Logic Programming

- Imperative programming models: compute via iteration and side effects
- Functional programming model: compute via recursion and substitution of parameters into functions
- Logic programming model: compute via resolution of logical statements, driven by the ability to unify variables and terms

Functional Programming

- E.g., Lisp, Scheme
- Formalism: Church's lambda calculus
- Key idea: no mutable state/side effects; everything done by composing functions

Functional Programming Design Features and Issues

- First-class and higher-order functions
- Polymorphism
- Recursion
- Garbage collection
- Control flow and evaluation order
- Support for list-based data

Logic Programming Model

- E.g., Prolog
- Formalism: Predicate calculus
- Key idea: collection of axioms from which theorems can be proven

Logic Programming Design Issues

- Horn clauses and terms
- Resolution and unification
- Search and execution order
- List manipulation
- High-order predicates for inspection and modification of the database

Horn Clauses

- Consists of a head consisting of term *H* and a body consisting of terms B_i
  - $H \leftarrow B_1, B_2, \ldots, B_n$
- *H* is true if $B_1, B_2, \ldots, B_n$ are all true
- Terms can be constants ("Rochester is rainy") or predicates applied to atoms or to variables (called a structure)
  - Constant is an atom or number or quoted string
  - Variable takes on values at run time
  - Structures consist of an atom called the functor and a list of arguments
    - Can be thought of as either a logical predicate or a data structure

Running Prolog

- The Prolog interpreter has a collection of facts and rules (clauses) in its database
  - Facts: axioms – assumed true (Horn clause without a right-hand side)
  - Rules: theorems – provably true, allows inference
- Run by asking the interpreter a question
  - a hypothesis or goal or query (Horn clause with an empty left-hand side)
  - Done by stating a theorem (asserting a predicate) that the interpreter tries to prove

How is a predicate satisfied?

- Unification – process by which compatible statements are merged (instantiating variables or determining their equivalence)
  - Equality - the goal 'A = B' or '=(A, B)' succeeds if and only if A and B can be unified
- Resolution – substitution of one clause inside another when its head unifies with one of the terms in the body of the other
  - Does not generally distinguish between input and output arguments (as opposed to imperative or functional languages that apply functions to arguments to generate results)

Arithmetic

- Built-in functor "is"

  is(X, 1+2).

  or

  X is 1+2.

  X = 3

Unification Rules for Prolog

- A constant unifies with itself
- Two structures unify if and only if they have the same functor and the same number of arguments, and the corresponding arguments unify recursively
- A variable unifies with anything and is instantiated if the other thing has a value or associated (considered equivalent) if not

List manipulation

- [a,b,c]
- Optional vertical bar that separates the tail of the list
- E.g.

  member(X, [X|T]).

  member(X, [H|T]) :- member(X, T).

Search/Execution Order

- Backward chaining – start with the goal and work backward (e.g., Prolog)
- Forward chaining – start with existing clauses and work forward

Imperative Control Flow

- The *cut* ('!') predicate – zero-argument predicate that prevents a goal or sub-goals to the left from succeeding more than once
- The *not* predicate – built using call (satisfy P as a goal), cut, and fail – not(P) succeeds if the interpreter is unable to prove P
  - Call can be used to execute new pieces of the program written on the fly (Prolog is homoiconic, as is Scheme – can represent itself)

Database Manipulation

- assert – built-in predicate to add clauses
- retract – built-in predicate to remove clauses

Example: Sorting

```
sort(L1, L2) :- permutation(L1, L2), sorted(L2).
sorted([]).
sorted([X]).
sorted([X|[Y|L]]) :- X=<Y, sorted([Y|L]).
permutation(L, [H|T]) :- append(V, [H|U], L),
   append(V, U, W), permutation(W, T).
permutation([], []).
append([], L, L).
append([H|T], L, [H|L2]) :- append(T, L, L2).
```

Example: Quicksort

```
quicksort([], []).
quicksort([A|L1], L2) :- partition(A, L1, P1,
   S1), quicksort(P1, P2), quicksort(S1, S2),
   append(P2, [A|S2], L2).
partition(A, [], [], []).
partition(A, [H|T], [H|P], S) :- A >= H,
   partition(A, T, P, S).
partition(A, [H|T], P, [H|S]) :- A =< H,
   partition(A, T, P, S).
```

Example: tic tac toe

```
ordered_line(1,2,3).
ordered_line(4,5,6).
ordered_line(7,8,9).
ordered_line(1,4,7).
…
line(A,B,C) :- ordered_line(A, B, C).
line(A, B, C) :- ordered_line(A, C, B).
…
empty(A) :- not x(A), not o(A)

…
```

Followed by rules for next move, ordering of which is important