Logic Programming

- Based on *first-order predicate calculus*
- Operators
  - Conjunction, disjunction, negation, implication
- Universal and existential quantifiers
  - $\forall x$ – for all $x$ ...
  - $\exists x$ – for some $x$ ...
- Statements
  - Sometimes true, sometimes false, often unknown
  - Axioms - assumed true
  - Theorems - provably true
  - Hypotheses (goals) - things we'd like to prove true
Logic Programming Concepts

- If we make certain restrictions on the format of statements you can prove theorems mechanically
  - We insist that all statements be in the form of Horn Clauses
  - The language implementation finds a collection of axioms and inference steps that imply the goal

- A Horn Clause consists of a Head and a Body
  \[ H \leftarrow B_1, B_2, \ldots B_n \]

  - A head, H, is a consequent term, a body consists of terms, \( B_i \)
  - Meaning is “H if \( B_1, B_2, \ldots \) and \( B_n \)”
Logic Programming Concepts (cont’d)

- **Resolution**
  - To derive a new statement, a logic programming system combines existing statements and *deduces* statements

\[
C \leftarrow A, B \\
E \leftarrow C, D \\
E \leftarrow A, B, D
\]

- **Unification**
  - During resolution, free variables may acquire values through unification with expressions in matching terms

\[
\text{flowery}(X) \leftarrow \text{rainy}(X) \\
\text{rainy(suwon)} \\
\text{flowery(suwon)}
\]
Horn Clauses in Prolog

A statement in the form of Horn clause

- Means that the conjunction of the terms in the body implies the head ($H \leftarrow B_1, B_2, \ldots, B_n$ means $H$ if $B_1, B_2, \ldots$ and $B_n$)
  - A clause with an empty body is called a fact
  - A clause with both sides is a rule
  - A clause with an empty head is a query, or top-level goal

The Prolog interpreter has a collection of facts and rules in its database

- Facts are axioms - things the interpreter assumes to be true
- Using inferences with rules, resolve the query
Prolog

- Prolog can be declarative or imperative
  - We’ll emphasize the declarative semantics for now, because that's what makes logic programming interesting
  - We'll get into the imperative semantics later

- Prolog allows you to state a list of axioms
  - Then you pose a query (goal) and the system tries to find a series of inference steps (deductions and unifications) that allow it to prove your query starting from the axioms
Atoms, Variables, Structures in Prolog

- **Atoms**
  - Identifier beginning with a lowercase letter, a sequence of “punctuation” characters, or a quoted character string
    
    foo    my_Const    +    'Hi, Mom'

- **Variables**
  - Identifier beginning with a uppercase letter
    
    Foo    My_var    X

- **Structures**
  - Consists of an atom (*functor*) and a list of arguments
    
    rainy(seoul)    teach(prof, pl)
    bin_tree(foo, bin_tree(bar, glarch))
  - Logically, structures are *predicates* (part of clause except the subject)
Facts, Rules in Prolog

- **Facts**
  - Horn clauses without right sides
    
    sunny(suwon).

- **Rules** are theorems that allow the interpreter to infer things
  - To be interesting, rules generally contain variables
    
    employed(X) :- employs(Y,X).

    can be read:
    
    for all X, X is employed
    
    if there exists a Y such that Y employs X

- **Note the direction of the implication**
  
  $A :- B$ means $A \leftarrow B$
Variables in Prolog

- Scope of a variable is the clause in which it appears
  - Variables whose first appearance is on the left hand side of the clause have implicit **universal quantifiers** (for all, $\forall$)
  - Variables whose first appearance is in the body of the clause have implicit **existential quantifiers** (there exist, $\exists$)

Another example

- $\text{grandmother}(A, C) : - \text{mother}(A, B), \text{mother}(B, C)$.

  for all A, C [A is the grandmother of C if there exists a B such that A is the mother of B and B is the mother of C].
Queries in Prolog

To run a Prolog program

- One asks the interpreter a question by stating a theorem which the interpreter tries to prove
- A query is entered with a special `?-`
  - The interpreter says yes or no
  - If your predicate contains variables, the interpreter prints the values which make the predicate true

```prolog
rainy(seoul).
rainy(suwon).
?- rainy(C).
C = seoul ; % find an answer which comes first in database
C = suwon ; % typing a semicolon(;) gives the next answer
No % gives No if no more answers
```
Resolution in Prolog

Resolution principle

If $C_1$ and $C_2$ are Horn clauses and the head of $C_1$ matches with one of the terms in the body of $C_2$, we can replace the term in $C_2$ with the body of $C_1$.

\[
\begin{align*}
A & \leftarrow B, C \\
X & \leftarrow A, D \\
X & \leftarrow B, C, D
\end{align*}
\]

\text{takes(jane, cs320).}
\text{takes(john, cs320).}
\text{classmate(X, Y) :- takes(X, Z), takes(Y, Z).}

\text{X=jane, Z = cs320}

\text{classmate(jane, Y) :- takes(jane, cs320), takes(Y, cs320)}

\text{Resolution Principle}

\text{classmate(jane, Y) :- takes(Y, cs320).}
Unification in Prolog

- **Unification**
  - The pattern matching process used to associate a variable with a specific value (e.g., associate \( X \) with \( jane \))
  - A variable is given a value as a result of unification. Then we say the variable is instantiated

- **Unification rules**
  - A constant unifies only with itself
  - Two structures unify if and if only they have the same *functor* and the same *arity*, and the corresponding arguments unify recursively
  - A variable unifies with anything (If the other thing has a value, the variable is instantiated)
Equality in Prolog

- Equality is defined in terms of unifiability
  - \( = (A, B) \) succeeds if and if only \( A \) and \( B \) can be unified
  - We can also write \( A = B \), instead of \( =(A, B) \)

```
?- a = a.
Yes
?- a = b.
No
?- foo(a,b) = foo(a,b).
Yes
?- X = a.
X = a ;  % unified only once with constant
No
```
Arithmetic in Prolog

- Arithmetic operators are predicates not functions
  - +(2, 3) or 2 + 3 is a structure with arity 2
    - ?- (2 + 3) = 5.
      - No % the structure (2 + 3) will not unify with 5

- Built-in predicate is
  - is unifies its first argument with the arithmetic value of the second argument
    - ?- is(X, 1+2).
      - X = 3
    - ?- X is 1+2.
      - X = 3
    - ?- 1+2 is 4-1.
      - No % the first arg (1+2) is already instantiated
    - ?- X is Y.
      - ERROR % the second arg must be already instantiated
Prolog Interpreter

- The interpreter works by *Backward Chaining*
  - Begins with the thing to prove and works backwards looking for things that would imply it, until it gets to facts

- Work forward from the facts
  - In theory, it is possible to see if any of the things you can prove from the facts
  - But can be very time-consuming
  - Fancier logic languages use both kinds of chaining with special smarts or hints from the user to bound the searches
Prolog Interpreter

- Steps to find the goal
  - Starts at the beginning of your database (which contains facts and rules in the order specified)
  - Searches facts or rules with which to the current goal can be unified
    - If it finds a fact, great; it succeeds
    - If it finds a rule, it attempts to satisfy the terms in the body of the rule in depth first and left-to-right
  - The term to satisfy is the subgoal and start at the beginning of the database and searches facts or rules again
  - If a subgoal fails, returns to the previous goal and attempts to satisfy in a different way (backtracking)
Backtracking Search in Prolog

facts, rules
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).

?- snowy(C).  % goal

Original goal

Candidate clauses

Subgoals

Candidate clauses

X = seattle
rainy(X) OR

cold(seattle) fails; backtrack

X = rochester
rainy(rochester) cold(rochester)
Search Tree in Prolog

- Search tree
  - The root is the top-level goal and the leaves are facts
  - The children of the goal (or the subgoal) are all the facts and the heads of the rules with which the goal can unify
    - The interpreter does an OR across them: one of them must succeed in order for the goal to succeed
  - The children of a head are the terms in the body of the rule
    - The interpreter does an AND across these: all of them must succeed in order for parent to succeed
  - The overall search tree then consists of alternating AND and OR levels
Search Order in Prolog

- Prolog is **not purely declarative**
  - The ordering of the database and the left-to-right pursuit of sub-goals gives a deterministic imperative semantics to searching and backtracking
- Changing the order of statements in the database can give you different results
  - It can lead to infinite loops
  - It can certainly result in inefficiency
Infinite Regression in Prolog

edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).

?- path(a, a).

\[ X_1 = a, Y_1 = a \]
\[ X_2 = X_1, Y_2 = Y_1, Z_1 = ? \]
\[ X_3 = X_2, Y_3 = Y_2 \]
\[ X_4 = X_3, Y_4 = Y_3, Z_2 = ? \]
\[ \ldots \]
\[ \ldots \]
SWI-Prolog

- Prolog interpreter in Linux
  - Install SWI-prolog from “Interpreters”

```bash
$ pl
?- [user].
| : factorial(0,1).
| : factorial(A,B) :- A > 0, C is A-1, factorial(C, D), B is A*D.
| : <ctrl-D>
Yes
?- factorial(5, What).

What = 120 ;
No
?- ['myprog.pl']. % load a file which contains facts and rules
?- halt.
$
```