Lecture 6

Logic Programming

introduction to Prolog, facts, rules

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2012

UC Berkeley

Today

Introduction to Prolog

Assigned reading: a Prolog tutorial (link at the end)

Today is no-laptop Thursday

- but you can use laptops to download SWI Prolog and solve excercises during lecture.

Software

Software:

download SWI Prolog

Usage:

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

Content of file likes.pl:

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

After loading, we can ask query:

```
?- likes(X,mary).    #who likes mary?
X = john ;           # type semicolon to ask “who else?”
false.               # no one else
```

Facts and queries

Facts:

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

Boolean queries

```
?- likes(john,jim).
false
```

Existential queries

```
?- likes(X,jim).
mary
```

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(X,mary). *# query: who likes mary?*

Variables in facts are universally quantified

- for all X, it is true that X likes X

Variables in queries are existentially quantified

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

Generalization (a deduction rule)

Facts

father(abraham,isaac).

Query

?- father(abraham,X). *# this query is a generalization above fact*

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

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

instantiate this rule to look like the query

Query

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

yes

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

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

Load family.pl

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

Database programming

A database programming rule

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

like(X,X) :- not disliker(X).

disliker(mary).

disliker(jimmy).

In cs164, we will translate SQL-like queries to Prolog.
But Prolog can also express richer (recursive) queries:

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

Compound terms

Compound term = functors and arguments.

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

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

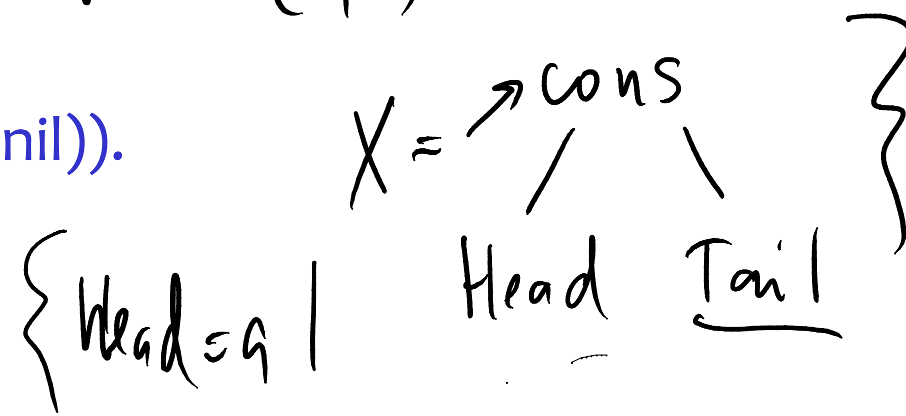
A rule:

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

?- car(a, X).

Query:

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



Must answer to queries be fully grounded?

Program:

```
eat(thibaud, vegetables).
eat(thibaud, Everything).
eat(lion, thibaud).
```

a variable (starts with a cap)

X is bound to a variable

Queries:

```
eat(thibaud, X)?
```

X = veg ;

X = _Everything

1st solution

2nd solution

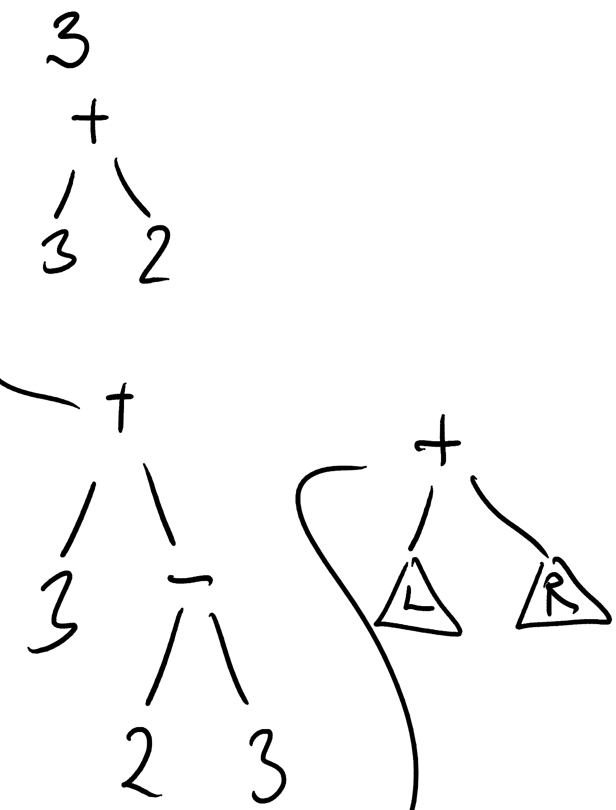
a free variable (Thibaud eats everything)

A simple interpreter

Res is Lv + Rv
Res = Lv + Rv

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
```

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]

Am a list? predicate

Let's test is 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([a,b,c],X).   X=a
cdr " "   X=[b,c]

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

how you define car, cdr, cons given [|].

[H|T] syntax for head, tail

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].
```

Hey, "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.”

Reading

Required

- download SWI prolog
- go through a good prolog tutorial, including lists, recursion

Recommended

- The Art of Prolog (this is required reading in next lecture)