Logic Programming

Prolog, and logic programming languages, take a formal approach to writing programs, using *Predicate Calculus* as the mathematical model. Unlike functional languages, logic languages provide a different type of programming interactivity: ***declarative style***. *Non-procedural programming* relies on an inbuilt computational engine within a language. Rather than describing how the computational engine should behave, a logic program describes the problem and the structure of the desired solution. The logic engine then searches possible structures to find the answer. This search process is not under user control, rather it is prescribed by the rules of logical inference. Logic program is synonymous with ***automated theorem proving***, in that the process of constructing an result is the same process as proving that result, given the input program as axioms.

Languages that do not require knowledge of control structures, internal implementations, and machine level details are called ***higher-level languages***. These languages represent the future of programming constructs. The syntax of high-level languages approaches direct mathematical description.

Example: The specification of a sort program for a set `S` of records might be:

```
For all i,j in S: if i<j then S[i] =< S[j]
```

Note that this program does not specify an algorithm; the choice of implementation strategy is under control of the logic engine.

Technique

Each statement, or clause, in a Prolog program makes an ***assertion***. An assertion can be either a statement of ***fact*** or a ***rule***. The difference between facts and rules is that facts do not contain variables, whereas rules do. Computation is initiated when a query is placed with the set of facts. ***Queries*** are the way to state the goals of a program.

A logic program returns the result of a query. When the query is answered, Prolog returns an example case for which the query is true. When a query result is false, either the query can be shown to be false through deduction, or there is simply no enough information in the program database to deduce a result to the given query.

Prolog generates answers purely through ***pattern-matching***. Patterns without variables must have exactly matching syntax. Variables within patterns can match arbitrary patterns. The matching technique is called ***unification***, which finds an assignment of values to the variables in a form which makes the goal statement syntactically identical to the head of some clause.

Prolog assumes that all available information is in its fact base. All statements that can be proved true are derived from the facts and rules of the Prolog program. This approach works well for most problems, especially for object-oriented and entity models.

It is possible to view Prolog clauses as definitions of procedures. Prolog relationships then become procedure invocations. One significant difference is that logic clauses can be processed ("evaluated") in any order, making logic programming a parallel process. Due to difficulties described below, Prolog itself cannot be executed in parallel. Another primary difference between conventional and logic programming is that the relational forms in logic do not distinguish between input and output.

No Data Types

There are no algorithms and no data structures in Prolog. Data types are defined implicitly by their properties. Properties are defined, as might be expected, by the mathematical model of the data structure. That is, Prolog data structures are necessarily and unavoidably abstract. Accessors, constructors and recognizers are all the same thing.

The implication token in Prolog, `:-`, can be read as "right-hand-side implies left-hand-side". Thus `A :- B` is "`A` is asserted if `B` is asserted".

Example, lists:

Prolog uses a shorthand notation for `cons`, `head`, and `tail`, the constructors and accessors of lists.

Mathematical	*Prolog*
`list = cons[head,tail]`	`list[[X|L]] :- list[L]`
`head = head[cons[head,tail]]`	`:- head[[X|L],X]`
`tail = tail[cons[head,tail]]`	`:- tail[[X|L],L]`

The "list" object is an abstraction, not an implementation. Therefore `[X|L]` can be interpreted as a generic decomposition operator; the underlying model and implementation could be of sets, list, arrays, or any other one dimensional data structure.

Example, append:

```
:- append[[],L,L]

append[ [X|L], M, [X|N] ] :- append[L,M,N]
```

This can be read: If appending `L` to `M` gives `N`, then appending `(cons X L)` to `M` gives `(cons X N)`.

The components of a list, or of any data structure are accessed implicitly, as part of the accessor-like decomposition of a form.

Example, set membership:

```
:- member[X,[X|T]]
member[X,[H|T]] :- member[X,T]
```

Example, sort:

```
:- sort[[]]
:- sort[[X]]
sort[[A,B|T]] :- A =< B, sort[[B|T]]
```

The form `[A,B|T]` separates the first two members of the list from the rest.

No Algorithms

Logic programming is designed to separate the control of a computation from the code. In fact, the ordering of clauses in a Prolog program determines the control flow. It is mandatory to put the base case of an inductive definition before the inductive case, since Prolog looks for matches by sequentially examining each clause in order.

Example, adding integers:

```
:- sum[0,X,X]

sum[succ[X],Y,succ[Z]] :- sum[X,Y,Z]
```

The successor function increments an integer by `1`. The first clause says:

```
0+X=X
```

This is the base of the recursive definition. The second clause is the inductive case. It says:

```
if X+Y=Z, then (X+1)+Y=(Z+1)
```

There are two evaluation methods for Prolog engines, top-down (also called backward-chaining), and bottom-up (called forward-chaining). In ***backward chaining***, we start from the goal and attempt to match clauses until the matching arrives at a fact. Backward chaining is similar to recursion. In ***forward-chaining***, we start from the facts and propagate matches until the goal is reached. Forward-chaining is similar to iteration.

Horn Clauses

The Satisfiability Problem for propositional calculus is NP, so logic programs can conceivably encounter great inefficiencies. To combat this, Prolog and other logic systems often restrict the form of logical assertions to those that are computationally tractable.

All logical propositions can be expressed in ***clausal form***. A clause has the structure:

```
A or B or C... :- W and X and Y...
```

One common subset of clauses is Horn clauses. A ***Horn clause*** is a logical assertion with at most one conclusion. Horn clause forms are:

```
A :- W and X and Y...          (one conclusion)
```
or
```
:- W and X and Y...            (no conclusions, a fact)
```

Prolog notation for Horn clauses uses a comma in place of `and` in the antecedent of the clause, so Prolog code would look like:

```
A :- W, X, Y,...
```

As well, Prolog drops to leading `:-` sign when the clause is a fact.

Resolution

We tend to think of logic as *natural deduction*, that form of logic which is closest to our natural language. Almost all machine implementations of logic, however, use ***resolution***, a computational form of logic. Resolution is a standardized representation of logic consisting of a conjunction of ***clauses***. Each clause consists of a set of ***literals***. A literal is either an atom or a negated atom.

Clauses can be of three different types:

Facts:	`:- mother[Mary, John]`	(Mary is the mother of John)
Goals:	`:- father[John, X]`	(John is the father of someone, X)
Rules:	`parent[X,Y] :- father[X,Y]`	(the father of Y is a parent of Y)

A clause specifies a ***relation*** between its components. In contrast to functions, relations do not have a direction of application; they can be used to find inputs given goals as easily as finding goals given inputs.

Functional form:	`F[X,Y] => Z`
Relational form:	`R[X,Y,Z]`

A relational form can optionally be read as a data structure and as a pattern to be matched.

The resolution technique converts clausal form into sets of literals:

```
A or B or C... :- W and X and Y...
```
becomes
```
{-W,-X,-Y,A,B,C}
```

A resolution step consists of combining two clauses, one with a positive literal and one with the negative literal for the same variable. The members of each clause are combined, while the target literal is omitted, generating a new clause which is asserted into the collection of existing clauses. An example:

```
{-A,B,-C,D} + {A,B,-E}  ==>  {B,-C,D,-E}
```

Horn clauses are those sets with at most one negated literal.

```
A :- B, C               (B and C imply A)
B :- E                  (E implies B)
```

becomes

```
{-A,B,C}
{-B,E}
```

A resolution step would match B with -B, delete both, and form the new clause with the rest of the literals in both the source clauses:

```
{-A,C,E}                (C and E imply A)
```

Control Structures

A Prolog program still has control structures, they are just not under control of the user. As in lambda calculus, control of pattern-matching can occur in two basic ways: from goals to facts, and from facts to goals Also like lambda calculus, the two evaluation strategies can be mixed in any order. *Example,* the Fibonacci relation:

```
fib[0,1]
fib[1,1]
fib[N,E] :- N=M+1, M=K+1, fib[M,G], fib[K,H], E=G+H, N>1
```

The inductive clause says that the Nth Fibonacci number is E, if it is the case that

1. M is N-1th Fibonacci number
2. K is the M-1th number, ie the N-2th number.
3. The Mth Fibonacci is G.
4. The Kth Fibonacci number is H.
5. E is the sum of G (the Mth number) and H (the Kth number).
6. N is greater than 1.

The matching algorithm is straight-forward. Suppose the following fact was asserted (the Fibonacci of 2 is R):

```
fib[2,R]
```

Using a backward chaining strategy, this pattern would be matched against all the left-hand-side goals of the logic program. The first match would be

```
fib[2,R]
fib[N,E] :- N=M+1, M=K+1, fib[M,G], fib[K,H], E=G+H, N>1
```

N gets the pattern 2, E gets the pattern R, generating a new clause with substitutions:

```
fib[2,R] :- 2=M+1, M=K+1, fib[M,G], fib[K,H], R=G+H, 2>1
```

This clause reduces by arithmetic to:

```
fib[2,R] :- M=1, M=K+1, fib[M,G], fib[K,H], R=G+H

fib[2,R] :-      1=K+1, fib[1,G], fib[K,H], R=G+H

fib[2,R] :-      K=0,   fib[1,G], fib[K,H], R=G+H

fib[2,R] :-             fib[1,G], fib[0,H], R=G+H
```

For the clause to be satisfied, each conjunct on the right-hand-side must be satisfied. Thus, the above clause would call `fib[1,G]` and `fib[0,H]`, both of which are ground cases:

```
fib[1,G]
fib[1,1]
```

`G` gets bound to `1`.

```
fib[0,H]
fib[0,1]
```

`H` gets bound to `1`. This leaves `R` equal to `2.`:

```
fib[2,R] :-             fib[1,1], fib[0,1], R=1+1
fib[2,2]
```

Note that the ordering both of clauses and of conjuncts within clauses strongly interacts with the efficiency of the matching algorithm. This means that Prolog fails to achieve control structure independence.

Using a forward-chaining strategy, the fact patterns are matched against the antecedents on the right-hand-side of the clauses.

```
fib[0,1]
fib[1,1]
fib[N,E] :- N=M+1, M=K+1, fib[M,G], fib[K,H], E=G+H, N>1
```

Matching `fib[0,1], M` gets `0, G` gets `1,` and simplifying:

```
fib[N,E] :- N=0+1, 0=K+1, fib[0,1], fib[K,H], E=1+H, N>1
```

This fails due to a contradiction, `N` is both equal to `1` and greater than `1.` The matching then continues with the next match opportunity, `K` gets `0`, `H` gets `1`:

```
fib[N,E] :- N=M+1, M=0+1, fib[M,G], fib[0,1], E=G+1, N>1

fib[N,E] :- N=M+1, M=1,   fib[M,G],           E=G+1, N>1
```

```
fib[N,E] :- N=1+1,        fib[1,G],          E=G+1, N>1

fib[2,E] :-               fib[1,G],          E=G+1, 2>1

fib[2,E] :-               fib[1,G],          E=G+1
```

The matching starts again form the top, with nothing matching `fib[0,1].` Using `fib[1,1], G` is bound to `1`:

```
fib[1,1]
fib[2,E] :-               fib[1,G],          E=G+1

fib[2,E] :-               fib[1,1],          E=1+1

fib[2,E] :-                                  E=2

fib[2,2] :-
```

With no more antecedents, the result `fib[2,2]` becomes a fact. This fact matches the query, binding `R` to `2`, and completing the computation:

```
fib[2,2]
fib[2,R]
```

Whether or not a backward or forward chaining strategy is better depends entirely upon the structure of the fact and rule bases. In the Fibonacci example, note that forward chaining avoids recomputing the Fibonacci numbers in each recursion. When a clause fails (as in the above example), the Prolog engine must ***backtrack***, that is, it must restart pattern-matching from where it previously left off, abandoning the exploration which ended in failure (and in unavoidable wasted effort). Also note that logic programming does not distinguish between input and output, since all structures are relations rather than functions. This makes questions such as the example below as easy to answer as finding output from input:

```
sum[X,3,5]                find X
```

Here we are asking what number added to `3` yields `5`? Subtraction, the inverse of addition, is not a necessary concept. Since Prolog treats structures relationally, inverses are not used.

In fact, there are dozens of matching strategies that have been explored by the theorem proving community.

Built-in Arithmetic

In the examples above, arithmetic simplification is built-in. This is not purely logical since it requires a different kind of evaluation mechanism. To have Prolog evaluate arithmetic using pattern-matching, the rules of arithmetic must be added to the rulebase. The new Fibonacci pattern is:

```
fib[0, succ[0]]
```

```
fib[succ[0], succ[0]]

fib[N,E] :-
      sum[M,succ[0],N], sum[K,succ[0],M], fib[M,G], fib[K,H], sum[G,H,E]
```

The successor pattern computes arithmetic relations by stepping though the numbers from `1` to `N`. However using the successor relation to define numbers is very clumsy since it provides no base system to abstract magnitude.

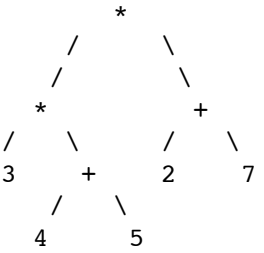
Data Structure Predicates

Compound terms can usually be expressed as trees. To represent a tree in Prolog, we label each node and specify a relationship between that node and its children.

Example: `(3 * (4 + 5)) * (2 + 7)`

String: `*[*[3,+[4,5]],+[2,7]]`

Tree:



Prolog:

```
sum[4,5,N1]
times[3,N1,N2]
sum[2,7,N3]
times[N2,N3,Ans]
```

Problems with Prolog

There are several problems with Prolog, as it was first formulated. Recently work on Prolog III has reduced these difficulties, turning Prolog into a pure constraint-based language.

Sequencing and Hard-wired Control

Logic programming still suffers from necessary sequencing of clauses and antecedents within clauses. This undermines the independence of program and control. The search strategy in Prolog is hardwired, it is depth-first. This also abandons the parallelism of a pure logic language. Since the language does not include control over different search approaches, Prolog search strategies have can be very inefficient, in some cases even non-terminating. Consider the following Prolog program:

```
ancestor[Mary,Tom]
ancestor[M,N] :- parent[M,N]
ancestor[X,Z] :- ancestor[X,Y], parent[Y,Z]
```

The intent of this code is to assert that a person is an ancestor if they are a parent, or if their parent is an ancestor. The problem is that binding ancestor results in an infinite sequence when backward chaining is used. In the first match,

`M` gets `Mary`, `N` gets `Tom`:

```
ancestor[Mary,Tom] :- parent[Mary,Tom]
```

The first rule yields the query "Is Mary Tom's parent?". This fails since there is no match to the `parent` pattern. The search backtracks, trying the second rule:

```
ancestor[Mary,Tom]
ancestor[X,Z] :- ancestor[X,Y], parent[Y,Z]
```

`X` gets `Mary`, `Z` gets `Tom`:

```
ancestor[Mary,Tom] :- ancestor[Mary,Y], parent[Y,Tom]
```

The search is looking for a missing link, `Y`, between `Mary` and `Tom`.

```
ancestor[Mary,Y]
ancestor[X',Z'] :- ancestor[X',Y'], parent[Y',Z']
```

Since the second rule has been used twice, the names of the logic variables must be changed to avoid name conflicts. The second rule is matched, in pursuit of the missing ancestor:

`X'` gets `Mary`, `Z'` gets `Y`:

```
ancestor[Mary,Y] :- ancestor[Mary,Y'], parent[Y',Y]
```

Two queries are generated, and the first one is explored. Again the first rule fails to identify the missing ancestor. The second rule is tried again:

```
ancestor[Mary,Y']
ancestor[X'',Z''] :- ancestor[X'',Y''], parent[Y'',Z'']
```

`X''` gets `Mary`, `Z''` gets `Y'`:

```
ancestor[Mary,Y'] :- ancestor[Mary,Y''], parent[Y'',Y']
```

This match continues by looking for a pattern to resolve the new antecedents:

```
ancestor[Mary,Y'']
ancestor[X''',Z'''] :- ancestor[X''',Y'''], parent[Y''',Z''']
```

The search has entered an infinite loop, looking for a binding for `Y, Y', Y''`, etc. Programming Prolog is not purely declarative, since a programmer must know the exact behavior of the search-and-match algorithm in order to avoid infinite looping.

Cuts

In order to stop needless and possibly infinite search, Prolog introduced the *cut operator*, **!**. In essence this is saying that Prolog does not have control orthogonality, the central reason for the language has been negated. For example, consider the false query "Is the factorial of 4 equal to 15?". The goal is stated as a conjunction of two assertions:

```
fac[4,A], A=15
fac[0,1]
fac[N,E] :- N1=N+1, fac[N1,R1], E=R1*N.
```

The first subgoal, `fac[4,A]`, succeeds, binding `A` to `24`. Next the second goal is attempted and it fails, since A cannot be both `24` and `15`. This causes backtracking to the last successful goal, which was the ground case of the factorial pattern as it was solved recursively. This is `fac[0,R]`. Prolog thinks the match to the first clause, `fac[0,1]`, must have been in error, so it abandons the (correct) ground case, and attempts to bind the current query to the second, inductive clause. With a forward-chaining strategy, this yields:

```
fac[0,R]
fac[N,E] :- N1=N+1, fac[N1,R1], E=R1*N.
```

`N1` gets `0, R1` gets `R:`

```
fac[N,E]  :- 0=N+1, fac[0,R], E=R*N.

fac[-1,E] :-                  E=R*-1
```

Prolog is now looking for factorial of -1, and about to enter an infinite loop. The solution to this problem is to use the cut operator:

```
fac[0,1] :- !
fac[N,E] :- N1=N+1, fac[N1,R1], E=R1*N.
```

This tells the engine to stop if a ground case is satisfied. The problem is that the programmer is now using a procedural metaphor, defining the steps of the actual computation.

Negation as Failure

Prolog uses negation is two senses. At the logical level, negation specifies the complement of its argument. At the control level, when a clause returns false (not true), negation is interpreted as the failure of a clause, which then triggers backtracking.

The essential problem is that the dual use of negation makes it impossible to make and verify negative assertions. The source of the problem is actually in using Horn clauses rather than a fully expressive logic. The limited expressability of Horn clauses eliminates the use of negated clauses (but not negated literals).

Example:

```
append[X,Y,[a,b,c]]
```

yields all possible solutions:

```
X=[], Y=[a,b,c]
X=[a], Y=[b,c]
X=[a,b], Y=[c]
X=[a,b,c], Y=[]
```

The assertion

```
not[not[append[X,Y,[a,b,c]]]]
```

should behave identically to append, since the double negation cancels out. However, the Prolog execution makes the following error. The outer not sets up the inner not as a goal. The inner not sets up append, which succeeds. Therefore the inner not fails, since `not[append[X,Y,[a,b,c]]]` is false. This failure deletes the prior successful bindings of `X` and `Y`. Finally the outer not succeeds, however the logical variables at this point are unbound.