

Prolog

Expressive Power: Logic programming, extensions.

Programming Methodology:

- Data as relations (unit clauses) of F.O.L.
Code as WFFs of F.O.L.
- Can give multiple answers
- No static type system
- No modularity
- Two phases to a program: **consult** data and program, **solve** a goal
- Answers are terms that unify with variables.

Implementation:

- Unification plus backtracking
- Efficiency concerns
 - Horn clause subset of logic
 - No occurs check in most systems
- Operational semantics dictates a particular search strategy

1

Prolog

Prolog programs in two parts:

- database of “rules” (Definite clauses)
each of form “infer conclusion from premises”
Premises may be empty — axioms (Unit clauses)
- “query” against database (Goal clause)

```
mercury.cs.uml.edu> pl
1 ?- consult(user).
|: man(socrates).
|: mortal(X) :- man(X).
|: ^D
% user://1 compiled 0.00 sec, 504 bytes
```

Yes

```
2 ?- mortal(socrates).
```

Yes

```
3 ?-
```

More usually

```
consult('my_program.pl').
abbreviated
['my_program.pl'].
```

2

Prolog syntax versus logic

Horn clauses:

Definite clause	$I_1 \vee \neg I_2 \vee \dots \vee \neg I_n$
A.k.a. “rule”	I_1 is called the “head” of the rule
Implication	$(I_2 \wedge \dots \wedge I_n) \rightarrow I_1$
Prolog	<code>l1 :- l2, ... ,ln.</code>
example	<code>mortal(X) :- man(X).</code>
Unit clause	I_1
A.k.a. “fact”	
Implication	$\rightarrow I_1$
Prolog	<code>l1.</code>
example	<code>man(socrates).</code>
Goal clause	$(\neg I_1 \vee \dots \vee \neg I_n)$
A.k.a. “query”	
Implication	$(I_1 \wedge \dots \wedge I_n) \rightarrow$
Prolog	<code>?- l1, ... , ln.</code>
example	<code>?- mortal(socrates).</code>

3

Prolog

Prolog’s answer to sources of non-determinism in logic programming

Q: What order do I try clauses in?

A: The order in which they were typed in.

Q: How do I solve a conjunction (and) of literals?

A: Left to right (with backtracking).

Q: How do I instantiate variables?

A: Use unification algorithm **without occurs check**.
(Most dialects).

4

Prolog

More Syntax

Comments: % to end of line or /* to */

Function and predicate symbols lowercase, variables uppercase.

Logic variables: are unified, maybe only partially instantiated. Instantiation of variables in goal printed in response to query.

`_` Underscore is “don’t care” – never instantiated!

`_Variable` is variable that you don’t want displayed in answer.

;`;` Semicolon is “solve again”.

Documentation style

```
%%% name (term_name_1, ..., term_name_n)
%%% description of terms
%%% intended direction of use
```

Some use just name/arity but more likely in reference that in documentation.

Directions: + expected instantiated. - expected uninstantiated. ? no expectation.

Reasoning with Prolog

```
%%% anc (Old, Young)
%%% =====
%%%
%%%      Old is a proper ancestor of Young.
%%%      This method is left-recursive
%%%      and wildly buggy.
```

```
anc(Old, Young) :-
    anc(Old, Mid),
    anc(Mid, Young).
anc(Old, Young) :-
    parent(Old, Young).
```

```
%%% parent(Parent, Child)
%%% =====
%%%
%%%      Parent is a parent of Child
```

```
parent(katherine, bertrand).
parent(amberley, bertrand).
parent(bertrand, kate).
parent(bertrand, john).
parent(bertrand, conrad).
```

Transcript

```
1 ?- [ancestor1].
% ancestor1 compiled 0.00 sec, 1,284 bytes

Yes
2 ?- anc(amberly,kate).
ERROR: Out of local stack
      Exception: (31,484) anc(amberly, _G306) ? a
% Execution Aborted
```

5

6

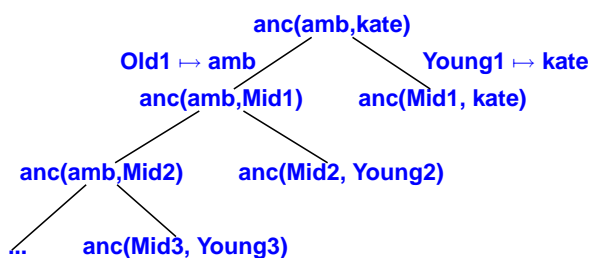
Left-recursive Rules and Base Cases

```
anc(Old, Young) :-
    anc(Old, Mid),
    anc(Mid, Young).
anc(Old, Young) :-
    parent(Old, Young).
```

Trace

```
[trace] 21 ?- anc(amberley, kate).
Call: (6) anc(amberley, kate) ?
Call: (7) anc(amberley, _G493) ?
Call: (8) anc(amberley, _G493) ?
Call: (9) anc(amberley, _G493) ?
Call: (10) anc(amberley, _G493) ?
Call: (11) anc(amberley, _G493) ?
Call: (12) anc(amberley, _G493) ?
```

Proof tree



7

Base cases

```
anc(Old, Young) :-
    parent(Old, Young).
anc(Old, Young) :-
    anc(Old, Mid),
    anc(Mid, Young).
```

`anc(amberly,kate).` works

`anc(kate,amberly).` loops – kate is not parent of anyone.

Idea

Do not write left-recursive rules before non-left-recursive rules.

Try to get rid of left recursion as much as possible.

8

Fixed

```
%%% anc2 (Old, Young)
%%% =====
%%%
%%% Old is a proper ancestor of Young. This method i
%%% properly defined.
```

```
anc2(Old, Young) :-
    parent(Old, Young).
anc2(Old, Young) :-
    parent(Old, Mid),
    anc2(Mid, Young).
```

Transcript

```
3 ?- [ancestor2].
% ancestor2 compiled 0.00 sec, 660 bytes
```

```
Yes
4 ?- anc2(amberley, kate).
```

```
Yes
5 ?- anc2(Ancestor, kate).
```

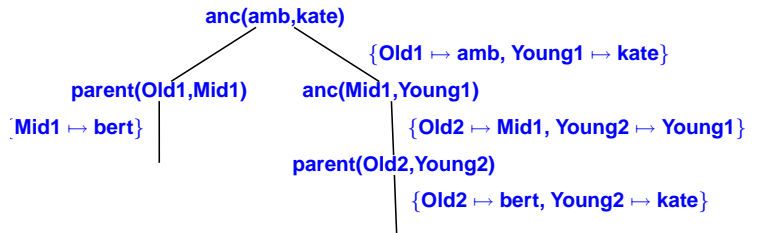
```
Ancestor = bertrand ;
Ancestor = katherine ;
Ancestor = amberley ;
```

```
No
```

Proof tree

```
anc2(Old, Young) :-
    parent(Old, Young).
anc2(Old, Young) :-
    parent(Old, Mid),
    anc2(Mid, Young).

[trace] 20 ?- anc2(amberley, kate).
Call: (6) anc2(amberley, kate) ?
Call: (7) parent(amberley, kate) ?
Fail: (7) parent(amberley, kate) ?
Redo: (6) anc2(amberley, kate) ?
Call: (7) parent(amberley, _G499) ?
Exit: (7) parent(amberley, bertrand) ?
Call: (7) anc2(bertrand, kate) ?
Call: (8) parent(bertrand, kate) ?
Exit: (8) parent(bertrand, kate) ?
Exit: (7) anc2(bertrand, kate) ?
Exit: (6) anc2(amberley, kate) ?
```



9

10

Lists

. is cons. [] is empty list.

```
%%% conc(Left, Right, LR)
%%% =====
%%%
%%% LR is concatenation of lists Left and Right,
%%% in that order.
```

```
conc([], List, List).
conc(.(Element, Rest), List, .(Element, LongRest)) :-
    conc(Rest, List, LongRest).
```

Built-in list notation

```
%%% conc(Left, Right, LR)
%%% =====
%%%
%%% LR is concatenation of lists Left and Right,
%%% in that order.
```

```
conc([], List, List).
conc([Element|Rest], List, [Element|LongRest]) :-
    conc(Rest, List, LongRest).
```

List notation [e1, e2 ..., en | rest-of-list]

List concatenation is a built-in operation: append/3

Lists continued

```
2 ?- conc([a,b], [c,d], Result).
```

```
Result = [a, b, c, d]
```

```
Yes
3 ?- conc(L, R, [a,b,c,d]).
```

```
L = []
R = [a, b, c, d] ;
```

```
L = [a]
R = [b, c, d] ;
```

```
L = [a, b]
R = [c, d] ;
```

```
L = [a, b, c]
R = [d] ;
```

```
L = [a, b, c, d]
R = [] ;
```

```
No
4 ?-
```

11

12

Terms as Data Structures

Construction create compound value from subcomponents

Selection select subcomponent of compound value

Predication is this value in specified form?

```
(define conc
  (lambda (list1 list2)
    (if (not (pair? list1)) list2      ; predication
        (cons                          ; construction
          (car list1)                  ; selection
          (conc                          ; selection
            (cdr list1)
            list2))))))
```

Predication: if not pair clause selection.

Construction: unify with uninstantiated:

[Element|LongRest]

Selection: unify with parts of instantiated:

[Element|Rest]

```
conc([], List, List).
conc([Element|Rest], List, [Element|LongRest]) :-
  conc(Rest, List, LongRest).
```

Predication: unify instantiated goal with instantiated head.

Construction: unify uninstantiated goal with instantiated head.

Selection: unify instantiated goal with uninstantiated head.

13

Example: Permutation Sort

(Ramsey & Kamin, page 460.

```
%%%
%%% permutation_sort(Unsorted,Sorted) (+,-)
%%%
permutation_sort(Unsorted, Sorted) :-
    permutation(Unsorted, Sorted), % generate
    sorted(Sorted).                % and test

%%%
%%% sorted(List_of_Numbers) (+)
%%%
sorted([]).
sorted([A]).
sorted([A, B|L]) :- A <= B, sorted([B|L]).

%%%
%%% permutation(One_permutation, Another_permutation)
%%% (+,-). If used (-,+), must give equal (known) length
%%% lists to avoid infinite loop after first result.
permutation([], []).
permutation(L, [H|T]) :-
    % H is an element of L. append used (-,-,+).
    append(L2, [H|U], L),
    % L3 is L with H removed. append used (+,+,-)
    append(L2, U, L3),
    % T is a permutation of L3
    permutation(L3, T).
```

14

Example: Merge Sort

(Periera & Shieber, page 51.)

```
merge(A, [], A).
merge([], B, B).
merge([A|RestAs], [B|RestBs], [A|Merged]) :-
    A < B, merge(RestAs, [B|RestBs], Merged).
merge([A|RestAs], [B|RestBs], [B|Merged]) :-
    B =< A, merge([A|RestAs], RestBs, Merged).
```

Note: merge([],[],[]). succeeds twice!

```
mergesort([], []).
mergesort([A], [A]).
mergesort([A,B|Rest], Sorted) :-
    split([A,B|Rest], L1, L2),
    mergesort(L1, SortedL1),
    mergesort(L2, SortedL2),
    merge(SortedL1, SortedL2, Sorted).
```

```
split([], [], []).
split([A], [A], []).
split([A,B|Rest], [A|RestA], [B|RestB]) :-
    split(Rest, RestA, RestB).
```

15

Incomplete Data: Difference Lists

Sequence 1,2,3 is difference between

[1,2,3,4,5] and [4,5], [1,2,3,8] and [8]

[1,2,3] and [], generally [1,2,3|X] and X.

diff(X, X) – empty list.

diff([1,2,3|Y], Y) – the list [1,2,3]

Consing still easy:

```
cons(X, diff(A,B), diff([X|A],B)).
```

Concatenation is easy!

```
diffappend(diff(L,X), diff(X,Y), diff(L, Y)).
```

“(L-X) + (X-Y) = (L-Y)”

Bi-directional conversion:

```
simplify(diff(X,X), []).
simplify(diff([X|Y], Z), [X|W]) :-
    simplify(diff(Y,Z), W).
```

General difference structures a Prolog programming idiom.

16

Not so fast!

```

1 ?- [user].
|: simplify(diff(X, X), []).
|: simplify(diff([X|Y], Z), [X|W]) :-
    simplify(diff(Y, Z), W).
|:
% user://1 compiled 0.00 sec, 516 bytes

Yes
2 ?- simplify(diff([3, 4|X], X), L).

X = [3, 4, 3, 4, 3, 4, 3, 4, 3|...]
L = [] ;

X = [4, 4, 4, 4, 4, 4, 4, 4, 4|...]
L = [3] ;

X = _G343
L = [3, 4] ;

X = [_G434, _G434, _G434, _G434, _G434, _G434, _G434, _G434, _G434]
L = [3, 4, _G434] ;

```

What went wrong? No occurs check in Prolog (occurs check in μ Prolog).

17

Occurs check 2

```

1 ?- [user].
|: simplify(diff(X, Y), []) :-
    unify_with_occurs_check(X,Y).
|: simplify(diff([X|Y], Z), [X|W]) :-
    simplify(diff(Y, Z), W).

2 ?- simplify(diff([3,4|X],X), L).
X = _G331
L = [3, 4] ;

But asking for more answers goes off into a loop as
simplify tries increasingly long lists for the second
parameter to diff. Lets try to cut down the search
space:

3 ?- [user].
|: simplify(diff(X, Y), []) :-
    unify_with_occurs_check(X,Y), !.
|: simplify(diff([X|Y], Z), [X|W]) :-
    simplify(diff(Y, Z), W).

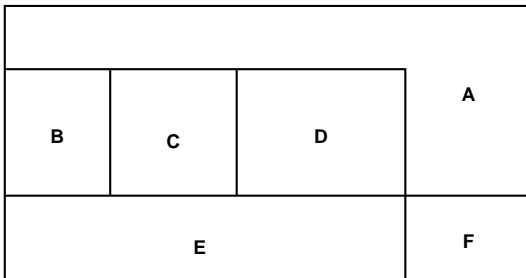
4 ?- simplify(diff([3,4|X],X), L).
X = _G508
L = [3, 4] ;
No

5 ?- simplify(diff(X,Y),[3,4]).
X = [3, 4|_G310]
Y = _G310 ;
No

```

18

Example: Map Coloring



Database for map coloring

```

different(yellow,blue).
different(blue,yellow).
different(yellow,red).
different(red,yellow).
different(blue,red).
different(red,blue).

coloring(A,B,C,D,E,F) :-
    different(A,B),
    different(A,C),
    different(A,D),
    different(A,F),
    different(B,C),
    different(B,E),
    different(C,E),
    different(C,D),
    different(D,E),
    different(E,F).

```

19

Another take on map coloring

Create rules that color any map given by query

Environments:

```

get(X,[assign(X,Y)|_T],Y).
get(X,[_|T],Y) :- get(X,T,Y).

```

Colorings: (A an environment binding colors to regions)
coloring OK if different colors for adjacent regions

```

color(_A,[]).
color(A,[adj(_X,[])|R]) :- color(A,R).
color(A,[adj(X,[Y|T])|R]) :-
    get(X,A,Xc),get(Y,A,Yc),different(Xc,Yc),
    color(A,[adj(X,T)|R]).

```

Assign colors to map regions — one color per region:

```

assignment([],[]).
assignment([assign(R,_)|S],[adj(R,_)|T]) :-
    assignment(S,T).

```

Search for assignment that colors correctly:

```

coloring(A,M) :- assignment(A,M), color(A,M).

```

Query is:

```

?- coloring(A,[adj(a,[b,c,d,f]),
               adj(b,[a,c,e]),
               adj(c,[a,b,d,e]),
               adj(d,[a,c,e]),
               adj(e,[b,c,d,f]),
               adj(f,[a,e])]).

```

```

A = [assign(a, yellow), assign(b, blue), ...
Yes

```

20

Traditional notions

Scoping of **variable names** is **within rule only**

- each rule has its own name space for variables

Scoping of **constructor names** is **across entire database**

Prolog is essentially untyped

(but name + arity can provide a weak notion of “type”)

Implementing Prolog—unification

Try to satisfy query by **unifying** it with some conclusion

- predicate/constructor/functor names must be identical
- number of arguments must be identical
- find substitution unifying arguments
Capitalized Names are variables

$a(b, c, d, E)$
with $x(\dots)$ **doesn't unify: a and x differ**

$a(b, c, d, E)$
 $a(-, -, -)$ **no: different # of args**

$a(b, c, d, E)$
 $a(b, f, G, H)$ **yes: by either $\{C \mapsto f, G \mapsto d, H \mapsto E\}$
or $\{C \mapsto f, G \mapsto d, E \mapsto H\}$**

$a(\text{pred}(X, j))$
 $a(B)$ **yes: $\{B \mapsto \text{pred}(X, j)\}$**

$a(\text{pred}(X, j))$
 $a(X)$ **problems $\{X \mapsto \text{pred}(X, j)\}$ in Prolog
(rejected by “occurs check” in μ Prolog)**

21

22

Implementing Prolog—backtracking

How to find the “right” conclusion? **Backtracking search**

To satisfy a goal:

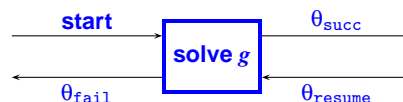
- Try to **unify with conclusion of first rule** in database
- if **successful**, apply substitution to **first premise**, try to satisfy resulting subgoals
- then apply both substitutions to **next subgoal** (premise), and so on...
- if **not successful**, go on to the **next rule** in database
- if all rules fail, try again (backtrack) to a previous subgoal

Substitutions accumulate, much as in type inference

See “Byrd box” for details of control flow

Sketch of backtracking

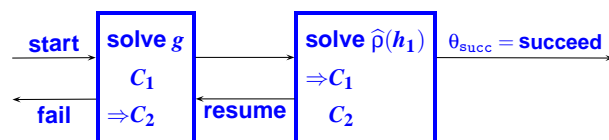
Revenge of CPS—the Byrd box:



Each Byrd box **tries every clause in sequence**

- may try all clauses and **fail**
- may find a good clause and **succeed**
- will try next clause if **resumed** after backtracking

Are strung together to solve subgoals:



Byrd box is

- Excellent **conceptual model**
- A very **simple implementation** (if not the most efficient)

23

24

Implementing Byrd Boxes using CPS

```
fun query database = let
  fun solveOne (g as (pred, args)) succ fail =
    find(pred, builtins) args succ fail
  handle NotFound _ => let
    fun search [] = fail ()
      | search (clause :: clauses) =
        let fun resume() = search clauses
            val G :- Hs = freshen clause
            val theta = unify (g, G)
        in solveMany (map (lift theta) Hs)
            theta succ resume
        end
      handle Unify => search clauses
  in search (potentialMatches (g, database))
  end
and solveMany [] 0 succ fail = succ 0 fail
  | solveMany (g::gs) theta succ fail =
    solveOne g
    (fn theta' => fn resume =>
      solveMany (map (lift theta') gs)
        (theta' o theta) succ resume)
    fail
in fn gs => solveMany gs (fn x => x)
end
```

25

Some Metalogical Facilities

Metalogical subset of extralogical facilities that deal with modifying logic programs.

Examples:

- `call/1` treat a term as a goal formula.
- `!` (cut) control backtracking.
- `\+` negation-as-failure.

```
?- conc([a,b], [c,d], Result).
Result=[a,b,c,d] ;
no
```

```
?- call( conc([a,b], [c,d], Result) ).
Result=[a,b,c,d] ;
no
```

inside a call, `:-`, `,` are function symbols!

`call` allows a goal to be a variable, so long as it is instantiated by the time that `call` literal is executed.

27

Metalogical and Extralogical Facilities

`print(X)` prints its argument, always succeeds

```
Z is X + Y succeeds if integers X+Y = Z
Z is X - Y succeeds if numbers X-Y = Z
```

- `X` and `Y` must be instantiated as numbers (prevents infinite backtrack)
- also supports `*` and `/`

`X < Y` succeeds if integers `X<Y`

- `X` and `Y` must be numbers (prevents infinite backtrack)
- also supports `>`, `<=`, and `>=`

...and many more...

Extralogical = Not logic programming, but useful extensions.

26

The Cut

A way of aborting in mid-backtrack: `G :- H, !, I.`

Backtracking can pass `!` in forward direction

- but in backward direction, entire goal `G` fails! (do NOT check more rules for `G`)

Prune search space (solutions to left, all clauses below)

A **green cut** adds efficiency, does not remove possible answers if the predicate that it is in is used in the documented fashion.

A **red cut** removes possible answers.

Example — inequality:

```
equal(X,X).
not_equal(X, Y) :- equal(X, Y), !, fail.
not_equal(X, Y).
```

Can use `\+` (not) as abbreviation for this idiom

```
not(Goal) :- call(Goal), !, fail.
not(Goal).
```

What is not? Not false but unprovable from known clauses.

closed universe assumption.

28

General uses of cut

If you found the right rule, cut out later ones

Example: sum integers up to N (2nd arg is sum)

```
sum_to(1,1) :- !.
sum_to(N, Answer) :-
    N1 is N - 1, sum_to(N1, A), Answer is A+N
```

Cut avoids infinite loop if sum_to(1, 1) in failing supergoal

```
ok :- sum_to(1, X), more(foo).
```

terminates even if more(foo) fails

Without cut:

```
sum_to(1,1) X = 1, ... backtrack
sum_to(1,X) N = 1, X = Answer
:- 0 is 1 - 1, sum_to(0,A1), (X is A1+1)
    N2 = 0, A1 = Answer2
:- -1 is 0 - 1, sum_to(-1,A2), (A1 is A2+0)
    N3 = -1. A2 = Answer3
:- ...
```

29

Another Cut

Restrict library facilities for overdue borrowers:

```
service(Patron, Fac) :-
    overdue(Patron, Book),
    !, % red cut limiting service
    basic_service(Fac).
service(Patron, Fac) :-
    any_service(Fac).
```

```
basic_service(reference).
basic_service(enquiries).
```

```
extra_service(borrowing).
extra_service(interlibrary_loan).
extra_service(audio_visual).
```

```
any_service(F) :- basic_service(F).
any_service(F) :- extra_service(F).
```

30

Metacircular Interpreter in Prolog

call/1 takes formula as term, evaluates it.

First metacircular interpreter:

```
prove(G) := call(G).
```

Second metacircular interpreter for logical Prolog:

- create predicate clause/1 “This term is a clause”
- simplify language by getting rid of unit clauses, except builtin literal true.

```
clause(( conc([], List, List) :- true ))).
clause(( conc([E|Rest], List, [E|LongRest]) :-
    conc(Rest, List, LongRest) ))).
```

- the , symbol is right-associative

```
prove(true).
prove(Goal) :-
    clause(( Goal :- Body )),
    prove(Body).
prove((Body1, Body2)) :-
    prove(Body1),
    prove(Body2).
```

31

Prolog in a nutshell

Prolog

- values are terms—and so is abstract syntax:

```
datatype term = VAR      of string
              | LITERAL of int
              | APPLY   of string * term list
```

```
variables = upper case
“functors” = lower case
```

- is untyped (everything is a term)
- has no data abstraction
- has no functional abstraction!
- has no mutable state
- has no explicit control flow.

Programs are declarative:

```
goal      = string * term list
clause    = goal :- goal list
database  = clause list
query     = goal list
```

- has an evaluation model based on backtracking and unification

32