

Chapter 13

Logic Programming

Overview

This chapter presents the goals and applications of logic programming and the historical and theoretical context in which it was developed.

In a logic language, the programmer defines predicates and writes logical formulas involving these predicates, constants, and variables. In this way, a data base of facts and relations is constructed.

Central to logic programming are the logical operation of implication and the process of inference. The programmer states rules which the language interpreter may use to infer new facts from those in the data base. This inference is done by a powerful reasoning technique called resolution.

Prolog is introduced as an example of a logic language. It was the first and is still widely known and used today. The relationship between deduction in an axiomatized system and computation is explored by comparing short Prolog programs to analogous programs in a procedural language.

In a procedural language, we define constants and variables and accept input data. We describe computation by defining functions and/or procedures which call preexisting functions and operators to compute new values from existing data. Assignment or binding is used to store the new data for future use. Because a program consists of a series of control units which will be executed sequentially, the order of computation is largely determined by the programmer. Within each control unit are other control units and function calls which largely (but not completely) define the sequence in which the actions and computations will be done.

In a logic language, we also define constants and variables. However, computation is done by finding data objects that satisfy a set of constraints. The programmer states facts, or relationships, about data objects, as well as inference rules by which conclusions may be drawn about those objects. A query may then be written—a statement whose truth or falsity the programmer does not know. The language interpreter attempts to prove that statement from the facts and rules previously provided.

The programmer does not define either the nature of the computational process or the exact order in which computations will happen. Those aspects are defined by the theorem-proving mechanisms built into the logic language interpreter. For this reason, logic languages are often described as “declarative” rather than “imperative”.

The traditional languages based on arithmetic, assignment, and sequential computation force programmers to over-specify the sequencing of their nonsequential algorithms and nonordered data. For some applications—for example, language recognition, artificial intelligence, data base query languages, and expert systems—it is easier and more comfortable to model the application declaratively. Interest in logic programming languages is, therefore, especially strong in these areas.

13.1 Predicate Calculus

The predicate calculus is a formal logical system for symbolizing facts about data structures and reasoning about them. It is also the basis on which logic programming languages are built. Before one can understand logic programming languages, one must be familiar with the symbols and concepts of logic. We review them here briefly.

13.1.1 Formulas

The components of the *first-order predicate calculus* are symbols for constants, variables, logical operations, functions, predicates, and quantifiers. Symbols may be combined to make formulas.

Constants

A constant denotes one specific object, which belongs to some specific domain, or set, of real or mathematical objects. This domain is also called a *universe of discourse*. Normally, several domains are involved in a single logical system, and different symbols are defined for each. In this discussion, we generally take the domain to be the set of numeric integers, and we will denote the integers by ordinary base-10 numerals. Names of other constants will be written using lowercase letters (including the underscore).

Variables

A variable ranges over a particular domain and represents some unspecified object from that domain. During a deduction, a variable may be *instantiated*, that is, bound to one constant object

Exhibit 13.1. Predicates and arity.

Predicate	Arity	Intent
odd(X)	1	X is an odd number.
father(F,S)	2	F is the father of S.
divide(N,D,Q,R)	4	N divided by D gives quotient Q and remainder R.

Instantiated predicate	Truth value
odd(2)	False.
father(David, Solomon)	True.
divide(23, 3, 7, 2)	True.
divide(23, 3, 7, N)	Neither true nor false until we instantiate N.

from its domain. In general, several variables will be simultaneously instantiated, each to a different value, and we use the term “instantiate” in this more general sense. We will use uppercase letters to name variables.

Functions

Symbols are used to name functions and may have parameters. A function name written with the appropriate collection of arguments denotes the result of executing that function on those arguments and is called a *term*. The arguments to functions may be constants, variables, or other terms. We will use lowercase letters (with the underscore) to name functions.

Predicates

A *predicate* is a symbol that denotes some property that an object, or several objects, might have. Thus if **gummy** and **biggerthan** are predicates and **j** and **k** are objects, then **gummy(k)** is an assertion that **k** has the property **gummy**, and **biggerthan(j,k)** asserts that **j** is **biggerthan** **k**. We will use lowercase letters to name predicates.

The arguments to a predicate are terms. A predicate applied to constants, or to variable-free terms, is called a *sentence* or a *proposition*. It has a truth value (true or false), depending on whether or not the predicate is true of the objects denoted by its arguments. A predicate may also be applied to variables, or to terms containing variables, but it does not form a sentence until all those variables are instantiated. If the instantiated predicate is true, then we say the predicate *is satisfied by* the instantiation.

A predicate may have one, two, or more arguments. [Exhibit 13.1] We use the word *arity* to denote the number of arguments, and describe a predicate with arity N as an N -ary predicate. Thus, the predicate **divide** in Exhibit 13.1 has arity 4 and is a 4-ary predicate.

Exhibit 13.2. Quantified predicates.

Quantified predicate	Truth value
1. $\forall X \text{ odd}(X)$	false.
2. $\exists X \text{ odd}(X)$	true, satisfied by $X=3$.
3. $\forall X (X=2*Y+1 \rightarrow \text{odd}(X))$	true.
4. $\forall X \exists Y \text{ divide}(X, 3, Y, 0)$	false for $X=2$.
5. $\exists X \exists Y \text{ divide}(X, 3, Y, 0)$	true, satisfied by $X=12, Y=4$.
6. $\exists X \forall Y \text{ divide}(X, 3, Y, 0)$	false, but harder to prove.

Quantifiers

Variables let us denote arbitrary objects in a domain; quantifiers let us denote sets of objects. There are two kinds of quantifiers: existential and universal. The *existential* quantifier is written $\exists X$ and read as “there exists an X such that”, and the *universal* quantifier is written $\forall Y$ and read as “for all Y ”. Quantifiers are written at the beginning of a formula and bind all occurrences of the quantified variable within the formula. The formula to which the quantifier is applied is called the *scope*. Thus, in Exhibit 13.2, both occurrences of the variable X on the third line are bound by the quantifier $\forall X$. The scope of the quantifier is $X = 2 * Y + 1 \rightarrow \text{odd}(X)$.

A quantified formula containing no free variables is a sentence and has a truth value. An existentially quantified formula is true if the scope of the quantifier(s) can be satisfied, that is, if there is at least one way to instantiate the variables (each from its proper domain) so that the resulting proposition is true. It is therefore easy to show that the quantified predicates on lines 2 and 5 in Exhibit 13.2 are true—we simply find suitable values for the variables.

Proving the truth of a universally quantified predicate is more difficult, especially if the universe of discourse is large or infinite. One must demonstrate that the scope is true for all objects in the

Exhibit 13.3. Quantified formulas.

Quantified predicate	Negation
1. $\forall X \text{ odd}(X)$	$\exists X \text{ not odd}(X)$
2. $\exists X \text{ odd}(X)$	$\forall X \text{ not odd}(X)$
3. $\forall X X=2*Y+1 \rightarrow \text{odd}(X)$	$\exists X \text{ not}(X=2*Y+1 \rightarrow \text{odd}(X))$ $\exists X \text{ not}(\text{not}(X=2*Y+1) \text{ or } \text{odd}(X))$ $\exists X (X=2*Y+1) \text{ and not odd}(X)$
4. $\forall X \exists Y \text{ divide}(X, 3, Y, 0)$	$\exists X \forall Y \text{ not divide}(X, 3, Y, 0)$
5. $\exists X \exists Y \text{ divide}(X, 3, Y, 0)$	$\forall X \forall Y \text{ not divide}(X, 3, Y, 0)$
6. $\exists X \forall Y \text{ divide}(X, 3, Y, 0)$	$\forall X \exists Y \text{ not divide}(X, 3, Y, 0)$

Exhibit 13.4. Semantics of not, or, and, and implies.

p	q	not p	p or q	p and q	q or not p	p implies q
true	true	false	true	true	true	true
true	false		true	false	false	false
false	true	true	true	false	true	true
false	false		false	false	true	true

universe. On the other hand, the negation of a universally quantified predicate is an existentially quantified predicate, as shown in Exhibit 13.3. Thus it is easy to show that a universally quantified statement is *false*; one just needs to find a single instantiation of the variables that makes the scope false, as shown on line 4 of Exhibit 13.2.

Logical Operations

Formulas can be combined to build bigger formulas using logical operations. Only a small number of logical operations, “and”, “or”, and “not”, are needed. The semantics of these operations are defined by the truth tables in Exhibit 13.4. Other logical operations such as “implies” are often used and can be defined in terms of the three basic operations.

13.2 Proof Systems

When we use the predicate calculus to represent some problem area, we start by defining symbols to represent objects, functions, and predicates from our external domains [Exhibit 13.5]. Then we write a set of sentences in terms of these symbols that represent facts about our system. These formulas, called *axioms*, usually capture only a portion of the logician’s semantic intent: that portion relevant to the current problem. Other aspects of semantic intent may be stated in the form of comments.

A *proof system* is a set of valid deduction rules that can be applied to axioms to deduce theorems. A *deduction* is an application of one of the valid inference rules of the proof system. In a simple deductive step, we derive a conclusion from the axioms and previously deduced theorems according

Exhibit 13.5. Logical symbols and sentences.

Constants: 17, gnu, apple_tree
 Variables: X, YY
 Functions: successor(X), product(Y,Z)
 Predicates: even(X), even(successor(17)), cousin(Y,Z), remainder(X, Y, Z)

to that rule. A *proof* is a series of sentences, ending with the sentence to be proved, such that each sentence is either an axiom or a valid deduction from some subset of the previous sentences in the proof. If there are N sentences in a proof, we will call it an N -step proof. A *refutation* is a proof of the negation of a sentence, which is the same as a proof that the sentence is inconsistent with the given axioms.

Two proofs of the same proposition can differ drastically in length and complexity. A logician constructs a proof or a refutation using insight, pattern recognition, and a bit of luck. With more insight (and luck) a proof is clear and brief; with less it can be tedious and murky. Even the clumsiest proof, however, takes some skill to construct.

A sentence that can be deduced from the axioms is called a *valid sentence*. Any valid sentence can also be called a *theorem*.¹ The set of all theorems derivable from a set of axioms is called a *theory*. We say that a theory is *consistent* if it does not contain contradictory theorems.

Classical Logic

Logicians have invented several different valid proof systems with very different deduction rules. One such system, often called *classical logic*, has been in use for centuries. It is based on the ancient deduction rule called “modus ponens” (the way of the bridge), which is the primary rule used to reason about conditional sentences. A *conditional sentence* is of the form:

$$C1 \rightarrow C2$$

where the arrow symbolizes the logical operation “implies”, and $C1$ and $C2$ are sentences. $C1 \rightarrow C2$ is read “ $C1$ implies $C2$ ”; $C1$ is called the *premise* and $C2$ is called the *conclusion*. Modus ponens says,

If the truth of the premise can be established, then one can infer that the conclusion is also true.

In symbols, this lets one derive the theorem $C2$ if both the conditional sentence $C1 \rightarrow C2$ and the sentence $C1$ have been previously proved (or are axioms).

Other proof rules are needed in order to deal with quantified sentences and with formulas containing free variables. For example, the rule of generalization says that if a formula $C(X)$ containing a free variable X can be proved (meaning that C is true no matter how X is instantiated), then one can infer the sentence $\forall X C(X)$.

Clausal Logic and Resolution

Clausal logic is an alternative to classical logic. It is based on a powerful deduction rule called *resolution* which is described in Section 13.4.1. *Resolution* often allows much shorter proofs because one resolution inference can take the place of many simple modus ponens inferences.

¹In other contexts this word is used only for particularly interesting or useful valid sentences.

Exhibit 13.6. A simple theory: the Theory of Ugliness.

Axioms for the Theory of Ugliness:

1. $\forall X \text{ ugly}(X) \rightarrow \text{not ugly}(X+1)$
2. $\forall X \text{ not ugly}(X+1) \rightarrow \text{ugly}(X)$
3. $2 = 1 + 1$

Theorems from the Theory of Ugliness:

1. $\forall X \text{ ugly}(X) \rightarrow \text{ugly}(X+2)$
2. $\forall X \text{ ugly}(X+2) \rightarrow \text{ugly}(X)$

Prolog itself is a proof system which uses clausal logic and resolution. Clausal logic generally writes “implies” using the left arrow, so that the conditional sentence $C1 \rightarrow C2$ becomes

$$C2 \leftarrow C1$$

and is read “ $C2$ if $C1$ ”.

13.3 Models

Recall that a theory is a set of formulas derivable from a set of axioms. Given a theory, we can try to find a *model* for it. A model is not something that can be written or discussed within a formal system. Rather, it is one possible interpretation of the semantic intent of a theory and is presented in a metalanguage. To make a model, we find interpretations for each constant, function, and predicate symbol in the axioms, so that the axioms are all true statements about those interpretations. Specifically, we must define each universe of discourse and show the correspondence between members of a universe, or domain, and the constants in the axioms. The predicate symbols in the axioms must also be assigned to properties of objects in these domains, and the function symbols must be assigned to functions that operate on objects from the specified domains. An inconsistent theory has no models because it is impossible to find interpretations that simultaneously satisfy contradictory theorems.

Because a set of axioms only partially “defines” the functions and predicates of a theory, one theory may have several models. For example, consider the very simple theory in Exhibit 13.6, which we will call the “Theory of Ugliness”. To find a model for this theory we must find a domain of discourse which contains interpretations for the constant symbols “1” and “2”, for the function symbol “+”, and for the predicate symbol “ugly”. This is easily done.

- Let our domain of discourse be the integers. Let “1” represent the integer one and let “2” represent the integer two.
- Let “+” represent the function of addition on the integers.

- Now there are two possible interpretations for “ugly”: the integer predicate “odd” or the integer predicate “even”.

We can now make two models for the Theory of Ugliness. In one model, all odd numbers are ugly, even numbers are not. In the other model, all even numbers are ugly, odd numbers are not. Both models are completely consistent with the axioms, yet the axioms do not let us identify even one ugly number with certainty. Nevertheless, we can reason about ugly numbers and prove, for example, that if X is ugly, then so is $X+2$. (This is the theorem mentioned in Exhibit 13.6.) The fact that a set of axioms can have more than one model is a very important matter in logic programming applications; we do not have to axiomatize all aspects of a model in order to reason about it.

13.4 Automatic Theorem Proving

Symbolic logic was first formulated by Frege², at the end of the nineteenth century. This work was extended during the next fifty years by several pioneers. Certain properties of logical theories have been central to this work. The first is completeness: a theory is *complete* if every sentence that is true in all models of the theory can be proved from its axioms by using its deductive rules. We say that such a sentence is a *logical consequence* of the axioms.

A second property, first studied by Hilbert, is *decidability*. A theory is decidable if one can decide (that is, prove) whether any given sentence is or is not valid in that theory. This is equivalent to saying that, for any sentence, either it or its negation can be proved. A theory can be complete but not decidable if there are sentences that are true in some models but not in others. Such sentences are not logical consequences of the axioms. For example, in the Theory of Ugliness, the sentence `ugly(3)` is true in one model, where the predicate `ugly` corresponds to the integer property `odd`, but not in the alternate model where `ugly` represents `even`. Thus `ugly(3)` is not a logical consequence of the theory, and neither `ugly(3)` nor `not ugly(3)` can be proved from the axioms.

By 1940, the logical foundations of mathematics had been thoroughly established and explored. The central theorems are as follows:

1. The predicate calculus is complete. That is, the proof techniques of the predicate calculus can be used to develop a formal proof for every logically true proposition. (Goedel, Herbrand, 1930.)
2. If a proposition can be proved at all, then it can be proved “in a direct way”, without requiring insight. That is, a mechanical proof method exists. (Gentzen, 1936.)
3. There is not, and cannot be, an algorithm that correctly identifies propositions that are *not* true and *cannot* be proved. Even though a proof procedure exists for true propositions, a proof may take an indefinitely long time to produce by this procedure. There is no time at

²Frege [1879].

which we can conclude that a proof will *not* be discovered in the future. As a consequence, Hilbert's decision problem is unsolvable for the predicate calculus. (Church, Turing, 1936.)

Since the invention of electronic computers, there has been interest in automating the proof process. At first this was only of academic interest. More recently, however, the community has come to understand that deduction and computation are analogous processes—indeed, computations can be written as deductions, and vice versa. Since 1940, logicians have been working on the problem of automatic theorem proving programs:

- We know that we can devise an algorithm (or write a program) that will take a set of axioms and enumerate and prove all possible propositions that can be derived from those axioms.
- Of more interest, we would like an automatic system which, given a single interesting proposition, will prove it.
- We would like this system to be efficient enough to be useful.
- We would like a language in which to express propositions, and a language processor that will find proofs for these propositions, if they exist.

As a consequence of the completeness result, we know that it is theoretically possible to build a program that can prove any theorem that is a logical consequence of a set of axioms. However, if we simply generate and prove propositions systematically, most of them will be simple variations of similar propositions and any interesting theorems would be buried in an avalanche of trivia. Moreover, the process would take forever. This is analogous to the old question of the monkeys and the typewriters: if you put 1000 monkeys in front of 1000 typewriters, and they all randomly type, day and night, will they ever produce any great literature? The answer is yes: eventually the random process will produce a poem, or more, but the valuable information will be lost in a sea of garbage. Finding the “great” stuff would be impractical.

Once we restrict our attention to proving a single proposition, the main problem is efficiency. If we start with the axioms and make every inference possible, a few might be relevant to our chosen proposition, but most are not, and we have no automatable way to tell which. Further, any deduction we make can lead to several other deductions, creating multiple, branching lines of reasoning. Visualize all possible deductions from a set of axioms as a tree structure. The width and bushiness of the tree are determined by the number of alternative deductions possible at each stage. A sentence at level one of the tree represents an axiom. A sentence at level two is a theorem with a two-step proof, and the sentences at level N are theorems with N step proofs.

If we could build an automatic theorem-prover to start with a set of axioms, make every possible inference, and follow every possible line of reasoning, our proof method would be complete. However, that would create an impossibly bushy and impossibly deep tree-structure of lines of deduction. We say that there is a *combinatorial explosion* of possible lines of reasoning, because of the huge number of different combinations of inferences that we might make.

We could try to generate proofs by constructing this tree in a breadth-first manner; that is, make all two-step deductions before starting on the three-step proofs. If we do so, we build a very bushy tree. By the time we have made all ten-step deductions, we could have so many possibilities that remembering and extending them all becomes unmanageable. And ten-step proofs are material for student exercises! Hundred-step proofs would be practically impossible. On the other hand, we could try to generate proofs in a depth-first manner; that is, we could follow one line of reasoning to its logical conclusion before exploring other lines. However, using this approach, we could get trapped in an infinite, circular line of reasoning, and our proof process would never terminate.

Some propositions which can be easily proved by a human with insight are very difficult to prove by constructing the entire reasoning tree. For example, Exhibit 13.7 states a very uninteresting proposition: “The number 65534 is a good number”. This is proved from axioms that the number zero is good and conditional rules about good numbers that use the arithmetic operations of addition, multiplication, and exponentiation to the base two. The complete line-of-reasoning tree for this foolish proposition would be immensely bushy, since four or five rules can be applied at each step and, usually, only one of them leads to this twenty-one-step proof or to a similar short proof. Moreover, one branch of the tree is at least 32,767 lines long. Finally, rules 3 and 4 can be applied alternately to create an infinite chain of deductions that makes no progress. Whether we search this tree breadth first or depth first, any systematic exploration of it will become mired in masses of uninteresting inferences. Admittedly, propositions like this are pathological cases and are of no real interest to anyone. However, they illustrate that, in order to be truly useful, a general proof algorithm must be able to avoid trouble when given such a proposition to prove.

13.4.1 Resolution Theorem Provers

Early in the history of automatic theorem proving, it was clearly understood that a brute-force, enumerate-them-all approach could never be practical. A usable automatic theorem-prover would have to use techniques that were different from the step-by-step deductions that humans were accustomed to making. Two breakthroughs happened. First, it was shown that a highly restricted form of the predicate calculus, called *clausal logic*, was still complete. In clausal logic, all sentences are a series of predicates, possibly negated, connected by “or” operators.

Second, a new proof technique named *resolution* was invented. The goal of *resolution* was to collapse many simple modus-ponens-like deduction steps into one large inference, thereby eliminating whole sections of the unmanageable deduction tree. Resolution operated on clausal sentences and relied on a pattern-matching algorithm called *unification* to identify sections of a proposition that could be simplified.

The process of resolution on two clauses is analogous to algebraic simplification; it is a way to derive a new, simpler fact from a pair of related facts. First, unification is used to identify portions of the two clauses that are related and can be “simplified” by a *cut operation*. Specifically, given the two assertions:

$$A \rightarrow (B \text{ or } C)$$

Exhibit 13.7. A proof that 65,534 is a good number.

- Let the predicate $\text{good}(X)$ mean that the number X is good.
- Let the predicate $\text{powtwo}(Y,Z)$ mean that 2 to the power Y is Z .
- Prolog syntax is used below to write the propositions. The comma in axiom 6 should be interpreted as the logical operator “and”.

Axioms.

1. $\text{good}(0)$.
2. $\text{powtwo}(0,1)$.
3. $\text{good}(A-2)$ \leftarrow $\text{good}(A)$.
4. $\text{good}(A+2)$ \leftarrow $\text{good}(A)$.
5. $\text{good}(A*2)$ \leftarrow $\text{good}(A)$.
6. $\text{good}(A)$ \leftarrow $\text{powtwo}(B,A), \text{good}(B)$.
7. $\text{powtwo}(P+1,A*2)$ \leftarrow $\text{powtwo}(P,A)$.

The shortest proof of the proposition that 65,534 is a good number:

- | | |
|-------------------------------|---|
| 8. $\text{good}(2)$ | by rule 4, $A=0$, with line 1. |
| 9. $\text{good}(4)$ | by rule 5, $A=2$, with line 8. |
| 10. $\text{powtwo}(1,2)$ | by rule 7, $P=0, A=1$, with rule 2. |
| 11. $\text{powtwo}(2,4)$ | by rule 7, $P=1, A=2$, with line 10 |
| 12. $\text{powtwo}(3,8)$ | by rule 7, $P=2, A=4$. |
| 13. $\text{powtwo}(4,16)$ | by rule 7, $P=3, A=8$. |
| 14. $\text{good}(16)$ | by rule 6, $A=16, B=4$. |
| 15. $\text{powtwo}(5,32)$ | by rule 7, $P=4, A=16$. |
| 16. $\text{powtwo}(6,64)$ | by rule 7, $P=5, A=32$. |
| 17. $\text{powtwo}(7,128)$ | by rule 7, $P=6, A=64$. |
| 18. $\text{powtwo}(8,256)$ | by rule 7, $P=7, A=128$. |
| 19. $\text{powtwo}(9,512)$ | by rule 7, $P=8, A=256$. |
| 20. $\text{powtwo}(10,1024)$ | by rule 7, $P=9, A=512$. |
| 21. $\text{powtwo}(11,2048)$ | by rule 7, $P=9, A=1024$. |
| 22. $\text{powtwo}(12,4096)$ | by rule 7, $P=9, A=2048$. |
| 23. $\text{powtwo}(13,8192)$ | by rule 7, $P=9, A=4096$. |
| 24. $\text{powtwo}(14,16384)$ | by rule 7, $P=9, A=8192$. |
| 25. $\text{powtwo}(15,32768)$ | by rule 7, $P=9, A=16384$. |
| 26. $\text{powtwo}(16,65536)$ | by rule 7, $P=9, A=32768$. |
| 27. $\text{good}(65536)$ | by rule 6, $A=65536, B=16$, with lines 14, 26. |
| 28. $\text{good}(65534)$ | by rule 3, $A=65536$, with line 27. |
-

and

$$(C \text{ or } D) \rightarrow E$$

then resolution lets us infer that

$$(A \text{ or } D) \rightarrow (B \text{ or } E)$$

Before making this inference, though, we must establish the correspondence between the C parts of the two formulas. Since these subformulas might be quite complicated and involve many predicates with variables and constants, unification (the process of establishing the correspondence or showing that no correspondence exists) is not a trivial matter.

A *resolution deduction* is a sequence of resolution inferences. A *resolution proof* is a resolution deduction whose conclusion is “false”; it establishes that the premises, taken together, form a contradiction. A resolution proof can be used to prove that the *negation* of any single premise can be deduced from the other premises by ordinary logical deduction.

The original resolution methods were better than the brute-force method of enumerating and testing all possible lines of reasoning but were still not practical or useful in real applications. For example, the proposition in Exhibit 13.7 would cause an early resolution-theorem-prover to become mired in irrelevant deductions. Since then, it has been shown that we can restrict the kinds of propositions even further, in ways that make the resolution more efficient. Happily, these restrictions affect only the way a proposition is stated; they do not affect the completeness of the system.

Some interesting theorems are easily derived using ordinary resolution. However, the computation for most theorems is very extensive and is still not of any practical use. A more advanced method, named *hyperresolution*, was developed that operates on Horn clauses, which are even more highly restricted sentences. It is this form of resolution on which logic programming languages are built.

Implementing Resolution Efficiently

Horn Clauses. A *Horn clause* is a clause with at most one unnegated predicate symbol. As shown in Exhibit 13.8, this is equivalent to saying that a sentence may contain a single “implies” operator, whose premise is several predicates connected by “and” and whose conclusion contains only one predicate symbol.

Unification Most of the time spent in finding a resolution proof is actually spent in identifying related phrases in two sentences that can be simplified, or cut. A hyperresolution inference depends on the result of a unification procedure. Hyperresolution says that, from the premises

$$A1 \text{ and } A2 \text{ and } \dots \text{ and } An$$

and

$$(B1 \text{ and } B2 \text{ and } \dots \text{ and } Bn) \rightarrow D$$

Exhibit 13.8. Ways to write a Horn clause.

We show that a Horn clause can be rewritten as an implication whose premise is a series of predicates connected by “and” operators and whose conclusion is a single predicate. Let each capital letter in the following derivation stand for a predicate symbol with its required arguments.

A Horn clause is a clause with at most one nonnegated predicate, such as:

$$\text{not } P \text{ or not } Q \text{ or not } R \text{ or } S \text{ or not } T$$

Because “or” is associative and commutative, we can move all the negated predicates to the right:

$$S \text{ or not } P \text{ or not } Q \text{ or not } R \text{ or not } T$$

Using DeMorgan’s law, this expression can be transformed to the form:

$$S \text{ or not}(P \text{ and } Q \text{ and } R \text{ and } T)$$

The operator “implies” is defined in terms of “not” and “or”, and can be written with either a right-pointing arrow or a left-pointing arrow. Thus the two expressions

$$B \leftarrow A \qquad A \rightarrow B$$

are defined to mean

$$B \text{ or not } A$$

We use the left-pointing “implies” arrow here to conform to the order of terms in Prolog syntax. Now we can rewrite the Horn clause as follows:

$$S \leftarrow (P \text{ and } Q \text{ and } R \text{ and } T)$$

we can infer D , where all variables in D have been instantiated with expressions that unify the set $\{\{A1, B1\}, \{A2, B2\}, \dots, \{An, Bn\}\}$.

The unification algorithm finds a set of substitutions for the variables in the two formulas so that, after making the substitutions, the two clauses are equal. This is a kind of pattern-matching process. Exhibit 13.9 shows examples of simple unification problems and their results.

We use the unification algorithm to determine whether or not a conditional Horn clause, C , can be used in a resolution, given the current set of unconditional clauses in the data base. To perform a hyperresolution step, we need to find a set of unconditional clauses that *cover* all the predicate symbols in C , and a single set of instantiations for all the variables in these clauses that unifies each unconditional clause with the corresponding predicate. Let us describe the process of unifying an unconditional clause whose predicate symbol is P , and a single predicate symbol, Q ,

Exhibit 13.9. Unifying clauses.

	Formula 1	Formula 2	Most General Unification
1.	pp(a,b)	pp(X,Y)	X=a, Y=b
2.	pp(Q(c,d),Z)	pp(Y,e)	Y=Q(c,d), Z=e
3.	pp(X,Y) and qq(Y,a)	pp(a,b) and qq(Z,W)	W=a, X=a, Y=b, Z=b
4.	qq(pp, gg(X,Y), X, Y)	qq(Y, Z, hh(U, kk), U)	U=pp, X=hh(pp,kk) Y=pp, Z= gg(hh(pp,kk), pp)

in the conditional clause C as follows:

- A cluster is a set of terms that have been mapped into one equivalence class and must be unified.
- To begin the unification process, map P and Q into a cluster.
 1. Choose any cluster. Look at all the terms in the cluster. (In unifying M items, there will be M terms.)
 2. If one term is a constant and if another term is a predicate or a constant that does not match, no unification is possible and the process fails. If all terms are matching constants, this branch is successfully unified. Otherwise proceed.
 3. If two or more terms are predicates and they are different, no unification is possible and the process fails. Otherwise, proceed, and let N be the arity of the cluster.
 4. We know we are merging a combination of one constant or one predicate with one or more variables. Merge all terms in the cluster into a single unified super-term. Temporarily, the super-term will inherit all the arguments from the predicate terms in the cluster, so that it will have M first arguments, M second arguments, ..., and M N th arguments.
 5. We must replace each collection of M N th arguments that was created in step 4 by a unified term. For j from 1 to N , look at all the N th arguments that belong to this super-term. (If we are unifying M terms, there will be M arguments to consider.) Make a new cluster by mapping together all the terms that are the j th arguments of the super-term. During this process, we may find that one of the terms to be mapped was mapped to a different cluster by a previous operation. In that case, the two clusters are mapped into one.
 6. Replace the cluster we have been processing by the super-term created in step 4 with its arguments replaced by the argument clusters from step 5.
 7. Repeat steps 1 through 6 until there are no remaining clusters in the formula.
- The process will end when it has descended through all levels of argument nesting.

13.5 Prolog

13.5.1 The Prolog Environment

Prolog is an interactive language for expressing predicates, relations, and axioms. The programmer may state facts or make queries about those facts. In response to a query, Prolog will attempt to find one instantiation of the query's variables that makes the formula true.

In addition to the language for writing logical assertions and queries, a Prolog system includes a metalanguage for managing the data base of facts and rules. Large data bases can be entered either from a file or from the keyboard by calling the input function `consult(filename)`. Once in the system, parts of the data base can be listed (by using `listing(predicate_name)`) or edited (by using `retract(Z)`), and the data base can be extended (by using `asserta(X)` or `assertz(X)`).

After entering a data base, the programmer may interactively enter queries, test hypotheses, and assert more facts. In response to a query, Prolog executes its proof-procedure and searches for a set of instantiations that satisfies a query's variables. If this search succeeds, Prolog responds by writing out the instantiations that were found. The programmer can indicate that she or he is satisfied (by typing a carriage return), and Prolog responds by saying "yes" and giving a new prompt. Alternatively, the programmer may ask Prolog to search for additional instantiations (by typing a ";" and a carriage return). This search process continues until the programmer decides she or he has seen enough or the entire data base has been searched and no further set of instantiations can be found, at which time Prolog responds by writing "no" and a new prompt.

13.5.2 Data Objects and Terms

The basic components of a Prolog program are objects (constants, variables, structures, lists), predicates, operators, functions, and rules.

- Constants: Integers are predefined and represented in base 10. User-defined constants, or *atoms*, have names that start with lowercase letters. Remember that a Prolog program, like a theory, may have many models (or none) and a Prolog atom may correspond to different objects in different models.
- Variables: All variable names start with uppercase letters. All occurrences of the same variable in a rule will be bound to the same object. The symbol "_" can be used in place of a variable name in cases where the program does not need to refer to this same object twice. If the "_" symbol occurs several times in a rule, it may be bound to several different objects.
- Structures: Compound data types (record types) may be defined by specifying a *functor* (the type name) and a list of *components* (fields), in parentheses:
 $\langle \text{type name} \rangle (\langle \text{component} \rangle, \dots, \langle \text{component} \rangle)$
- Lists may be denoted by using square brackets. The empty brackets "[]" denote the empty (null) list. The notation `[a,b,c]` denotes a list of three terms whose head is `a` and whose

last item is c . The notation $[A|X]$ denotes a list of length one or greater with head A and tail, or remainder, X , which may be a list, a single atom, or null. This notation is often used in function definitions to give local names (A and X) to the parts of a list.

This bracket notation is a “sugaring” of a less attractive basic syntax; “[Head|Tail]” is a syntactic variant of “. (Head, Tail)” and “[x, y, z]” is a variant of “. (x, . (y, . (z, [])))”. We will call both forms *list specifications*. A list specification can denote a fully defined list, such as $[a, b, c]$, a fully variable list, such as $[Head|Tail]$, or a list with a defined head and a variable tail, such as $[a, b|Z]$. The term *open list* refers to a list with a variable tail. During the proof process, an open list may be unified (matched) with any list that has the same constant as its head. The selector functions $head(x)$ and $tail(x)$ are defined to return the head and tail of a list argument.

- Predicates $=$, $\backslash=$, $<$, $=<$, $>$, and $=>$ are defined. Users may introduce their own predicate symbols, which are written in lowercase. Again, remember that the semantics of these symbols remains largely in the programmer’s mind. Within Prolog they have no meaning except that captured by the axioms, or rules, the programmer supplies.
- The integer operators $+$, $-$, $*$, $/$, and mod are defined and may be written either in standard functional notation: “ $+(3, X)$ ”, or in infix form: “ $3+X$ ”. When infix form is used, the operators have the usual precedence. These operators may be used in an “is” formula, thus: “ X is $(A+B) mod C$ ”. When Prolog processes such a formula it performs the computation using the current bindings of A , B , and C , and binds the result to X . An error is generated if any variable on the right of the “is” is not bound to a number.

A *term* is a predicate symbol or operator or function symbol, together with a parenthesized list of the right number of arguments.

13.5.3 Horn Clauses in Prolog

Deductive Rules

Rules are the axioms that form the basis for deductions in Prolog. A *rule* represents a statement of the form

If a set of premises are all true,
then we may infer that a given conclusion is also true.

The conclusion is written on the left, followed by a “:-” sign, and the premises are written on the right, separated by commas. The variables in a rule are implicitly, universally quantified. A rule is also called a *conditional clause*. Examples of rules are given in Exhibit 13.10.

A *fact* is an assertion that some object or objects satisfy a given predicate. Formally, a fact is a rule which has a conclusion but no conditions, and it is, therefore, sometimes called an *unconditional clause*. Examples of facts are given in Exhibit 13.10. The programmer adds a new fact to the data

Exhibit 13.10. Prolog rules and facts.

Conditional clauses	<code>pretty(X) :- artwork(X).</code>
(rules)	<code>pretty(X) :- color(X,red), flower(X).</code>
	<code>watchout(X) :- sharp(X,_).</code>
Unconditional clauses	<code>color(rose, red).</code>
(facts)	<code>sharp(rose, stem).</code>
	<code>sharp(holly, leaf).</code>
	<code>flower(rose).</code>
	<code>flower(violet).</code>
	<code>artwork(painting(Monet, haystack_at_Giverny)).</code>

base by *asserting* it. If X is a fact, then the predicate `asserta(X)` appends the fact X to the beginning of the data base, and the predicate `assertz(X)` appends it to the end.

Prolog rules and facts are Horn clauses. The conditions of a rule are a conjunction of terms, even though we write commas between the terms rather than “and” operators. Each single rule represents the “and” of a series of conditions. The conclusion is a single positive term, and the “:-” represents a \leftarrow sign. An underscore is a wild card. Thus a Prolog rule is a Horn clause written in the form shown on the last line of Exhibit 13.8. Prolog facts are unconditional Horn clauses.

Typically, one predicate may have several rules defined for it [Exhibit 13.13]. A list of rules for one predicate represents the “or” of those conditions—only one rule will be used to interpret a given call on that predicate, and the one used will be the first one that can be satisfied. Thus we can use a sequence of rules to express a generalized conditional semantic structure.

Rules may be recursive and thus may be used to implement repetition in an algorithm. A recursive predicate must, of course, have at least two rules (the base case and the recursive step).

Exhibit 13.11. Prolog queries.

```

?- pretty(rose).
yes
?- pretty(Y).
Y=painting(Monet, haystack_at_Giverny).
Y=rose
no
?- pretty(W),sharp(W,Z).
W=rose Z=stem
no

```

Exhibit 13.12. Interpreting Prolog rules.

Rules 2 and 3 from the gcd algorithm [Exhibit 13.13] are dissected here. In both rules, A and B are input parameters and D is an output parameter.

```
gcd(A, B, D) :- (A<B), gcd(B, A, D).
```

This right side is the analog of a Pascal IF...THEN statement. Prolog interprets it as follows: test whether (A<B) is true. If so, call predicate gcd with arguments (B, A, D). If that succeeds, D will be bound to the answer.

```
gcd(A, B, D) := (A>B), (B>0), R is A mod B, gcd(B, R, D).
```

This performs the equivalent of two Pascal if statements, an assignment, and a function call. Prolog interprets it thus: test whether (A>B) is true. If so, test (B>0). If both conditions hold, calculate A mod B and bind to R. Finally, call predicate gcd with arguments (B, R, D). If all succeed, D will be bound to the answer.

Queries

A query in Prolog is a request to Prolog to prove a theorem. Because the question posed is the goal of the proof process, a query is also called a *goal*. Syntactically, a query is a list of terms, separated by commas [Exhibit 13.11]. Semantically, the commas represent “and” operators. If the query has no variables, Prolog will attempt to prove it from the rules and facts previously given. Thus, in Exhibit 13.11, the query “?- pretty(rose)” can be proved from the second rule taken together with the first and fourth facts in Exhibit 13.10.

If the query does contain variables, the Prolog theorem-prover attempts to find a set of instantiations that satisfy the query. All variables in the query are, implicitly, quantified by “ \exists ”; that is, a query asks whether any set of objects exists that can satisfy the clause. Thus to respond to the query “?- pretty(Y)”, Prolog tries to find some object, Y, that has the property “pretty”. It will begin by using rule 1, the first rule given for the predicate “pretty”, and will combine this with fact 5, instantiating Y to a painting, and producing the output “Y = painting(Monet, haystack_at_Giverny)”. If the programmer types “;” to ask for another instantiation, Prolog will continue to search its data base and find the second instantiation, “Y = rose”. If the search is continued again, no further instantiations succeed, and the answer “no” is printed.

A query may contain a series of predicate terms which will always be processed in order, left to right, and processing will be aborted if any one predicate cannot be satisfied [Exhibit 13.12. This built-in conditional sequencing is used where “if” statements and sequences of assignments would be used in a procedural language.

13.5.4 The Prolog Deduction Process

A query establishes a goal for the deductive process, in the form of a conjunction of terms. To satisfy this goal, Prolog first takes each individual term in the goal, in order, as a subgoal, and recursively attempts to satisfy that subgoal.

The subgoal is a predicate with arguments. Prolog begins by finding, in the data base, the rules for that predicate which have the right arity for the subgoal, and then starts with the first rule. It attempts to unify the head (conclusion) of the rule with the subgoal; if there is a conflict because constant terms do not match, Prolog will go on to the next rule for that predicate. If the head can be unified with the subgoal, it means that this rule is, potentially, applicable.

Prolog then tries to satisfy each of the condition terms on the right side of that rule. To do this, it searches for a set of facts that “cover” the terms of the goal. If this process succeeds, the rule is applied to the goal, the goal’s variables are instantiated to the objects discovered by the unification algorithm, and the set of instantiations is returned to the calling context. At the top level, the instantiations are printed out.

We can summarize the operation of the Prolog proof process as follows:

1. Work on the leftmost subgoal first.
2. Select the first applicable rule.
3. Search the facts in the order they appear in the data base.

Careful attention to the order in which goals are written, rules are given, and facts are asserted can improve the performance of the proof system substantially. It makes good sense, when defining a multirule predicate, to make the first rule the one that will be used most frequently, if that is possible. For example, in the gcd algorithm [Exhibit 13.13], the rule that ends the recursion must be written before rule 2, to prevent rule 2 from being invoked with a zero divisor, which would generate an error. However, rules 2 and 3 could be written in either order, and rule 3 is placed last because it will be executed at most once, to swap the arguments on the first recursion, in the case that the first argument is smaller than the second.

The programmer must also be careful of the order of the conditions within a rule. Since these conditions are taken, in order, as subgoals, and variables become instantiated as the subgoals succeed, a condition that is intended to instantiate a variable must precede all conditions that expect that variable to be instantiated. This is like saying, in a procedural language, that variables must be initialized before they are used. Further, because the conditions are tested in order, a series of conditions is very much like a nested conditional. Sometimes a condition is simple to test; sometimes satisfying it can involve a great deal of computation. Where two conditions must both be true before proceeding with a rule, it is prudent to write the simple one first, so that, if it is false, it will abort the rule before making the costly test. A simple condition can thus “guard” a complex or difficult condition.

Exhibit 13.13. Euclid's algorithm in Prolog.

These three rules axiomatize Euclid's greatest common divisor algorithm. The numbers on the left are for reference only; they are not part of the Prolog code.

1. | `gcd(A, 0, A).`
2. | `gcd(A, B, D) :- (A>B), (B>0), R is A mod B, gcd(B, R, D).`
3. | `gcd(A, B, D) :- (A<B), gcd(B, A, D).`

Backtracking

The recursive descent from goal to subgoal ends successfully when a subgoal is reached that corresponds to a known fact in the data base. The instantiations used to satisfy the subgoal are passed back up and are used to instantiate the variables in the goal.

The recursive descent ends in failure if the `fail` predicate is encountered while processing a rule, or if the entire data base has been searched and no relevant information was found. Failure of a goal does not mean that the goal is false, only that it cannot be proven from the facts given. When a subgoal fails, control *backtracks*, that is, it passes back up to the level of the goal above, and Prolog attempts to find a different set of instantiations that satisfy the original goal. In terms of the tree of possible proofs, this amounts to backing up one path until an unexplored branching point is found, then going down the new branch of the tree. Exhibit 10.36, at the end of Chapter 10, illustrates this recursive backtracking.

13.5.5 Functions and Computation

A programmer trying to solve a problem or model a system starts with a set of inputs, in some given form, and wishes to derive from them a set of outputs. In a language like C, this is done by defining functions to carry out a series of calculations and manipulations on the data that produce a result in the desired form. We call this procedural programming.

We might view the same problem in a different way. Instead of specifying the method to reach the desired output, we could describe the output desired, in a declarative, axiomatic language. Prolog is a language for axiomatizing a desired result. The process of computation, in a C program, is replaced by the process of proof in Prolog, where a proof consists of *finding* a data object that satisfies the formula for the result. Thus ordinary computation, as well as data base searches, can be expressed in Prolog.

Computation

In place of a C function that returns an answer, a Prolog programmer writes a predicate with one additional parameter, which will be used for returning the answer. It is customary to write the output parameter or parameters last. Instead of instructions to perform a computation, the Prolog programmer writes axioms to verify that the answer is correct. During the proof process, the

Exhibit 13.14. Euclid's algorithm in C.

```
int gcd(int A, int B)
{   int D;
    if (B >A) D=B, B=A, A=D; /* swap */
    do D=(A % B), A=B, B=D;
    while (D >0);
    return A;
}
```

output parameter names become bound to values that have been verified, and these values are then returned to the calling environment. If the predicate involves an actual arithmetic computation, one of its rules will contain an `is` clause, which directs Prolog to actually evaluate an expression and bind its result to a variable so that it can be returned.

An example of ordinary computation expressed as a Prolog predicate is the implementation of Euclid's algorithm for calculating the greatest common divisor of `A` and `B` [Exhibit 13.13]. This recursive definition has three rules. The first rule stops the recursion when the gcd has been found. The second rule does the work. It is a recursive axiom that says that the gcd of two numbers, `A` and `B`, is also the gcd of `B` and `(A mod B)`, and the gcd of every remainder calculated. The third rule is invoked only when the second argument is larger than the first, and it simply reorders the arguments so that the larger one is first.

Let us compare the Prolog gcd program with the same algorithm expressed in C [Exhibit 13.14]. We see that the C program requires more syntactic details—such things as type declarations for the parameters and an explicit `return` statement. As in *Miranda*, Prolog variables are untyped, but objects are typed and their types are deduced by the system. Return values are implemented as output parameters in Prolog, and the value is returned by Prolog's action (during unification) of instantiating the output parameter.

The three Prolog rules for gcd are exactly echoed in the remaining three lines of the C program. Rule one corresponds to the `while`—both stop the recursion/iteration when the answer has been calculated. Rule two corresponds to the `do`—both perform one `mod` operation and shift the arguments to the left for the next recursion/iteration. Finally, rule three corresponds to the `if`—both just swap the inputs. Thus we see that an axiomatic expression of an algorithm can look and work very much like a procedural formulation.

Sorting

Algorithms, such as sorts, that manipulate data can also be expressed axiomatically. An axiomatization of quicksort is shown in Exhibit 13.15. The `quicksort` function is expressed as a recursive function with three rules. Rules 4 and 5 are the base cases for the recursion, and they stop it when

Exhibit 13.15. Quicksort in Prolog.

```

1. | split(_, [], [], []).
2. | split(Pivot, [Head|Tail], [Head|Sm], Lg) :-
   |     Head<Pivot, split(Pivot, Tail, Sm, Lg).
3. | split(Pivot, [Head|Tail], Sm, [Head|Lg]) :-
   |     Pivot<Head, split(Pivot, Tail, Sm, Lg).
4. | quicksort([], []).
5. | quicksort([Head|[]], Head).
6. | quicksort([Pivot|Unsorted], AllSorted) :-
   |     split(Pivot, Unsorted, Small, Large),
   |     quicksort(Small, SmSorted),
   |     quicksort(Large, LgSorted),
   |     append(SmSorted, [Pivot|LgSorted], AllSorted).

```

the list to be sorted has zero elements or one element. Rule 6 does most of the work. It separates the argument (a list) into three parts (smaller values, pivot, and larger values), sorts the parts, then appends them together in sorted order. `Quicksort` calls a subroutine, named `split`, that actually does the separation.

The task of `split` is to separate the list into a sublist of small values and a sublist of large values by comparing each list element to the `Pivot`. The input arguments are `Pivot`, the first value on the sublist that is being sorted, and `Unsorted`, the rest of that sublist. The value of `Pivot` will be compared to each element of `Unsorted`, and the elements of `Unsorted` will be divided into two lists, those smaller than `Pivot` and those that are larger. The last two arguments, `Small` and `Large`, are output parameters that `split` will use to return the sorted `Small` and `Large` sublists.

`Split` is implemented recursively, by three rules. Rule 1 is the base case for the recursion; if the argument is the null list, it will return immediately with null lists as its output parameters. Each time `split` is called with a list of at least one item, either the second rule or the third will be used. In both cases, the symbol `Head` is bound to the first item remaining unsorted list, and `Tail` is bound to whatever remains. On the next call, the first item of `Tail` will be peeled off. This will continue until the remaining tail is null, at which time the first rule for `split` will terminate the recursion.

The second rule for `split` is applied whenever the first item on the `Unsorted` list is smaller than the `Pivot`. This first item is peeled off and remembered, then `split` calls itself recursively to process the items remaining on the unsorted list. When this recursive call returns, all items smaller than the `Pivot` will be on the list `Sm`, and all larger ones will be on the list `Lg`. Since the `Head` element was smaller than `Pivot`, it belongs on the `Sm` list, and so it is appended to the front of that list. The extended `Sm` list and the `Lg` list are then returned to the calling routine.

Exhibit 13.16. Quicksort in C.

Notes on this code are given in Exhibit 13.17.

```

int * split( int *first, int *last )
{
    int *small;           /* small is a left-to-right scanner. */
    int *large;          /* large is a right-to-left scanner. */
    int swap, pivot = *first;
    int *scan;

    for( small=first, large=last+1; ; )
    {
        while (* ++small < pivot); /* Scan until large item is found. */
        while (* --large > pivot); /* Scan until small item is found. */
        if (small >= large) break; /* Quit if scanners have crossed. */
        swap = *large; *large = *small; *small = swap;
    }
    *first = *large; *large = pivot;
    return large;          /* This marks the split point. */
}

void quicksort(int * first, int * last)
{
    int * split_point;

    if (last<=first) return; /* Only one item -- no action needed. */
    split_point = split (first, last);
    quicksort (first, split_point-1);
    quicksort (split_point+1, last);
}

```

The third rule for `split` is applied whenever the first item on the `Unsorted` list is larger than the `Pivot`. Its operation is just like the second rule, except that, because the `Head` element is bigger than the `Pivot`, it is concatenated to the `Lg` list.

Let us look at how the `Prolog` proof system is being used here to implement a sort procedure, and compare it to a quicksort program written in a procedural language. In `C`, the code for `quicksort` uses recursion, explicit iteration, explicit conditionals, comparisons, pointers, and assignments [Exhibits 13.16 and 13.17]. In contrast, the `Prolog` code uses only recursion and comparison. How is the rest of the work done? First, note that in both languages, the main `quicksort` routine is recursive. However, the `split` routine is tail-recursive in `Prolog` but iterative in `C`, for efficiency. The act of calling a routine recursively looks just the same in the two languages, and it has the same semantic effect.

The data structures used are different; the `Prolog` version sorts a list, while the `C` version sorts

Exhibit 13.17. Notes on the C quicksort.

1. The parameters `first` and `last` must be pointers to the beginning and end of an array of integers. On exit from `quicksort`, the values in this array are in sorted order. Pointers, rather than subscripts, are used to index the array.
2. A sentinel value equal to `maxint` must follow the last data element to stop the left-to-right scanning pointer in the case that the pivot value is the largest value in the array.
3. The inner `while` loops identify the leftmost and rightmost elements that are stored in the wrong part of the array. These two elements are then swapped.
4. After exit from the `for` loop, all items to the left of `small` are small (less than or equal to pivot value) and all items to the right of `large` are large. The `large` scanner points to a small element, and `small` either points to the same thing or points to the large element on `large`'s right.
5. Before returning, the pivot element is swapped into the middle, between the small and large elements. This is its final resting place. The small and large areas remain to be sorted by future recursive calls on `quicksort`.

an array. The action of moving through the elements of the list is accomplished in `Prolog` by recursively binding a local name, `Tail`, to the list with its head removed. This recursive binding takes the place of the increment (“++”) and decrement (“--”) operations in C, both of which have assignment as a side effect.

Binding and concatenation are used in `Prolog` in place of the assignment operations in C. As `Prolog`'s `split` pulls each item off the unsorted list, it binds the item to a local variable name, `Head`. Since the function is called recursively, once for each element on the unsorted list, enough bindings are created to hold all the list elements. As each call to `split` returns, the bound item is appended to the list returned by the lower level. In contrast, the C version uses iteration instead of recursion to perform the split operation. New storage is not created; rather, assignment is used to swap pairs of values in the original storage area.

The sequential execution of C statements is echoed exactly in the sequential application of the subgoals in each `Prolog` rule. Thus the main routine, `quicksort`, looks almost the same in the two languages.

Finally, in the C code, explicit `if` statements are used to end recursion in `quicksort` and to end the `split` operation, and `while` statements are used to determine whether an element belongs in the small or the large part of the array. In `Prolog` all this is accomplished by the elaborate pattern matching algorithm (unification) that is built into the proof system. `Prolog` uses unification to

select which rule to apply when `split` is called, which determines whether the next step will add an element to the small list or the large list, or end the recursion.

13.5.6 Cuts and the “not” Predicate

Two major theoretical results have had a strong bearing on Prolog: clausal logic is *complete* but *not decidable*. So although every true clausal theorem can be proved, no effective procedure can ever exist that will always produce a proof and terminate in a finite amount of time. This means that if the Prolog proof system relied on resolution alone, a programmer might not know of not knowing whether a given query would ever be answered. Prolog does have a way, called a cut, to control the proof process so that a programmer can avoid being trapped in lengthy deductions that seem likely to be fruitless. However, when the cut operation is used for this purpose in Prolog, it destroys the completeness of the proof system and leaves open the possibility that a provable goal might fail.

Cuts

A cut is written as “!” and may appear as one of the conditions in a rule.³ Informally, a *cut* prunes off one branch of the proof-search tree by telling the proof system to abandon a chain of reasoning under certain conditions. In some ways it is analogous to a `break` instruction.

Perhaps the best way to think of a cut is to imagine that it is a barrier, placed by the programmer in a rule, to stop fruitless backtracking. Consider a rule with several terms:

$$P: -Q, R, S, !, T, U, V.$$

In trying to satisfy this rule, the proof system starts by searching for a unification of conditions Q , R , and S . Backtracking might occur several times during this search, and control might go back as far as condition Q . If Prolog fails to satisfy this part of the rule, it will go on to try the next rule for P . However, if the conditions Q , R , and S on the left are eventually satisfied, control passes through the cut to the conditions T , U , and V on the right. At this point, all variable bindings for conditions Q , R , and S are frozen, or committed, and the information that would permit Prolog to backtrack back through these conditions is discarded.

The proof system now begins to try to find a unification for conditions T , U , and V that is consistent with the frozen bindings. Again, a great deal of backtracking can happen among these clauses, and, perhaps, some unification of the whole rule may be found. In this case, the rule succeeds, and a unification is returned. If (during backtracking) control ever returns to the cut, it means that the attempt to unify conditions T , U , and V with the frozen bindings has failed. At this point, the two pruning actions of the cut take place:

1. Instead of returning to reinstantiate the left part of the rule, the entire rule fails immediately.

³Do not confuse this meaning of the term “cut” with the meaning of “cut” in a resolution step of clausal logic.

Exhibit 13.18. “Cutting” off a search.

These predicates axiomatize what it means to be on academic probation at a hypothetical university. Loosely, the requirement is that the closer a student is to graduation, the closer the student’s grade point average must be to the minimum gpa for graduation, which is 2.00. (We use the integer form, 200, rather than the decimal form, 2.00, in this code.) Each student is represented in the data base by a fact of the form shown in Exhibit 13.19.

```

1. | year(S, Y) :- student(S,Y,_).
2. | gpa( S, G) :- student(S,_,G).

3. | probation(S, X):-year(S,fr), !, gpa(S,X), X<150.
4. | probation(S, X):-year(S,so1),!, gpa(S,X), X<160.
5. | probation(S, X):-year(S,so2),!, gpa(S,X), X<170.
6. | probation(S, X):-year(S,ju1),!, gpa(S,X), X<180.
7. | probation(S, X):-year(S,ju2),!, gpa(S,X), X<190.
8. | probation(S, X):-year(S,se), gpa(S,X), X<200.
```

2. The goal that caused this rule to be processed also fails and no more attempts are made to satisfy the predicate, even if there are more, untried rules for it.

Safe Cuts. A safe cut is one that cannot possibly cause a provable goal to fail. These are used for the sake of efficiency, in situations where the conditions that guard the various rules for a predicate are mutually exclusive. This is illustrated by the code in Exhibit 13.18, which determines whether a student should be put on academic probation. (Students are listed in Exhibit 13.19.) The grade point average needed to avoid probation becomes higher each year. Thus one rule is included in the `probation` predicate for each grade-point level involved.

If we ask “`probation(tal)`”, the query will fail, because `tal` is a fine student. In the process of answering this query, Prolog will have looked at all the rules for the predicate `probation` and will have failed to satisfy the “`year(S,Y)`” term on all but the last rule. In this case, the cuts in the prior rules have no effect. Similarly, the cuts have no effect during processing of the query “`year(S,se), probation(S,G)`”, and Prolog will return with the instantiation “`S=rae, G=195`” indicating that `rae` is, indeed, in trouble.

Exhibit 13.19. Data for the probation predicate.

```

student(ali, so1, 195).    student(dale, ju2, 189).    student(jan, fr, 372).
student(jess, fr, 142).   student(ken, fr, 199).     student(les, so2, 315).
student(mark, so1, 152). student(nan, so2, 170).    student(pat, ju1, 175).
student(rae, se, 195).    student(sal, ju2, 298).    student(tal, se, 400).
```

Exhibit 13.20. The “not” in Prolog.Definition of `not`:

1. | `not(X) :- X, !, fail.`
2. | `not(_).`

The condition `not(X)` succeeds if `X` fails, and fails if `X` succeeds.

Similarly, there is no inefficiency problem if we ask “`probation(jess,G)`”. Prolog tries rule 3 first, finds that it describes jess well, and succeeds with the instantiation `G=142`. No backtracking ever happens.

However, if we ask “`probation(jan,G).`”, Prolog finds that rule 3 for probation should be processed because jan is a freshman. It then finds jan’s gpa, compares it to the minimum for freshmen, and fails. This failure initiates backtracking, and control backs up to the cut without finding any other instantiations for `X`. If the cut were not there, backtracking would continue; rule 3 would then fail, and rules 4 through 8 would all be tried and fail on the `year` condition. All this work is nonproductive because a student who is a freshman cannot simultaneously fall into any of the other categories. The presence of the cut in rule 3 acts like an `else` clause and eliminates the fruitless testing of all the other rules. In a chain of exclusive conditions, therefore, each rule except the last should be written with a cut.

Cuts Implement the Operator `not`. Perhaps the reader has noted the conspicuous absence of “`not`” in any of the examples given. One of the restrictions placed on both rules and queries is that the conditions must all be *nonnegative*. This is required in order to permit the use of resolution as a proof system.

Of course, several of the built-in predicates have a negative form also—for example, we have both “`=`” and “`\=`”. The Prolog language does include a built-in `not` operator which can be applied to a condition term. However, `not` is problematical and the results obtained by using it can be misleading. Consider what happens during processing of `not`, whose definition is shown in Exhibit 13.20:

- If `X` is false, rule 1 fails before passing through the cut.
- In that case, rule 2 is used. This rule always succeeds, no matter what its argument is, and it instantiates nothing.
- If `X` is true, control passes through the cut in rule 1 and comes to the `fail`. This term always fails, initiating backtracking.
- When the backtracking reaches the cut, rule 1 fails, and since rule 2 is not used because of the cut, the predicate fails.
- The result is that `not(X)` succeeds if `X` fails and fails if `X` succeeds.

Exhibit 13.21. The trouble with “not”.

Suppose we have the simple predicate `test`:

```
test(S, T) :- S = T.
```

The problems with `not` are illustrated by this transcript of a test run:

```
?- test(3, 5).
no
?- test(5, 5).
yes
?- not( test(5, 5) ).
no
?- test(X, 3), R is X+2.
X=3
R=5.
?- not( not( test(X, 3))), R is X+2.
!error in arithmetic expression: not a number.
```

The responses to the first four queries are exactly what one expects. However, the final one is a surprise, since the query is the double-`not` of the preceding query, but the response is different!

The disturbing fact about `not` is that the results of evaluating the predicates `P` and `not(not(P))` are not necessarily the same, and the results of evaluating the predicates `P` and `not(P)` are sometimes the same! This is illustrated by the simple example in Exhibit 13.21. This difficulty is caused by the fact that `not` is implemented using a cut, and when we backtrack to a cut, the goal fails. Thus we see that the last query in Exhibit 13.21 fails, even though it seems that it should not.

One last difficulty with `not` arises from the fact that clausal logic is not decidable. In practical terms, this means that sometimes we can prove a theorem, T , sometimes we can prove its negation, $\text{not } T$, and sometimes we cannot prove either! In the last case, the theorem is true in some models of the theory but not in others, and we certainly cannot conclude that T is false just because we cannot prove that T is true. However, when Prolog cannot prove that T is true, the predicate `not(T)` succeeds. This is so, even if the reason for the failure of T is that the data base contains too little relevant data or that the programmer made a program-sequencing error! Obviously, we must be extremely careful when using `not`.

Unsafe Cuts. Sometimes a programmer might decide to use a cut when implementing some heuristic part of a computation. In this case, the programmer knows that there is some possibility that the desired solution may lie on that part of the proof-tree that is being cut off. Use of the cut for such purposes is considered to be “impure”, since it destroys the completeness of the proof system. However, in many artificial intelligence and optimization problems, it may not be possible,

because of time constraints, to fully explore the proof-tree. Pruning the tree might eliminate the best solution, and it might even eliminate the only solution, but if the heuristics are skillfully chosen, these unhappy outcomes can be largely avoided. On the positive side, skillfully used heuristic cuts can dramatically speed up processing and reduce the memory requirements of a computation to be within practical limits.

13.5.7 Evaluation of Prolog

Prolog will not be appropriate in applications where efficiency is a major consideration. Its performance is limited by its interactive, interpretive nature, and by the lack of destructive assignment operations. All computation is done through parameter binding, and parameter binding is done by the complex and relatively slow unification algorithm. Further, the programmer has only limited control over execution order. Ordinary applications that can be programmed directly in a procedural language such as C will be more efficient than the same applications in Prolog.

However, Prolog is a very attractive language for applications in which the programmer does not know how to organize the data or the computational process. It can be used to express some kinds of information much more directly than the common procedural languages, and it makes it possible to integrate procedural program elements with nonprocedural ones.

Finally, because order of evaluation is left largely unspecified, Prolog is useful for applications in which parallel evaluation is required. It will be an appropriate language for implementation on computers with highly parallel architecture, such as the Connection Machine.

Exercises

1. What is the difference between computation and deduction? In what sense are they analogous processes?
2. What is a universe of discourse?
3. What does it mean to instantiate a variable? How are variables instantiated in the process of deduction?
4. What is a term? A predicate? A sentence? Make clear what the differences are among these concepts.
5. Give an example of a 3-ary predicate.
6. Given the predicates in Exhibit 13.1, are the following instantiated predicates true or false?
 - a. `odd(13)`
 - b. `odd(7+3)`
 - c. `divide(45,5,9,0)`

- d. `divide(56,5,11,3)`
 - e. `father(Isaac, Abraham)`
7. What is the difference between the meaning of $\exists X$ and $\forall X$?
 8. Which of the following quantified predicates is true? What is the negation of each?
 - a. $\forall Y \text{ even}(Y)$
 - b. $\exists X \text{ even}(X)$
 - c. $\forall X X=X+0$
 - d. $\exists X \forall Y X=Y+1$
 - e. $\forall X \exists Y X=Y*1$
 9. Define and explain the relationship between axioms and theories.
 10. How is it possible for a theory to have more than one model?
 11. What is a proof? A refutation?
 12. Write an example of a Horn clause in the form of a disjunction of terms. Write a logical sentence that is not a Horn clause.
 13. If a theory has only unconditional axioms (no conditional axioms), the theory has very few theorems and all the proofs are quite boring. Explain.
 14. Write examples of conditional and unconditional Horn clauses. Write a Prolog sentence that corresponds to each.
 15. What are the elements of the Prolog programming environment?
 16. A Prolog rule specifies a set of constraints that must be simultaneously true to draw a conclusion. Explain how each part of a rule fits this description.
 17. Given the data base in Exhibit 13.10, how would Prolog respond to the following queries?
 - (a) a. `pretty(violet).`
 - (b) b. `watchout(holly).`
 - (c) c. `watchout(X),pretty(X).`
 - (d) d. `watchout(X),color(X,green).`
 18. Prolog cannot always prove a true assertion and it cannot always disprove a false assertion. Explain.
 19. In Prolog, a fact is a single nonnegated term, such as `pretty(mary)`. The same term, or a similar one such as `pretty(pat)` or `pretty(X)`, can be a query. However, a term that is a fact and one that is a query have different meanings. Explain.

20. Given these facts and rules:

facts	rules
flower(crocus, spring, white).	color(F,C) :- flower(F,S,C).
flower(violet, spring, blue).	color(T,green) :- tree(T).
flower(iris, summer, blue).	pretty(F) :- color(F,red).
flower(rose, summer, red).	pretty(F) :- color(F,blue).
flower(marigold, summer, orange).	grows(X) :- tree(X).
tree(holly)	grows(X) :- flower(X,Y,Z).

Find all sets of instantiations that unify each of the following pairs of clauses:

- (a) a. flower(F, spring, Y), pretty(F).
 - (b) b. flower(F, summer, Y), pretty(F).
 - (c) c. grows(X), pretty(X).
 - (d) d. flower(Z, Y, orange), pretty(Z).
21. Discuss the difference in efficiency between the C quicksort [Exhibit 13.16] and the Prolog version [Exhibit 13.15].
 22. Compare and contrast the operation and efficiency of the Miranda gcd algorithm [Exhibit 12.12] and the Prolog version [Exhibit 13.13].
 23. Compare the clarity of code in the C quicksort [Exhibit 13.16] and the Prolog version [Exhibit 13.15]. Which do you think is easier to understand? Why?
 24. What are the syntactic similarities and differences between the Miranda quicksort script [Exhibit 12.19] and the Prolog quicksort [Exhibit 13.15]?
 25. Compare and contrast the operation and efficiency of the Miranda quicksort script [Exhibit 12.19] and the Prolog quicksort [Exhibit 13.15]?