Lecture 6

Logic Programming

rule-based programming with Prolog

Ras Bodik
with Mangpo, Ali

Hack Your Language!
CS164: Introduction to Programming Languages and Compilers, Spring 2013
UC Berkeley

Today

Why study Prolog

- our abstraction stack

Introduction to Prolog

- facts and queries
- generalization, instantiation, and the Prolog semantics
- rules
- functors: AST and a simple interpreter
- working with lists
- a simple AST rewrite engine

Reading

Compulsory:

Adventures in Prolog, a tutorial (Chapters 1-11)

Optional:

The Art of Prolog, on reserve in Engineering Library (starting Friday evening, Feb 8).

Why logic programming?

Our abstraction stack

Parsing (PA4), Analysis of Programs, Types

- we'll express each with Prolog rules
- (not all will require backtracking)

Logic programming (PA3)

- a convenient layer over backtracking
- enables rule-based programming, inference

Coroutines (PA2)

- enable lazy iterators and backtracking
- ex: regex matching: for all matches of patt1, match patt2

Closures and lexical scoping (PA1)

- enable, for example, iterators
- ex: `d3.select(someNodes).each(aClosure)`

Infrastructure

Software

Software:

install SWI Prolog

Usage:

?- [likes]. *# loads file likes.pl*

Content of file likes.pl:

```
likes(john,mary).
likes(mary,jim).
```

After loading, we can ask a query:

?- likes(X,mary). *#who likes mary?*

X = john ; # type semicolon to ask “who else?”

false. # no one else

Facts, queries, and variables

Facts and queries

(Database of) two facts:

likes(john, mary). # *relation*: facts with same name, eg likes

likes(mary, jim). # relation is a.k.a. *predicate*

Boolean queries (answers are *true* or *false*)

?- likes(john,jim). # sometimes we use syntax *likes(a,b)?*

false

Existential queries (is there X s/t likes(X,jim) holds?)

?- likes(X,jim). # variables start with capital letters

mary # atoms start with capital letters

goals, facts, and queries

Syntactically, facts and queries look similar

goal: likes(jim, mary)

notice that there is no dot at the end of goal

fact: likes(jim, mary).

states that the goal likes(jim,mary) is true

query: likes(jim, mary)? sometimes we write ?- goal.

asks whether the goal likes(jim, mary) is true

Terminology

Ground terms (do not contain variables)

father(a,b). *# fact (a is father of b)*

?- father(a,b). *# query (is a father of b?)*

Non-ground terms (contain variables)

∀ likes(X,X). *# fact: everyone likes himself*

∃ ?- likes(Y,mary). *# query: who likes mary?*

Variables in facts are universally quantified

for *whatever* X, it is true that X likes X

Variables in queries are existentially quantified

does there *exist* an X such that X likes mary?

Example

A single fact

likes(X,X).

Queries:

?- likes(a,b).

false.

?- likes(a,a).

true.

?- likes(X,a).

X = a.

?- likes(X,Y).

X = Y. <-- Answers to queries need not be fully grounded

?- likes(A,A).

true.

Generalization and Deduction via Substitution for Variables

Generalization (a deduction rule)

Facts

```
father(abraham,isaac).
```

Query

```
?- father(abraham,X).
```

This query is a *generalization* of the fact

We answer the query by finding a *substitution* {X=isaac}.

- This substitution turns the query into a fact that exists in the database, leading to *true*. Well, the answer also shows the substitution.

Instantiation (another deduction rule)

Rather than writing

plus(0,1,1). plus(0,2,2). …

We write

plus(0,X,X). *# 0+x=x*

plus(X,0,X). *# x+0=x*

Query

?- plus(0,3,3). *# this query is instantiation of plus(0,X,X).*

yes

We answer by finding a substitution {X=3}.

Prolog semantics

How the Prolog interpreter answers a query:

```
p(a,a).     # fact 1
p(a,b).     # fact 2
p(b,c).     # fact 3

?- p(a,A).  # This query raises one goal, p(a,A)
A = a ;     # going top down across facts, the goal matches 1
A = b.      # and when asked for next match, it matches 2.
```

We'll generalize this algorihm when we add rules.

Rules

Rules

Rules define new relationships in terms of existing ones

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandfather(X,Y) :-father(X,Z), parent(Z,Y).
```

Assume facts

```
father(john,mary).
mother(mary,jim).
```

Now ask the query

```
?- grandfather(X,Y).
X = john,
Y = jim ;
false.
```

The Prolog semantics

Prolog algorithms in the presence of rules:

```
father(john,mary).
mother(mary,jim).
grandfather(fim, fum).        # 1
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandfather(X,Y) :-father(X,Z), parent(Z,Y).   # 2
?- grandfather(X,Y).
X = fim, Y = fum ;    # matches fact 1 from relation 'grandfather'
X = john, Y = jim ;   # matches head (lhs) of rule 2, which then
                      # creates two new goals from the rhs of 2.
false.                # lhs = left-hand-side
```

So Prolog can do a simple inference!

1801 - Joseph Marie Jacquard uses punch cards to instruct a loom to weave "hello, world" into a tapestry. Redditers of the time are not impressed due to the lack of tail call recursion, concurrency, or proper capitalization.

…

1972 - Alain Colmerauer designs the logic language Prolog. His goal is to create a language with the intelligence of a two year old. He proves he has reached his goal by showing a Prolog session that says "No." to every query.

http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html

Database programming

A database programming rule

```
brother(Brother, Sib) :-
    parent(P, Brother),
    parent(P, Sib),
    male(Brother),
    Brother \= Sib.          # same as \=(Brother,Sib)
```

This rule assumes that we have defined relations parent and male. (The \= relation is a built-in.)

Database programming

In cs164, we will translate SQL-like queries to Prolog.

But Prolog can also express richer (recursive) queries:

same X — distinct Ys

```
descendant(Y,X) :- parent(X,Y).
descendant(Y,X) :- parent(X,Z), descendant(Y,Z).
```

Order of rules and clauses matters

1) Given a goal, Prolog matches facts and rules top-down as they appear in the file.

   ex: on slide 19, #1 matches before #2 matches.

2) If the rhs of a rule raises multiple goals, they are answered left-to-right.

   ex: on slide 19, match 2, father(X,Z) is resolved before parent(Z,Y).

Test yourself

Make sure you understand why these three variants of descendants have different behaviors:

v1:

```
descendant(Y,X) :- parent(X,Y).
descendant(Y,X) :- parent(X,Z), descendant(Y,Z).
```

v2:

```
descendant(Y,X) :- parent(X,Z), descendant(Y,Z).
descendant(Y,X) :- parent(X,Y).
```

v3:

```
descendant(Y,X) :- parent(X,Y).
descendant(Y,X) :- descendant(Y,Z), parent(X,Z).
```

Compound terms

Compound terms

Compound term = functors and arguments.

Name of functor is an atom (lower case), not a Var.

example: cons(a, cons(b, nil))

A rule:

```
car(Head, List) :- List = cons(Head,Tail).
car(Head, cons(Head,Tail)).        # equivalent to the above
```

Query:

```
?- car(Head, cons(a, cons(b, nil)).
```

A simple interpreter

A representation of an abstract syntax tree

```
int(3)
plus(int(3),int(2))
plus(int(3),minus(int(2),int(3)))
```

An interpreter

```
eval(int(X),X).
eval(plus(L,R),Res) :-
    eval(L,Lv),
    eval(R, Rv),
    Res is Lv + Rv.
eval(minus(L,R),Res) :-
    # same as plus
```

Working with lists

Lists

Lists are just compounds with special, clearer syntax.

Cons is denoted with a dot '.'

.(a,[])	is same as	[a|[]]	is same as	[a]
.(a,.(b,[]))		[a|[b|[[]]]		[a,b]
.(a,X)		[a|X]		[a|X]

predicate “Am I a list?”

Let’s test whether a value is a list

```
list([]).
list([X|Xs]) :- list(Xs).
```

Note the common Xs notation for a list of X’s.

Let’s define the predicate member

Desired usage:

```
?- member(b, [a,b,c]).
true
```

Lists

```
car([X|Y],X).
cdr([X|Y],Y).
cons(X,R,[X|R]).
```

meaning ...

- *The head (car) of [X|Y] is X.*
- *The tail (cdr) of [X|Y] is Y.*
- *Putting X at the head and Y as the tail constructs (cons) the list [X|R].*

From: http://www.csupomona.edu/~jrfisher/www/prolog_tutorial

An operation on lists:

[a, b, c] → [a, [b, [c, []]]] → .(a, .(b, .(c, nil)))

[] is just sugar

The predicate member/2:

```
member(X,[X|R]).
member(X,[Y|R]) :- member(X,R).
```

One can read the clauses the following way:

X is a member of a list whose first element is X.

X is a member of a list whose tail is R if X is a member of R.

List Append

```
append([],List,List).
append([H|Tail],X,[H|NewTail]) :-
    append(Tail,X,NewTail).
```

```
?- append([a,b],[c,d],X).
X = [a, b, c, d].
?- append([a,b],X,[a,b,c,d]).
X = [c, d].
```

- This is "bidirectional" programming
  - Variables can act as both inputs and outputs

More on append

```
?- append(Y,X,[a,b,c,d]).
Y = [],
X = [a, b, c, d] ;
Y = [a],
X = [b, c, d] ;
Y = [a, b],
X = [c, d] ;
Y = [a, b, c],
X = [d] ;
Y = [a, b, c, d],
X = [] ;
false.
```

Exercise for you

Create an append query with infinitely many answers.

```
?- append(Y,X,Z).
Y = [],
X = Z ;

Y = [_G613],
Z = [_G613|X] ;

Y = [_G613, _G619],
Z = [_G613, _G619|X] ;

Y = [_G613, _G619, _G625]
```

Another exercise: desugar AST

Want to rewrite each instance of 2*x with x+x:

```
rewrite(times(int(2),R), plus(Rr,Rr)) :-
    !, rewrite(R,Rr).
rewrite(times(L,int(2)), plus(Lr,Lr)) :-
    !, rewrite(L,Lr).
rewrite(times(L,R),times(Lr,Rr)) :-
    !, rewrite(L,Lr),rewrite(R,Rr).
rewrite(int(X),int(X)).
```

And another exercise

Analyze a program:

1) Translate a program into facts.
2) Then ask a query which answers whether a program variable is a constant at the of the program.

Assume the program contains two statement kinds

S ::= S* | def ID = n | if (E) ID = n

You can translate the program by hand

Some other cool examples to find in tutorials

compute the derivative of a function

this is example of symbolic manipulation

solve a math problem by searching for a solution:

"Insert +/- signs between 1 2 3 4 5 so that the result is 5."