fails - go back to last choice point and start procedure from there
     $\Rightarrow$ unification of C_1 or choice point "higher up"
2. substitute C_1 in Q with conditions in body of the clause *cl*
3. repeat 2.) and 3.) 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 goals 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) :-
    human(Person),
    human(Ancestor),
    is_parent_of(Parent, Person),
    is_ancestor_of(Ancestor, Parent).
```

Base case & recursive definition

Don't forget the **base case**
⇒ 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**
$\Rightarrow$ different behaviour

```
p :- p
```

- declarative meaning: “If `p` holds then `p` holds”
- in logic: $p \to p$
  $\Rightarrow$ a tautology – it's always true
- procedural meaning: “To prove `p` you must prove `p`”
  $\Rightarrow$ `?- p` returns `no`

So: Prolog is not a full logic programming language!

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`