Prolog, part 1
CS 440: Programming Languages and Translators
Lecture 24, Wed 4/24

Prolog and Logic
- Prolog is a programming language based on logic. E.g., there exists a fish Tyler; all fishes are animals; and so on.
- The base of Prolog is **declarative**: Instead of writing programs and arguing that they meet some specification, in Prolog you describe a set of logical facts and rules, make queries, and Prolog tries to prove that your queries follow logically from the given information.
  - E.g., if you want to solve a puzzle, instead of building a data structure for the puzzle and writing a search method to try to find a solution, in Prolog you give the logical rules describing the puzzle and Prolog tries to find a solution.
- Prolog does have some imperative programming-like features, so it's not completely declarative.
  - E.g., to sort a list of values, you could define a predicate that asserts that a list is sorted, another predicate that says one list is a permutation of another, and then make a query that asks for a list that is sorted and a permutation of the unsorted list. Depending on how the notion of permutation is phrased, there might be ways to cut down the set of possible proofs Prolog searches through. Adding these to a Prolog program makes it not purely declarative but more efficient.

First-Order Logic
- Prolog uses first-order predicate calculus: You have primitive predicates \( P(a_1, a_2, \ldots, a_n) \), conjunction (\( \land \)), disjunction (\( \lor \)), negation (\( \neg \)), implication (\( \rightarrow \)), and universal (\( \forall \)) and existential (\( \exists \)) quantifiers.
- **First-order** means that the quantifiers range over values, not predicates, so you might ask if there exists a fish named wanda, but you can't ask if there is a relationship on wanda.
- **Statements** are true or false (also, they can be known to be true or known to be false, or neither). **Axioms** are statements that we take as basic facts, like there exists an empty set. **Rules of inference** describe the relationship "if we know X is true, then we know Y is true". **Assumptions** are statements that we take to be true for the sake of discussion. **Theorems** are statements that are provably true (follow by rules of inference from axioms and postulates).

Basic Datatypes
- **Terms** include
  - **Atoms**, which are are named constants; e.g., eggplant, purple, vegetable.

---

• **Variables** can have values; e.g., X might stand for *eggplant*. Variables begin with an upper case letter or underscore.

• Numbers are integer or floating-point. There are arithmetic operations on numbers, relations such as "less than", and a couple of notions of equality (we'll see later).

• `true` and `false` are the boolean constants.

• **Compound terms** such as `likes(alex, eggplant)`. Here, `likes` is the **functor** (different meaning from Haskell's Functor classtype). It has an **arity** of 2 because it takes two arguments, `alex` and `eggplant`.

• Lists like `[1, alex, likes(alex, eggplant)]` are surrounded by square brackets. `[]` is the empty list.

• Strings "like this" are treated as atoms or as lists of characters.

**Facts and rules**

• A predicate is a compound term or arithmetic or equality relation, such as `needs(eggplant, salt), X < 3, or [A, 3, rhubarb] = [ice, 3, B]`

• A **fact** has the form of a compound term such as `color(eggplant, purple)` or `parent(charlie, joey)`.

• There's also a terminating period.

• Rules have the form of an implication written backward (a conclusion is implied by a body), followed by a period.

  • The conclusion is a single predicate; the body consists of predicates separated by commas.

  • The "is implied by" symbol is `:-`

  • So e.g., `grandparent(charlie, max) :- parent(charlie, X), parent(X, max).`

    • (Charlie is a grandparent of max if there is an X where Charlie is a parent of X and X is a parent of Max.)

  • Another example: `grandparent(A, B) :- parent(A, X), parent(X, B).`

• **Quantification**: A variable that appears in the head is universally quantified; a variable that appears in the body but not the head is existentially quantified.

  • So `grandparent(A, B) :- parent(A, X), parent(X, B).` means "for every A and B, A is a grandparent of B if there exists an X such that A is a parent of X and X is a parent of B".

• **Horn Clause**

  • Written in standard predicate logic form, a Prolog rule looks like `p₁ ∧ p₂ ∧ ... ∧ pₙ → q` or equivalently, `p₁ ∧ p₂ ∧ ... ∧ pₙ ∧ ¬q`. A predicate of this form (a conjunction of logical term where exactly one is negative [negated]) is a **Horn clause**.

  • Prolog facts and rules are Horn clauses. (A fact like `parent(charlie, joey)`. implicitly has a "`:- true" attached to it.)

**Prolog Execution**

• A Prolog program consists of a **database** of facts and rules, plus queries.

  • A query or goal is a logical term followed by a period.
• Prolog tries to prove that a query is true through backtracking search with unification of variables. If it can't find a proof, it comes back with false (or at least, can't be proved). If it does find a proof, it says so and gives the values of variables it used in the proof. Prolog implementations allow you to ask for another proof, so you can look for multiple collections of variable values that make the query true.

• **Backward Chaining**
  • Prolog considers $p_1 \land p_2 \land ... \land p_n \rightarrow q$ to have been proved if the $p_i$'s are facts.
  • If, say $p_1$ is not a fact, then Prolog looks for an instance of $p_1$ in the database. (Either exactly $p_1$ or it is something that unifies with $p_1$.) If it finds $p_1$, then it's done with that and moves on to $p_2$, etc.
  • If $p_1$ is not a fact, then it tries to find $p_1$ as the conclusion of some rule in the database (again, either exactly $p_1$ or unifies with $p_1$). If it finds a rule, say $r_1 \land r_2, ... \land r_m \rightarrow p_1$, then it replaces $p_1$ in our original implication $p_1 \land p_2 \land ... \land p_n \rightarrow q$ with $r_1 \land r_2, ... \land r_m \land p_2 \land ... \land p_n \rightarrow q$. (If $m > 1$, then we have more things to prove but presumably things that are easier to prove.)
    • (This replacement is called the resolution principle.)
  • If $p_1$ doesn't unify with any fact or rule in the database, then Prolog has to backtrack.
    • It unapplies the last fact or rule application (including undoing any unification it did) and tries to find a different proof of the term the fact or rule that application used. (If there is no such way to backtrack, then the overall proof attempt has failed.)
  • Conceptually, Prolog maintains a stack of goals already proved (along with the fact or rule that it applied plus any unification substitution it used), so backtracking consists of popping off the top of the stack so that our new goal is the one we just popped off, and the proof search begins with the next database item after the fact or rule we just popped off. (When the query is started, the stack is empty and our set of goals is just the query itself. If the set of unproved goals is empty, we've found a proof.)

**Declarative vs Imperative Steps**
• When Prolog looks for a fact or rule to apply to the current goal, it searches the database top-to-bottom, so facts and rules earlier in the database take priority over later facts and rules. This makes the search not purely declarative.
• Also, when Prolog tries to prove the goals, it goes left-to-right through the body of each rule. This means that a recursive property rule should begin with the base case, not the recursive case otherwise you might get an infinite loop: `reln(…) :- base(…), reln(…).` , not `reln(…) :- reln(…), base(…).`.
• Arithmetic expressions can be compared textually or after evaluation: $e_1 = e_2$ means the two expressions unify; $e_1 =:= e_2$ means their values are equal.
  • $2+2 = 4$ is false but $2+2 =:= 4$ is true. Similarly, $1+3 = 3+1$ is false but $1+3 =:= 3+1$ is true.
• Note: $X+2 = 3+Y$ if $X = 3$ and $Y = 2$, but $X+2 =:= 3+Y$ fails if we don't already have values for $X$ and $Y$.
  • So you can't ask "Are there values for $X$ and $Y$ that make $X+2$ and $3+Y$ evaluate to the same result?" unless you have a rule that describes the sets of possible values for $X$ and $Y.
Running Prolog on SWI

- SWI Prolog
- Easy-to-use prolog environment; access prolog through a web page
- SWI Prolog: http://www.swi-prolog.org/ > Try SWI-Prolog online > Create a Program
- Left-hand window ("Your Prolog rules and facts go here ...")
- Bottom right-hand window ("Your query goes here ...")
  - Press control-enter to execute query
- Top right-hand window contains results
  - If you get a result, you can press Next to find another one.
  - E.g., with ice_cream example above, first solution is X = charlie; pressing Next gives finley, etc.
  - To get rid of a result (frees up memory), press circled x at top right of result.
- To get rid of all results, find the triple bar icon at the top right of the top right window (the one with the owl).
  Select Clear to clear everything

Random Examples

Likes and dislikes

likes(sam, pizza).
likes(charlie, ice_cream).
likes(mr_whiskers, chicken).
likes(finley, pizza).
likes(finley, ice_cream).
likes(finley, chicken).
likes(X,ice_cream) :- someone(X). % everyone likes ice cream
someone(sam).
someone(charlie).
someone(finley).
someone(mr_whiskers).

% Query
likes(X,ice_cream).

Without the rule likes(X,ice_cream) :- someone(X), we'd say only that charlie and finley like ice_cream.

Standard parent / ancestor example

% a is the parent of b
parent(a,d).
parent(a,k).
parent(k,l).
parent(k,m).
parent(b,e).
parent(b,f).
parent(f,g).
parent(f,h).
parent(f,i).

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).

% note swapping ancestor and parent in previous rule causes infinite loop.

List of unique values
% list contains different values (no repetitions)?
%
different([]).
different([H|T]) :- not_in(H,T), different(T).

not_in(_,[]).
not_in(X,[Y|Z]) :- X \= Y, not_in(X,Z).

% Lists are in square brackets
% [Y | Z] is Y consed onto Z ( Y : Z in H.)
% X = Y means X unifies with Y
% X \= Y means X does not unify with Y
% %
in(X,[X|_]).
in(X,[_|Y]) :- in(X,Y).

% You can ask for the AND of terms:
in(X,[3,6]), not_in(X,[2,5]). % is a query solved by X=3 and by X=6

Range check
• The between predicate below checks for $Y \leq X \leq Z$, but it also will generate all the values between $Y$ and $Z$ when asked repeatedly for $X$'s.
  \[
  \text{between}(X,Y,_) :- X \text{ is } Y.
  \]
  \[
  \text{between}(X,Y,Z) :- Y < Z, \text{ Yp1 is } Y+1, \text{ between}(X,\text{Yp1},Z).
  \]
• Here's a slight variation that also works but illustrates how building up terms and doing calculations differ. If you ask for more results from \text{between1}, the recursive calls build terms $Y+1, Y+1+1, Y+1+1+1, \text{etc}$. It's not obvious that this happens partly because the terms are evaluated to actual values before they're returned.
  \[
  \text{between1}(X,Y,_) :- X \text{ is } Y.
  \]
between1(X, Y, Z) :- write(Y), nl, Y < Z, between1(X, Y+1, Z). % (uses Y+1 instead)