Logic Programming in Perl

Let the computer do the hard work

by

Curtis "Ovid" Poe

Logic Programming in Perl

Let the computer do the hard work

by

Curtis "Ovid" Poe

Introduction

```
A programmer who hasn't been exposed to all four of the imperative,
functional, objective, and logical programming styles has one or more
conceptual blindspots. It's like knowing how to boil but not fry.

Programming is not a skill one develops in five easy lessons.

-- Tom Christiansen
```

By now, many Perl programmers know that the language offers support for imperative, objective, and functional programming. However, logic programming seems to be a lost art. This article attempts to shed a little light on the subject. It's not that Prolog can do things that Perl cannot or vice versa. Instead, Prolog does some things more naturally than Perl -- and vice versa. In fact, while I introduce you to Prolog, I won't be teaching a bunch of nifty tricks to make your Perl more powerful. I can't say what needs you may have and, in any event, the tools that allow logic programming in Perl are generally alpha quality, thus making them unsuitable for production environments.

What is Logic Programming?

Logic programming is somewhat of a mystery to many Perl programmers because, unlike imperative, objective, and functional styles, Perl does not have direct support for logic programming. There is, however, much interest in bringing logic programming to Perl 6. With luck the information presented here will not be merely theoretical.

Logic programming is not as alien as programmers might think. Regular expressions, SQL and grammars are all closely related to logic programming. The shared component of these seemingly disparate technologies is how they *describe* their goals rather than state how to achieve it. This is the essence of logic programming. Rather than tell the computer how to achieve a given goal, we tell the computer what the goal looks like and let it figure out how to get there. Because of this, some refer to logic programming as "specification-based programming" because the specification and the program are one and the same.

This article will focus on `AI::Prolog`. Prolog, though not a "pure" logic programming language, is the most widely used in the field. `AI::Prolog` is a Prolog engine written entirely in Perl and it's very easy to install and use. However, if you start doing serious work in Prolog, I strongly recommend you take a look at Salvador Fandiño's `Language::Prolog::Yaswi`. This module allows access to SWI-Prolog (http://www.swi-prolog.org/) from within Perl. It's more difficult to compile and get running, but it's faster and much more powerful. Luke Palmer's new `Logic` distribution takes a somewhat different approach to the same problem space.

3 Things to Learn

Most programming in Prolog boils down to learning three things: facts, rules, and queries. The basics of these can be learned in just a few minutes and will let you read many Prolog programs.

Facts

Facts in Prolog are stored in what is called the *database* or *knowledge base*. This isn't a PostgreSQL or SQLite database, but a plain-text file. In fact, you would call it a program and I'd be hard-pressed to argue with you, so I won't. In fact, I'll often refer to these as Prolog "programs" just to avoid confusion.

Facts look like this:

```
gives(tom, book, sally).
```

Note: While the order of the arguments is technically not relevant, there is a convention to more or less read the arguments left to right. The above fact can be read as "tom gives the book to sally." Of course, it is sometimes read as "sally gives the book to tom", but whichever order you adopt, you must keep it consistent.

A fact consists of a *functor* (also known as the *head*) and 0 or more arguments, followed by a period. Allowed names for functors generally follow allowed named for subroutines in Perl.

The number of arguments to the functor are known as the *arity*. The functor, followed by a slash and the arity ("gives/3" in this example) is called the *predicate*. A predicate can have as many clauses as you like.

```
gives(tom, book, sally).
gives(tom, book, martin).
gives('George Bush', grief, liberals).
gives('Bill Clinton', grief, conservatives).
```

Note that those are not function calls. Those are merely facts stating the relationship of the arguments to one another. Additionally, "tom" and "book" are each repeated. In Prolog, those each refer to the same entity.

Of course, facts can have a varying number of arguments, even for the same functor. The following are all legal:

```
parent(bob, tim). % parent/2
parent(sue, alex, william). % parent/3
male(sue). % male/1
female(sue). % female/1
frobnitz. % frobnitz/0
```

You can name a predicate just about anything that makes sense, but note that some predicates are built-in and their names are reserved. The document `AI::Prolog::Builtins` has a list of the predicates that `AI::Prolog` directly supports. If you're in the `aiprolog` shell (explained later), you can type `help.` to get a list of built-in predicates and `help('functor/arity')` (e.g., `help('consult/1')`) to get description of how a particular built-in predicate works.

And that pretty much covers most of what you need to know about facts.

Rules

Rules, like facts, are stored in the program. Rules describe how we can infer new facts, even though we haven't explicitly stated them.

```
gives(tom, book, SOMEONE) :-
    person(SOMEONE),
    likes(tom, SOMEONE).
```

This rule states that "Tom will give a book to anyone who Tom likes." Note that we are not telling Prolog how to figure out to whom Tom will give books. Instead, we have merely defined the conditions under which Tom is willing to part with his material possessions.

To understand rules, read the neck operator, `:-`, as "if" and commas outside of argument lists as "and." Further, arguments beginning with upper-case letters are *variables*, such as `SOMEONE` in the rule above. Note that only the first letter needs to be capitalized; `Someone` would also be a variable, as would `SomeOne` or `SOmeoNe`.

Of course, we could simply enumerate the relationships:

```
gives(tom, book, alice).
gives(tom, book, bob).
gives(tom, book, martin).
gives(tom, book, charlie).
gives(tom, book, ovid).
```

However, this quickly become unweildy as the number of people in our program grows.

Queries

Now that we have a rough understanding of facts and rules, we need to know how to get answers from them. Queries are typically entered in the "interactive environment." This would be analogous to the query window in a database GUI. With `AI::Prolog`, a shell named `aiprolog` is included for demonstration purposes though we'll mainly focus on calling Prolog from within a Perl program.

```
?- gives(tom, book, SOMEONE).
```

This looks just like a fact, but with some minor differences. The `?-` at the beginning of the query is a query prompt seen in interactive environments.

You'll also note that `SOMEONE` is capitalized, making it a variable (only the first letter needs to capitalized.) Thus, this query is asking who Tom will gives books to.

```
?- gives(WHO, book, SOMEONE).
```

This query looks like the previous one, but since the first argument is capitalized, we're asking "who will give books to whom?".

```
?- gives(WHO, WHAT, WHOM).
```

Finally, because all arguments are variables, we're asking for everyone who will give anything to anybody.

Note that no code changes are necessary for any of this. Because Prolog facts and rules define relationships between things, Prolog can automatically infer additional relationships if they are logically supported by the program.

Let's take a closer look at this, first focusing on the `aiprolog` shell. Assuming you've installed `AI::Prolog` and said "yes" to installing the `aiprolog` shell, enter the following text in a file and save it as *gives.pro*. Note that lines beginning with a percent sign (`%`) are single-line comments.

```
% who are people?
person(bob).
person(sally).
person(tom).
person(alice).

% who likes whom?
likes(tom, bob).
likes(tom, alice).
likes(alice, bob).

% who has what
has(tom, book).
has(alice, ring).
has(bob, luck).

% who will give what to whom?
gives(WHO, WHAT, WHOM) :-
    has(WHO, WHAT),
    person(WHOM),
    likes(WHO, WHOM).

gives(tom,book,harry).
```

When starting the shell, you can read in a file by supplying as a name on the command line:

```
$ aiprolog gives.pro
```

Alternately, you can *consult* the file from the shell:

```
$ aiprolog

Welcome to AI::Prolog v 0.732
Copyright (c) 2005, Curtis "Ovid" Poe.
AI::Prolog comes with ABSOLUTELY NO WARRANTY. This library is free software;
you can redistribute it and/or modify it under the same terms as Perl itself.

Type '?' for help.

?- consult('gives.pro').
```

The second notation allows you to consult multiple files and add all of them to the knowledge base.

After issuing the `consult/1` command, the shell will appear to hang. Hit *Enter* to continue. We'll explain this behavior in a moment.

Now that you've loaded the program into the shell, issue the following query:

```
?- gives(X,Y,Z).
```

The shell should respond:

```
gives(tom, book, bob)
```

It will appear to hang. It wants to know if you wish for more results or if you are going to continue. Typing a semicolon (;) tells Prolog that you want it to *resatisfy* the goal. In other words, you're asking Prolog if there are more solutions. If you keep hitting the semicolon, you should see something similar to the following:

```
?- gives(X,Y,Z).

gives(tom, book, bob) ;

gives(tom, book, alice) ;

gives(alice, ring, bob) ;
```

```
gives(tom, book, harry) ;

 No
?-
```

That final "No" is Prolog telling you that there are no more results which satisfy your goal (query). (If you hit *Enter* before Prolog prints "No", it will print "Yes", letting you know that it found results for you. This is standard behavior in Prolog.)

One thing you might notice is that the last result, `gives(tom, book, harry)`, does not match the rule we set up for `gives/3`. However, we get this result because we chose to hard-code this fact as the last line of the Prolog program.

How this works

At this point, it's worth having a bit of a digression to explain how this works.

Many deductive systems in artificial intelligence are based on two algorithms: backtracking and unification. You're probably already familiar with backtracking from regular expressions. In fact, regular expressions are very similar to Prolog in that you specify a pattern for your data and let the regex engine worry about how to do the matching.

In a regular expression, if a partial match is made, the regex engine remembers where the end of that match occurred and tries to match more of the string. If it fails, it backtracks to the last place a successful match was made and sees if there are alternative matches it can try. If that fails, it keeps backtracking to the last successful match and repeats that process until it either finds a match or fails completely.

Unification, described in a fit of wild hand-waving, attempts to take two logical terms and "unify" them. Imagine you have the following two lists:

```
( 1, 2, undef, undef,     5 )
( 1, 2,     3,     4, undef )
```

Imagine that undef means "unknown". We can unify those two lists because every element that is known corresponds in the two lists. This leaves us with a list of the integers one through five.

```
( 1, 2, 3, 4, 5 )
```

However, what happens if the last element of the first list is unknown?

```
( 1, 2, undef, undef, undef )
( 1, 2,     3,     4, undef )
```

We can still unify the two lists. In this case, we get the same five element list, but the last item is unknown.

```
( 1, 2, 3, 4, undef )
```

If corresponding terms of the two lists are both bound (has a value) but not equal, the lists will not unify:

```
( 1, 23, undef, undef, undef )
( 1,  2,     3,     4, undef )
```

Logic programming works by pushing these lists onto a stack and walking through the stack and seeing if you can unify everything (sort of). But how to unify from one item to the next? We assign names to the unknown values and see if we can unify them. When we get to the next item in the stack, we check to see if any named variables have been unified. If so, the engine will try to unify them along with the other known variables.

That's a bad explanation, so here's how it works in Prolog. Imagine the following knowledge base:

```
parent(sally, tom)
parent(bill, tom)
parent(tom, sue)
parent(alice, sue)
parent(sarah, tim)

male(bill)
male(tom)
male(tim)
```

Now let's assume we have a rule that states that someone is a father if they are a parent and they are male.

```
father(Person) :-
  parent(Person, _),
  male(Person).
```

In the above rule, the underscore is called an "anonymous vairable" and means "I don't care what this value is." Prolog may still bind the variable internally (though this behavior is not guaranteed), but its value will not be taken into account when trying to determine if terms unify.

Taking the first term in the rule, the logic engine might try to unify this with the first fact in the knowledge base, `parent(sally, tom)`. `Person` unifies with *sally*. The underscore, `_`, unifies with *tom* but since we stated this unification is unimportant, we can ignore that.

We now have a fact which unifies with the first term in the rule, so we push this information onto a stack. Since there are still additional facts we can try, we set a "choice point" in the stack telling us which fact we last tried. If we have to backtrack to see a choice point, we move on to the next fact and try again.

Moving on to the next term in the rule, `male(Person)`, we know that "sally" is unified to `Person`, so we now try to unify `male(sally)` with all of the corresponding rules in the knowledge base. Since we can't, the logic engine backs up to the last item where we could make a new choice and sees `parent(bill, tom)`. `Person` gets unified with *bill*. Then in moving to the next rule we see that we unify with `male(bill)`. Now, we check the first item in the rule and see that it's `father(Person)`. and the logic engine reports that *bill* is a father.

Note that we can then force the engine to backtrack and by continuously following this procedure, we can determine who all the fathers are.

And that's how logic programming works. Simple, eh? (Well, individual items can be lists or other rules, but you get the idea).

Executing Prolog in Perl

Getting back to Perl, how would we implement that in a Perl program?

The basic process for using `AI::Prolog` looks something like this:

```
use AI::Prolog;
my $prolog = AI::Prolog->new($prolog_code);
```

Create a new `AI::Prolog` object, passing Prolog code as the argument. If you prefer, you can wrap the constructor in a `BEGIN` block:

```
my $prolog;
BEGIN {
  $prolog = AI::Prolog->new(<<'  END_PROLOG');
    % some Prolog code goes here
  END_PROLOG
}
```

This is not strictly necessary, but if your Prolog code has a syntax error, it will be a compile-time error, not a run-time error, and you'll get an error message similar to:

```
Unexpected character: (Expecting: ')'. Got (.)) at line number 12.
BEGIN failed--compilation aborted at test.pl line 7.
```

Note that the line number for "Unexpected character" is relative to the Prolog code, not the Perl code.

After the contructor, issue your query:

```
$prolog->query($some_query);
```

And do something with the results:

```
while ( my $results = $prolog->results ) {
    print "@$results\n";
}
```

Results are usually each returned as an array reference with the first argument being the functor and subsequent arguments being the values. If any value is a list, it will be represented as an array reference. We'll see more on that later as we cover lists.

Now let's see the full program:

```
#!/usr/bin/perl
use strict;
use warnings;
use AI::Prolog;

my $prolog;

# If reading from DATA, we need a CHECK block to ensure that
# DATA is available by the time the constructor is called
CHECK {
  $prolog = AI::Prolog->new( do { local $/; <DATA> } );
}

$prolog->query( 'father(WHO).' );
while ( my $results = $prolog->results ) {
    print "@$results\n";
}

__DATA__
parent(sally, tom).
parent(bill, tom).
parent(tom, sue).
parent(alice, sue).
parent(sarah, tim).

male(bill).
male(tom).
male(tim).

father(Person) :-
    parent(Person, _),
    male(Person).
```

If you run this program, it will quite happily print out "father bill" and "father tom." In fact, if you really want to see what's going on internally, after you issue the query you can "trace" the execution:

```
$prolog->query('father(Who)');
$prolog->trace(1); # after the query, before the results
while ( my $result = $prolog->results ) {
  ...
}
```

Running the program again produces a lot of output, the beginning of which matches our description of how logic programming works internally:

```
= Goals:
        father(A)
==> Try:  father(A) :-
        parent(A, B),
        male(A)

= Goals:
        parent(A, C),
        male(A)
==> Try:  parent(sally, tom) :- null

= Goals:
        male(sally)
==> Try:  male(bill) :- null
 <<== Backtrack:

= Goals:
        male(sally)
==> Try:  male(tom) :- null
 <<== Backtrack:

= Goals:
        male(sally)
==> Try:  male(tim) :- null
 <<== Backtrack:

[etc.]
```

Now if you really want to have fun with it, notice how you can rearrange the clauses in the program at will and Prolog will return the same results (though the order will likely change). This is because when one programs in a purely declarative style, the order of

the statements no longer matters. Subtle bugs caused by switching two lines of code usually go away.

Prolog versus Perl

Now that you have a beginning understanding of what Prolog can do and how it works internally, let's take a look at some of the implications of this. By now, you know that the following can be read as "Ovid loves Perl":

```
loves(ovid, perl).
```

Assuming you've consulted a file with that fact in it, querying to find out what I love is simple:

```
?- loves(ovid, WHAT).
```

In Perl, it's also pretty simple:

```
%loves = ( ovid => 'perl' );
$what  = $loves{ovid};
```

But how do we find out who loves Perl? In Prolog:

```
loves(WHO, perl).
```

In Perl, however, we have two options, neither of them particularly good. We can scan the `%loves` hash for entries whose value is Perl, but this is `O(n)` when we want to stick with our simple `O(1)` check. Thus, a more common solution is to reverse the hash:

```
%who_loves = reverse %loves;
$who       = $who_loves{perl};
```

(`AI::Prolog`, being written in Perl, is slow. So the `O(n)` versus `O(1)` argument doesn't hold. Versions written in C are far more practical in this regard.)

But this fails, too. The first problem is that we have duplicated data. If we have to add additional data to the `%loves` hash, we'll have to remember to synchronize the hashes. The second problem is that Perl is just a little too popular:

```
loves(ovid,    perl).
loves(alice,   perl).
loves(bob,     perl).
loves(charlie, perl).
```

If we simply reverse the hash for those entries, we lose three of those names. So we have to play with this some more.

```
while ( my ($person, $thing) = each %loves ) {
    push @{ $who_loves{$thing} }, $person;
}
```

Oh, wait. This fails too. You see, I'm fickle.

```
loves(ovid, perl).
loves(ovid, prolog).
loves(ovid, 'Perl 6').
```

(The quotes around "Perl 6" tell Prolog that this is a single value and that it's a constant, not a variable.)

How do I find out everything Ovid loves? In Prolog, the query doesn't change:

```
loves(ovid, WHAT).
```

In Perl, our original hash wasn't enough.

```
my %loves = (
    ovid    => [ qw/perl prolog Perl6/ ],
    alice   => [ 'perl' ],
    bob     => [ 'perl' ],
    charlie => [ 'perl' ],
);

my %who_loves;
while ( my ($person, $things) = each %loves ) {
    foreach my $thing ( @$things ) {
        push @{ $who_loves{$thing} }, $person;
    }
}
```

Now that's starting to get really ugly. To represent and search that data in Prolog is trivial. Now do you really want to have fun?

```
gives(tom, book, sally).
```

How would you represent that in Perl? There could be multiple gift-givers, each gift-giver could give multiple things. There can be multiple recipients. Representing this cleanly in Perl would not be easy. In fact, when handling relational data, Perl has several weaknesses in relation to Prolog.

- Data must often be duplicated to handle bi-directional relations.
- You must remember to synchronize data structures if the data changes.
- Often you need to change your code if your relations change.
- The code can get complex, leading to more bugs.

Prolog versus SQL

At this point, there's a good chance that you're thinking that you would just stuff this into a relational database. Of course, this assumes you need relational data and want to go to the trouble of setting up a database and querying it from Perl. This is a good solution if you only need simple relations. In fact, Prolog is often used with relational databases as the two are closely related. SQL is a *special purpose* declarative language whereas Prolog is a *general purpose* declarative language.

Firing up SQLite, let's create two tables and insert data into them.

```
sqlite> CREATE TABLE parent_2 (parent VARCHAR(32), child VARCHAR(32));
sqlite> CREATE TABLE male_1   (person VARCHAR(32));

sqlite> INSERT INTO parent_2 VALUES ('sally', 'tom');
sqlite> INSERT INTO parent_2 VALUES ('bill', 'tom');
sqlite> INSERT INTO parent_2 VALUES ('tom', 'sue');
sqlite> INSERT INTO parent_2 VALUES ('alice', 'sue');
sqlite> INSERT INTO parent_2 VALUES ('sarah', 'tim');

sqlite> INSERT INTO male_1   VALUES ('bill');
sqlite> INSERT INTO male_1   VALUES ('tom');
sqlite> INSERT INTO male_1   VALUES ('tim');
```

We can then find out who the fathers are with the following query.

```
SELECT parent
FROM   parent_2, male_1
WHERE  parent_2.parent = male_1.person;
```

This is very similar to Prolog but we are forced to explicitly state the relationship. Many Prolog queries are conceptually similar to SQL queries but the relationships are listed directly in the program rather than in the query. When working with a relational database, we can get around this limitation with a view:

```
CREATE VIEW father AS
    SELECT parent
    FROM   parent_2, male_1
    WHERE  parent_2.parent = male_1.person;
```

And finding out who the fathers are is trivial:

```
SELECT * FROM father;
```

Further, databases, unlike many Prolog implementations, support indexing, hashing, and reordering of goals to reduce backtracking. Given all of that, why on earth would someone choose Prolog over SQL?

As stated earlier, Prolog is a general purpose programming language. Imagine if you had to write all of your programs in SQL. Could you do it? You could with Prolog. Further, there's one huge reason why one might choose Prolog: recursion. Some SQL implementations have non-standard support for recursion, but usually recursive procedures are simulated by multiple calls from the programming language using SQL.

Let's take a look at recursion and see why this is a win.

Recursion and Lists

In Prolog, lists are not data structures. They're actually implemented in terms of something referred to as the "dot functor" (./2). For our purposes, we'll ignore this and pretend they're really data types. In fact, there's going to be a lot of handwaving here so forgive me if you already know Prolog. If you know LISP or Scheme, the following will be very familiar.

A list in Prolog is usually represented by a series of terms enclosed in square brackets:

```
owns(alice, [bookcase, cats, dogs]).
```

A list consists of a head and a tail. The tail, in turn, is another list with a head and tail, continuing recursively until the tail is an empty list. This can be represented as follows:

```
[ HEAD | TAIL ]
```

Note that the head and tail are separated by the pipe operator.

Now if Alice owns things, how do we find out if she owns cats? First, we need to define a predicate that succeeds if a given term is an element of a given list.

```
member(X, [ X | _ ]).
member(X, [ _ | TAIL ]) :-
   member(X, TAIL).
```

The first clause in this predicate states that `X` is a member of a list if it's the head of a list. The second clause states that `X` is a member of a list if it's a member of the tail of the list. Here's how this works out:

```
?- member(3, [1,2,3,4]).
```

Prolog checks the first clause and sees that 3 is not the head of the list: `member(3, [1|2,3,4])`. It then checks to see if it's a member of the tail of the list: `member(3, [2|3,4])`. Again this fails so Prolog tries again and we succeed: `member(3, [3|4])`.

So how do we see if Alice has cats?

```
has(PERSON, THING) :-
    owns(PERSON, STUFF),
    member(THING, STUFF).
```

And the query `has(alice, cats)` will now succeed. However, as mentioned previously, Prolog predicates can often be reused to deduce information that you and I could deduce. Thus, we can find out who owns cats (`has(WHO, cats)`), everything Alice has (`has(alice, WHAT)`), or everything that everyone has (`has(WHO, WHAT)`.

Why does that work? Well, going back to the `member/2` predicate, we can find out if a term is an element of a list:

```
member(ovid, SOMELIST).
```

Or we can find all members of a list:

```
member(X, SOMELIST). % assumes SOMELIST is bound to a list
```

Now that might seem kind of nifty, but appending lists in Prolog is truly sublime. Many consider understanding the power of the `append/3` predicate the true gateway to appreciating logic programming. So, without further ado:

```
append([], X, X).
append([HEAD|X], Y, [HEAD|Z]) :-
    append(X, Y, Z).
```

You'll probably want to put that in a file named *append.pro* and in the shell `consult('append.pro')` to follow along with some of the examples you're about to see.

Explaining how that works would take a bit of time, so I recommend you work it out on your own. Instead, I'll focus on its implications.

It looks like a lot of work to define how to append two lists. Perl is much simpler:

```
my @Z = (@X, @Y);
```

In fact, that hides quite a bit of complexity for us. Anyone who has had to concatenate two arrays in C can appreciate the simplicity of the Perl approach. So what would the complexity of the `append/3` predicate gain us? Naturally, we can use this to append two lists. (The following output is similar to what one would see in regular Prolog systems. It's been reproduced here for clarity.)

```
?- append([1,2,3], [4,5], Z).

Z = [1,2,3,4,5]
```

Of course, by now you should know that we can reuse this. We can use it to figure out which list to append to `X` to create `Z`.

```
?- append([1,2,3], Y, [1,2,3,4,5]).

Y = [4,5]
```

We can also use this to figure out all combinations of two lists can be appended together to create `Z`.

```
?- append(X, Y, [1,2,3,4,5]).

X = [], Y = [1,2,3,4,5]
X = [1],  Y = [1,2,3,4]
X = [1,2],  Y = [1,2,3]
X = [1,2,3],  Y = [1,2]
X = [1,2,3,4],  Y = [1]
X = [1,2,3,4,5], Y = []
```

And the Perl equivalent:

```
use AI::Prolog;

my $prolog = AI::Prolog->new(<<"END_PROLOG");
    append([], X, X).
    append([W|X], Y, [W|Z]) :- append(X, Y, Z).
END_PROLOG

my $list = $prolog->list( qw/1 2 3 4 5/ );

$prolog->query("append(X,Y,[$list]).");
while ( my $results = $prolog->results ) {
    my ( $x, $y, $z ) = @{ $results }[ 1, 2, 3 ];
    $" = ', '; # Array separator
    print "[@$x],     [@$y],     [@$z]\n";
}
```

As you can see, Prolog lists will be returned to Perl as array references. `$results->[0]` is the name of the predicate, `append`, and the next three elements are the successive values generated by Prolog.

PROBLEMS WITH PROLOG

This article wouldn't be complete without listing some of the issues that have hampered Prolog.

Perhaps the most significant is the depth-first exhaustive search algorithm used for deduction. This is slow and due to the dynamically typed nature of Prolog, many of the optimizations that have been applied to databases cannot be applied to Prolog. Many Prolog programmers, understanding how the language works internally, use the "cut" operator, `!` (an exclamation point), to prune the search trees. This leads to our next problem.

The "cut" operator is what is known as an "extra-logical" predicate. This, along with I/O and predicates that assert and retract facts in the database are a subject of much controversy. They cause issues because they frequently cannot be backtracked over and the order of the clauses sometimes becomes important when using them. They can be very useful and simplify many problems, but they are more imperative in nature than logical and can introduce subtle bugs.

Math is also handled poorly in Prolog. Consider the following program:

```
convert(Celsius, Fahrenheit) :-
     Celsius is (Fahrenheit - 32) * 5 / 9.
```

You can then issue the query `convert(Celsuis, 32)` and it will dutifully report that the celsius value is zero. However, `convert(0, 32)` fails. This is because Prolog is not able to solve the right hand side of the equation unless `Fahrenheit` has a value at the time that Prolog starts examining the equation. One simplistic way of getting around this limitation is with multiple predicates and testing which argument is an unbound variable:

```
convert(Celsius, Fahrenheit) :-
    var(Celsius),
    not(var(Fahrenheit)),
    Celsius is (Fahrenheit - 32) * 5 / 9.

convert(Celsius, Fahrenheit) :-
    var(Fahrenheit),
    not(var(Celsius)),
```

```
Fahrenheit is (Celsius * (9/5)) + 32.
```

This has a variety of problems, not the least of which is that it offers no advantages over imperative programming.

ISO-Prolog does not define it, but many Prolog implementations support constraints and these, though beyond the scope of this article, can get around this and other issues. They can allow "logical" math and ensure that "impossible" search branches are never explored. This can help to alleviate many of the aforementioned concerns.

CONCLUSION

We've barely scratched the surface of what the Prolog language can do. The language has many detractors and some criticisms are quite valid. However, the language's simplicity and ease-of-use has kept it around. It's an excellent starting point for understanding logic programming and exploration of AI. In fact, many common uses for Prolog are heavily used in the AI arena:

- Agent-based programming
- Expert Systems
- Fraud prevention (via inductive logic programming)
- Natural language processing
- Rules-based systems
- Scheduling systems
- And much more (it was even embedded in Windows NT)

At the present time, I would not recommend `AI::Prolog` for production work. Attempts to bring logic programming to Perl are still relatively recent and no module (to this author's knowledge) is currently ready for widespread use. The `AI::Prolog` distribution has a number of sample programs in the `examples/` directory and two complete, albeit small, games in the `data/` directory.

REFERENCES

Online Resources

- Adventure in Prolog http://amzi.com/AdventureInProlog/index.htm

  I highly recommend Amzi! Prolog's free online book "Adventure in Prolog." It's clearly written and by the time you're done, you've built Prolog programs for an adventure game, a genealogical database, a customer order inventory application and a simple expert system.

- Building Expert Systems in Prolog http://amzi.com/ExpertSystemsInProlog/xsipfr-top.htm

  This free online book, also by Amzi!, walks you through building expert systems. It includes expert systems for solving Rubik's cube, tax preparation, car diagnostics and many other examples.

- Databases Vs AI http://lsdis.cs.uga.edu/~cartic/SemWeb/DatabasesAI.ppt

  This Powerpoint presentation discusses AI in Prolog and compares it to SQL databases.

- W-Prolog http://goanna.cs.rmit.edu.au/~winikoff/wp/

  Dr. Michael Winikoff kindly gave me permission to port this Java application to Perl. This formed the basis of the earliest versions.

- X-Prolog http://www.iro.umontreal.ca/~vaucher/XProlog/

  Jean Vaucher, Ph.D., allowed me to borrow from X-Prolog for the first implementation of math in `AI::Prolog`. Currently, `AI::Prolog` is a hybrid of these two applications, but it has evolved in a much different direction.

Books

- Programming in Prolog http://portal.acm.org/citation.cfm?id=39071

  This book is sometimes referred to simply as "Clocksin/Mellish". First published in the early 80s, this is the book that brought Prolog into the mainstream.

- The Art of Prolog htp://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=8327

  This excellent MIT textbook is very in-depth and covers "proving" Prolog programming, second-order logic, grammars, and many working examples.

CREDITS

Many thanks to Rebekah Golden, Danny Werner and David Wheeler for their excellents comments and insights.