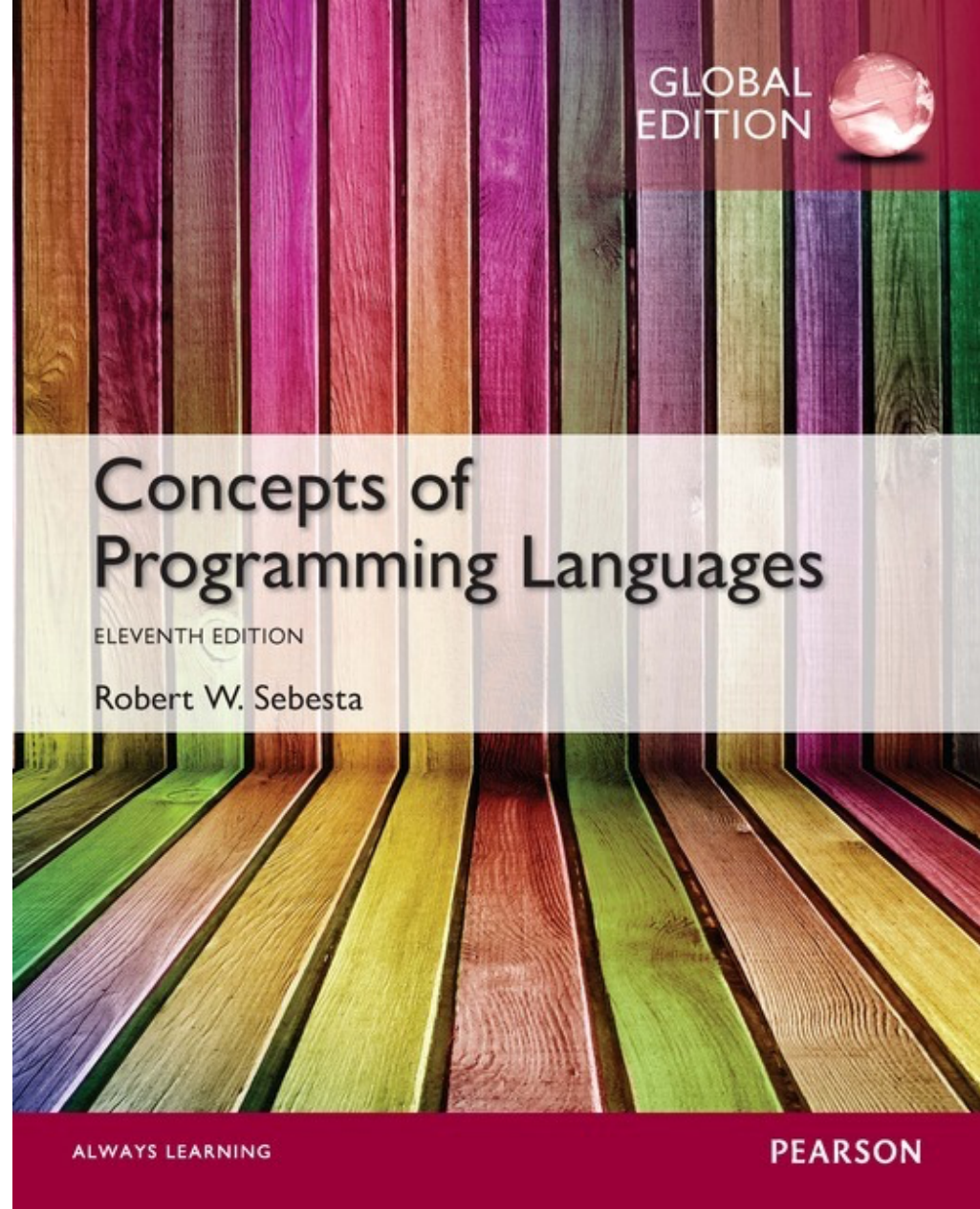


# Chapter 16

## Logic Programming Languages



# Chapter 16 Topics

---

- Introduction
- A Brief Introduction to Predicate Calculus
- Predicate Calculus and Proving Theorems
- An Overview of Logic Programming
- The Origins of Prolog
- The Basic Elements of Prolog
- Deficiencies of Prolog
- Applications of Logic Programming

# Introduction

---

- Programs in logic languages are expressed in a form of symbolic logic
- Use a logical inferencing process to produce results
- *Declarative* rather than *procedural*:
  - Only specification of *results* are stated (not detailed *procedures* for producing them)

# Proposition

---

- A logical statement that may or may not be true
  - Consists of objects and relationships of objects to each other

# Symbolic Logic

---

- Logic which 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
- Particular form of symbolic logic used for logic programming called *predicate calculus*

# Object Representation

---

- Objects in propositions are represented by simple terms: either constants or variables
- *Constant*: a symbol that represents an object
- *Variable*: a symbol that can represent different objects at different times
  - Different from variables in imperative languages

# Compound Terms

---

- *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

# Parts of a Compound Term

---

- Compound term composed of two parts
  - Functor: function symbol that names the relationship
  - Ordered list of parameters (tuple)
- Examples:

```
student(jon)
```

```
like(seth, OSX)
```

```
like(nick, windows)
```

```
like(jim, linux)
```



# Forms of a Proposition

---

- 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

---

Name	Symbol	Example	Meaning
negation	$\neg$	$\neg a$	not a
conjunction	$\cap$	$a \cap b$	a and b
disjunction	$\cup$	$a \cup b$	a or b
equivalence	$\equiv$	$a \equiv b$	a is equivalent to b
implication	$\supset$	$a \supset b$	a implies b
	$\subset$	$a \subset b$	b implies a

# Quantifiers

---

Name	Example	Meaning
universal	$\forall X.P$	For all X, P is true
existential	$\exists X.P$	There exists a value of X such that P is true

# Clausal Form

---

- Too many ways to state the same thing
- Use a standard form for propositions
- *Clausal form*:
  - $B_1 \cup B_2 \cup \dots \cup B_n \subset A_1 \cap A_2 \cap \dots \cap A_m$
  - means if all the  $A$ s are true, then at least one  $B$  is true
- *Antecedent*: right side
- *Consequent*: left side

# Predicate Calculus and Proving Theorems

---

- A use of propositions is to 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

# Resolution

---

- *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

# Proof by Contradiction

---

- *Hypotheses*: a set of pertinent propositions
- *Goal*: negation of theorem stated as a proposition
- Theorem is proved by finding an inconsistency

# Theorem Proving

---

- Basis for logic programming
- When propositions used for resolution, only restricted form can be used
- *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



# Overview of Logic Programming

---

- Declarative semantics
  - There is a simple way to determine the meaning of each statement
  - Simpler than the semantics of imperative languages
- Programming is nonprocedural
  - Programs do not state how a result is to be computed, but rather the form of the result

# Example: Sorting a List

---

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

$\text{sort}(\text{old\_list}, \text{new\_list}) \subset \text{permute}(\text{old\_list}, \text{new\_list}) \cap \text{sorted}(\text{new\_list})$

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

# The Origins of Prolog

---

- University of Aix–Marseille (Calmerauer & Roussel)
  - Natural language processing
- University of Edinburgh (Kowalski)
  - Automated theorem proving

# Terms

---

- This book uses the Edinburgh syntax of Prolog
- *Term*: a constant, variable, or structure
- *Constant*: an atom or an integer
- *Atom*: symbolic value of Prolog
- Atom consists of either:
  - a string of letters, digits, and underscores beginning with a lowercase letter
  - a string of printable ASCII characters delimited by apostrophes

# Terms: Variables and Structures

---

- *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  
functor (*parameter list*)

# Fact Statements

---

- Used for the hypotheses
- Headless Horn clauses

```
female(shelley) .
```

```
male(bill) .
```

```
father(bill, jake) .
```

# 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)

# Example Rules

---

```
ancestor(mary, shelley) :- mother(mary, shelley) .
```

- Can use variables (*universal objects*) to generalize meaning:

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

```
parent(X, Y) :- father(X, Y) .
```

```
grandparent(X, Z) :- parent(X, Y) , parent(Y, Z) .
```



# Goal Statements

---

- For theorem proving, theorem is in form of proposition that we want system to prove or disprove – *goal statement*
- Same format as headless Horn

`man(fred)`

- Conjunctive propositions and propositions with variables also legal goals

`father(X, mike)`

# 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:

$P_2 \text{ :- } P_1$

$P_3 \text{ :- } P_2$

...

$Q \text{ :- } P_n$

- Process of proving a subgoal called matching, satisfying, or resolution

# Approaches

---

- *Matching* is the process of proving a proposition
- Proving a subgoal is called *satisfying* the subgoal
- *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

# 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

# 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!
  - The following is illegal:

`Sum is Sum + Number.`

# 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.
```

**A query:** `distance(chevy, Chevy_Distance) .`

# Trace

---

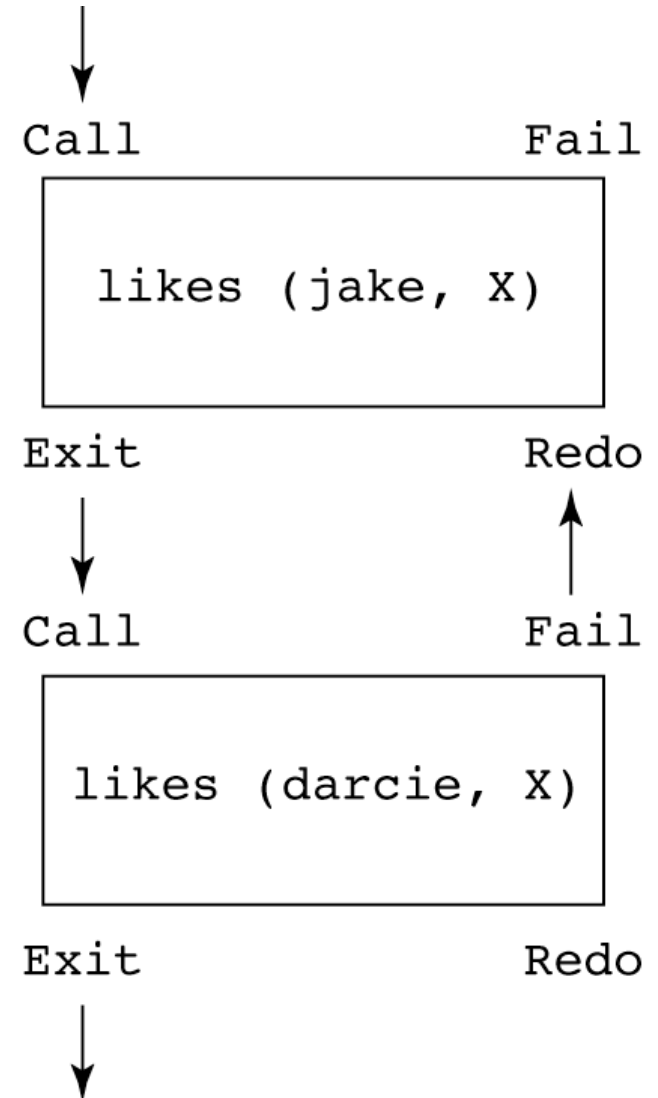
- Built-in structure that displays instantiations at each step
- *Tracing model* of execution – four events:
  - *Call* (beginning of attempt to satisfy goal)
  - *Exit* (when a goal has been satisfied)
  - *Redo* (when backtrack occurs)
  - *Fail* (when goal fails)



# Example

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

trace.
likes(jake, X), likes(darcie, X).
(1) 1 Call: likes(jake, _0)?
(1) 1 Exit: likes(jake, chocolate)
(2) 1 Call: likes(darcie, chocolate)?
(2) 1 Fail: likes(darcie, chocolate)
(1) 1 Redo: likes(jake, _0)?
(1) 1 Exit: likes(jake, apricots)
(3) 1 Call: likes(darcie, apricots)?
(3) 1 Exit: likes(darcie, apricots)
X = apricots
```



# List Structures

---

- Other basic data structure (besides atomic propositions we have already seen): list
- *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*)

[X | Y]           (*head X and tail Y*)

# Append Example

---

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

# More Examples

---

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

```
member(Element, [Element | _]).  
member(Element, [_ | List]) :-  
    member(Element, List).
```

The underscore character means an anonymous variable—it means we do not care what instantiation it might get from unification

# Deficiencies of Prolog

---

- Resolution order control
  - In a pure logic programming environment, the order of attempted matches is nondeterministic and all matches would be attempted concurrently
- The closed-world assumption
  - The only knowledge is what is in the database
- The negation problem
  - Anything not stated in the database is assumed to be false
- Intrinsic limitations
  - It is easy to state a sort process in logic, but difficult to actually do—it doesn't know how to sort

# Applications of Logic Programming

---

- Relational database management systems
- Expert systems
- 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