Background

- 1972, University of Aix-Marseille
- Original goal: Natural language processing
- At first, just an interpreter written in Algol
  - Compiler created at Univ. of Edinburgh
Logic Programming

- At a high level, logic programs model the relationship between “objects”
  1. Programmer specifies relationships at a high level
  2. Language builds a database
  3. Programmer then queries this database
  4. Language searches for answers
Features of Prolog

- **Declarative**
  - Specify what goals you want to prove, not how to prove them (mostly)
- **Rule based**
- **Dynamically typed**
- **Several built-in datatypes**
  - Lists, numbers, records, … but no functions
- **Several other logic programming languages**
  - Datalog is simpler; CLP and λProlog more feature-ful
  - Erlang borrows some features from Prolog

A Small Prolog Program – Things to Notice

Use /* */ for comments, or % for 1-liners

```
/* A small Prolog program */
female(alice).
male(bob).
male(charlie).
father(bob, charlie).
mother(alice, charlie).

% "X is a son of Y"
son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).
```

- Lowercase logically terminates
- Program consists of facts and rules
- Uppercase denotes variables
- Period ends statements
Running Prolog (Interactive Mode)
Navigating location and loading program at top level

?- working_directory(C, C).
  C = 'c:/windows/system32/'.

?- working_directory(C, 'c:/Users/me/desktop/p6').
  C = 'c:/Users/me/desktop/'.

?- ['01-basics.pl'].
  % 01-basics.pl compiled 0.00 sec, 17 clauses
  true.

?- make.
  true.

Load file 01-basics.pl

Running Prolog (Interactive Mode)
Listing rules and entering queries at top level

?- listing(son).
  son(X, Y) :-
    father(Y, X),
    male(X).
  son(X, Y) :-
    mother(Y, X),
    male(X).
  true.

?- son(X,Y),
  X = charlie,
  Y = bob;
  X = charlie,
  Y = alice.

List rules for son

User types ; to request additional answer

Multiple answers

User types return to complete request
Style

One predicate per line

blond(X) :-
  father(Father, X),
  blond(Father), % father is blond
  mother(Mother, X),
  blond(Mother). % and mother is blond

Descriptive variable names

Inline comments with % can be useful

Outline

- Syntax, terms, examples
- Unification
- Arithmetic / evaluation
- Programming conventions
- Goal evaluation
  - Search tree, clause tree
- Lists
- Built-in operators
- Cut, negation
Prolog Syntax and Terminology

Terms
- **Atoms**: begin with a lowercase letter
  - horse, underscores_ok, numbers2
- **Numbers**: 123, -234, -12e-4
- **Variables**: begin with uppercase or _, “don’t care” variables
  - X, Biggest_Animal, _the_biggest1
- **Compound terms**: functor(arguments)
  - bigger(horse, duck)
  - bigger(X, duck)
  - f(a, g(X, _), Y, _)

No blank spaces between functor and (arguments)

Prolog Syntax and Terminology (cont.)

Clauses
- **Facts**: define predicates, terminated by a period
  - bigger(horse, duck).
  - bigger(duck, gnat).

  Intuitively: “this particular relationship is true”
- **Rules**: Head :- Body
  - is_bigger(X,Y) :- bigger(X,Y).
  - is_bigger(X,Y) :- bigger(X,Z), is_bigger(Z,Y).

  Intuitively: “Head if Body”, or “Head is true if each of the subgoals can be shown to be true”

A program is a sequence of clauses
Queries

- To “run a program” is to submit queries to the interpreter
- Same structure as the body of a rule
  - Predicates separated by commas, ended with a period
- Prolog tries to determine whether or not the predicates are true

?- is_bigger(horse, duck).
?- is_bigger(horse, X).

“What does there exist a substitution for X such that is_bigger(horse, X)?”

Unification – The Sine Qua Non of Prolog

Without which, nothing

Two terms unify if and only if

- They are identical
  
  ?- gnat = gnat.
  
  true.

- They can be made identical by substituting variables
  
  ?- is_bigger(X, gnat) = is_bigger(horse, gnat).
  
  X = horse. This is the substitution: what X must be
  
  for the two terms to be identical.

  ?- pred(X, 2, 2) = pred(1, Y, X)
  
  false

  Sometimes there are multiple possible substitutions; Prolog can
  
  be asked to enumerate them all.

  ?- pred(X, 2, 2) = pred(1, Y, _)
  
  X = 1,
  
  Y = 2.
The = Operator

- For unification (matching)
  - `?- 9 = 9.`
    true.
  - `?- 7 + 2 = 9.`
    false.
- Why? Because these terms do not match
  - 7+2 is a compound term (e.g., +(7,2))
- Prolog does not evaluate either side of =
  - Before trying to match

The Is Operator

- For arithmetic operations
- “LHS is RHS”
  - First evaluate the RHS (and RHS only!) to value V
  - Then match: LHS = V
- Examples
  - `?- 9 is 7+2.`
    true.
  - `?- 7+2 is 9.`
    false.
  - `?- X = 7+2.`
    X = 7+2.
  - `?- X is 7+2.`
    X = 9.
No Assignment

- = and is operators do not perform assignment

Example

- `foo(....X) :- ... X = 1,... % true only if X = 1`
- `foo(....X) :- ... X = 1, ..., X = 2, ... % always fails`
- `foo(....X) :- ... X is 1,... % true only if X = 1`
- `foo(....X) :- ... X is 1, ..., X is 2, ... % always fails`

X can’t be unified with 1 & 2 at the same time

Function Parameter & Return Value

- Code example

```prolog
increment(X,Y) :-
    Y is X+1.
?- increment(1,Z).
Z = 2.
?-- increment(1,2).
true.
?- increment(Z,2).
ERROR: incr/2: Arguments are not sufficiently instantiated
```

Can’t evaluate X+1 since X is not yet instantiated to int

Parameter

Return value
Function Parameter & Return Value

- Code example

```prolog
addN(X, N, Y) :-
    Y is X+N.
```

?- addN(1, 2, Z).
Z = 3.

Recursion

- Code example

```prolog
addN(X, 0, X).
addN(X, N, Y) :-
    X1 is X+1,
    N1 is N-1,
    addN(X1, N1, Y).
```

?- addN(1, 2, Z).
Z = 3.
Factorial

- Code
factorial(0, 1).
factorial(N, F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, F1),
    F is N * F1.

Tail Recursive Factorial w/ Accumulator

- Code
tail_factorial(0, F, F).
tail_factorial(N, A, F) :-
    N > 0,
    A1 is N * A,
    N1 is N - 1,
    tail_factorial(N1, A1, F).
And and OR

- And
  - To implement X && Y (use , in body of clause)
  - Example
    \[ Z \leftarrow X, Y. \]

- OR
  - To implement X || Y (use two clauses)
  - Example
    \[ Z \leftarrow X. \]
    \[ Z \leftarrow Y. \]

Goal Execution

- When submitting a query, we ask Prolog to substitute variables as necessary to make it true
- Prolog performs goal execution to find a solution
  - Start with the goal
  - Try to unify the head of a rule with the current goal
  - The rule hypotheses become subgoals
    - Substitutions from one subgoal constrain solutions to the next
  - If it reaches a dead end, it backtracks
    - Tries a different rule
  - When it can backtrack no further, it reports false
- More advanced topics later – cuts, negation, etc.
Goal Execution (cont.)

Consider the following:

- "All men are mortal"
  \[ \text{mortal}(X) \iff \text{man}(X). \]
- "Socrates is a man"
  \[ \text{man}(\text{socrates}). \]
- "Is Socrates mortal?"
  \[ ?- \text{mortal}(\text{socrates}). \text{true}. \]

How did Prolog infer this?

1. Sets \text{mortal}(\text{socrates}) as the initial goal
2. Sees if it unifies with the head of any clause:
   \[ \text{mortal}(\text{socrates}) = \text{mortal}(X). \]
3. \text{man}(\text{socrates}) becomes the new goal (since \( X = \text{socrates} \))
4. Recursively scans through all clauses, backtracking if needed …

Clause Tree

- Clause tree
  - Shows (recursive) evaluation of all clauses
  - Shows value (instance) of variable for each clause
  - Clause tree is true if all leaves are true

Factorial example

\begin{align*}
\text{factorial}(0,1) & . \\
\text{factorial}(N,F) \iff & \begin{cases} 
N > 0, \\
N1 \text{ is } N-1, \\
\text{factorial}(N1,F1), \\
F \text{ is } N*F1.
\end{cases}
\end{align*}
### Tracing

- `trace` lets you step through a goal’s execution
  - `notrace` turns it off

```prolog
?- trace.
true.

(trace) ?- my_last(X, [1,2,3]).
```

#### Example

- `my_last(X, [X]).`
- `my_last(X, [_|T]) :- my_last(X, T).`

```prolog
Call: (6) my_last(G2148, [1, 2, 3]) ? creep
Call: (7) my_last(G2148, [2, 3]) ? creep
Call: (8) my_last(G2148, [3]) ? creep
Exit: (8) my_last(3, [3]) ? creep
Exit: (7) my_last(3, [2, 3]) ? creep
Exit: (6) my_last(3, [1, 2, 3]) ? creep
X = 3
```

### Goal Execution – Backtracking

- Clauses are tried in order
- If clause fails, try next clause, if available

#### Example

- `jedi(luke).`
- `jedi(yoda).`
- `sith(vader).`
- `sith(maul).`
- `fight(X,Y) :- jedi(X), sith(Y).`

```prolog
?- fight(A,B).
A=luke,
B=vader;
A=luke,
B=maul;
A=yoda,
B=vader;
A=yoda,
B=maul.
```
Prolog (Search / Proof / Execution) Tree

?- fight(A,B).
A=X, B=Y

?- jedi(X), sith(Y).
X=luke, Y=vader
X=yoda, Y=maul

?- sith(vader).
?- sith(maul).

Lists In Prolog

- [a, b, 1, ‘hi’, [X, 2]]
- But really represented as compound terms
  - [] is an atom
  - [a, b, c] is represented as .(a, .(b, .(c, [])))
- Matching over lists
  - ?- [X, 1, Z] = [a, _, 17]
    X = a,
    Z = 17.
List Deconstruction

- Syntactically similar to Ocaml: \([H|T]\) like \(h::t\)
  
  \[
  \text{?- [Head | Tail] = [a,b,c].}
  \]
  
  Head = a,
  
  Tail = [b, c].

  \[
  \text{?- [1,2,3,4] = [_, X | _].}
  \]
  
  X = 2

- This is sufficient for defining complex predicates

- Let’s define \texttt{concat}(L1, L2, C)

  \[
  \text{?- concat([a,b,c], [d,e,f], X).}
  \]
  
  X = [a,b,c,d,e,f].

Example: Concatenating Lists

- To program this, we define the “rules” of concatenation
  
  - If \(L1\) is empty, then \(C = L2\)
    
    \[
    \text{concat( [ ], L2, L2 ).}
    \]

  - Prepending a new element to \(L1\) prepends it to \(C\), so long as \(C\) is the concatenation of \(L1\) with some \(L2\)

    \[
    \text{concat( [E | L1], L2, [E | C] ) :-}
    \]
    
    \[
    \text{concat(L1, L2, C).}
    \]

- … and we’re done
Why Is The Return Value An Argument?

- Now we can ask what inputs lead to an output

?- concat(X, Y, [a,b,c]).

\[
\begin{align*}
X &= [], \\
Y &= [a, b, c] ; \\
X &= [a], \\
Y &= [b, c] ; \\
X &= [a, b], \\
Y &= [c] ; \\
X &= [a, b, c], \\
Y &= [] ; \\
\end{align*}
\]

User types ; to request additional answers

More Syntax: Built-in Predicates

- Equality (a.k.a. unification)
  \[
  X = Y \quad f(1,X,2) = f(Y,3,_) \]

- fail and true

- "Consulting" (loading) programs
  
  \[
  ?- consult('file.pl') \quad ?- ['file.pl']
  \]

- Output/Input

  \[
  ?- write('Hello world'), nl \quad ?- read(X).
  \]

- (Dynamic) type checking

  \[
  ?- atom(elephant) \quad ?- atom(Elephant)
  \]

- help
The == Operator

- For identity comparisons
- X == Y
  - Returns true if and only if X and Y are identical
- Examples
  - ?- 9 == 9. true.
  - ?- X == X. true.
  - ?- 9 == 7+2. false.
  - ?- X == Y. false.
  - ?- 7+2 == 7+2. true.

The ::= Operator

- For arithmetic operations
- “LHS ::= RHS”
  - Evaluate the LHS to value V1 (Error if not possible)
  - Evaluate the RHS to value V2 (Error if not possible)
  - Then match: V1 = V2
- Examples
  - ?- 7+2 ::= 9. true.
  - ?- X ::= 9. Error: ::=/2: Arguments are not sufficiently instantiated
  - ?- 7+2 ::= 3+6. true.
  - ?- X ::= 7+2
Example – Towers of Hanoi

**Problem**

- Move stack of disks between pegs
- Can only move top disk in stack
- Only allowed to place disk on top of larger disk

To move a stack of \( n \) disks from peg \( X \) to \( Y \)

- **Base case**
  - If \( n = 1 \), move disk from \( X \) to \( Y \)
- **Recursive step**
  1. Move top \( n-1 \) disks from \( X \) to 3\(^{rd} \) peg
  2. Move bottom disk from \( X \) to \( Y \)
  3. Move top \( n-1 \) disks from 3\(^{rd} \) peg to \( Y \)

Iterative algorithm would take much longer to describe!
Towers of Hanoi

- Code
  ```prolog
  move(1,X,Y,_) :-
      write('Move top disk from '), write(X),
      write(' to '), write(Y), nl.
  move(N,X,Y,Z) :-
      N>1,
      M is N-1,
      move(M,X,Z,Y),
      move(1,X,Y,_),
      move(M,Z,Y,X).
  ```

Prolog Terminology

- A query, goal, or term where variables do not occur is called **ground**; else it’s **nonground**
  - `foo(a,b)` is ground; `bar(X)` is nonground
- A substitution $\theta$ is a partial map from variables to terms where $\text{domain}(\theta) \cap \text{range}(\theta) = \emptyset$
  - Variables are terms, so a substitution can map variables to other variables, but not to themselves
- $A$ is an **instance** of $B$ if there is a substitution such that $A = B\theta$
- $C$ is a **common instance** of $A$ and $B$ if it is an instance of $A$ and an instance of $B$
Prolog’s Algorithm Solve()

Solve(goal \( G \), program \( P \), substitution \( \theta \)) =

- Suppose \( G \) is \( A_1, \ldots, A_n \). Choose goal \( A_1 \).
- For each clause \( A :- B_1, B_2, \ldots, B_k \) in \( P \),
  - if \( \theta_1 \) is the mgu of \( A \) and \( A_1 \theta \) then
    - If \( \text{Solve}(\{ B_1, \ldots, B_k, A_2, \ldots, A_n \}, P, \theta \cdot \theta_1) = \text{some } \theta' \) then return \( \theta' \)
    - (else it has failed, so we continue the for loop)
  - (else unification has failed, so try another rule)
- If loop exits return fail
- Output: \( \theta \) s.t. \( G \theta \) can be deduced from \( P \), or fail

! : a.k.a. “cut”

- When a \(!\) is reached, it succeeds and commits Prolog to all the choices made since the parent goal was unified with the head of the clause the cut occurs in
  - Suppose we have clause \( C \) which is \( A :- B_1, \ldots, B_k, !, \ldots, B_n \).
  - If the current goal unifies with \( A \), and \( B_1, \ldots, B_k \) further succeed, the program is committed to the choice of \( C \) for the goal.
    - If any \( B_i \) for \( i > k \) fail, backtracking only goes as far as the cut.
    - If the cut is reached when backtracking, the goal fails
Cut

- Limits backtracking to predicates to right of cut

Example

```
jedi(luke).
jedi(yoda).
sith(vader).
sith(maul).
fight2(X,Y) :- jedi(X), !, sith(Y).
fight3(X,Y) :- jedi(X), sith(Y), !.
```

```
?- fight2(A,B).
A=luke, B=vader;
A=luke, B=maul.
A=yoda, B=vader.
```

Prolog Search Tree Limited By Cut

```
?- fight2(A,B).
A=x, B=y

?- jedi(X), !, sith(Y).
X=x

- jedi(luke), !, sith(Y).
Y=vader

- sith(vader).
Y=maul

- sith(maul).
```

```
?- fight3(A,B).
A=x, B=y

?- jedi(yoda), sith(Y).
X=yoda

- jedi(yoda), sith(Y).
Y=vader

- sith(vader).
Y=maul

- sith(maul).
```
What Exactly Is Cut Doing?

- Prunes all clauses below it
- Prunes alternative solutions to its left
- Does not affect the goals to its right

Note: Cut only affects this call to merge. Does not affect backtracking of functions calling merge, or later recursive call to merge past cut

merge([X|Xs], [Y|Ys], [X|Zs]) :- !, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X,Y|Zs]) :- X =:= Y, !, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X > Y, !, merge([X|Xs], Ys, Zs).
merge([], [], Xs) :- !.
merge([], Ys) :- !.

Why Use Cuts?

- Save time and space, or eliminate redundancy
  - Prune useless branches in the search tree
  - If sure these branches will not lead to solutions
  - These are green cuts

- Guide to the search to a different solution
  - Change the meaning of the program
  - Intentionally returning only subset of possible solutions
  - These are red cuts
Negation As Failure

- Cut may be used implement negation (not)
- Example
  not(X) :- call(X), !, fail.
  not(X).
- If X succeeds, then the cut is reached, committing it; fail causes the whole thing to fail
- If X fails, then the second rule is reached, and the overall goal succeeds.
  - FYI, X here refers to an arbitrary goal
  - Effect of not depends crucially on rule order

Not

- Not is tricky to use
  - Does not mean “not true”
  - Just means “not provable at this time”
- Example
  jedi(luke).
  jedi(vader).
  sith(vader).
  Cannot prove either jedi(leia) or sith(leia) are true, so not( ) returns true

  ?- not(sith(luke)).
  true.
  ?- not(sith(vader)).
  false.
  ?- not(jedi(leia)).
  true.
  ?- not(sith(leia)).
  true.
Not (cont.)

- Not is tricky to use
  - Does not mean “not true”
  - Just means “not provable at this time”

- Example
  - `jedi(luke).`  
  - `jedi(vader).`  
  - `sith(vader).`  

Because not(sith(X)) does not mean “Can prove sith(X) is false for some X”
Instead, it means “Cannot prove sith(X) is true for some X”. So X=vader causes not(sith(X)) to fail and return false

Huh? Why not return X=luke?

Not – Search Tree

Will search for all X such that sith(X) is true.
**Not (cont.)**

- Ordering of clauses matters
- Example
  
  ```prolog
  jedi(luke).
  jedi(vader).
  sith(vader).
  true_jedi1(X) :-
    jedi(X), not(sith(X)).
  true_jedi2(X) :-
    not(sith(X)), jedi(X).
  ```

  X=vader causes not(sith(X)) to fail; Will not backtrack to X=luke, since sith(luke) is not a fact

---

**True_jedi2 – Search Tree**

```prolog
?- true_jedi2(X).
true.
?- true_jedi1(X).
X=luke.
?- true_jedi2(luke).
true.
?- true_jedi2(X).
false.
```
Not and \\(!=\\)

- **Built-in operators**
  - \(+\) is not
  - \(X \neq Y\) is same as not(\(X=Y\))
  - \(X \neq= Y\) is same as not(\(X==Y\))
- So be careful using \(!=\\)
  - Ordering of clauses matters
  - Try to ensure operands of \(!=\\) are instantiated

---

**Example Using \(!=\\)**

- **Example**
  
  `jedi(luke).`
  
  `jedi(yoda).`
  
  `help2(X,Y) :- jedi(X), jedi(Y), X \neq Y.`
  
  `help3(X,Y) :- jedi(X), X \neq Y, jedi(Y).`
  
  `help4(X,Y) :- X \neq Y, jedi(X), jedi(Y).`

  
  ?- help2(X,Y).
  
  X=luke,
  
  Y=yoda;
  
  X=yoda,
  
  Y=luke.

  
  ?- help3(X,Y).
  
  X=yoda.

  
  ?- help3(X,Y).
  
  false.

  
  After selecting X, can choose Y=X and fail X \neq Y.
Help3 – Search Tree

not(X=Y) :- X=Y, !, fail.
not(X=Y).

jedi(luke).
jedi(yoda).

X=luke

Y=luke

?- jedi(luke), luke \= Y, jedi(Y).
luke=luke, !, fail

Y=yoda

?- jedi(yoda), yoda \= Y, jedi(Y).
yoda=yoda, !, fail

Using \=

- In fact, given X \= Y
  - will always fail if X or Y are not both instantiated

\[ X \not\!\!\not= a \] // fails for X=a
\[ a \not\!\!\not= Y \] // fails for Y=a
\[ X \not\!\!\not= Y \] // fails for X=Y
Example Using \(\neq\)

**Example**

```prolog
jedi(luke).
jedi(yoda).
help2(X,Y) :- jedi(X), jedi(Y), X \(\neq\) Y.
help3(X,Y) :- jedi(X), X \(\neq\) Y, jedi(Y).
help4(X,Y) :- X \(\neq\) Y, jedi(X), jedi(Y).
```

?- help4(X,luke).
false.

?- help4(yoda,luke).
true.

Built-in List Predicates

**length(List,Length)**

```prolog
?- length([a, b, [1,2,3] ], Length).
Length = 3.
```

**member(Elem,List)**

```prolog
?- member(duey, [huey, duey, luey]).
true.

?- member(X, [huey, duey, luey]).
X = huey; X = duey; X = luey.
```

**append(List1,List2,Result)**

```prolog
?- append([duey], [huey, duey, luey], X).
X = [duey, huey, duey, luey].
```
Built-in Predicates

- `sort(List, SortedList)`
  ```prolog
  ?- sort([2,1,3], R).
  R = [1,2,3].
  ```

- `findall(Elem, Predicate, ResultList)`
  ```prolog
  ?- findall(E, member(E, [huey, duey, luey]), R).
  R = [huey, duey, luey].
  ```

- `setof(Elem, Predicate, ResultSortedList)`
  ```prolog
  ?- setof(E, member(E, [huey, duey, luey]), R).
  R = [duey, huey, luey].
  ```

- See documentation for more
  ```plain
  • http://www.swi-prolog.org/pldoc/man?section=builtin
  ```

Prolog Summary

- General purpose logic programming language
  - Associated with AI, computational linguistics
  - Also used for theorem proving, expert systems

- Declarative programming
  - Specify facts & relationships between facts (rules)
  - Run program as queries over these specifications

- Natural support for
  - Searching within set of constraints
  - Backtracking