# An Introduction to the PROLOG Programming Language

Jack L. Watkin
Department of Electrical and Computer Engineering
University of Dayton
Dayton, Ohio 45469–0232 USA
watkinj1@udayton.edu

## ABSTRACT

We provide an introduction to PROLOG—a programming language based on predicate calculus.

## KEYWORDS

Logic programming, predicate calculus, PROLOG.

Jack L. Watkin. 2017. An Introduction to the PROLOG Programming Language. *CPS 499-03: Emerging Languages, University of Dayton, Ohio 45469–0232 USA, Spring 2017,* 3 pages.

## 1 INTRODUCTION

Prolog is a logic programming language developed in the early 1970s for artificial intelligence applications. PROLOG's programming paradigm causes it to stand in contrast to procedural programming languages. Fig. 1 situates PROLOG in relation to other programming paradigms and languages [1]. Since logic programming is based on first-order predicate logic (FOPL), rather than describing *how* to find a solution, PROLOG programs describe *what* a solution to a problem looks like. Then, through the processes of *resolution* and *unification*, the PROLOG engine attempts to search for a solution that proves a goal true. If PROLOG cannot prove a goal, then PROLOG assumes the goal to be false.

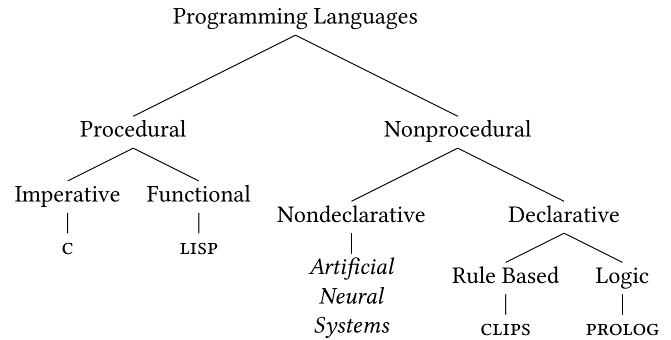


**Figure 1: A hierarchy of programming paradigms and languages (adapted from [1]).**

## 2 FOPL AND HORN CLAUSES

Fopl is a formal system of logic which uses variables, predicates, quantifiers, and logical connectives to produce clauses. PROLOG relies on a restricted form of this called Horn clauses. Horn clauses are FOPL clauses which conform to one of the three forms detailed in Table 1 [2]. All statements in PROLOG match one of these three forms for Horn clauses.

**Table 1: Types of Horn clauses.**

| Form | Horn Clause Type | PROLOG Name |
|---|---|---|
| $\{\} \subset B_1 \wedge \ldots \wedge B_n, n \geq 1$ | Headless | Goal |
| $A \subset \{\}$ | Headed | Fact |
| $A \subset B_1 \wedge \ldots \wedge B_n, n \geq 1$ | Headed | Rule |

## 3 RESOLUTION AND UNIFICATION

A PROLOG program is consists of facts and rules. To run the program, the user must ask questions (called *goals*) of the program. When a goal is given, the PROLOG engine attempts to match the goal with the head of a headed Horn clause, which can be either a fact or a rule. Prolog works backward from the goal through a process called *resolution* to find a combination of facts and rules which can be used to prove the goal. Thus, PROLOG is called a *backwards chaining* system. The following is an example of a series of facts (sometimes called a database) that describe a binary tree and a rule which defines a path between two nodes [2].

```
edge(a,b).
edge(a,c).
edge(b,d).
edge(c,e).

path(X,Y) :- edge(X,Z),path(Z,Y).
path(X,Y) :- edge(X,Y).
```

The `path` predicate defines a path between the two nodes in two ways. A path is either either an edge from `X` to `Y` or from `X` to some `Z` which there is also a path from `Z` to `Y`. The user can then query the program by giving goals to determine whether the goal is true or to find all instantiations of variables which make the statement true. One such statement may be `path(a, c)`, which asks if there exists a path between nodes `a` and `c`. Another goal, `path(a, E)`, queries the values of `E` that make this goal true. Note that the case of the first letter of the variable indicates whether the term is interpreted literally (lowercase) or as a variable (uppercase).

To produce these results PROLOG, undergoes the processes of resolution and unification. When the goal `path(a, c)` is given,

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
*CPS 499-03: Emerging Languages, University of Dayton, Ohio 45469–0232 USA*
© 2017 Copyright held by the owner/author(s).

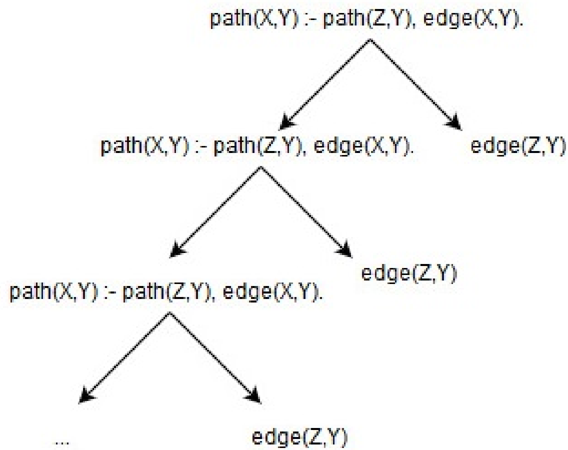


**Figure 2: Resolution graph illustration of infinite recursion.**

PROLOG performs its resolution algorithm through the following steps:

```
1. {} :- path(a, c).
2. path(X, Y) :- edge(X, Y).
3. path(a, c) :- edge(a, c).
4. edge(a, c) :- {}.
5. path(a, c) :- {}.
6. {} :- {}
```

At step 3, PROLOG performs pattern matching between `path(a, c)` and `path(X,Y)`. In matching these terms, PROLOG unifies `X` and `Y` with the values `a` and `c`. On steps 4 and 5, the fact `edge(a, c)` is resolved with the statement `path(a, c) :- edge(a, c)` to deduce that `path(a, c)` is true.

## 4 TREES AND RECURSIVE LOGIC

In PROLOG, statements are evaluated from left to right. This causes a problem when utilizing recursive predicates. Consider following rewrite of the earlier tree searching example:

```
path(X,Y) :- path(Z,Y),edge(X,Z).
path(X,Y) :- edge(X,Y).
```

In FOPL, a clause of a similar format would cause no issue, because there is no explicit order in FOPL, meaning that Z will be bound before the predicate `path(Z,Y)` is evaluated. However, because PROLOG statements are evaluated from left to right, Z will never be bound to a value. The tree in Fig. 2 illustrates this idea. The left to right evaluation of PROLOG causes this tree to be searched in a depth first fashion, which leads to infinite expansion of the rule. Because of this, it is important to ensure that terms can be bound during resolution to values before they are used recursively.

## 5 PRACTICAL APPLICATIONS

### 5.1 Natural Language Processing

One application of PROLOG is *natural language processing* because PROLOG's reliance on FOPL causes it naturally to act as a parser. One could imagine facts as terminals and rules as non terminals or production rules. Consider the following simple grammar for English:

| | | | |
|---|---|---|---|
| ($r_1$) | *<sentence>* | ::= | *<noun phrase> <verb phrase>* |
| ($r_2$) | *<noun phrase>* | ::= | *<determiner> <adj noun phrase>* |
| ($r_3$) | *<noun phrase>* | ::= | *<adj noun phrase>* |
| ($r_4$) | *<adj noun phrase>* | ::= | *<adj> <adj noun phrase>* |
| ($r_5$) | *<adj noun phrase>* | ::= | *<noun>* |
| ($r_6$) | *<verb phrase>* | ::= | *<verb> <noun phrase>* |
| ($r_7$) | *<verb phrase>* | ::= | *<verb>* |

Using this grammar, a PROLOG program can be written to verify the grammatical correctness of a sentence: Note that the input is a list where each element in the list is a single word in the language (e.g., `sentence(["The","dog","runs","fast"])`).

```
sentence(S) :- append(NP,VP,S),
               noun_phrase(NP),
               verb_phrase(VP).

noun_phrase(NP) :- append(ART,NP2,NP),
                   det(ART),
                   noun_phrase_adj(NP2).

noun_phrase(NP) :- noun_phrase_adj(NP).

noun_phrase_adj(NP) :- append(ADJ,NPADJ,NP),
                       adjective(ADJ),
                       noun_phrase_adj(NPADJ).

noun_phrase_adj(NP) :- noun(NP).

verb_phrase(VP) :- append(V,NP,VP),
                   verb(V),
                   noun_phrase(NP).

verb_phrase(VP) :- verb(VP).
```

One drawback of using PROLOG to implement a parser is that left recursive grammars cannot be implemented for the same reasons discussed in § 4.

### 5.2 Graph Algorithms

The following PROLOG program can be used to find cycles in a graph. This predicate finds cycles by searching the unidirectional edges that connect the nodes. A cycle is any path of nodes where the starting node appears at the beginning and end of the list and any other node only appears once in the list. A *cycle* is found when the program finds a list in which the starting node and ending node are the same. This example shows how logic can be directly translated

into PROLOG, allowing for the PROLOG engine to find a solution rather than the programmer.

```
edge(a,b).
edge(b,a).
edge(a,c).
edge(c,d).
edge(d,a).

cycle(Start, Visited) :-
    cycle(Start, Start, [Start], Visited).

cycle(Orig, Start, Path, Visited) :-
    edge(Start,Orig),
    reverse([Orig|Path], Visited).

cycle(Orig, Start, Path, Visited) :-
    edge(Start, Next),
    \+ member(Next, Path),
    cycle(Orig, Next,
          [Next|Path], Visited).
```

## 6 EXERCISES

The following are programming exercises that incorporate essential PROLOG concepts:

(1) A *multiplexer* is a device that selects one of many inputs to output based on a select line input. Define a PROLOG predicate that acts as a 4-input 2-bit *multiplexer*.
Examples:

```
?> mux("1","2","3","4",1,1,Output).
Output = "4".

?> mux("1","2","3","4",0,1,Output).
Output = "2".
```

(2) Define a PROLOG predicate that takes two cities and a route and returns if that route is a valid route. Excluding fact declarations, this program should be approximately 15 lines of code. The roads need not be implicitly bi-directional.
Sample list of cities:

```
road(paris,rouen).
road(paris,lyon).
road(lyon,marseille).
road(marseille,nice).
road(paris,bordeaux).
road(paris,caen).
road(bordeaux,madrid).
road(madrid,cuenca).
```

Examples:

```
?> route(paris,caen,[paris, caen]).
true.

?> route(paris,cuenca,Route).
Route = [paris, bordeaux, madrid, cuenca].
```

(3) Define a PROLOG predicate which takes an infix numerical expression and the result and returns whether the given result is the correct result. The predicate need not handle a divide by 0 error. Use the following grammar:

| | | | |
|---|---|---|---|
| ($r_1$) | *<expression>* | ::= | *<number>* *<op>* *<expression>* |
| ($r_1$) | *<expression>* | ::= | *<number>* *<op>* *<number>* |
| ($r_3$) | *<op>* | ::= | + \| - \| * \| / |

Examples:

```
?> expr([3,*,39,+,3], 120).
true.

?> expr([3,*,39,+,3], 39).
false.

?> expr([3,*,39,+,3], X).
X = 120.
```

## 7 CONCLUSION

PROLOG's reliance on FOPL sets the language apart from other programming languages. Problems are solved in PROLOG by specifying a description of the solution, not a series of instructions to compute a solution. PROLOG obtains this powerful abstraction through the processes of resolution and unification. These ideas allow for PROLOG code to take on a mathematical form which allows for terse programs in comparison to procedural programs.

## REFERENCES

[1] J.C. Giarratano and G. Riley. 1998. *Expert systems principles and programming.* PWS Publishing Company, Boston, MA.
[2] P. Lucas and L. van der Gaag. 1991. *Principles of expert systems.* Addison-Wesley, Boston, MA.