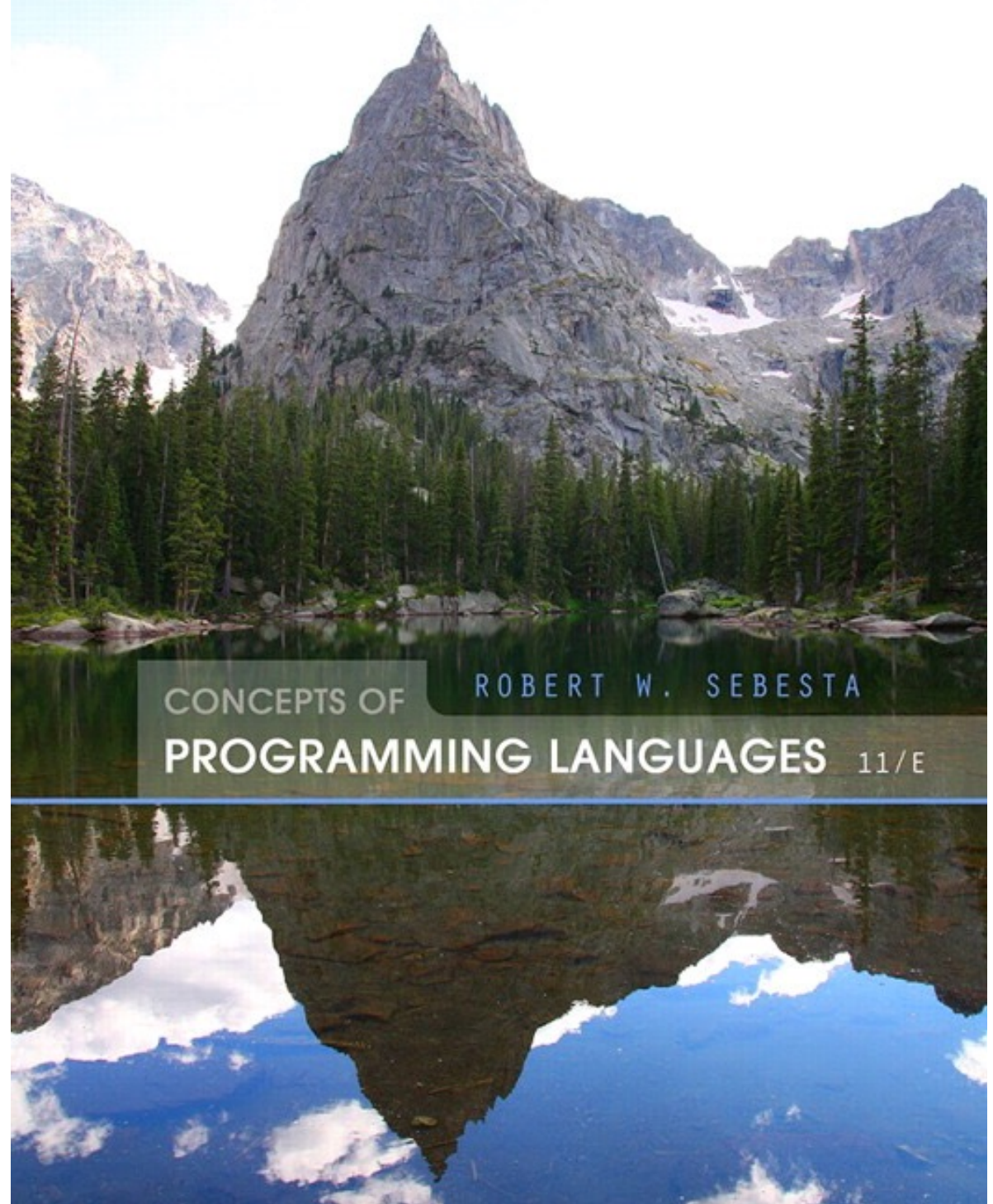


Chapter 16

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

man (jake)

like (bob, steak)

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*

First-order logic quantifies only variables that range over individuals; **second-order logic**, in addition, also quantifies over sets; **third-order logic** also quantifies over sets of sets, and so on.

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 or list

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)` → 1-tuple
`like(seth, OSX)`
`like(nick, windows)` → 2-tuple
`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 \subset	$a \supset b$ $a \subset b$	a implies 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

Examples

$$a \cap b \supset C$$

$$a \cap \neg b \supset d \implies (a \cap (\neg b)) \supset d$$

$$\forall X. (\text{woman}(X) \supset \text{human}(X))$$

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

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

Clausal Form Characteristics

- » Existential quantifiers are not required
- » Universal quantifiers are implicit in the use of variables in the atomic propositions
- » No operators other than conjunction and disjunction are required: disjunction on the left side (**consequent**) and conjunction on the right side (**antecedent**).

Example

- » $\text{likes}(\text{bob}, \text{trout}) \subset \text{likes}(\text{bob}, \text{fish}) \cap \text{fish}(\text{trout})$
- » $\text{father}(\text{louis}, \text{al}) \cup \text{father}(\text{louis}, \text{violet}) \subset$
 $\text{father}(\text{al}, \text{bob}) \cap \text{mother}(\text{violet}, \text{bob})$
 $\cap \text{grandfather}(\text{louis}, \text{bob})$

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

Example

- » $\text{father}(\text{bob}, \text{jake}) \cup \text{mother}(\text{bob}, \text{jake}) \subset \text{parent}(\text{bob}, \text{jake})$
- » $\text{grandfather}(\text{bob}, \text{fred}) \subset \text{father}(\text{bob}, \text{jake}) \cap \text{father}(\text{jake}, \text{fred})$
- » resolution says that:
 - » $\text{mother}(\text{bob}, \text{jake}) \cup \text{grandfather}(\text{bob}, \text{fred}) \subset \text{parent}(\text{bob}, \text{jake}) \cap \text{father}(\text{jake}, \text{fred})$
- » In English:
- » if: bob is the parent of jake implies that bob is either the father or mother of jake
- » and: bob is the father of jake and jake is the father of fred implies that bob is the grandfather of fred
- » then: if bob is the parent of jake and jake is the father of fred
 then: either bob is jake's mother or bob is fred's grandfather

Producing the Inferred Rule:

- » Or all the Left hand side terms
- » And all the right hand side terms
- » Remove terms that appear on both sides

Resolution when using variables

- *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
 - $\text{likes}(\text{bob}, \text{trout}) \subset \text{likes}(\text{bob}, \text{fish}) \cap \text{fish}(\text{trout})$
 - *Headless*: empty left side (used to state facts)
 - $\text{father}(\text{bob}, \text{jake})$
- 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
 - Predicate calculus supplies the basic form of communication to the computer, and resolution provides the inference technique.

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
- *All Prolog statement, as well as Prolog data, are constructed from terms.*
- *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) .
```

```
female(mary) .
```

```
male(jake) .
```

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

```
father(bill, shelley) .
```

```
mother(mary, jake) .
```

```
mother(mary, shelley) .
```

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:

$P_2 :- P_1$

$P_3 :- P_2$

...

$Q :- P_n$

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

Example

- » Goal: man(bob)
 - » If the following fact and inference rule is found in the database:
 - » father(bob).
 - » $\text{man}(X) \text{ :- father}(X).$ —> instantiate X temporarily to bob.
 - » The the goal is true
- » Goal: man(X)
 - » match the goal against the propositions in the database, and instantiate with the first object found

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

Backward Chaining

» Given this knowledge base:

1. If someone is a third year, then they need a job.
2. If someone is a third year, then they live in.
3. If someone needs a job, they will apply to be an accountant.
4. John is a third year

» **Goal: Is there anyone who is going to become an accountant?**

- » The system begins by searching either for a fact that gives the answer directly, or for a rule by which the answer could be inferred.
- » There's one rule whose conclusion, if true, would supply an answer, and that's rule 3.
- » The system next checks the rule's conditions. Is there anyone who needs a job? facts or rules?
- » There are no facts, but rule 1 is relevant.
- » So we now check its conditions. Is there a third year? This time, there is a fact that answers this: John is a third year. So we've proved rule 1, and that's proved rule 3, and that's answered the question.

Forward Chaining

- » It is a repeated application of modus ponens (if a conditional statement 'if p then q' is accepted, and the consequent does not hold 'not-q' then the negation of the antecedent 'not-p' can be inferred.)
- » Forward chaining is a popular implementation strategy for expert systems, business and production rule systems.
- » Use knowledge base, and infer to produce more data until a goal is reached.

Subgoal Strategies

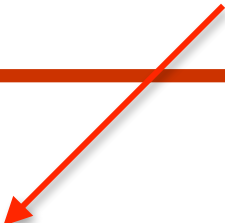
- When goal has more than one subgoal, can use either
 - Depth-first search: find a complete proof for the first subgoal before working on others
 - Breadth-first search: work on all subgoals in parallel
- Prolog uses depth-first search
 - Can be done with fewer computer resources

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

Backtracking Example

what happens if you reverse the order of the subgoals?



- » Goal: male(X), parent(X, shelley).
- » Prolog finds the first fact in the database with male as its functor.
- » It then instantiates X to the parameter of the found fact, say mike.
- » Then, it attempts to prove that parent(mike, shelley) is true.
- » If it fails, it backtracks to the first subgoal, male(X), and attempts to re-satisfy it with some alternative instantiation of 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!
 - 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) .



2205

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

trace.

distance(chevy, Chevy_Distance).

(1) 1 Call: distance(chevy, _0)?

(2) 2 Call: speed(chevy, _5)?

(2) 2 Exit: speed(chevy, 105)

(3) 2 Call: time(chevy, _6)?

(3) 2 Exit: time(chevy, 21)

(4) 2 Call: _0 is 105*21?

(4) 2 Exit: 2205 is 105*21

(1) 1 Exit: distance(chevy, 2205)

Chevy_Distance = 2205

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)
```

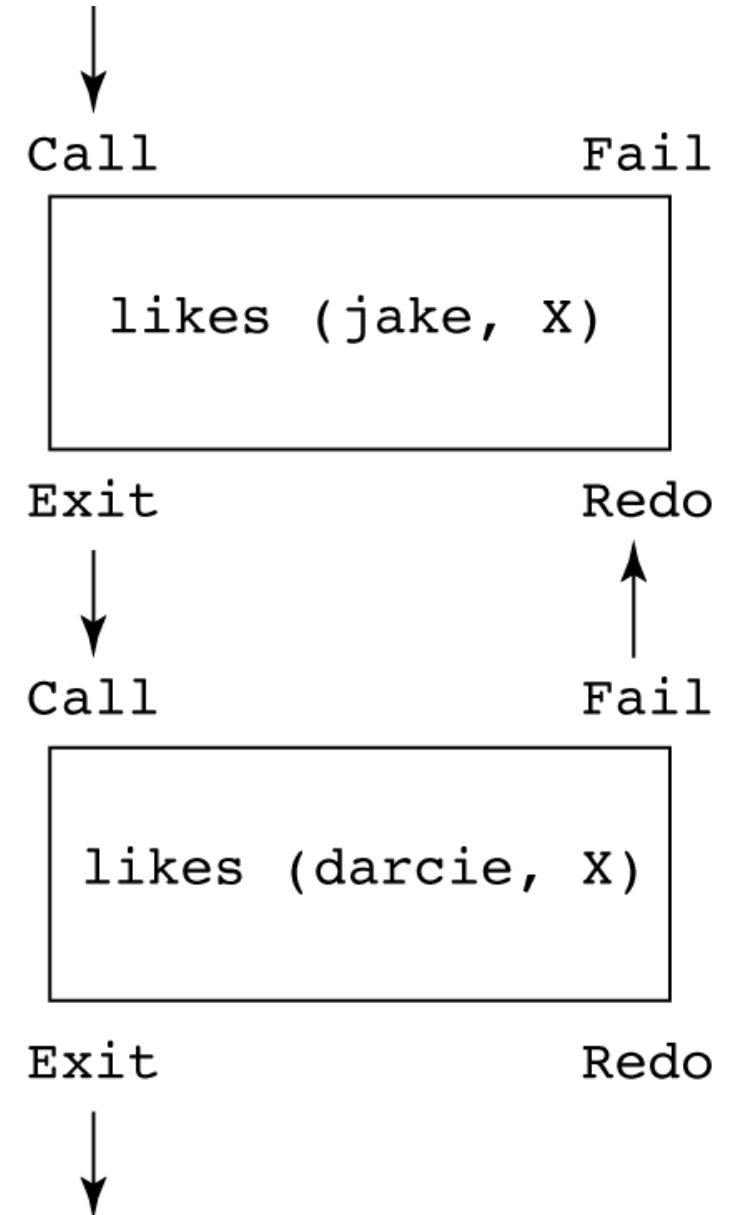
```
(1) 1 Redo: likes(jake, _0)?
```

```
(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)*

`new_list([apple, prune, grape, kumquat]).` → *creates a list*

`new_list([apricot, peach, pear])` → *creates a list*

Append Example

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

The first proposition specifies that when the empty list is appended to any other list, that other list is the result.
(recursion terminating condition)

The second means that appending the list [Head | List_1] to any list List_2 produces the list [Head | List_3], but only if the list List_3 is formed by appending List_1 to List_2. In LISP, this would be:

```
(CONS (CAR FIRST) (APPEND (CDR FIRST) SECOND))
```

Prolog's append is a predicate: it does not return a list, it returns yes or no.

Append Example:

trace.

append([bob, jo], [jake, darcie], Family).

(1) 1 Call: append([bob, jo], [jake, darcie], _10)?

(2) 2 Call: append([jo], [jake, darcie], _18)?

(3) 3 Call: append([], [jake, darcie], _25)?

(3) 3 Exit: append([], [jake, darcie], [jake, darcie])

(2) 2 Exit: append([jo], [jake, darcie], [jo, jake,
darcie])

(1) 1 Exit: append([bob, jo], [jake, darcie],
[bob, jo, jake, darcie])

Family = [bob, jo, jake, darcie]

yes

Append Query Example

» `append(X, Y, [a, b, c]).`

» Determines what two lists can be appended to get `[a, b, c]`

» Output:

```
X = []  
Y = [a, b, c];  
X = [a]  
Y = [b, c];  
X = [a, b]  
Y = [c];  
X = [a, b, c]  
Y = []
```

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: `not(not(some_goal))`.
- 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