Lecture 16: Logic Programming in Prolog

COMP 524 Programming Language Concepts
Stephen Olivier
March 26, 2009

Based on slides by A. Block, notes by N. Fisher, F. Hernandez-Campos, and D. Stotts

Goal of Lecture

- Understand concepts associated with logic programming
- Program in the Prolog language

Axioms and Goals

- Logic Programming is based on a series of axioms.
  - Axioms define the language
  - After the axioms have been stated the user states a goal and the logic language attempts to find a series of axioms to satisfy the goal.

```
C←A,B
D←C
----------
D←A,B
```

This `C←A,B` should be read as "C, if A and B."

C is the **head** and A and B define the **body**.

The statement is called a **Horn Clause**

the logical language attempts to find a series of axioms to satisfy the goal.

```
C←A,B
D←C
----------
D←A,B
```

Prolog

- A prolog interpreter runs in the context of a database of clauses.
- Each clause is composed of terms, which may be constants, variables, or structures.
- An **atom** is Prolog is an identifier beginning with a lowercase letter, a sequence of “punctuation” characters, or a quoted character string:

```
a b + 'Hi, Mom'
```

Numbers

- Numbers resemble integers and floating point constants of other languages

Variables

- Variables look like identifier with an upper case letter

```
X Y Zebra
```

- Variables can be instantiated to take on arbitrary values at run time.

Structures

- Structures consist of an atom called the functor and a list of arguments.

```
apple(bar, qud).
bin_tree(foo, bin_tree(pear,larch))
```

- The parentheses must come directly after the atom (no white space)

Clauses

- Clauses are classified as facts or rules each of which ends with a period.
- A fact is a Horn clause without a right-hand side.

```
loser(dook).
```

- A rule has a right hand side.

```
lame(X) :- loser(X), expensive(X).
```

Query

- It is possible to right a query or a goal, which are statements with no “left-hand side”

```
?-loser(X).
```

- Queries can return multiple answers

```
X=dook;
X=state
```

Closed World Assumption

- Suppose our database includes only the following:

```
loser(dook).
loser(state).
```

- Now we query on Virginia Tech:

```
-?loser(vt).
no
```

- Does that mean VT really isn't a loser?
  - Their mascot is a turkey!

Closed World Assumption

- The interpretation here is that Prolog does not have sufficient knowledge in the database to prove that VT is a loser.
- As far as Prolog is concerned, all that is true about the world can be proved from the database.
  - So-called **Closed World Assumption**

Resolution Principle

- Resolution Principle states that for two Horn clauses A and B. If the **head** of A matches one of the **terms** in B, then the **body** of A can replace the **term** in B.

```
takes(jane_doe,comp524).
takes(jane_doe,comp121).
takes(john_smith,comp524).
takes(john_smith,art101).
classmates(X,Y):-takes(X,Z),takes(Y,Z).
```

```
X= jane_doe. Z=comp524.
classmates(jane_doe,Y):-takes(Y,comp524).
```

Unification

- Unification is the process of pattern-matching process used to associate a variable with values. Variables that are given values as a result of unification are said to be instantiated.

```
...
classmates(X,Y):-takes(X,Z),takes(Y,Z).
?-classmates(john_smith,jane_doe).
```

Unification rules

- A constant unifies only with itself
- Two structures unify iff they have the same functor and the same number of arguments, and the corresponding arguments unify recursively
- A variable unifies with anything. If the other thing has a value, then the variable is instantiated. If the other thing is an uninstantiated variable, then the two variables are associated in such a way that if either is given a value later, that the value will be shared by both.

Unification rules

- A constant unifies only with itself

```
?-a=a.
yes
?-a=b.
no
```

Unification rules

- Two structures unify iff they have the same functor and the same number of arguments, and the corresponding arguments unify recursively

```
?-foo(a,b)= foo(a,b).
yes
?-foo(a,b)=foo(X,b).
X = a;
no
```

Unification rules

- A variable unifies with anything. If the other thing has a value, then the variable is instantiated. If the other thing sis an uninstantiated variable, then the two variables are associated in such a way that if either is given a value later, that the value will be shared by both.

```
?-foo(a,b)= foo(a,b).
yes
?-foo(a,b)=foo(X,b).
X = a;
no
```

Arithmetic

- Can’t unify arithmetic

```
?-(2+3) = 5.
no
?-X is 1+2.
X=3
?-1+2 is 4-1
no
?-X is Y
<error>% Y isn't instantiated
?-Y is 1+2, X is Y.
X=3
Y=3
```

Assert and retract

- assert() adds a statement
  - To allow modification of existing facts later, use dynamic at the top of the knowledge base file

```
:- dynamic father/2.
```

- retract() removes a statement

```
assert(father(vader,luke)).
retract(raining(carrboro)).
```

Other functions

- write -- writes value to output
- nl -- writes newline
- read -- read from input
- get -- gets a character (from input)
- put -- puts a character (into output)
- consult -- Read database clauses from a file
  - Shorthand: [file].
- listing -- show the contents of the database

Search/Execution Order

- Two approaches
  - Start with existing clauses and work forward (forward chaining)
    - Okay if there are many rules and few facts
  - Start with the goal and work backwards (backward chaining)
    - Often the better way to go
    - This is what Prolog uses

Backtracking DFS Search

- Prolog uses depth-first search with backtracking.
- Try this example with trace turned on

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X):-rainy(X),cold(X).
```

```
?- trace.
[trace] ?- snowy(X).
```

Infinite Recursion

- Since Prolog takes a depth first approach for search states, this can cause problems if not careful.
  - Could encounter an infinite branch
  - Order is important!
- Consider the example of a graph...

Infinite recu

```
edge(a,b).edge(b,c).edge(c,d).
edge(d,e).edge(b,e).edge(d,f).
path(X,X).
path(X,Y):-edge(Z,Y),path(X,Z).
```

```
edge(a,b).edge(b,c).edge(c,d).
edge(d,e).edge(b,e).edge(d,f).
path(X,X).
path(X,Y):-path(X,Z),edge(Z,Y).
```

```
edge(a,b).edge(b,c).edge(c,d).
edge(d,e).edge(b,e).edge(d,f).
path(X,Y):-path(X,Z),edge(Z,Y).
path(X,X).
```

The cut

- The **cut** commits the interpreter to whatever choices have been made since unifying the parent goal with the left-hand side of the current rule
  - Written as ! in Prolog
- So, it “prunes” the tree

The not

- Alternatively, we can use not to guarantee that only one statement is returned

```
not(x=y). %true
not(x=x). %false
```

The not

- Alternatively, we can use not to guarantee that only one statement is returned

```
not(P):-call(P), !, fail.
not(P).
```

if...then...else

- We can use the cut to make if then else

```
statement:-if_clause, !, then_part.
statement:-else_part.
```

Lists

- We can write a list as [a,b,c] or
- [a | [b,c]]

```
member(X,[X|T]).
member(X,[H|T]):-member(X,T).
sorted([]).
sorted([X]).
sorted(A,B|T]):-A=<B,sorted([B|T]).
```

Looping with fail

```
append([],A,A).
append([H|T],A,[H|L]):-append(T,A,L).
print_part(L):-append(A,B,L),
           write(A), write(' '), write(B), nl,
           fail.
```

```
[] [a,b,c]
[a] [b,c]
[a,b] [c]
[a,b,c] []
no
```

Looping with an unbounded generator

```
natural(1).
natural(N):-natural(M), N is M+1.
```

```
my_loop(N):-natural(I),
        write(I), nl,
        I>=N, !, fail.
```