

Logic Languages

Logic languages grew out of the AI research field in the 1970s. One of the main results of symbolic AI was an understanding of how to execute searches within a state space. A state space is a graph, where nodes in the graph represent locations in a virtual space. These locations can be real-valued or symbolic. The connections between nodes may be defined by their relative location in the state space, or they may be defined by a series of rules. In either case, given a specific location in state space, there are some number of other nodes (including zero) that are connected to that location. When executing a search, the nodes may exist in a database, or they may be generated algorithmically as the search moves through the space.

Search is the process of identifying A) if two nodes are connected in the graph, and B) the path connecting the two nodes. The primary concerns of search are efficiency and optimality. Efficiency is concerned with the total time and/or memory required to answer part A. Optimality is concerned with whether the path connecting the two nodes is the shortest path. Different search strategies result in different tradeoffs between time efficiency, memory efficiency, and optimality.

A logic language is an interface to a search engine. It is a syntax for specifying nodes and connections in a state space and then making queries to a search engine about the state space. Like other programming languages, it has naming conventions, types, assignments, expressions and statements. Prolog is a Turing complete language, so it is possible to write interpreters, compilers, and anything else in Prolog.

The unique characteristic of logic languages is that the user's program is generally a specification of facts, followed by a series of queries. The program generally does not look like a sequence of operations to be executed by the computer. Each query specifies a starting location in the state space and one or more goal nodes. The search engine then identifies if the start node and one or more goals are connected.

To summarize, A program written in a logic programming language consists of:

- a set of facts about objects and their relationships
- a set of rules about objects and their relationships
- a set of queries about objects and their relationship

Prolog is the primary surviving logic language. Prolog uses propositional and predicate logic to specify facts (nodes) and relationships (connections). For example,

`hobbit(frodo)`

is a statement that the predicate (property) `hobbit` applies to the atom (symbol) `frodo`. A Horn clause is a statement that connects a single predicate to a set of conditions for that predicate to apply. For example, a Hobbit hole is dry, comfortable, and underground, which is expressed by the following Horn clause. The way to read the clause is that the

hobbitHole predicate applies to **X** if the predicates dry, comfortable, and underground also apply to **X**.

```
hobbitHole(X) :- dry(X), comfortable(X), underground(X)
```

A Horn clause defines a new predicate as a function of other predicates. The predicate on the left applies to the variable **X** only if all of the predicates on the right also apply to **X**. By itself, a Horn clause is not meaningful. However, when combined with factual predicates, it defines relationships in a state space.

Following the rules of logic, a Horn clause of the form $p(A) \text{ :- } q(A)$ is equivalent to $p(A)$ or $(\text{not } q(A))$, which is a Boolean statement of facts, rather than an if-then relationship. Given a Boolean statement of facts, we can use search to identify if we can satisfy the expression, and whether there are multiple possible solutions.

Resolution is the process of identifying when the head (the left side) of Horn clause A exists as one of the terms (the right side) of a different Horn clause B and substituting the terms of A into the list of terms for B. Resolution is the transitive property of logic. In the example below, if we have the first two Horn clauses, then the third one is also true.

```
likesfood(X) :- hobbit(X)
likessecondbreakfast(X) :- likesfood(X)

likessecondbreakfast(X) :- hobbit(X)
```

A simple predicate of the form **comfortable(bagend)** is the head of a Horn clause with no terms, which means it is a fact. Resolution gives us the ability to start searching a database of facts by connected together Horn clauses. As resolution proceeds, it finds atoms, like **bagend** it can bind to variables. Binding a variable with a value is called **instantiation**. The process of identifying the set of values for a set of variables in a Horn clause that make it true is called **unification**.

For example, consider the following set of facts and the predicate **hobbitHole** defined as a Horn clause.

```
dry(ashes).
dry(desert).
dry(bagend).
dry(num2bagshotrow).
dry(prancingpony).

comfortable(chair).
comfortable(sweatshirt).
comfortable(bagend).
comfortable(num2bagshotrow).
comfortable(prancingpony).

underground(sewer).
```

```
underground(goblinhole).
underground(bagend).
underground(num2bagshotrow).

hobbitHole(X) :- dry(X), comfortable(X), underground(X).
```

If we were to make a query about what atoms satisfied the expression `hobbitHole(X)`, the search engine would use resolution to identify possible substitutions for the terms in the `hobbitHole` clause. Both `bagend` and `num2bagshotrow` satisfy all three of the terms.

Prolog Syntax

Prolog programs consist of a series of terms. Each term is a statement of facts or a relationship between terms. Terms can contain, or consist of constants, variables, and structures. Constants, or atoms, are names beginning with lower-case letters or strings enclosed in quotes. A variable is a name that begins with a capital letter. A structure is a predicate followed immediately by zero or more terms in parentheses.

A Horn clause that relates predicates has the following syntax, which states that `term0` is true only if the terms on the right side are true. The following are both Horn clauses. The first is a statement of fact that is always true. The second has a head term that is true if all of the tail (right side) terms are true. The code above shows specific examples.

```
term0.
term0 :- term1, term2, ...
```

All facts, rules, and queries in Prolog must end with a period, and a Prolog program file must end with an end-of-line character.

When searching for a solution, the search engine searches rules in the order in which they appear in the program file. When resolving the terms of a Horn clause, the search engine proceeds from left to right. By default, Prolog stops and returns the first value that satisfies the query. It is also possible to tell Prolog to continue searching from where it left off, forcing it to backtrack and find other possible solutions until it has exhausted the state space.

When writing Prolog, it is important to keep in mind that you are directing a search engine. If you want it to explore a set of alternative solutions, you have to give it enough direction that it can find those alternatives at the appropriate time in the search. This observation is most important when you are specifying that a particular solution should not be equal to a specific value. A **not equals** relationship means that the search engine has found a solution that is not equal to a particular value. The expression, on its own, does not give the engine any particular direction as to where to search for those potential solutions.

Here are examples. We don't include the queries, but they are generally of the form. thing(X), where we want X to be found.

Example: What do people like?

```
/* Example from Programming in Prolog by
Clocksin and Mellish.
Draw the resolution on the board (p. 12 from book).

terminology:
likes is a predicate
mary is a constant
X is a variable.
resolution: making an inference from multiple rules.
instantiation: the assignment of variable to value during resolution is called instantiation.
unification: the pattern-matching process involving multiple resolutions.
*/
likes( mary, food ). /* fact */
likes( mary, wine ).
likes( john, wine ).
likes( john, mary ).
likes( mary, mary ).
common_like( X ) :- likes( mary, X ), likes( john, X ). /* rule */
```

Example: Speaking languages

```
/* speaks.pl
Which pairs of people can communicate with a common language?
This example illustrates that we need to add the requirement that the two people are not the same.
*/
speaks(allen,russian).
speaks(bob,english).
speaks(mary,russian).
speaks(mary,english).
talkswith(Person1,Person2) :- speaks(Person1,L), speaks(Person2,L), Person1 \= Person2.
```

Example: CS Course Dependencies

/* Stephanie's attempt to determine the prereqs for CS courses
at Colby.

If I say the indirect depends on prereq instead of on direct, then
I run into a problem with infinite recursion.

e.g. prereq(232, 300). is a problem. the stack runs outs.

*/

```
course(151).
course(231).
course(232).
course(251).
course(333).
direct( 151, 231 ).
direct( 151, 232 ).
direct( 231, 251 ).
direct( 231, C ) :- C > 300, course(C).
indirect( A, C ) :- direct( A, B ), direct( B, C ).
prereq( A, B ) :- course(A), course(B), direct( A, B ).
prereq( A, B ) :- course(A), course(B), indirect( A, B ).
```

Example: Logic Puzzle

```
/*
Logic Puzzle. Stephanie lost the reference. It is either from a book by
Clocksin and Mellish or from logic-puzzles.org.
1. The miner working at York River finished with 3 ounces more gold than the
   miner working at Belle Hart.
2. Dilbert Duke finished with 3 fewer ounces of gold than Gil Gonzalez.
3. The miner who found 17 ounces of gold worked the Laurel Ridge claim.
4. Elroy Ellis found 17 ounces of gold.
*/

miner(adamAlford).
miner(dilbertDuke).
miner(elroyEllis).
miner(gilGonzalez).
ounces(11).
ounces(14).
ounces(17).
ounces(20).
claim( belleHart ).
claim( laurelRidge ).
claim( tyrolCreek ).
claim( yorkRiver ).

threeMore( A, B ) :- A is B+3.

earnings( [ finish( adamAlford, _, _ ), finish( dilbertDuke, _, _ ),
           finish( elroyEllis, _, _ ), finish( gilGonzalez, _, _ ) ] ).

solution( X ) :- earnings( X ),
    member( finish( M1, yorkRiver, G1 ), X ), miner(M1), ounces(G1),
    member( finish( M2, belleHart, G2 ), X ), miner(M2), ounces(G2), threeMore(G1,G2),
    member( finish( M3, laurelRidge, 17 ), X ), miner(M3),
    member( finish( elroyEllis, C4, 17 ), X ), claim(C4),
    member( finish( M5, tyrolCreek, G5 ), X ), miner(M5), ounces(G5),
    member( finish( dilbertDuke, C6, G6 ), X ), claim(C6), ounces(G6),
    member( finish( gilGonzalez, C7, G7 ), X ), claim(C7), ounces(G7),
    threeMore( G7, G6 ), C7 \= C6,
    M1 \= M2, M2 \= M3, M1 \= M3,
    G1 \= G2, G1 \= 17, G2 \= 17, G5 \= 17, G1 \= G5, G2 \= G5.
```

Note from Stephanie: These notes are a mash-up of notes from Bruce Maxwell, Ying Li, and Stephanie Taylor