01 Jan 1986

# PROLOG.

Ralph W. Wilkerson
*Missouri University of Science and Technology*, ralphw@mst.edu

# Prolog

Ralph W. Wilkerson

## A programming language for fifth-generation computing

A programming language was developed in the early 1970s by Alain Colmerauer, based upon the work of Robert Kowalski and others. Prolog, a contraction of "PROgramming in LOGic," uses the formalism of mathematical logic as its primary design principle. It has been studied extensively ever since the Japanese announced their intention to build a new series of fast, intelligent computer systems using this language. This project is popularly referred to as the "Fifth Generation Computing Systems Project."

Prolog has attracted the attention of the artificial intelligence community because of its applications in relational databases, natural languages, automated reasoning, and other areas of symbolic processing. Its declarative nature distinguishes it from other languages such as Fortran and Pascal, which are primarily procedural; that is, programs in these languages consist of statements that specify actions which need to be executed in order to achieve the desired result. In other words, the flow of control necessary to perform some computation is explicitly specified.

Declarative languages such as Prolog, on the other hand, specify the flow of data in a program, and programs become descriptions of a collection of relations or functions to be computed. Thus, the execution of a Prolog program is an application of the definitions (rules) to find an output corresponding to some given input. This type of programming is sometimes called pattern-directed rule-based programming, and it is typical of the activity which takes place in the development of expert systems. For example, WARPLAN, a Prolog program created by D.H.D. Warren, is a general planning system prototype that provides a base for tasks such as programming robots to assemble auto parts or writing programs for constructing possible floor plans for businesses.

### The structure

Currently, there are several Prolog implementations available for microcomputers, with a fairly large difference in their terminology and syntax. The syntax used here is that of micro-Prolog, with its simple front-end that provides a user-friendly environment much closer to the syntax of English. It should be noted that other Prolog syntaxes may not have the same features as micro-Prolog, therefore the items discussed here may not function exactly the same in other forms of Prolog.

Prolog programs are composed of facts and rules. Facts are simple statements about objects and their relationships which contain no logical connectives. We call a collection of facts a database. For example, the statement, "Mary likes calculus," contains two objects, "Mary" and "calculus," and a relationship, "likes." As a Prolog fact, this would be expressed as: likes (Mary calculus).

A rule is an abstract statement about objects and their relationships which contains one or more logical connectives. For example, "Everyone who likes calculus is intelligent," can be represented more formally as, "IF *x* likes calculus THEN *x* is intelligent," where *x* stands for any object. However, Prolog requires that such sentences be in Horn clause Form, which places the conclusion before the condition. Thus, our sentence becomes "*x* is intelligent IF *x* likes calculus," which in micro-Prolog syntax is the rule "intelligent (_*x*) if likes (_*x* calculus)."

Notice that the above rule contains the variable _*x* (micro-Prolog requires that all variables begin with an underscore). In Prolog, a variable only has scope in the statement in which it is defined. The conclusion of the rule, intelligent (_*x*), is referred to as the goal, and the condition of the rule, likes (_*x* calculus), is called a subgoal. In order for the goal to be satisfied, the subgoal must be verified.

One executes a Prolog program by asking questions within the Prolog environment, which makes use of the facts and rules to derive an answer to that question. In micro-Prolog these questions take the form of "is" or "which" queries. In the first case, the answer to the question is either "yes" or "no," and in the latter, it is a list of answers which satisfy the query, if any. The actual answers are arrived at by using depth-first search and backtracking.

Asking questions about a collection of facts expressed in Prolog form is exactly like querying a relational database of facts. For example, "is (likes (Mary calculus))" is equivalent to, "Is it true that Mary likes calculus?" If Prolog is able to confirm this fact in the database, then it will respond "yes"; otherwise, it will respond "no." In attempting to verify the above query, Prolog utilizes a powerful pattern matching technique called unification to find a match in the relation name (also called the predicate name) and the corresponding arguments. Simply, unification attempts to find a substitution which makes two or more statements identical.

Consider "is (likes (Mary _*x*))", which asks whether there is anything that Mary likes. Again, Prolog will respond "yes." When the question is asked, the variable _*x* is uninstantiated, and Prolog searches for any relation and first argument which unifies with "likes" and "Mary." But since the second argument is a variable, Prolog will instantiate the variable _*x* to "calculus" and display the answer to the user.

Suppose we then ask is (intelligent (_*x*)), or, "Is there anyone who is intelligent?" In this case, the clause "intelligent (_*x*)" is matched with the goal of the rule for intelligent. In order for this goal to be satisfied, the subgoal "likes (_*x* calculus)" must be verified. Prolog proceeds to check the "likes" relation to determine if there is a relation where someone likes calculus. Since the fact "likes (Mary calculus)" is in the database, the subgoal will be confirmed. The goal intelligent (_*x*) will hence be confirmed and Prolog will respond with "yes."

It should be noted here that if we actually wanted a list of all the intelligent persons in the previous query, we would have had to use the "which" form of the question. Furthermore, an "is" query will terminate its search once it has confirmed the query. However, a "which" query continues to search the database until all answers have been exhausted.

This brings us to another point in the construction of Prolog programs. The order in which facts and rules are entered into the Prolog database affects the order in which they are evaluated. For example, suppose we entered the fact "likes (Bob calculus)" into our database after the fact "likes (Mary calculus)". Now the query: which (_*x*: intelligent (_*x*)), which is read, "Give the names of all individuals that are intelligent," would respond with the

```
grandparent-of( _x _z) if grandfather-of( _x _z)
grandparent-of( _x _z) if grandmother-of( _x _z)
grandfather-of( _x _z) if father-of( _x _y) and parent-of ( _y _z)
grandmother-of( _x _z) if mother-of( _x _y) and parent-of ( _y _z)
parent-of( _x _y) if father-of( _x _y)
parent-of( _x _y) if mother-of( _x _y)
brother-of ( _x _y) if male ( _x) and parent-of ( _z _x) and parent-of ( _z _y)
                and NOT (EQ( _x _y))
sister-of ( _x _y) if female ( _x) and parent-of ( _z _x) and parent-of ( _z _y)
                and NOT (EQ( _x _y))
uncle-of ( _x _y) if male ( _x) and parent-of ( _z _x) and grandparent-of ( _z _y)
aunt-of ( _x _y) if female ( _x) and parent-of ( _z _x) and grandparent-of ( _z _y)
cousin-of ( _x _y) if parent-of ( _z _x) and parent-of ( _w _y) and brother-of ( _z _w)
cousin-of ( _x _y) if parent-of ( _z _x) and parent-of ( _w _y) and sister-of ( _z _w)
husband-of ( _x _y) if father-of ( _x _z) and mother-of ( _y _z)
wife-of ( _x _y) if husband-of ( _y _x)
father-of (Bob Rick)
father-of (Bob Bill)
father-of (Bob Mary)
father-of (Rick Jane)
mother-of (Ann Rick)
mother-of (Ann Bill)
mother-of (Ann Mary)
mother-of (Mary Mark)
male (Bob)
male (Rick)
male (Bill)
male (Mark)
female (Ann)
female (Mary)
female (Jane)
```

*Fig. 1.*

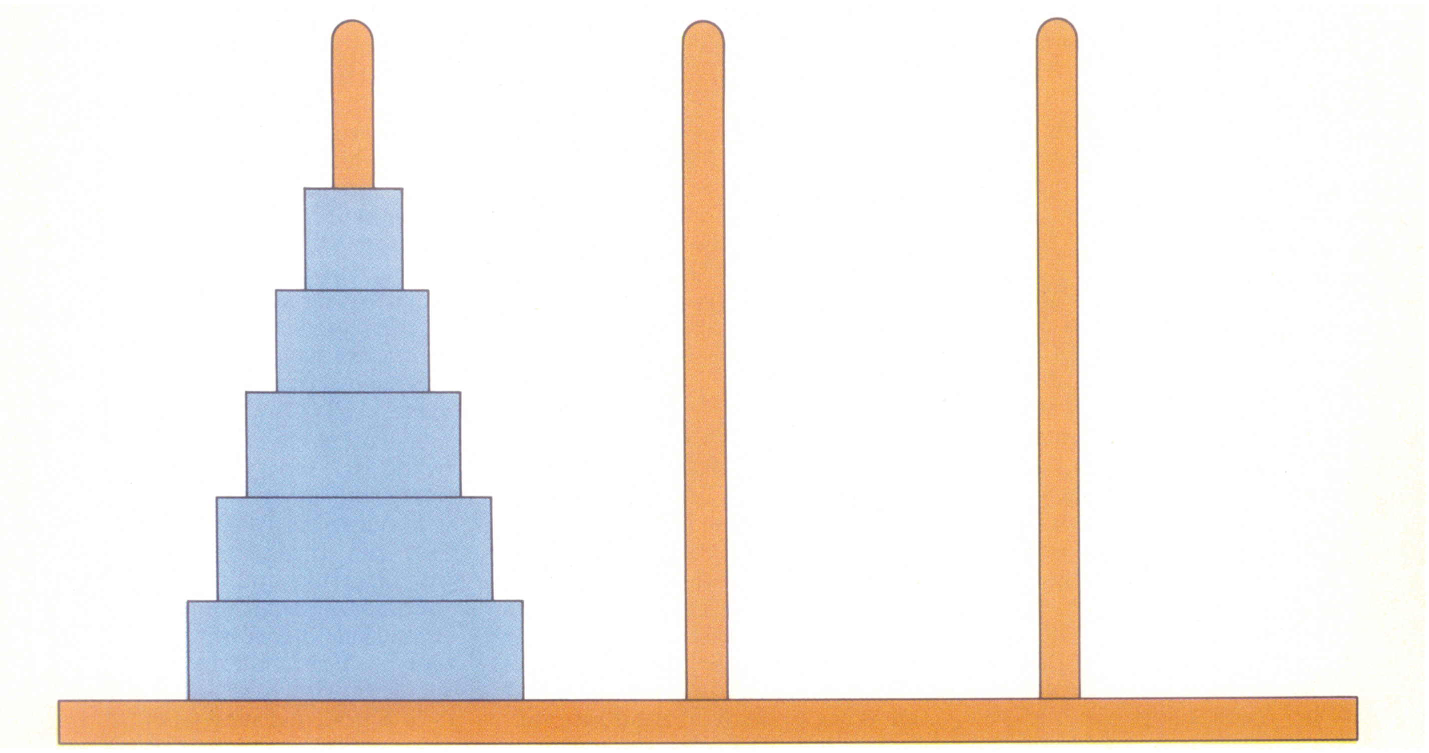

*Fig. 2*

```
Move disk 1 from peg a to peg c
Move disk 2 from peg a to peg b
Move disk 1 from peg c to peg b
Move disk 3 from peg a to peg c
Move disk 1 from peg b to peg a
Move disk 2 from peg b to peg c
Move disk 1 from peg a to peg c
```

*Fig. 3.*

answers "Mary, Bob," in that order. The answer "Bob" was found by resuming the search through the database until other answers were found or the search failed.

Sometimes we want to ask a question or state a rule which involves more than one condition or goal. For example, suppose we add the facts male (Bob) and female (Mary) to our database and ask the question "is (likes (_*x* calculus) and female (_*x*))." In this situation, we have a conjunction of the two conditions "likes (_*x* calculus)" and "female (_*x*)", which must both be satisfied in order for the query to succeed. That is, an instantiation of the variable _*x* must be found which makes both conditions verifiable in our database. In a similar manner, we can have a conjunction of conditions or subgoals in the statement of a Prolog rule. For example, we could change our intelligent rule to "intelligent (_*x*) if likes (_*x* calculus) and reads (_*x* Potentials)."

In addition to being able to query the Prolog program about facts and rules, Prolog has a large number of built-in predicates to aid in the construction of new predicates. These include predicates for various arithmetic operations, string operations, input/output operations, and interfacing with the disk operating system, just to name a few.

## A database program

The logic program in Fig. 1 expresses the kinship relations of a collection of people. The *EQ*(_*x*_*y*) relation contained in the "brother-of" predicate is built-in and attempts to unify its two arguments, succeeding if it can do so.

Consider the question "is (cousin-of (Mark Jane))." In the first cousin-of predicate, Mark will be bound to the variable _*x*, and Prolog will attempt to satisfy the first subgoal "parent-of (_*z* Mark)." This subgoal will unify with the first parent-of relation, and an attempt will be made to satisfy the new subgoal "father-of (_*x* Mark)." A search of the father-of facts fails, so the second parent-of rule is tried. This succeeds when _*x* is unified with Mary in the fact "mother-of (Mary Mark)." Thus, the parent-of relation is satisfied, and the first subgoal of the cousin-of relation has now been satisfied.

Next, the subgoal "parent-of (_*w* Jane)" is attempted, and it is solved in a similar manner as the first subgoal, but _*w* is unified with Rick. Thus, the first two subgoals of the first cousin-of rule have been satisfied. Now the third subgoal, "brother-of (Mary Rick)," is tried, but this goal fails, since the subgoal male (Mary) fails. Hence, Prolog backtracks and attempts to find another _*w* to satisfy the second subgoal, "parent-of (_*w* Jane)," which also fails. Consequently, Prolog backtracks again and attempts to resatisfy the first subgoal and find another _*z* such that "parent-of (_*z* Mark)" is true. But this also fails, and thus the first cousin-of rule fails. In the same manner, the second cousin-of relation is tried, and, as in the first case, the first two parent-of subgoals are satisfied.

Now the subgoal "sister-of (Mary Rick)" is pursued, with the first subgoal female (Mary) being satisfied. The second subgoal, "parent-of (_*z* Mary)," is solved, with _*z* unified with Bob, and the fact "parent-of (Bob Rick)" is confirmed, so that the third subgoal is also true. Finally, since "NOT (EQ(Mary Rick)" is true, the sister-of rule is confirmed, and hence the original "cousin-of (Mark Jane)" goal is verified.

## Application to problem solving

As a second example, consider the game called the Towers of Hanoi. The initial situation is depicted in Fig. 2, where there are three pegs with five disks stacked on the left-hand peg. The disks are stacked such that each one is slightly smaller than the one under it. The object of the game is to move all the disks from the left hand peg to the right hand peg, subject to the conditions that only one disk can be moved at a time, and no disk is ever allowed to be placed on top of a smaller disk. (In the original story, the game has 64 disks of gold stacked on diamond needles. At the time of creation, priests began moving the disks, and when the transfer is complete, the universe will cease to exist.) It can easily be shown that the number of moves in this game is $2^N - 1$, where $N$ is the number of disks to be moved.

A relatively simple algorithm solves this problem using recursion, which makes this a very appropriate problem for using Prolog. Move $N - 1$ disks from the left-hand peg to the center peg using the right-hand peg as an auxiliary peg. Move the $N$th disk from the left-hand peg to the right-hand peg. Finally, move the $N - 1$ disks from the center peg to the right-hand peg using the left-hand peg as an auxiliary peg. For a specific example of this algorithm in action, study Fig. 3 when $N = 3$.

The logic program for the Towers of Hanoi puzzle is given in Fig. 4. In this case, the predicate "towers-of-hanoi" consists of two rules which require four arguments. The first arguments coincide with the peg labels given in Fig. 2, which are used in the transfer of disks, and the fourth argument is the number of disks which are to be moved. The "PP" predicate found in both rules is the built-in print predicate which prints the string following PP and the current values of any variables which might appear in the string. Also, the "SUM" predicate in the second rule subtracts 1 from the current value of _*N* and instantiates _*X* to this value.

The output in Fig. 3 was produced by this logic program by the query "is (towers-of-Hanoi (*a c b* 3))." In attempting to verify this query, Prolog first tries to match the query with the first rule and fails, since the fourth argument is three. Hence, the second rule is tried with _*N* instantiated to three, and the first subgoal succeeds with _*X* instantiated to two. The second subgoal of this rule is a recursive invocation of the towers-of-hanoi predicate, with which the fourth argument is now

```
towers-of-hanoi (_FROM _TO _AUX 1) if
          PP (Move disk 1 from peg _FROM to peg _TO)
towers-of-hanoi (_FROM _TO _AUX _N) if
          sum (_X 1 _N) and
          towers-of-hanoi (_FROM _AUX _TO _X) and
          PP (Move disk _N from peg _FROM to peg _TO) and
          towers-of-hanoi (_AUX _TO _FROM _X)
```

*Fig. 4.*

equal to two. Again, the first rule fails to match, but the second matches, and this time $\_X$ is instantiated to one by the first subgoal. The recursive call to towers-of-hanoi is made again with the first rule matching this time. The PP subgoal always succeeds, and thus the first line in Fig. 3 is printed out. Thus, the second subgoal of the second rule succeeds, and then the third subgoal of this rule succeeds, with the second line of output being printed. Finally, for the fourth subgoal to succeed, the call "towers-of-hanoi (*c b a* 1)" must succeed, which it does, with the third line of output resulting. The reader is encouraged to walk through the remainder of the execution of this query, carefully writing all the recursive calls made along the way.

## Yet to come

Prolog, a logic programming language, combines the use of goal-oriented logic within a framework which is closely related with the manner in which humans think. Prolog has the capability to explain its decision-making process when it attempts to verify a goal. Currently, there are a number of researchers exploring parallel execution of the subgoal conditions within a rule in order to speed up the search process. A new language called PARLOG, for Parallel Programming in Logic, is presently being used to study the parallel evaluation of rules. Prolog is still a relative newcomer in computer languages and is continually under modification as new ideas about its structure and purpose change. Whatever its final form, however, logic programming languages will play a major role in the development of fifth-generation computing methodologies for knowledge processing and artificial intelligence.

## About the author

Ralph W. Wilkerson is Associate Professor of Computer Science at the University of Missouri-Rolla. □

Fujitsu

*Using advanced systemization, the Fujitsu Software Plant programming goal is achieving high reliability and quality in creating programs for language processing, general/special purpose system control, and native language/graphics/voice processing.*