Chapter 10:
Logic Programming Languages

Principles of Programming Languages

Contents

- Predicate Calculus and Proving Theorems
- The Basic Elements of Prolog
- Simple Arithmetic and List in Prolog
- Deficiencies of Prolog
- Applications of Logic Programming

Introduction

- Another programming methodology:
  - Express programs in a form of symbolic logic
  - Use a logical inferencing process to produce results
- Prolog

Proposition

- Proposition: a statement that may or may not be true
  - James I is a male
  - Catherine is a female
  - James I is parent of Charles I
  - Catherine is parent of Sophia
- Compound term: functor + parameter lists

  male(james1)

  female(catherine)

  parent(james1, charles1)

  parent(catherine, sophia)
- Two modes: facts or queries

Logic Operators

$\neg a$	negation
$a \wedge b$	conjunction
$a \vee b$	disjunction
$a \rightarrow b$	implication
$a \leftrightarrow b$	equivalence

- Examples:

  $(a \wedge b) \rightarrow c$

Predicates

- When we employ variables, the truth value of a statement is unknown until the variable has a specific value
  - X is a male
  - male(X)
- Predicates turn into propositions if they have quantifiers

$$\forall X\,(\text{woman}(X) \rightarrow \text{human}(X))$$

$$\exists X\,(\text{mother}(\text{mary}, X) \wedge \text{male}(X))$$

Clausal Form

- A standard form for propositions

$$B_1 \vee B_2 \vee \ldots \vee B_n \leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_m$$

- Examples

mother(catherine, sophia) ← parent(catherine, sophia) ∧ female(catherine)

father(louis, al) ∨ father(louis, violet) ← father(al, bob) ∧ mother(violet, bob) ∧ grandfather(louis, bob)

- RHS = antecedent
- LHS = consequent

Automatic Theorem Proving

- Resolution: if we have

  older(joanne, jake) ← mother(joanne, jake)

  wiser(joanne, jake) ← older(joanne, jake)
- Using resolution, we can construct

  wiser(joanne, jake) ← mother(joanne, jake)
- Mechanics:
  1. AND two LHS to make the new LHS
  2. AND two RHS to make the new RHS
  3. Remove any terms that appear in the both sides

Questions

father(bob, jake) ∨ mother(bob, jake) ← parent(bob, jake)

grandfather(bob, fred) ← father(bob, jake) ∧ father(jake, fred)

What can we infer?

Horn Clauses

- When propositions are used for resolution, only a restricted kind of clausal form can be used
- There are two forms of Horn clauses:
  - Headed: They have single atomic proposition on the LHS

    $likes(bob, trout) \leftarrow likes(bob, fish) \wedge fish(trout)$

  - Headless: Empty LHS

    $father(bob, jake)$

Resolution

- Ability to detect any inconsistency in a given set of propositions
- Theorem is proved by
  - We are given a set of pertinent propositions
  - Negation of theorem as a new proposition
  - Resolution is used to find inconsistency
- *Hypotheses*: a set of original propositions
- *Goal*: negation of theorem stated as a proposition

An Overview of Logic Programming

- Declarative, rather that procedural or functional
- Specify **relation** between objects
  - larger(3, 2)
  - father(tom, jane)
- Separate logic from control:
  - Programmer declares **what** facts and relations are true
  - System determines **how** to use facts to solve problems
  - System **instantiates** variables in order to make relations true

Example: Sorting a List

- Describe the characteristics of a sorted list, not the process of rearranging a list

$$\text{sort}(\text{old\_list}, \text{new\_list}) \leftarrow \text{permute}(\text{old\_list}, \text{new\_list}) \wedge \text{sorted}(\text{new\_list})$$

$$\text{sorted}(\text{list}) \longleftarrow \forall j \text{ such that } 1 \leq j < n, \text{list}(j) \leq \text{list}(j+1)$$

The Origins of Prolog

- Alain Colmerauer at Marseille in 1970s
  - Natural language processing
- University of Edinburgh
  - Automated theorem proving
- 1981, Japanese Fifth Generation Computing Systems choose Prolog as the basis
- Despite the great assumed potential of logic programming and Prolog, little great significance had been covered

Facts

```
male(charlie).
male(bob).

female(alice).
female(eve).

parent(charlie, bob).
parent(eve, bob).
parent(charlie, alice).
parent(eve, alice).
```

- Headless Horn clauses
- Every Prolog statement is terminated by a period

Rules

- Headed Horn clauses

$$\text{mother}(X, Y) \leftarrow \text{female}(X) \wedge \text{parent}(X, Y)$$

```
mother(X, Y) :- female(X), parent(X, Y).
```

Queries

- Let the computer know the facts, then ask it some questions – queries
- Ask the system to prove or disprove a theorem

```
?- mother(sophia, george1).
true
```

- Ask the system to identify the instantiation of variables that make the query true

```
?- mother(X, sophia).
X = Elizabeth
false
```

Some Prolog Syntax

- Layer 0: **constants** and **variables**
- Constants are:
  - Atoms: any string consisting of alphanumeric characters and underscore (_) that begins with a lowercase letter
  - Numbers
- Variables are strings that begin with "_" or an uppercase letter

Some Prolog Syntax

```
const     -> atom | number
var       -> ucase string | _ string
atom      -> lcase string
letter    -> ucase | lcase
string    -> epsilon | letter string
          | number string | _ string
ucase     -> A | ... | Z
lcase     -> a | ... | z
number    -> ...
```

Some Prolog Syntax

- Layer 1: **terms**
- These are inductively defined:
  - Constants and variables are terms
  - Compound terms – applications of any *n*-ary **functor** to any *n* terms – are terms
  - Nothing else is a term

Some Prolog Syntax

- Layer 2: **atomic formulae**
- These consist of an *n*-ary relation (also called **predicate**) applied to *n* terms
- Formulae are either true or false; terms denote entities in the world
- In Prolog, atomic formulae can be used in three ways:
  - As **facts**: in which they assert truth
  - As **queries**: in which they enquire about truth
  - As components of more complicated statements

Some Prolog Syntax

```
term -> const | var |
        functor “(“ term (“,” term)* “)”

pred -> pname “(“ term (“,” term)* “)”
```

Prolog Queries

- A **query** is a proposed fact that is to be proven
  - If the query has no variables, returns true/false
  - If the query has variables, returns appropriate values of variables (called a substitution)

```
?- male(charlie).
true

?- male(eve).
false

?- female(Person).
Person = alice;
Person = eve;
false
```

Terms

- Horn Clause = Clause
- Consequent = Goal = Head
- Antecedents = Subgoals = Tail
- Horn Clause with No Tail = Fact
- Horn Clause with Tail = Rule

Some Prolog Syntax

```
clause-> pred . |
            pred :- pred (“,” pred)* .

term -> const | var |
        functor “(“ term (“,” term)* “)”

pred -> pname “(“ term (“,” term)* “)”

const -> ...
var  -> ...
```

Variables

- Variables may appear in the LHS or RHS of a clause. Prolog interprets free variables of a rule **universally**

  ```
  mother(X) :- female(X), parent(X, Y).
  ```

  In first-order logic:

  $\forall X \text{ mother}(X) \leftarrow \exists Y \text{ parent}(\text{X}, \text{Y}) \wedge \text{female}(\text{X})$

- Note that the Prolog **query**

  ```
  ?-q(X1, ..., Xn)
  ```

  means

  $\exists X1,...,Xn \text{ q}(\text{X1}, \ldots, \text{Xn})$

Execution of Prolog Programs

- **Unification**: variable bindings
- **Backward Chaining/Top-Down Reasoning/Goal Directed Reasoning**: reduces a goal to one or more subgoals
- **Backtracking**: systematically searches for all possible solutions that can be obtained via unification and backchaining.

Unification

- Two atomic formulae unify iff they can be made syntactically identical by replacing their variables by other terms
- For example:
  - `parent(bob,Y)` unifies with `parent(bob,sue)` by replacing `Y` by `sue`

```
?-parent(bob,Y)=parent(bob,sue)
Y = sue ;
false
```

  - `parent(bob,Y)` unifies with `parent(X,sue)` by replacing `Y` by `sue` and `X` by `bob`

```
?-parent(bob,Y)=parent(X,sue).
Y = sue
X = bob ;
false
```

Unification Algorithm

Equation	Action
$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$	Replace with $s_1 = t_1, \ldots, s_n = t_n$
$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_n)$ where $f \neq g$	Halt with failure
$X = X$	Delete the equation
$t = X$ where t is not a variable	Replace with $X = t$
$X = t$ where X does not occur in t and it occurs elsewhere	Apply the substitution $X = t$ to all other terms
$X = t$ where X occurs in t (but $t \neq X$)	Halt with failure

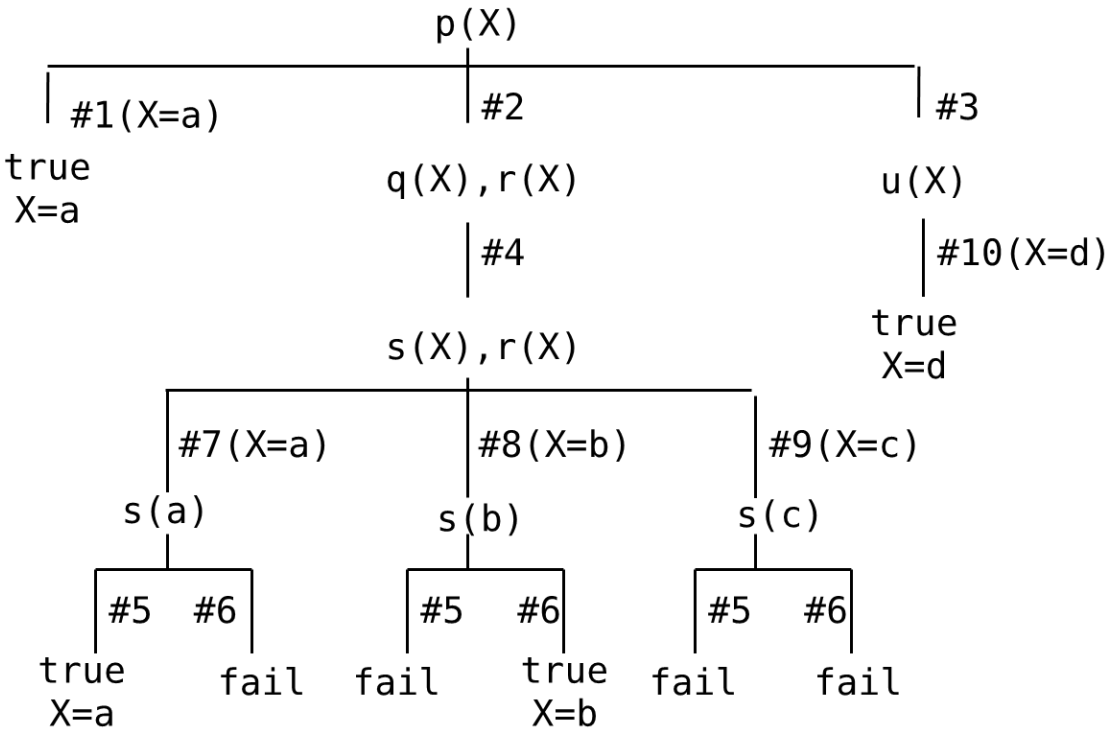
Prolog Search Trees

- Encapsulate unification, backward chaining, and backtracking
  - Internal nodes are ordered list of subgoals
  - Leaves are success nodes or failures, where computation can proceed no further
  - Edges are labeled with variable bindings that occur by unification
- Describe **all possible computation** paths
  - There can be many success nodes
  - There can be infinite branches

Prolog Search Tree

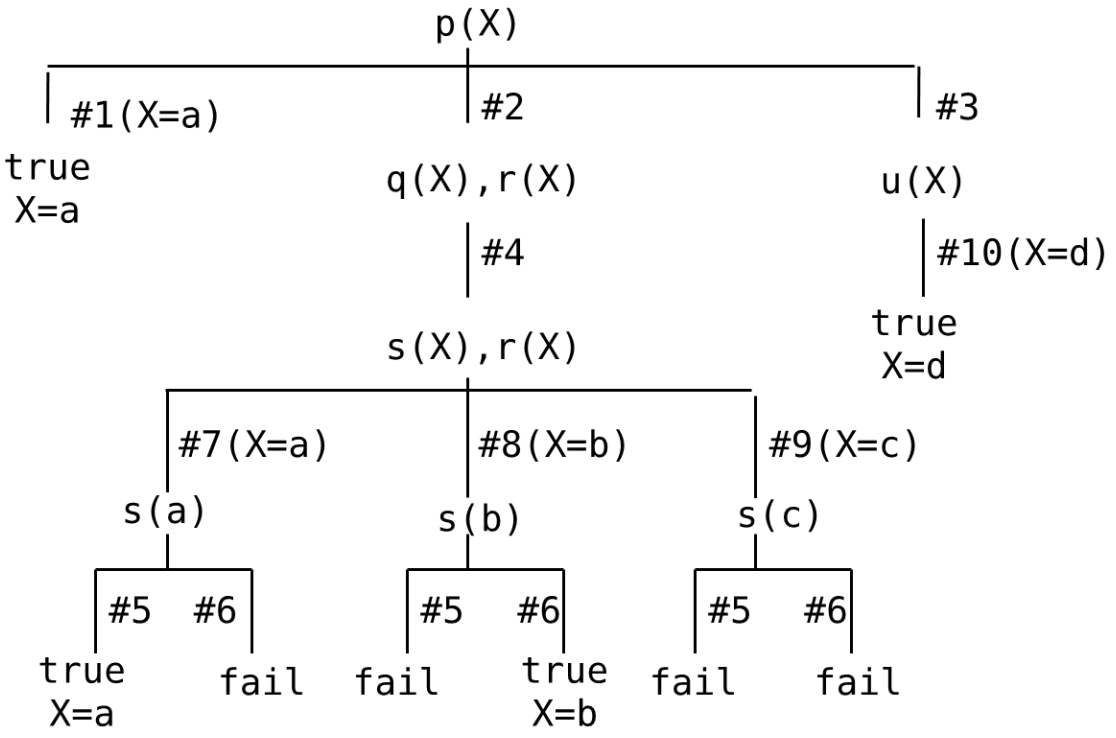
```
/* program P clause # */

p(a). /* #1 */
p(X) :- q(X), r(X). /* #2 */
p(X) :- u(X). /* #3 */

q(X) :- s(X). /* #4 */

r(a). /* #5 */
r(b). /* #6 */

s(a). /* #7 */
s(b). /* #8 */
s(c). /* #9 */

u(d). /* #10 */
```

?- p(X)



```
[trace]  ?- p(X).
   Call: (7) p(_G312) ? creep
   Exit: (7) p(a) ? creep

X = a ;
   Redo: (7) p(_G312) ? creep
   Call: (8) q(_G312) ? creep
   Call: (9) s(_G312) ? creep
   Exit: (9) s(a) ? creep
   Exit: (8) q(a) ? creep
   Call: (8) r(a) ? creep
   Exit: (8) r(a) ? creep
   Exit: (7) p(a) ? creep

X = a ;
   Redo: (9) s(_G312) ? creep
   Exit: (9) s(b) ? creep
   Exit: (8) q(b) ? creep
   Call: (8) r(b) ? creep
   Exit: (8) r(b) ? creep
   Exit: (7) p(b) ? creep

X = b ;
   Redo: (9) s(_G312) ? creep
   Exit: (9) s(c) ? creep
   Exit: (8) q(c) ? creep
   Call: (8) r(c) ? creep
   Fail: (8) r(c) ? creep
```

```
Redo: (7) p(_G312) ? creep
   Call: (8) u(_G312) ? creep
   Exit: (8) u(d) ? creep
   Exit: (7) p(d) ? creep

X = d ;

No
```



Approaches

- Prolog implementations use **backward chaining**
- 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
- 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

Subgoal Strategies

- When goal has more than one subgoal, can 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

Backtracking

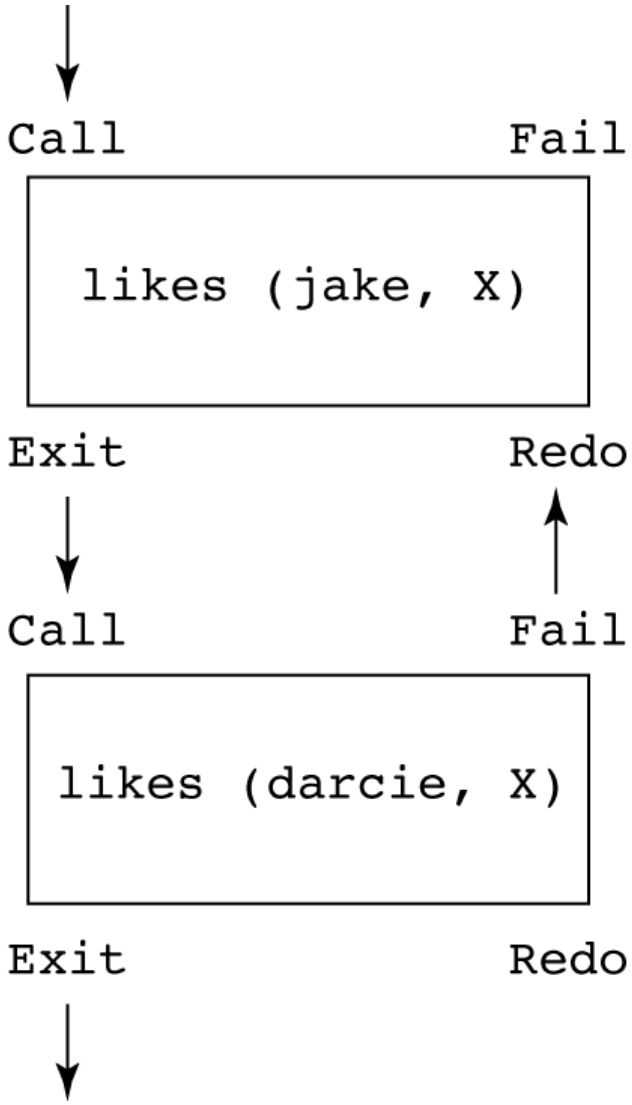
- With a goal with multiple subgoals, if fail to show truth of one of subgoals, reconsider previous subgoal to find an alternative solution: **backtracking**
- Begin search where previous search left off
- Can take lots of time and space because may find all possible proofs to every subgoal

Trace

- Built-in structure that displays instantiations at each step
- *Tracing model* of execution - four events:
  - *Call* (beginning of attempt to satisfy goal)
  - *Exit* (when a goal has been satisfied)
  - *Redo* (when backtrack occurs)
  - *Fail* (when goal fails)

Example

```
likes(jake,chocolate).
likes(jake,apricots).
likes(darcie,licorice).
likes(darcie,apricots).

trace.
likes(jake,X),
likes(darcie,X).
```



Simple Arithmetic

- Prolog supports integer variables and integer arithmetic
- `is` operator: takes an arithmetic expression as right operand and variable as left operand

  ```
  A is B / 17 + C
  ```

- Not the same as an assignment statement!

Example

```
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.
```

List Structures

- *List* is a sequence of any number of elements
- Elements can be atoms, atomic propositions, or other terms (including other lists)

```
[apple, prune, grape, kumquat]
[]          (empty list)
[H | T]     (head H and tail T)
```

Append Example

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3]) :-
                    append (List_1, List_2, List_3).
```

Another Example

```
whatisme([], []).
whatisme([Head | Tail], List) :-
          whatisme(Tail, Result),
          append(Result, [Head], List).
```

- Try to write `member` rule to check if an element is a member of a list

Deficiencies of Prolog

- Resolution order control
  - Prolog always matches in the same order – user can control the ordering
  - Prolog allows explicit control of backtracking via cut operator -- “ ! ”, which is actually a goal, not an operator
    - Always succeed but cannot be resatisfied through backtracking

```
a, b, !, c, d
```

  - It is detrimental to one of the important advantages of logic programming – programs do not specify how solutions are to be found.

Deficiencies of Prolog

- The closed-world assumption
  - In Prolog, the truths are those that can be proved using its database
  - If there is insufficient information in the database to prove a query, it is not actually false, it fails
  - It relates to negation problem

Deficiencies of Prolog

- The negation problem
  - Stating that two atoms are not equal is not straightforward in Prolog
  - Bad solution: adding facts stating that they are not the same
  - Alternative: state in the goal that X must not be the same as Y

```
sibling(X, Y) :- parent(M, X), parent(M, Y), not(X=Y)
```

  - However, not operator is not logical NOT, not(not(some_goal)) is not equivalent to some_goal
  - **Reason**: the use of Horn clause form prevents any negative conclusions

Deficiencies of Prolog

- Intrinsic limitations
  - Prolog has no idea of how to sort, other than simply to enumerate all permutations of the given list until it happens to create the one that has the list in sorted order
  - We must specify the details of how that sorting can be done – not logic programming any more
  - Resolution is not capable of doing this sorting

Applications of Logic Programming

- Relational database management systems
  - Logic programming is a natural match to the needs of implementing a RDBMS

    SQL is goal, tables are facts
- Expert systems
  - Computer systems designed to emulate human expertise in some particular domain
  - In addition to the initial knowledge base, ES can learn from the process of being used
- Natural language processing

Summary

- Symbolic logic provides basis for logic programming
- Logic programs should be nonprocedural
- Prolog statements are facts, rules, or goals
- Resolution is the primary activity of a Prolog interpreter
- Although there are a number of drawbacks with the current state of logic programming it has been used in a number of areas