Prolog

Variously:

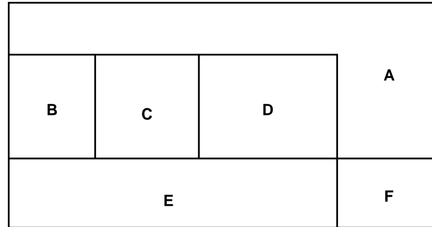
- logic programming
- declarative programming
- unification with backtracking

Roots in "Horn Clauses" of first-order logic

Prolog programs in two parts:

- database of "rules"
  each of form "infer conclusion from premises"
  (premises may be empty — axioms)
- "query" against database

Reasoning with Prolog



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).
```


Another take on map coloring

Create rules that color *any* map given by query

Environments:

```
get(X,[assign(X,Y)|_],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(C,_)|R],[adj(C,_)|T]) :-
  assignment(R,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])]).
```


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 unusual evaluation model based on backtracking and unification

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
- unification is like ML pattern matching,
  except the "value" can have variables
  and Prolog patterns are non-linear!

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

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")

Unification examples

`a(b,C,d,E)` with `x( ... )`	doesn't unify: `a` and `x` differ
`a(b,C,d,E)` `a(_,_,_)`	no: different # of args
`a(b,C,d,E)` `a(j,f,G,H)`	no: `b` $\neq$ `j`
`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(pred(X,j))` `a(pred(k,j))`	yes: $\{X \mapsto k\}$
`a(pred(X,j))` `a(B)`	yes: $\{B \mapsto pred(X,j)\}$
`a(pred(X,j))` `a(X)`	no! — no infinite trees allowed

(rejected by "occurs check", sometimes unimplemented)
(but `X`'s may differ because of scoping)

Simulating functions

No functions, only relations

- "function result" becomes extra argument to predicate

Example: `append (X, Y, Z)` ≡ "`Z` is the result of appending `X` and `Y`"

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X,Y,Z).
```

Difference lists:

```
contents(A, dl(L,E)) :- append(A, E, L).
```

Representation invariant: `E` is a suffix of `L`
Abstraction function: given by `contents` rule

More on Difference Lists

`dl([a,b|E], E)` is a "difference list" containing `a` and `b`.

`E` could be anything—more difference lists containing `a` and `b`.

```
dl([a,b], [])
dl([a,b,c], [c])
dl([a,b,c|F], [c|F])
```

Consing on to the front of a difference list:

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

Consing on to the end of a difference list:

```
consend(X,dl(A,[X|B]), dl(A,B)).
```

Extracting the contents of a difference list:

```
contents(X, dl(L,E)) :- append(X,E,L).
```

Appending lists without recursion!

```
diffappend(dl(L,X), dl(X,Y), dl(L, Y))
```

"$(L-X)+(X-Y)=(L-Y)$"

Built-in Predicates

`print(X, Y, Z, ...)` prints its arguments, always succeeds

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

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

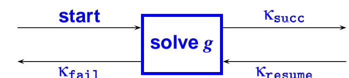
`X < Y` succeeds if integers `X<Y = Z`

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

...and in real Prolog, many more...

Sketch of backtracking

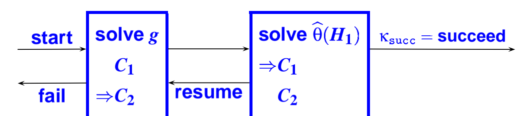
Revenge of CPS—the Byrd box:



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

Can be strung together to solve multiple goals at once:



Byrd box is

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

Implementing Byrd Boxes using CPS

```
fun query database = let
  fun solveOne (g as (func, args)) succ fail =
   find(func, 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 [] θ succ fail = succ θ 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
```


Running Prolog

Try μProlog:

```
~cs152/bin/uprolog
```

Or Stony Brook Prolog

```
~cs152/bin/xsb -i
```

To load a file `my_file.P` into memory:

```
[my_file].
```

or to type a database in interactively:

```
[user].
```

For XSB, this interactive stream needs to be terminated by `^D`

Also you can type queries in directly at the

```
?-
```

prompt.

Logic or Procedure?

My opinion:

- anyone who tells you "Prolog is logic" is lying

Why?

- real Prolog programs have to worry about infinite loops
- the "cut"

Both are significant real problems
and both depend only on "order of evaluation"
(that is, a particular proof strategy)

The Cut

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

Backtracking can pass ! in forward direction

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

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

... a trap for the unwary
BUT recommended anyway

`not` is easier to understand than general cut

Another Cut

Restrict library facilities for overdue borrowers:

```
service(Patron, Fac) :- overdue(Patron, Book),
                        !, 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).
```


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

To fail without trying alternatives:

```
..., !, fail.
```

Stop generating alternatives

Example: Tic-Tac-Toe

Two players, `x` and `o`:

```
otherplayer(x, o).
otherplayer(o, x).
```

Board has 9 squares:

```
square(1). square(2). square(3). square(4).
square(5). square(6). square(7). square(8).
square(9).
```

Win is three in a row across, down, or diagonally:

```
in_row(1, 2, 3). in_row(1, 4, 7). in_row(1, 5, 9).
in_row(4, 5, 6). in_row(2, 5, 8). in_row(3, 5, 7).
in_row(7, 8, 9). in_row(3, 6, 9).
```

Need access to lists by element number:

```
nth(0, [X|_], X).
nth(I, [_|Y], Z) :- I1 is I - 1, nth(I1, Y, Z).
```

Can also change lists by element number:

```
update([_|Y], 0, P, [P|Y]).
update([X|Y], I, P, [X|Z]) :-
  I1 is I - 1, update(Y, I1, P, Z).
```


Tic-Tac-Toe board and moves

Board "configuration" is player to move, plus squares:

```
to_move(B, P) :- nth(0, B, P).
```

Initial board has designated player, empty squares:

```
initial([P|S], P) :- initial_squares(S, 9).
initial_squares([], 0) :- !.
initial_squares([none|X], N) :-
  N1 is N - 1, initial_squares(X, N1).
```

Board `B` is a win for player `P` if player `P` has 3 in a row
(note implicit search!)

```
won_for(B, P) :-
  in_row(X, Y, Z),
  nth(X, B, P),
  nth(Y, B, P),
  nth(Z, B, P).
```

On board `B`, a move to square `S` leads to board `BB`:

```
move(B, S, BB) :-
  square(S),
  nth(S, B, none),
  to_move(B, P),
  update(B, S, P, BBB),
  otherplayer(P, Q),
  update(BBB, 0, Q, BB).
```


Tic-Tac-Toe predictions

Game of complete info has 1 of 3 outcomes for player to move:

```
forecast(B, won)  :- forced_win(B).
forecast(B, lost) :- lost_position(B).
forecast(B, tied) :- guaranteed_tie(B).
```

Player can force a win if

- player has already won, or
- player hasn't lost, and can put opponent in lost position

```
forced_win(B) :- to_move(B, P), won_for(B, P).
forced_win(B) :-
  to_move(B, P), not(lost_for(B, P)),
  move(B, _, BB), lost_position(BB).
```

Need to know losses and ties:

```
lost_for(B, P) :-
   otherplayer(P, Q), won_for(B, Q).
tied(B) :- won_for(B, _), !, fail.
tied(B) :- move(B, _, _), !, fail.
tied(B).
```


Predictions, continued

Can avoid loss if already tied, or opponent can't force win

```
saveable(B) :- tied(B).
saveable(B) :- move(B, _,BB), not(forced_win(BB)).

lost_position(B) :- to_move(B, P), lost_for(B, P).
lost_position(B) :- not(saveable(B)).

guaranteed_tie(B) :-
  not(forced_win(B)), not(lost_position(B)).
```


Power of Prolog

Pattern matching (with untyped constructors!)

Backtracking

Unification

Moreover

- rules, goals are also data!
  - `assert`, `retract` modify database
  - `call(P)` treats `P` as a goal!

The cut is needed but can be very hard to understand

The logical model is cute but it's a chimera

- must understand evaluation model

Very cute language as a "thought experiment."

People do use it for real searching problems, also for AI problems

- but more unusual than otherwise