Logic Programming Languages

- Express programs in a form of symbolic logic
- Use a logical inferencing process to produce results
- Declarative programming language
- Declarative rather than procedural:
  - only specification of results are stated
  - not detailed procedures for producing them
- Applications of Logic Programming
  - Edinburgh Syntax
  - Relational database management systems
  - Expert systems
  - Natural language processing
  - Education

Predicate Calculus

- Proposition
  - a logical statement that may or may not be true
  - Consists of objects and relationships of objects to each other
- Symbolic logic
  - can be used for the basic needs of formal logic:
    - express propositions
    - express relationships between propositions
    - describe how new propositions can be inferred from other propositions
- Predicate Calculus
  - Form of symbolic logic used for logic programming

Predicate Calculus

- Propositions
  - Objects in propositions are represented by simple terms
  - Constant:
    - a symbol that represents an object
  - Variable:
    - a symbol that can represent different objects at different times
    - different from variables in imperative languages
  - Atomic propositions
    - consist of compound terms
  - Compound term:
    - one element of a mathematical relation, written like a mathematical function
      - Mathematical function is a mapping
      - Can be written as a table
    - Examples:
      - student(jon)
      - like(seth, OSX)
      - like(nick, windows)
      - like(jim, linux)
Predicate Calculus

• Propositions can be stated in two forms:
  – Fact
    • proposition is assumed to be true
  – Query
    • truth of proposition is to be determined

• Compound proposition:
  – Have two or more atomic propositions
  – Propositions are connected by operators

## Logical Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>(\neg)</td>
<td>(\neg a)</td>
<td>not a</td>
</tr>
<tr>
<td>conjunction</td>
<td>(\land)</td>
<td>(a \land b)</td>
<td>a and b</td>
</tr>
<tr>
<td>disjunction</td>
<td>(\lor)</td>
<td>(a \lor b)</td>
<td>a or b</td>
</tr>
<tr>
<td>equivalence</td>
<td>(\equiv)</td>
<td>(a \equiv b)</td>
<td>a is equivalent to b</td>
</tr>
<tr>
<td>implication</td>
<td>(\rightarrow)</td>
<td>(a \rightarrow b)</td>
<td>a implies b</td>
</tr>
</tbody>
</table>

## Quantifiers

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>universal</td>
<td>(\forall X.P)</td>
<td>For all X, P is true</td>
</tr>
<tr>
<td>existential</td>
<td>(\exists X.P)</td>
<td>There exists a value of X such that P is true</td>
</tr>
</tbody>
</table>

## Proving Theorems

• Propositions can discover new theorems that can be inferred from known axioms and theorems
• Resolution
  – an inference principle that allows inferred propositions to be computed from given propositions
• Unification
  – finding values for variables in propositions that allows matching process to succeed
• Instantiation
  – assigning temporary values to variables to allow unification to succeed
  – After instantiating a variable with a value, if matching fails, may need to backtrack and instantiate with a different value

• Use proof by contradiction
• Hypotheses
  – a set of pertinent propositions
• Goal
  – negation of theorem stated as a proposition
• Theorem is proved by finding an inconsistency
• Basis for logic programming
• Horn clause - can have only two forms
  – Headed: single atomic proposition on left side
  – Headless: empty left side (used to state facts)
  – Most propositions can be stated as Horn clauses
Logic Programming

- Declarative semantics
  - Method to determine the meaning of each statement
  - Simpler than the semantics of imperative languages
- Programming is nonprocedural
  - Programs do not state now a result is to be computed
  - Just rather the form of the result
- Sorting a List
  - Describe the characteristics of a sorted list, not the process of rearranging a list

\[
\begin{align*}
\text{sort(old_list, new_list)} & \cap \text{permute(old_list,new_list)} \\
\text{sorted(list)} & \land \forall j\text{ such that }1 < j < n, \text{ list}(j) < \text{list}(j+1)
\end{align*}
\]

Prolog

- University of Aix-Marseille -
  - Natural language processing
- University of Edinburg -
  - Automated theorem proving
- Advantages:
  - Prolog programs based on logic, so likely to be more logically organized and written
  - Processing is naturally parallel, so Prolog interpreters can take advantage of multi-processor machines
  - Programs are concise, so development time is decreased – good for prototyping

Prolog

- Term
  - a constant, variable, or structure
- Constant
  - an atom or an integer
- Atom
  - symbolic value of Prolog
  - string of letters, digits, and underscores beginning with a lowercase letter
  - string of printable ASCII characters delimited by apostrophes

Prolog

- Variable:
  - any string of letters, digits, and underscores beginning with an uppercase letter
- Instantiation:
  - binding of a variable to a value
  - Lasts only as long as it takes to satisfy one complete goal
- Structure:
  - represents atomic proposition
  \[\text{functor(parameter list)}\]
- Fact Statements
  - Used for the hypotheses
  - Headless Horn clauses
    - student(jonathan).
    - sophomore(ben).
    - brother(tyler, cj).

Prolog

- Rule Statements
  - Used for the hypotheses
  - Headed Horn clause
  - Right side: antecedent (if part)
    - May be single term or conjunction
  - Left side: consequent (then part)
    - Must be single term
  - Conjunction:
    - multiple terms separated by logical AND operations (implied)

parent(kim,kathy):= mother(kim,kathy).

Prolog

- Rule Statements
  - Can use variables (universal objects) to generalize meaning:
    - parent(X,Y):~ mother(X,Y).
    - sibling(X,Y):~ mother(M,X), mother(M,Y), father(F,X), father(F,Y).
- Goal Statements
  - Theorem we want system to prove or disprove in proposition form
  - Same format as headless Horn
    - student(james)
  - Conjunctive propositions and propositions with variables
    - also legal goals
    - father(X, joe)
Prolog

- Inferencing Process of Prolog
  - Queries are called goals
  - If a goal is a compound proposition,
    - each of the facts is a subgoal
  - To prove a goal is true, must find a chain of inference rules and/or facts. For goal Q:
    
    \[
    \begin{align*}
    B & :- A \\
    C & :- B \\
    & \ldots \\
    Q & :- P
    \end{align*}
    \]
  - Process of proving a subgoal called matching, satisfying, or resolution

- Inferencing Process
  - Bottom-up resolution, forward chaining
    - Begin with facts and rules of database and attempt to find sequence that leads to goal
    - works well with a large set of possibly correct answers
  - Top-down resolution, backward chaining
    - begin with goal and attempt to find sequence that leads to set of facts in database
    - works well with a small set of possibly correct answers
  - Prolog implementations use backward chaining

- Inferencing Process
  - When goal has more than one subgoal, use either
    - Depth-first search: find a complete proof for the first subgoal before working on others
    - Breadth-first search: work on all subgoals in parallel
  - Prolog uses depth-first search
    - Can be done with fewer computer resources
    - With a goal with multiple subgoals, if fail to show truth of one of subgoals, reconsider previous subgoal to find an alternative solution: backtracking
    - Begin search where previous search left off
    - Can take lots of time and space because may find all possible proofs to every subgoal

- Simple Arithmetic
  - Prolog supports integer variables and integer arithmetic
  - is operator: takes an arithmetic expression as right operand and variable as left operand
  - Not the same as an assignment statement!

speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :- speed(X,Speed), time(X,Time),
                 Y is Speed * Time.

- Trace
  - Displays instantiations at each step
  - Call (attempt to satisfy goal)
  - Exit (when a goal has been satisfied)
  - Redo (when backtrack occurs)
  - Fail (when goal fails)
  - Not the same as an assignment statement!
Prolog

• List Structures
  – Other basic data structure
  – Sequence of any number of elements
  – Elements can be atoms, atomic propositions, or other terms (including other lists)

  \[\text{[apple, prune, grape, kumquat]}\]
  [\text{[empty list]}]
  [\text{[X | Y]}] (head X and tail Y)

Prolog

• Definition of append function:

\[
\text{append([], List, List).}
\]

\[
\text{append([Head | List_1], List_2, [Head | List_3]) :-}
\]

\[
\text{append (List_1, List_2, List_3).}
\]

• Definition of reverse function:

\[
\text{reverse([], [], []).}
\]

\[
\text{reverse([Head | Tail], List) :-}
\]

\[
\text{reverse (Tail, Result),}
\]

\[
\text{append (Result, [Head], List).}
\]

Prolog

\[\text{?- append([1,2],[3,4],Z).}
\]

\[\text{Z = [1, 2, 3, 4]}
\]

Yes

Predefined \text{append(X, Y, Z)} succeeds if and only if \text{Z} is the result of appending the list \text{Y} onto the end of the list \text{X}

SWI-Prolog

Welcome to SWI-Prolog (Version 3.4.2)
Copyright (c) 1990-2000 University of Amsterdam.
Copy policy: GPL-2 (see www.gnu.org)

For help, use \text{?- help(Topic).} or \text{?- apropos(Word).}

?-

• Prompting for a query with \text{?-}
• Normally interactive: get query, print result, repeat

SWI-Prolog

?- \text{consult(relations).}
% relations compiled 0.00 sec, 0 bytes

Yes

?-

• The \text{consult} predicate
  – Predefined predicate to read a program from a file into the database
  – File \text{relations} (or \text{relations.pl}) contains our parent facts

SWI-Prolog

?- \text{parent(margaret,kent).}

Yes

?- \text{parent(fred,pebbles).}

No

?-

• Queries
  – A query asks the language system to prove something
  – The answer will be \text{Yes} or \text{No}
  – (Some queries, like \text{consult}, are executed only for their side-effects)
Final Period
- Queries can take multiple lines
- If you forget the final period, Prolog prompts for more input with !

Queries With Variables
- Any term can appear as a query, including a term with variables
- The Prolog system shows the bindings necessary to prove the query

SWI-Prolog

```
?- parent(margaret,kent).
Yes
?- parent(P,jean).
P = herbert
Yes
?- parent(P,esther).
No
```

Here, it waits for input. We hit Enter to make it proceed.

Conjunctions
- A conjunctive query has a list of query terms separated by commas
- The Prolog system tries prove them all (using a single set of bindings)

```
?- parent(margaret,X), parent(X,holly).
X = kim
Yes
```

Multiple Solutions are Possible
- There might be more than one way to prove the query
- By typing ; rather than Enter, you ask the Prolog system to find more

```
?- parent(margaret,Child).
Child = kim ;
Child = kent ;
No
```

```
?- parent(Parent,kim), parent(Grandparent,Parent).
Parent = margaret
Grandparent = esther ;
Parent = margaret
Grandparent = herbert ;
No
?- parent(ester,Child), parent(Child,Grandchild), parent(Grandchild,GreatGrandchild).
Child = margaret
Grandchild = kim
GreatGrandchild = holly
Yes
```
A rule says how to prove something: to prove the head, prove the conditions
To prove greatgrandparent(GGP,GGC), find some GP and P for which you can prove parent(GGP,GP), then parent(GP,P) and then finally parent(P,GGC)

This shows the initial query and final result
Internally, there are intermediate goals:
- The first goal is the initial query
- The next is what remains to be proved after transforming the first goal using one of the clauses (in this case, the greatgrandparent rule)
- And so on, until nothing remains to be proved
A program consists of a list of clauses
A clause is either a fact or a rule, and ends with a period

Rules Using Other Rules
- Same relation, defined indirectly
- Note that both clauses use a variable P
- The scope of the definition of a variable is the clause that contains it

Recursive Rules
- X is an ancestor of Y if:
  - Base case: X is a parent of Y
  - Recursive case: there is some Z such that Z is a parent of Y, and X is an ancestor of Z
- Prolog tries rules in the order you give them, so put base-case rules and facts first
**SWI-Prolog**

\[ \text{ancestor}(X,Y) :- \text{parent}(X,Y). \]
\[ \text{ancestor}(X,Y) :- \text{parent}(Z,Y), \text{ancestor}(X,Z). \]

?- ancestor(jean,jean).
No
?- ancestor(kim,holly).
Yes
?- ancestor(A,holly).
A = kim ;
A = margaret ;
A = esther ;
A = herbert ;

**The Anonymous Variable**

- The variable _ is an anonymous variable
- Every occurrence is bound independently of every other occurrence
- In effect, much like ML's _ : it matches any term without introducing bindings

**Example**

\[ \text{tailof}(\text{_}, A, A). \]
\[ \text{tailof}([\text{_}, A], A). \]

- This \text{tailof}(X,Y) succeeds when X is a non-empty list and Y is the tail of that list
- Don’t use this, even though it works:

\[ \text{tailof}(\text{_}(\text{Head}, A), A). \]

**The \text{not} Predicate**

?- member(1,[1,2,3]).
Yes
?- not(member(4,[1,2,3])).
Yes

- For simple applications, it often works quite a bit like logical negation
- But it has an important procedural side...

**Negation As Failure**

- To prove \text{not}(X), Prolog attempts to prove X
- \text{not}(X) succeeds if X fails
- The two faces again:
  - Declarative: \text{not}(X) = \neg X
  - Procedural: \text{not}(X) succeeds if X fails, fails if X succeeds, and runs forever if X runs forever

**Example**

\[ \text{sibling}(X,Y) :- \text{not}(X=Y), \text{parent}(P,X), \text{parent}(P,Y). \]

?- sibling(kim,kent).
Yes
?- sibling(kim,kim).
No
?- sibling(X,Y).
No
A Classic Riddle

• A man travels with wolf, goat and cabbage
• Wants to cross a river from west to east
• A rowboat is available, but only large enough for the man plus one possession
• Wolf eats goat if left alone together
• Goat eats cabbage if left alone together
• How can the man cross without loss?

Configurations

• Represent a configuration of this system as a list showing which bank each thing is on in this order: man, wolf, goat, cabbage
• Initial configuration: [w,w,w,w]
• If man crosses with wolf, new state is [e,e,w,w] – but then goat eats cabbage, so we can’t go through that state
• Desired final state: [e,e,e,e]

Moves

• In each move, man crosses with at most one of his possessions
• We will represent these four moves with four atoms: wolf, goat, cabbage, nothing
• (Here, nothing indicates that the man crosses alone in the boat)

Moves Transform Configurations

• Each move transforms one configuration to another
• In Prolog, we will write this as a predicate: move(Config,Move,NextConfig)
  – Config is a configuration (like [w,w,w,w])
  – Move is a move (like wolf)
  – NextConfig is the resulting configuration
    – (in this case: [e,e,w,w])

The move Predicate

change(e,w).
change(w,e).
move([X,X,Goat,Cabbage],wolf,[Y,Y,Goat,Cabbage]) :- change(X,Y).
move([X,Wolf,X,Cabbage],goat,[Y,Wolf,Y,Cabbage]) :- change(X,Y).
move([X,Wolf,Goat,X],cabbage,[Y,Wolf,Goat,Y]) :- change(X,Y).
move([X,Wolf,Goat,C],nothing,[Y,Wolf,Goat,C]) :- change(X,Y).

Safe Configurations

• A configuration is safe if
  – At least one of the goat or the wolf is on the same side as the man, and
  – At least one of the goat or the cabbage is on the same side as the man

oneEq(X,X,_,_).
oneEq(X,_,X,_,_).
safe([Man,Wolf,Goat,Cabbage]) :- oneEq(Man,Goat,Wolf),
oneEq(Man,Goat,Cabbage).
Solutions

- A solution is a starting configuration and a list of moves that takes you to \([e, e, e, e]\), where all the intermediate configurations are safe

\[
\text{solution([e,e,e,e],[]).}
\]

\[
\text{solution(Config,[Move|Rest]) :-}
\]

\[
\begin{align*}
& \text{move(Config,Move,NextConfig),} \\
& \text{safe(NextConfig),} \\
& \text{solution(NextConfig,Rest).}
\end{align*}
\]

Prolog Finds A Solution

?- length(X,7), solution([w,w,w,w],X).
X = [goat, nothing, wolf, goat, cabbage, nothing, goat]
Yes

- Note: without the length(X, 7) restriction, Prolog would not find a solution
- It gets lost looking at possible solutions like [goat, goat, goat, goat, goat...]

Unification

- Two Prolog terms \(t_1\) and \(t_2\) unify if there is some substitution \(s\) (their unifier) that makes them identical: \(s(t_1) = s(t_2)\)
  
  - \(a\) and \(b\) do not unify
  - \(f(X,b)\) and \(f(a,Y)\) unify: a unifier is \(\{X \mapsto a, Y \mapsto b\}\)
  - \(f(X,b)\) and \(g(X,b)\) do not unify
  - \(a(X,X,b)\) and \(a(b,X,X)\) unify: a unifier is \(\{X \mapsto b\}\)
  - \(a(X,X,b)\) and \(a(c,X,X)\) do not unify
  - \(a(X,f)\) and \(a(X,f)\) do unify: a unifier is \(\{\}\)

Multiple Unifiers

- \(\text{parent}(X,Y)\) and \(\text{parent}(fred,Y)\):
  
  - one unifier is \(s_1 = \{X \mapsto fred\}\)
  
  - another is \(s_2 = \{X \mapsto fred, Y \mapsto mary\}\)

- Prolog chooses unifiers like \(s_1\) that do just enough substitution to unify, and no more
- That is, it chooses the MGU—the Most General Unifier

MGU

- Term \(x_1\) is more general than \(x_2\) if \(x_2\) is an instance of \(x_1\) but \(x_1\) is not an instance of \(x_2\)
  
  - Example: \(\text{parent}(fred,Y)\) is more general than \(\text{parent}(fred,mary)\)

- A unifier \(s_1\) of two terms \(t_1\) and \(t_2\) is an MGU if there is no other unifier \(s_2\) such that \(s_2(t_1)\) is more general than \(s_1(t_1)\)

- MGU is unique up to variable renaming

Unification For Everything

- Parameter passing
  
  - \(\text{reverse([1,2,3],X)}\)

- Binding
  
  - \(X=0\)

- Data construction
  
  - \(X=\{1,2,3\}\)

- Data selection
  
  - \(\{1,2,3\}=\{X,Y\}\)
The Occurs Check

- Any variable \( X \) and term \( t \) unify with \( \{ X \mapsto t \} \):
  - \( X \) and \( b \) unify: an MGU is \( \{ X \mapsto b \} \)
  - \( X \) and \( f(a,g(b,c)) \) unify: an MGU is \( \{ X \mapsto f(a,g(b,c)) \} \)
  - \( X \) and \( f(a,Y) \) unify: an MGU is \( \{ X \mapsto f(a,Y) \} \)
- Unless \( X \) occurs in \( t \):
  - \( X \) and \( f(a,X) \) do not unify, in particular not by \( \{ X \mapsto f(a,X) \} \)

Quoted Atoms As Strings

- Any string of characters enclosed in single quotes is a term
- In fact, Prolog treats it as an atom:
  - \('abc'\) is the same atom as \(abc\)
  - \('hello world'\) and \('Hello world'\) are atoms too
- Quoted strings can use \( \backslash n, \backslash t, \backslash ', \backslash \)\)

Input and Output

- Simple term input and output.
- Also the predicate \( \text{read} \): equivalent to \( \text{write}('\text{\textquoteleft\text{\textquoteleft'})} \)

Debugging With \texttt{write}

\begin{verbatim}
?- write('Hello world').
Hello world
Yes
?- read(X).
| hello.
X = hello
Yes
\end{verbatim}

- \texttt{write('Hello world')}.
- \texttt{Yes}
- \texttt{?~ read(X).}
- \texttt{| hello.}
- \texttt{X = hello}
- \texttt{Yes}

The \texttt{assert} Predicate

\begin{verbatim}
?- assert(parent(joe,mary)).
Yes
?- parent(joe,mary).
No
?- assert(parent(joe,mary)).
Yes
?- parent(joe,mary).
Yes
\end{verbatim}

- Adds a fact to the database (at the end)

The \texttt{retract} Predicate

\begin{verbatim}
?- parent(joe,mary).
Yes
?- retract(parent(joe,mary)).
Yes
?- parent(joe,mary).
No
\end{verbatim}

- Removes the first clause in the database that unifies with the parameter
- Also \texttt{retractall} to remove all matches
An Adventure Game

• Prolog comments
  - /* to */, like Java
  - Also, % to end of line

/*
This is a little adventure game. There are three
entities: you, a treasure, and an ogre. There are
six places: a valley, a path, a cliff, a fork, a maze,
and a mountaintop. Your goal is to get the treasure
without being killed first.
*/

/*
First, text descriptions of all the places in
the game.
*/
description(valley,
  'You are in a pleasant valley, with a trail ahead.').
description(path,
  'You are on a path, with ravines on both sides.').
description(cliff,
  'You are teetering on the edge of a cliff.').
description(fork,
  'You are at a fork in the path.').
description(maze(_),
  'You are in a maze of twisty trails, all alike.').
description(mountaintop,
  'You are on the mountaintop.').

/*
report prints the description of your current
location.
*/
report : -
at(you, X),
description(X, Y),
write(Y), nl.

?- assert(at(you, cliff)).
Yes
?- report.
You are teetering on the edge of a cliff.
Yes
?- retract(at(you, cliff)).
Yes
?- assert(at(you, valley)).
Yes
?- report.
You are in a pleasant valley, with a trail ahead.
Yes

/*
These connect predicates establish the map.
The meaning of connect(X, Dir, Y) is that if you
are at X and you move in direction Dir, you
get to Y. Recognized directions are
forward, right and left.
*/
connect(valley, forward, path).
connect(path, right, cliff).
connect(path, left, cliff).
connect(path, forward, fork).
connect(fork, left, maze(0)).
connect(fork, right, mountaintop).
connect(maze(0), left, maze(1)).
connect(maze(0), right, maze(2)).
connect(maze(1), right, maze(2)).
connect(maze(2), left, fork).
connect(maze(0), right, maze(3)).
connect(maze(_), _, maze(0)).

/*
move(Dir) moves you in direction Dir, then
prints the description of your new location.
*/
move(Dir) : -
at(you, Loc),
connect(Loc, Dir, Next),
retract(at(you, Loc)),
assert(at(you, Next)),
report.
/*
But if the argument was not a legal direction,
print an error message and don’t move.
*/
move(_) : -
write('That is not a legal move.\n'),
report.
/*
 * Shorthand for moves.
 */
forward :- move(forward).
left :- move(left).
right :- move(right).

/*
 * If you and the ogre are at the same place, it kills you.
 */
ogre :-
at(ogre,Loc),
at(you,Loc),
write('An ogre sucks your brain out through\n'),
write('your eyesockets, and you die.\n'),
retract(at(you,Loc)),
assert(at(you,done)).
/*
 * But if you and the ogre are not in the same place, nothing happens.
 */
ogre.

/*
 * If you are at the cliff, you fall off and die.
 */
cliff :-
at(you,cliff),
write('You fall off and die.\n'),
assert(at(you,done)).
/*
 * But if you are not at the cliff nothing happens.
 */
cliff.

/*
 * Main loop.  Stop if player won or lost.
 */
main :-
at(you,done),
write('Thanks for playing.\n').
/*
 * Main loop.  Not done, so get a move from the user and make it. Then run all our special behaviors. Then repeat.
 */
main :-
write('\nNext move -- '),
read(Move),
call(Move),
ogre,
treasure,
cliff,
main.

The predefined predicate call(X) tries to prove X as a goal term.
This is the starting point for the game. We assert the initial conditions, print an initial report, then start the main loop.

```prolog
/*
This is an adventure game. Legal moves are left, right or forward. End each move with a period.

You are in a pleasant valley, with a trail ahead. Next move -- forward. You are on a path, with ravines on both sides. Next move -- forward. You are at a fork in the path. Next move -- right. You are on the mountaintop. There is a treasure here. Congratulations, you win! Thanks for playing.
*/
go :-
  retractall(at(_,_)), % clean up from previous runs
  assert(at(you,valley)),
  assert(at(ogre,maze(3))),
  assert(at(treasure,mountaintop)),
  write('This is an adventure game. \n'),
  write('Legal moves are left, right or forward.\n'),
  write('End each move with a period.\n'),
  report,
  main.
```

**Unevaluated Terms**

- Prolog operators allow terms to be written more concisely, but are not evaluated
- These are all the same Prolog term:
  ```prolog
  +(1,*[2,3])
  1+ *[2,3]
  +(1,2*3)
  (1+(2*3))
  1+2*3
  ```
- That term does not unify with 7

**Evaluating Expressions**

```prolog
?- X is 1+2*3.
X = 7
yes
```

- The predefined predicate `is` can be used to evaluate a term that is a numeric expression
- `is(X,Y)` evaluates the term `Y` and unifies `X` with the resulting atom
- It is usually used as an operator

**Instantiation Is Required**

```prolog
?- Y=X+2, X=1.
Y = 1+2
X = 1
yes
?- Y is X+2, X=1.
ERROR: Arguments are not sufficiently instantiated
?- X=1, Y is X+2.
X = 1
Y = 3
yes
```

**Evaluable Predicates**

- For `X is Y`, the predicates that appear in `Y` have to be evaluable predicates
- This includes things like the predefined operators `+`, `-`, `*` and `/`
- There are also other predefined evaluable predicates, like `abs(Z)` and `sqrt(Z)`
Real Values And Integers

There are two numeric types: integer and real.

Most of the evaluable predicates are overloaded for all combinations.

Prolog is dynamically typed; the types are used at runtime to resolve the overloading.

But note that the goal 2=2.0 would fail.

Comparisons

• Numeric comparison operators:
  <, >, =<, >=, =:=, =\= 

• To solve a numeric comparison goal, Prolog evaluates both sides and compares the results numerically

• So both sides must be fully instantiated

Comparisons

?– 1+2 < 1*2.
   No
?– 1<2.
   Yes
?– 1+2>=1+3.
   No
?– X is 1-3, Y is 0-2, X =:= Y.
   X = -2
   Y = -2
   Yes

Equalities In Prolog

• We have used three different but related equality operators:
  – X is Y evaluates Y and unifies the result with X: 3 is 1+2 succeeds, but 1+2 is 3 fails
  – X = Y unifies X and Y, with no evaluation: both 3 = 1+2 and 1+2 = 3 fail
  – X =:= Y evaluates both and compares: both 3 =:= 1+2 and 1+2 =:= 3 succeed

• Any evaluated term must be fully instantiated

Example: mylength

mylength([],0).
mylength([\_Tail], Len) :-
    mylength(Tail, TailLen),
    Len is TailLen + 1.

?– mylength([a,b,c],X).
   X = 3
   Yes
?– mylength(X,3).
   X = [\_G266, \_G269, \_G272]
   Yes

Counterexample: mylength

mylength([],0).
mylength([\_Tail], Len) :-
    mylength(Tail, TailLen),
    Len = Taillen + 1.

?– mylength([1,2,3,4,5],X).
   X = 0+1+1+1+1
   Yes
Example: **sum**

\[
\begin{align*}
\text{sum([],0).} \\
\text{sum([Head|Tail],X) :-} \\
& \text{sum(Tail,TailSum),} \\
& \text{X is Head + TailSum.}
\end{align*}
\]

?- sum([1,2,3],X).
X = 6
Yes
?- sum([1,2,5,3],X).
X = 6.5
Yes

Example: **gcd**

\[
\begin{align*}
\text{gcd}(X,Y,Z) :& \text{ --- Note: not just} \\
& X =:= Y, \\
& Z \text{ is } X. \\
\text{gcd}(X,Y,\text{Denom}) :& \text{ --- gcd}(X,X) \\
& X < Y, \\
& \text{NewY is } Y - X, \\
& \text{gcd}(X,\text{NewY},\text{Denom}). \\
\text{gcd}(X,\text{Denom}) :& \text{ --- gcd}(X,X,\text{Denom}) \\
& X > Y, \\
& \text{NewX is } X - Y, \\
& \text{gcd}(\text{NewX},Y,\text{Denom}).
\end{align*}
\]

The **gcd** Predicate At Work

?- gcd(5,5,X).
X = 5
Yes
?- gcd(12,21,X).
X = 3
Yes
?- gcd(91,105,X).
X = 7
Yes
?- gcd(91,X,7).
ERROR: Arguments are not sufficiently instantiated

Example: **factorial**

\[
\begin{align*}
\text{factorial}(X,1) :& \text{ --- } \text{factorial}(X,1) \\
& X =:= 1. \\
\text{factorial}(X,Fact) :& \text{ --- }\text{factorial}(X,Fact) \\
& X > 1, \\
& \text{NewX is } X - 1, \\
& \text{factorial}(\text{NewX},NF), \\
& \text{Fact is } X \times NF.
\end{align*}
\]

?- factorial(5,X).
X = 120
Yes
?- factorial(20,X).
X = 2.4329e+018
Yes
?- factorial(-2,X).
No

A Procedural View

- Three views of Prolog’s execution model
  - Procedural
  - Implementational
  - Abstract

- One way to think of it: each clause is a procedure for proving goals
  - p :- q, r. – To prove a goal, first unify the goal with p, then prove q, then prove r
  - s. – To prove a goal, unify it with s
- A proof may involve “calls” to other procedures
Simple Procedural Examples

\[
p :- q, r.
q :- s.
r :- s.
s.
\]

\[
boolean p() \{ return q() && r(); \}
\]

\[
boolean q() \{ return s(); \}
\]

\[
boolean r() \{ return s(); \}
\]

\[
boolean s() \{ return true; \}
\]

Backtracking

- One complication: backtracking
- Prolog explores all possible targets of each call, until it finds as many successes as the caller requires or runs out of possibilities
- Consider the goal \( p \) here: it succeeds, but only after backtracking

1. \( p :- q, r. \)
2. \( q :- s. \)
3. \( q. \)
4. \( r. \)
5. \( s :- 0=1. \)

Substitution

- Another complication: substitution
- A hidden flow of information

\[
\sigma_1 = \text{MGU}(p(f(Y)) \to \sigma) \text{ is applied to all subsequent conditions in the clause}
\]

\[
\sigma_2 = \text{substitution developed by } q \text{ to prove } \sigma_1(q(Y)) \text{, is applied to all subsequent conditions in the clause}
\]

Resolution

- The hardwired inference step
- A clause is represented as a list of terms (a list of one term, if it is a fact)
- Resolution step applies one clause, once, to make progress on a list of goal terms

Resolution Example

Given this list of goal terms:
\( \{ p(X), s(X) \} \)
And this rule to apply:
\( p(f(Y)) :- q(Y), r(Y). \)
The MGU of the heads is \( \{ X = f(Y) \} \), and we get:
\[
\text{resolution}(\{ p(f(Y)), q(Y), r(Y) \}, \{ p(X), s(X) \}) = \{ q(Y), r(Y), s(f(Y)) \}
\]

A Prolog Interpreter

function solve(goals):
if goals is empty then succeed()
else for each clause c in the program, in order
if head(c) does not unify with head(goals) then do nothing
else solve(resolution(c.goals))
• solve tries each of the four clauses in turn
  – The first works, so it calls itself recursively on the result of the resolution step (not shown yet)
  – The other three do not work: heads do not unify with the first goal term

Program:
1. p(f(Y)) :- q(Y), r(Y).
2. q(g(Z)).
3. q(h(Z)).
4. r(h(a)).

A partial trace for query p(X):
1. solve([p(X)])
   1. solve([q(Y), r(Y)])
   2. nothing
   3. nothing
   4. nothing

2. nothing
3. nothing
4. nothing

A complete trace for query p(X):
1. solve([p(X)])
   1. solve([q(Y), r(Y)])
   2. nothing
   3. nothing
   4. nothing

2. nothing
3. nothing
4. nothing

A complete trace for query p(X), expanded:
1. solve([p(X)])
   1. solve([q(Y), r(Y)])
   2. nothing
   3. nothing
   4. nothing

2. nothing
3. nothing
4. nothing

Collecting The Substitutions

function resolution(clause, goals, query):
    let sub = the MGU of head(clause) and head(goals)
    return (sub(tail(clause) concatenated with tail(goals)), sub(query))

function solve(goals, query)
    if goals is empty then succeed(query)
    else for each clause c in the program, in order
        if head(c) does not unify with head(goals) then do nothing
        else solve(resolution(c, goals, query))

• Modified to pass original query along and apply all substitutions to it
• Proved instance is passed to succeed

Prolog Interpreters

• The interpreter just shown is how early Prolog implementations worked
• All Prolog implementations must do things in that order, but most now accomplish it by a completely different (compiled) technique
Proof Trees

• We want to talk about the order of operations, without pinning down the implementation technique
• Proof trees capture the order of traces of `prove`, without the code:
  – Root is original query
  – Nodes are lists of goal terms, with one child for each clause in the program

Example

Simplifying

• Children of a node represent clauses
• They appear in the order they occur in the program
• Once this is understood, we can eliminate the nothing nodes, which represent clauses that do not apply to the first goal in the list

Example

Prolog Semantics

• Given a program and a query, a Prolog language system must act in the order given by a depth-first, left-to-right traversal of the proof tree
• It might accomplish that using an interpreter like our `prove`
• Or it might do it by some completely different means

Infinite Proof Tree,

`p :- p.`

`p.`

```
prove [p]

prove [p]

prove [p]

prove [p]

prove [p]

prove [p]

prove [p]

prove [p]

prove [p]
```

leftmost path is infinite
A Problem

• All three of the models of Prolog execution we have seen are flawed
• They work on the examples we chose
• On other examples they would not agree with common sense, or with the actual behavior of a Prolog language system
• For instance, reverse([1,2],X)

Variable Renaming

• To avoid capture, use fresh variable names for each clause, every time you apply it
• The first application of reverse might be:
  reverse([Head|Tail],X) :-
  reverse(Tail,Y),
  append(Y,[Head],X).
• And the next might be:
  reverse([Head2|Tail2],X2) :-
  reverse(Tail2,Y2),
  append(Y2,[Head2],X2).
• And so on…
Rename Everywhere

- This renaming step is required for all three of our models of Prolog execution
- Every time a clause is used, it must have a fresh set of variable names
- This implements clause scope as required: the scope of a definition of a variable is the clause containing it