MACLENNAN83
Bruce J. MacLennan; Chapter 13, Logic Programming: PROLOG, Principles of Languages: Design, Evaluation and Implementation. New York: Holt, Rinehart and Winston, 1983, pp. 499-521.

This textbook has descriptions of a number of interesting languages including PROLOG. PROLOG, which stands for PROgramming in LOGic, is a very interesting programming language originally designed for Artificial Intelligence work in theorem proving. It was invented in 1972 by Phillippe Roussel of the Groupe d' Intelligence Artificielle of the University of Marseille. Most of the early papers about it were in French and little notice was taken of PROLOG until the Japanese adopted it for use in their Fifth Generation Computer Project around 1980. There are a number of different interpreters and compilers for PROLOG written in ALGOLW, FORTRAN, Pascal, and PROLOG itself, so it is a "real" language, although it is not clear whether any large systems have been written in it.

There is only one form in PROLOG:

```
P <- Q
```

which is read "P is true if Q is true". P and Q are called predicates, and either may be empty. P is the conclusion and Q is the goal. If Q is empty, the statement reads "P is true always" and is used to assert facts. If P is empty, then this is taken to be a statement of negation that the system tries to disprove by finding counter examples. For example:

```
Parent(x,y) <- Father(x,y).
        {x is the parent of y if x is the father of y}

Parent(x,y) <- Mother(x,y).
        {x is the parent of y if x is the mother of y;
```

these two statements together form an "or":

```
Parent(x,y) if Father(x,y) or Mother(x,y)}

GrandParent(x,z) <- Parent(x,y), Parent(y,z).
        { "," here means "and"}

Father(George, Sue) <-.
        {a fact, George is the father of Sue always}

Mother(Sue, John) <-.
        {capital letters on variables denote constants}

<- GrandParent(George, x)
        {there is no x such that George is the grandparent
         of x.  The system disproves this by finding such
         an x, namely John}
```

The clauses all have zero or one conclusion and zero or more goals which are "and"ed together using commas. This is called Horn clause form. Of course, there may be multiple definitions of the same conclusion, and the definitions may be recursive. Since the PROLOG system will often choose the wrong clause to apply, backtracking is an important part of all PROLOG implementations. The process of applying clauses to solve a problem is a form of pattern-matching, and is called "unification" in PROLOG.

There is much interest in PROLOG due to its unique properties. First, it is claimed to be the first totally non-procedural language to be implemented. It completely separates the logic of programs from the control of how the operations should be carried out. Thus, the statements in PROLOG are unordered;

moving a statement around does not change the meaning of a PROLOG program, as it would in almost all other programming languages. PROLOG also has no data structures per se, and all data objects are inherently abstract, since you can only access data through the predicates that operate on them. There is no distinction between input and output variables in PROLOG, so "<- Fib(3,x)" asks for the third Fibonacci number, and "<- Fib(x,3)" asks for the number whose Fib equals 3. PROLOG programs tend to be very short and clearly do not contain extraneous detail, and therefore should be easy to verify (especially since the control portions are entirely absent). In addition, PROLOG programs seem ideally suited to parallel and coroutine implementations since there are often a number of different predicates to be tried and there is no ordering constraints on the various goals. Additionally, PROLOG may be considered a "data flow" language since predicates can wait around for input and then operate in parallel.

One of the big questions about PROLOG is whether it can be implemented efficiently. It is clear that the control aspect of the problem, i.e. which predicates are applied in which order, will have a significant impact on overall performance. The text suggests that some implementations provide external mechanisms (e.g. a separate language from PROLOG) for specifying hints about the control. It should be emphasized that these hints cannot affect the program's correctness; only its performance. Kowalski points out (see below) that data base systems using the relational model have for a long time successfully separated logic from control, so there is clearly hope for programming in general.

Another problem with PROLOG is that there seems to be no way to express negation as a fact, e.g. "there is no x such that P(x)". Similarly, PROLOG cannot answer "Is it true that for all x, P(x)" in the affirmative because the absence of a counter-example does not imply truth. Clearly, the Japanese believe that these problems can be overcome and that PROLOG is the language to use for AI research. The text does a good job of introducing PROLOG and showing its strengths and apparent weaknesses.

Some of the information in this summary actually comes from Robert A. Kowalski, "Algorithms = Logic + Control" CACM 22, 7 (July 1979) pp. 424-436, which explains some more about logic programming and the ideas in PROLOG and provides more insight into the benefits of logic programming in general.