	Operation/Arguments	0,0	0,1	1,0	1,1
0	Constant zero	0	0	0	0
1	Conjunction (AND)	0	0	0	1
2	Greater than >	0	0	1	0
3	Left side	0	0	1	1
4	Less than <	0	1	0	0
5	Right side	0	1	0	1
6	Exclusive OR	0	1	1	0
7	Disjunction (OR)	0	1	1	1
8	Not-OR (NOR)	1	0	0	0
9	Equivalent ⇔	1	0	0	1
10	Not right side	1	0	1	0
11	Reverse implication ⇐	1	0	1	1
12	Not left side	1	1	0	0
13	Implication ⇒	1	1	0	1
14	Not-AND (NAND)	1	1	1	0
15	Constant 1	1	1	1	1

gate may be implemented using two switches in series. Similarly, an OR gate may be implemented using two switches in parallel. Each input wire to the logic gate controls one of the switches. In each case, if the current is flowing, the switch is closed. For an AND gate, both switches must be closed before a current may flow. For an OR gate, closing either switch allows current to flow.

Typically, the logic gates used in implementing **central processing units** (CPUs) and memory (**ram** or **rom**) are connected together in fixed patterns; however, in programmable logic arrays, the interconnections are specified by programs and thus can change. Understanding logic gates will remain an essential part of computer science, since they are the ultimate building blocks of information processing systems.

FURTHER READING

Gajski, Daniel D. *Principles of Digital Design*. Upper Saddle River, N.J.: Prentice Hall, 1997.

Knuth, Donald. *Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*. Reading, Mass.: Addison-Wesley, 1973.

May, Mike. "How the Computer Got into Your Pocket." *American Heritage of Invention and Technology*, Vol. 15, No. 4, 2000.

Schröder, Ernst. *Vorlesungen über die Algebra der Logik*. Leipzig: Teubner, 1890; 2nd ed., Bronx, N.Y.: Chelsea, 1966.

Sloan, M. E. *Computer Hardware and Organization*. Chicago: Science Research Associates, 1973; 2nd ed., 1983.

Sowa, J. F. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Pacific Grove, Calif.: Brooks/Cole, 1999.

—David Whitten

Logic Programming

In conventional programming languages, a program consists of a linear sequence of direct commands that set the values of variables or establish the program flow. The programmer has to imagine, in advance and in detail, how the program will behave and keep in mind the values stored in the variables at any given execution stage. Logic programming uses a different approach: Logic programs can be considered to consist of purely logical assertions about the problem at hand and its solution. The programmer does not have to consider the exact sequence of operations, only the relations between the variables and objects. He or she concentrates on the logical problem at hand, and not on the coding details.

There are several languages for logic programming, but the best known is **Prolog** (Programming in Logic). This language derives from work done by Alain Colmerauer (1941–) in France and Robert Kowalski (1941–) at Edinburgh University in the early 1970s. Simple examples of actual code in Prolog are a good way of grasping the essential difference between programs written in, for example, **Fortran** and logic programming.

Consider the problem of defining a database of genealogical relations. In Prolog, facts can be stated by writing them followed by a point, as in:

```
father(adam,abel).
father(adam,cain).
```

In Prolog the strings "adam," "abel," and "cain" are called atoms and are used to identify individual objects. The structure "father(adam,abel)" states that Adam is the father of Abel. Lowercase letters are used, because uppercase letters are reserved for variables.

Apart from facts, rules can be also defined in Prolog. If two persons have the same father, they are related. This rule can be formulated in Prolog by writing:

```
related(X,Y) :- father(Z,X), father(Z,Y).
```

The rule states that the variable X (any person) is related to the variable Y if there is a Z that is the father of both X and Y. Of course, there are other possibilities for two persons to be related, but the rule above can be complemented with additional rules that cover the other cases. The syntax of Prolog is so simple that even without knowing the language, the examples given above can be understood immediately. Writing equivalent information using Fortran would be rather cumbersome, because Prolog is geared toward symbolic and Fortran toward numeric processing.

In logic programming, once the facts and rules have been stated, queries can be started using the logic predicates defined in the program. It is possible to ask whether Abel and Cain are related by querying the system with: "related(abel, cain)." In this case the system responds with "yes." In other cases, when the answer is negative, the system responds with "fail." The programmer does not have to code the exact internal operations that the machine uses to arrive at the answer; he is interested only in the logical relations (in this case, the genealogical facts and the rules). Logic programming languages always provide logical inference machinery to fill that gap.

Logic programming is based in the language of predicate logic. The whole of mathematics can be developed in this framework. However, predicate logic is too complex for the computer, because when proving a statement, many alternative logical steps can be followed (i.e., there are many different roads toward a logical proof). In logic programming a query is always handled as an assertion that is either proved or refuted. If proved, the values of the variables that make the assertion true are returned to the user. In our biblical example above, we could ask the system "father(X,abel)"—if there is an X that is the father of Abel. The system responds with "yes" and "X=adam"; that is, it gives back the instantiation of X that makes the assertion true. However, efficiency can be guaranteed only when predicate logic is restricted to a simple subset of logic called *Horn logic*. In this formalism, not all predicate logical formulas can be written. However, Horn logic is a large enough subset of predicate logic so that interesting applications are still possible.

The types of statements that can be formulated in Horn logic are of the form "A and B and C imply D." This is written in Prolog using the inverted notation:

```
D :- A,B,C.
```

The commas between A and B, and B and C represent logical conjunction. The colon and the hyphen represent the inverted implication symbol (this combination is used because there is no symbol on the keyboard for an inverted arrow).

In Horn logic, therefore, statements such as the following can be written directly: "IF (person A is old) AND (person A worked) THEN (person A is retired)." However, a statement such as "IF (person A is human) THEN (person A is a man OR person A is a woman)" cannot be formulated, because in Horn logic the implication cannot lead to a disjunction. Horn logic is therefore incomplete, but proving assertions is much simpler than with the full predicate logic because there is a simple automated procedure that can be used to reduce Horn formulas, called *resolution*.

The resolution rule is more general than other logical inference mechanisms. The proposition *Modus ponens*, for example, is based on the inference scheme

(IF A is B) AND (IF B is C) THEN (A is C)

For example, if Aristotle is human, and humans are mortal, then Aristotle is mortal.

The resolution rule consists of observing that two logical formulas can sometimes be reduced to a simpler form. Assume that the formula (A OR B) is true, and also the formula (C OR NOT(B)). Then, if B is true, necessarily C is true, because NOT(B) is false and we said that (C OR NOT(B)) is true. But if B is false, then necessarily A is true, because we said that (A OR B) is true. In one case C must be true, in the other A. We conclude, therefore, that the formula (A OR C) is true regardless of the truth value of B. We have thus derived a new formula out of the original two, at the same time suppressing the statement B.

The resolution rule can thus be stated as follows: Given two disjunctive clauses, if one statement Z appears in both, once negated and once not, we can

combine all other statements disjunctively, suppressing Z. It can be proved that resolution includes modus ponens and other inference rules used by logicians. However, the resolution rule is more general and can be applied mechanically, just as computers do. Using resolution, Horn formulas can be proved or refuted, and therefore resolution can be used as the sole inference engine for logic programming.

Another important ingredient of logic programming is *unification*, which is used to instantiate variables and match queries with rules. In logic programming variables can be assigned a value only once, not repetitively, as in imperative languages such as Fortran and **Pascal**.

When we pose the query "father(X,abel)," what Prolog does is to look for a matching rule or fact (i.e., one with the same name [functor] and the same number of arguments). Then it "unifies" the query with the rule or fact. In our example the only matching fact is "father(adam,abel)." Both query and fact are made equal by instantiating the variable X with the value "adam." Once assigned this value, X cannot be assigned any other value.

Now it should be clear how logic programming works: The programmers write a set of rules and facts and then query the system. A matching rule or fact is sought, and resolution is used to match the query to one of them. If a matching rule or fact is found, the query is processed further using the resolution rule, until the query is proved or disproved. The entire process is completely automatic.

The dream of the logic programming community has always been to reach the stage where a programmer only "declares" what he or she wants, not how to do it (declarative programming). Prolog uses a special mechanism, called *backtracking*, to look for all possible ways of proving or disproving a query. The programmer states the problem, then the system looks for an answer. This can be very inefficient when the system has to test every possible answer before finding the correct one.

Other languages for logic programming are Gödel and FunLog. The latter tries to incorporate functions into the logic programming framework. Other logic programming schemes work with constraints (special conditions on the variables) and the result is called *constraint logic programming*.

After a surge during the 1980s, the interest in logic programming subsided in the 1990s. However, it remains, together with functional and imperative programming, one of the principal computational paradigms for high-level programming languages.

FURTHER READING

Hill, Patricia, and John W. Lloyd. *The Gödel Programming Language*. Cambridge, Mass.: MIT Press, 1994.

Kowalski, Robert. *Logic for Problem Solving*. New York: North-Holland, 1985.

Lloyd, John W. *Foundations of Logic Programming*. Berlin: Springer-Verlag, 1993.

Shapiro, Ehud, and Leon Sterling. *The Art of Prolog*. Cambridge, Mass.: MIT Press, 1994.

—*Raúl Rojas*

"Look and Feel" Lawsuit

The expression *look and feel* refers to the overall gestalt of a computer or computer program: how the **interfaces** work, how the user interacts with the machine or **software**, and so on. The appropriation of one program's look and feel by another has been the subject of multiple lawsuits over the years. By far the most famous is the lawsuit brought by **Apple Computer** against **Microsoft**, which attempted to prevent Microsoft **Windows** from copying the "look and feel" of Apple's **graphical user interface** (GUI). Microsoft's eventual victory left that company free to continue using and improving the GUI of its popular Windows family of personal computer operating systems.

In the 1980s, Apple introduced its Lisa and **Macintosh** computers with, in the words of the U. S. Court of Appeals that ruled against Apple, a GUI that was "a user-friendly way for ordinary mortals to communicate with the Apple computer." This GUI, based at least in part on ideas developed at **Xerox Palo Alto Research Laboratory**, gave Apple an advantage in the efforts to expand the group of people who were comfortable working with a computer.

Apple's lawsuit was based on copyright law, a claim the Court of Appeals made clear was not about the possibility that the program code could be registered as a literary work. Instead, this case was about a claim of