Prolog

Lecture KRR-7

Lucia Gomez and Brandon Bennett

University of Leeds

Introduction

- Prolog was one of the first logic programming languages, and remains the most popular among such languages today.

Introduction

- Prolog was one of the first logic programming languages, and remains the most popular among such languages today.

- Logic programming is a programming paradigm in which languages are largely based on formal logic.

Introduction

- Prolog was one of the first logic programming languages, and remains the most popular among such languages today.

- Logic programming is a programming paradigm in which languages are largely based on formal logic.

- Logic programming languages are declarative, they allow the programmer to state *what* is to be done while leaving the details of how is to be done to the language implementation. This is very different from imperative languages.

Reasons to Learn a Bit of Prolog

- The Logic Programming paradigm is radically different from the traditional imperative style; so knowledge of Prolog helps develop a broad appreciation of programming techniques.

- Although not usually employed as a general purpose programming language, Prolog is well-suited for certain tasks and is used in many research applications.

- Prolog has given rise to the paradigm of Constraint Programming which is used commercially in scheduling and optimisation problems.

Pitfalls in Learning Prolog

One reason that Prolog is not more widely used is that a beginner can often encounter some serious difficulties.

Pitfalls in Learning Prolog

One reason that Prolog is not more widely used is that a beginner can often encounter some serious difficulties.

Trying to crowbar an imperative algorithm into Prolog syntax will generally result in complex, ugly and often incorrect code. A good Prolog solution must be formulated from the beginning in the logic programming idiom.

- Forget about *functions*, think of relations.
- Forget ”for” or ”while” loops, think of recursion.

Pitfalls in Learning Prolog

One reason that Prolog is not more widely used is that a beginner can often encounter some serious difficulties.

Trying to crowbar an imperative algorithm into Prolog syntax will generally result in complex, ugly and often incorrect code. A good Prolog solution must be formulated from the beginning in the logic programming idiom.

- Forget about *functions*, think of relations.
- Forget ”for” or ”while” loops, think of recursion.

Another difficulty is in getting to grips the execution sequence of Prolog programs. This is subtle because Prolog is written declaratively. Nevertheless, its search for solutions does proceed in a determinate fashion and unless code is carefully ordered it may search forever, even when a solution exists.

Prolog Syntax

Basic idea of Prolog

- Describe the situation of interest with facts and rules
- Ask a question (query)
- Prolog:
  - Logically deduces new facts about the situation we described
  - Gives us its deductions back as answers

Terms

- Q: What exactly are facts, rules and queries built out of?
- A: Prolog terms

Simple terms

- Variables Start with either an uppercase letter or an underscore. E.g. `X, Y, Variable, Vincent, _tag`

- Constants
  - Atoms Start with a lowercase letter or are enclosed in single quotes. E.g. `butch, big_kahuna_burger, 'playGuitar'`
  - Numbers Can be integers or floats. E.g. `12, -56, 2346.765`

Complex terms

Coding for many kinds of application is facilitated by representing data in a structured way. Prolog provides generic syntax for structuring data that can be applied to all manner of particular requirements.

A complex term takes the following form:

```
functor(term1, ..., termN)
```

Where `term1, ..., termN` may also be complex terms.

Complex terms

Coding for many kinds of application is facilitated by representing data in a structured way. Prolog provides generic syntax for structuring data that can be applied to all manner of particular requirements.

A complex term takes the following form:

```
functor(term1, ..., termN)
```

Where `term1, ..., termN` may also be complex terms.

Although such a term looks like a function, it is not evaluated by applying the function to the arguments. Instead, Prolog tries to find values for which it is true by matching it to the facts and rules given in the code.

Facts and Rules

```
happy(yolanda).                          %Fact
listens2Music(mia).                      %Fact
listens2Music(yolanda) :- happy(yolanda). %Rule
```

There are two facts in this code: listens2Music(mia) and happy(yolanda) .

The last item it contains is a rule.

Facts and Rules

Rules state information that is conditionally true of the situation of interest. For example, this rule says that Yolanda listens to music if she is happy. More generally,

- the `:-` should be read as "if", or "is implied by".
- The part on the left hand side of the `:-` is called the **head** of the rule, the part on the right hand side is called the **body**.
- **If the body of the rule is true, then the head of the rule is true too**.

If a program contains a rule `head :- body`, and Prolog knows that body follows from the information in the program, then Prolog can infer head.

Prolog and Logic

Clearly, Prolog has something to do with logic...

- Implication in Prolog is `:-`
- Conjunction in Prolog is `,`
- Disjunction in Prolog is `;`

And it uses inference (modus ponens)!

(What about negation? Well, it is `\+` but that is a whole other story. We'll be discussing it later)

Forming Disjunctions with ‘;’

Sometimes we may want to test whether either of two goals is satisfied. We can do this with an expression such as:

```
( clever(X) ; rich(X) )
```

This will be true if there is a value of `X` , such that both the `clever` and `rich` predicates are true for this value. Thus, we could define:

```
powerful( X ) :- (clever(X) ; rich(X) ).
```

However, we will normally express this as a pair of rules, which is equivalent:

```
powerful( X ) :- clever(X).
powerful( X ) :- rich(X).
```

The Negation as failure Operator '\+'

It is often useful to check that a particular goal expression does not succeed.

This is done with the '`\+`' operator.

E.g.

```
\+loves(john, mary)
\+loves(john, X )
```

By using brackets one can check wether two or more predicates cannot be simultaneously satisfied:

```
\+( member( Z, L1), member(Z, L2 ) )
```

But, beware that Negation as failure is not logical negation!

Combining Operators Example

In general, we can combine several operators to form a complex goal which is a (truth functional) combination of several simpler goals.

E.g.:

```
eligible( X ) :- good_CV( X ),
                 (clever(X) ; hard_working(X)),
                 \+ conflictive( X ).
```

Combining Operators Example

In general, we can combine several operators to form a complex goal which is a (truth functional) combination of several simpler goals.

E.g.:

```
eligible( X ) :- good_CV( X ),
                 (clever(X) ; hard_working(X)),
                 \+ conflictive( X ).
```

Beware of the negation, want to see what happens if we change the order?

```
eligible( X ) :- \+ conflictive( X ),
                 (clever(X) ; hard_working(X)),
                 good_CV( X ).
```

Unification and Equality

Unification (Pattern Matching) and Equality

When evaluating a query containing one or more variables, Prolog tries to match the query to a stored (or derivable) fact, in which variables may be replaced by particular instances. In Prolog terms this is usually called unification. In fact the matching always goes both ways. If we have the code:

```
likes( X, X ).
```

Then the query `?- likes( john, john )` will give `yes` , even though there are no facts given about `john` , because the query matches the general fact — which says that everyone likes themself.

Similarly, the query `?- X=whateverRandomThing` also will give yes.

Matching Complex Terms

Prolog will also find matches of complex terms as long as there is a instantiation of variables that makes the terms equivalent:

```
loves( brother(X), daughter(tom) ).

?- loves( brother( susan ), Y ).
X = susan,
Y = daughter(tom)
```

Matching Complex Terms

Prolog will also find matches of complex terms as long as there is a instantiation of variables that makes the terms equivalent:

```
loves( brother(X), daughter(tom) ).

?- loves( brother( susan ), Y ).
X = susan,
Y = daughter(tom)
```

We can make Prolog perform a match using the equality operator:

```
?-  f( g(X), this, X )  =  f( Z, Y, that ).
X = that,
Y = this,
Z = g(that)
```

Note that no evaluation of `f` and `g` takes place.

Coding by Matching

A surprising amount of functionality can be achieved simply by pattern matching.

Consider the following code:

```
vertical( line(point(X,Y), point(X,Z)) ).
horizontal( line(point(X,Y), point(Z,Y)) ).
```

Here we are assuming a representation of line objects as pairs of point objects, which in turn are pairs of coordinates.

Coding by Matching

A surprising amount of functionality can be achieved simply by pattern matching.

Consider the following code:

```
vertical( line(point(X,Y), point(X,Z)) ).
horizontal( line(point(X,Y), point(Z,Y)) ).
```

Here we are assuming a representation of line objects as pairs of point objects, which in turn are pairs of coordinates. Given just these simple facts, we can ask queries such as:

```
?- vertical( line(point(5,17), point(5,23)) ).
```

More Useful Prolog Operators and Built-Ins

Math Evaluation with *'is'*

Though possible, it would be rather tedious and very inefficient to code basic mathematical operations in term of Prolog facts and rules (e.g. `add(1,1,2).` , `add(1,2,3).` etc).

However, the *'is'* enables one to evaluate a math expression and bind the value obtained to a variable. For example after executing the code line

```
X is sqrt( 57 + log(10) )
```

Prolog will bind `X` to the appropriate decimal number:

```
X = 7.700817170469251
```

Recursion

Prolog predicates can be defined recursively

A predicate is recursively defined if one or more rules in its definition refers to itself

```
even(2).
even(X) :- Y is X-2, even(Y).
```

Recursion

Prolog predicates can be defined recursively

A predicate is recursively defined if one or more rules in its definition refers to itself

```
even(2).
even(X) :- Y is X-2, even(Y).
```

However, there is a little problem with this code...

Recursion

Prolog predicates can be defined recursively

A predicate is recursively defined if one or more rules in its definition refers to itself

```
even(2).
even(X) :- Y is X-2, even(Y).
```

However, there is a little problem with this code... What if I check for more solutions?

The Cut Operator, '!'

Prolog predicates can be defined recursively

The so-called cut operator is used to control the flow of execution, by preventing Prolog backtrack past the cut to look for alternative solutions. Consider the following mini-program:

```
red(a).          black(b).
color(P,red)     :- red(P), !.
color(P,black)   :- black(P), !.
color(P,unknown).
```

Consider what happens if we give the following queries:

```
?- color(a, Col).
?- color(c, Col).
```

What would be the difference if the cuts were removed?

The Cut Operator, *'!'*

The so-called cut operator is used to control the flow of execution, by preventing Prolog backtrack past the cut to look for alternative solutions.

In our even we get into an infinite recursion without a cut operator

```
even(2).
even(X) :- Y is X-2, even(Y).
```

The Cut Operator, '!'

The so-called cut operator is used to control the flow of execution, by preventing Prolog backtrack past the cut to look for alternative solutions.

In our even we get into an infinite recursion without a cut operator

```
even(2).
even(X) :- Y is X-2, even(Y).
```

How can we fix it?

The Cut Operator, *'!'*

The so-called cut operator is used to control the flow of execution, by preventing Prolog backtrack past the cut to look for alternative solutions.

In our even we get into an infinite recursion without a cut operator

```
even(2).
even(X) :- Y is X-2, even(Y).
```

How can we fix it?

```
even(X) :- X is 2, !.
even(X) :- Y is X-2, even(Y).
```

Other basic concepts

Have a look at the meanin of these terms in Prolog. We will be talking about them so it is good to know what is going on.

- Clause
- Predicate
- Arity

Lists in Prolog

Lists

Lists are important recursive data structures often used in Prolog programming.

Lists

Lists are important recursive data structures often used in Prolog programming.

A list is a finite sequence of elements.

Lists are enclosed in square brackets and can contain all sorts of Prolog terms as elements.

E.g.:

```
[mia, vincent, jules, yolanda]
[mia, robber(honeybunny), X, 2, mia]
[ ]
[mia, [vincent, jules], [butch, friend(butch)]]
[[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]
```

The List Constructor Operator

The basic syntax that is used either to construct or to break up lists in Prolog is the head-tail structure:

[Head|Tail]

Here Head is any term. It could be an atom, a variable or some complex structure.

Tail is either a variable or any kind of list structure.

The structure [Head|Tail] denotes the list formed by adding Head at the front of the list Tail.

Thus [a | [b,c,d]] denotes the list [a,b,c,d].

```
?- [a | [b,c,d]] = [a,b,c,d]
True
```

Head and tail of empty list

- The empty list has neither a head nor a tail
- For Prolog, [ ] is a special simple list without any internal structure
- The empty list plays an important role in recursive predicates for list processing in Prolog

Matching Lists

Consider the following query:

```
?-   [ H | T ] = [a, b, c, d, e].
```

This will return:

```
Head = a,
Tail = [b,c,d,e]
```

Note that this is identical in meaning to:

```
?-    [a, b, c, d, e] = [ H | T ].
```

Matching Lists

What if we are interested on the first n elements?

```
?- [X1,X2,X3,X4|Tail] =
              [mia, vincent, marsellus, jody, yolanda].

X1 = mia
X2 = vincent
X3 = marsellus
X4 = jody
Tail = [yolanda]
yes
?-
```

Anonymous variable

What if we are interested only on the second and the fourth elements?

We can use _, which is the anonymous variable.

```
?- [_,X2,_,X4|_] =
                [mia, vincent, marsellus, jody, yolanda].

X2 = vincent
X4 = jody
yes
?-
```

The anonymous variable is used when you need to use a variable, but you are not interested in what Prolog instantiates it to.

Recursion Over Lists

The following example is of utmost significance in illustrating the nature of Prolog. It combines, pattern matching, lists and recursive definition. Learn this:

```
in_list( X, [X | _] ).
in_list( X, [_ | Rest] ) :- in_list( X, Rest ).
```

Given this definition, `in_list(elt,list)` will be true, just in case `elt` is a member of `list`. (In fact this same functionality is already provided by the inbuilt member predicate.)

Check if Arrays Share a Value in C

```
#include <stdio.h>
int main() {
    int arrayA [4] = {1,2,3,4};
    int arrayB [3] = {3,6,9};
    if ( arrays_share( arrayA, 4, arrayB, 3 ) )
        printf("YES\n");
    else printf("NO\n");
}
int arrays_share(int A1[],int len1,int A2[],int len2){
    int i, j;
    for (i = 0; i < len1; i++ ) {
        for (j = 0; j < len2; j++) {
            if (A1[i] == A2[j]) return 1;
        }
    }
    return 0;
}
```

Check if Arrays Share a Value in Prolog

```
lists_share_element( L1, L2 ) :- member( X, L1 ),
                                 member( X, L2 ).

?- lists_share_element( [1,2,3,4], [3,6,9] ).
yes
```

Some examples

Data Record Processing Example

Here is a typical example of how some data might be stored in the form of Prolog facts:

```
%%%    ID     Surname    First Name   User Name   Degree
student(101, 'SMITH',     'John',    cs2010js,    'CS').
student(102, 'SMITH',     'Sarah',   cs2010ss,    'CS').
student(103, 'JONES',     'Jack',    cs2010jj,    'CS').
student(104, 'MORGAN',   'Augustus',lg2010adm, 'Logic').
student(105, 'RAMCHUNDRA','Lal',     lg2010lr,   'Logic').

coursework( 1, cs2010js,  45 ).
coursework( 1, cs2010ss,  66 ).
coursework( 1, cs2010jj,  35 ).
coursework( 1, lg2010adm,  99 ).
coursework( 1, lg2010lr,   87 ).
```

Deriving Further Info from the Data

Given the data format used on the previous slide, the following code concisely defines how to derive the top mark for a given coursework:

```
top_mark( CW, [First, Sur] ) :-
            coursework( CW, User, Mark1 ),
            \+( ( coursework( CW, _, Mark2 ),
                  Mark2 > Mark1
                )
              ),
            student( _, Sur, First, User, _ ).
```

Note: the extra layer of bracketing in `\+(( ... ))` is required in order to compound two predicates to form a single argument for the `\+` operator.

More Data Extraction Rules

Here are some further useful rules for extracting information from the student and coursework data:

```
student_user( U ) :- student( _, _, _, U, _).

user_pass_cw( U, CW ) :-
                        coursework( CW, U, Mark ),
                        Mark >= 40.

user_name( U, [S,F] ) :- student( _, S, F, U, _).
```

Example Use of *setof*

The `setof` operator enables us to straightforwardly derive the whole set of elements satisfying a given condition

```
pass_list( CW, PassList ) :-
                setof( Name, U^(user_pass_cw(U,CW),
                user_name(U,Name)), PassList).
```

Note: the construct `U^( ... )` is a special operator that is used within setof commands to tell prolog that the value of variable `U` can be different for each member of the set

Conclusion

Conclusion

The nature of Prolog programming is very different from other languages.

In order to program efficiently you need to understand typical Prolog code examples and build your programs using similar style.

Trying to put imperative ideas into a decarative form can lead to overly complex and error prone code.

Although Prolog code is very much declarative in nature, in order for programs to work correctly and efficiently one must also be aware of how code ordering affects the execution sequence.