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

More Information On Prolog

- Various tutorials available online
- Links on webpage
- We will use SWI Prolog
  http://www.swi-prolog.org/swipl, on Grace

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

/* 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).

Use /* */ for comments, or % for 1-liners
Period ends statements

Lowercase logically terminates
Program consists of facts and rules
Uppercase denotes variables

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.

Use /* */ for comments, or % for 1-liners
Period ends statements

Lowercase logically terminates
Program consists of facts and rules
Uppercase denotes variables

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.

User types ; to request additional answer
Multiple answers
User types return to complete request
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 _
  - 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)

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 (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
Prolog Syntax and Terminology (cont.)

- **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

  ```prolog
  ?- is_bigger(horse, duck).
  ?- is_bigger(horse, X).
  “Does there exist a substitution for X such that is_bigger(horse, X)?”
  ```

Unification – The Sine Qua Non of Prolog

- **Two terms unify if and only if**
  - They are identical
    ```prolog
    ?- gnat = gnat.
    true.
    ```
  - They can be made identical by substituting variables
    ```prolog
    ?- 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.

    ```prolog
    ?- pred(X, 2, 2) = pred(1, Y, X)
    false
    ```
    ```prolog
    ?- pred(X, 2, 2) = pred(1, Y, _)
    X = 1,
    Y = 2.
    ```
    Sometimes there are multiple possible substitutions; Prolog can be asked to enumerate them all

The = Operator

- For unification (matching)

  ```prolog
  ?- 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
    ```prolog
    ?- 9 is 7+2.
    true.
    ?- 7+2 is 9.
    false.
    ```
    ```prolog
    ?- X = 7+2.
    X = 9.
    ```
No Variable 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>increment(X,Y) :-</td>
<td></td>
</tr>
<tr>
<td>Y is X+1.</td>
<td></td>
</tr>
</tbody>
</table>

?- increment(1,Z). Query
Z = 2. Result

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

ERROR: incr/2: Arguments are not sufficiently instantiated

Function Parameter & Return Value

- Code example

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>addN(X,N,Y) :-</td>
<td></td>
</tr>
<tr>
<td>Y is X+N.</td>
<td></td>
</tr>
</tbody>
</table>

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

Recursion

- Code example

<table>
<thead>
<tr>
<th>Base case</th>
<th>Inductive step</th>
<th>Recursive call</th>
</tr>
</thead>
<tbody>
<tr>
<td>addN(X,0,X).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addN(X,N,Y) :-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X1 is X+1,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N1 is N-1,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addN(X1,N1,Y).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

?- addN(1,2,Z). Query
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 :- X,Y.`

- **OR**
  - To implement X || Y (use two clauses)
  - Example
    `Z :- X.
     Z :- 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”
  mortal(X) :- man(X).
• “Socrates is a man”
  man(socrates).
• “Is Socrates mortal?”
  ?- mortal(socrates).

How did Prolog infer this?

1. Sets mortal(socrates) as the initial goal
2. Sees if it unifies with the head of any clause:
   mortal(socrates) = mortal(X).
3. man(socrates) becomes the new goal (since X=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

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

Tracing

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

Goal Execution – Backtracking

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

Example

?- fight(A,B).
A=luke,
B=vader;
A=yoda,
B=vader;
A=yoda,
B=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 related to Ocaml: \([H|T]\) like \(h::t\)
  - \(?- [\text{Head} | \text{Tail}] = [a,b,c].\)
    - \(\text{Head} = a,\)
    - \(\text{Tail} = [b, c].\)
  - \(?- [1,2,3,4] = [\_, X | \_].\)
    - \(X = 2\)
- This is sufficient for defining complex predicates
- Let’s define \(\text{concat}(L1, L2, C)\)
  - \(?- \text{concat}([a,b,c], [d,e,f], X).\)
    - \(X = [a,b,c,d,e,f].\)

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

More Syntax: Built-in Predicates

Equality (a.k.a. unification)

\[
X = Y \\
f(1,X,2) = f(Y,3,\_)
\]

fail and true

“Consulting” (loading) programs

?- consult('file.pl')
?- ['file.pl']

Output/Input

?- write('Hello world'), nl
?- read(X).

(Dynamic) type checking

?- atom(elephant)
?- atom(Elephant)

help

The \texttt{==} Operator

For identity comparisons

\[
X == Y
\]

- Returns true if and only if X and Y are identical

Examples

?- 9 == 9. 
  true. 
  ?- 9 == 7+2. 
  false.
?- X == 9.
  False.
?- X == X.
  true.

The \texttt{:==} 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.
?- 7+2 =:= 3+6.
  true.
?- X =:= 9.
?- X =:= 7+2
Error: :==/2: Arguments are not sufficiently instantiated
Example – Towers of Hanoi

- Problem
  - Move full stack of disks to another peg
  - Can only move top disk in stack
  - Only allowed to place disk on top of larger disk

Example – Towers of Hanoi

- To move a stack of \( n \) disks from peg X to Y
  - Base case
    - If \( n = 1 \), transfer disk from X to Y
  - Recursive step
    1. Move top \( n-1 \) disks from X to 3rd peg
    2. Move bottom disk from X to Y
    3. Move top \( n-1 \) disks from 3rd 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).
```