CS206 Lecture 09

Logic Progamming (Prolog)

G. Sivakumar

Computer Science Department

IIT Bombay

siva@iitb.ac.in

http://www.cse.iitb.ac.in/∼siva

Fri, Jan 17, 2003

Plan for Lecture 9

- Prolog Vocabulary Syntax
- Numbers and Structured Data

Syntax of Prolog Language

A Prolog program is a collection of two types of logical formulae:

1. Facts (or Unit Clauses)
2. Rules (Head :- Body.)

To build formulae we use:

Terms: These are the basic data objects in the domain built using:
- Variables
- Constants
- Function Symbols.

Predicate Symbols: These symbols take terms as arguments to return unit clauses which should have the meaning of *true* or *false* only.

Connectives Like " :- " " , " and " . ".

Constants and Variables

Constants are the basic items (atoms) in the domain of the problem. Denoted by a sequence of alphanumeric characters **not** beginning with a capital letter or underscore (_).

Examples:

```
ganga, kaveri, jaya, 0, 23, 12.7, iitb
```

In simple (flat) domains answers to queries will instantiate variables in the query to one of these constants.

Variables are denoted by a sequence of alphanumeric characters beginning with a capital letter or underscore (_). **Examples:**

```
X, X2, River, State, List1,
Arg2, _XX23, _, __, Constant.
```

A *variable*, as the name implies, will be instantiated to take different values (terms) from the underlying domain when solving goals in which they occur. Typically, we do not use underscore (_) to start variable names, and we use suggestive names that make the clauses easy to understand.

Function Symbols

In reasonably complex domains, we cannot represent all objects simply by constants, but need a more structured representation. For this we use function symbols.

Example1: Consider Natural Numbers:

$$0, 1, 2, 3, 4, 5, \ldots$$

This representation needs as many atoms as there are numbers (infinite).
Let us introduce one unary function symbol `s` (for successor).
Using `0` and `s` we can construct all the numbers as follows

`0, s(0), s(s(0)), s(s(s(0))), ...`

In general we can write $\mathtt{s}^n(\mathtt{0})$ to denote the natural number n.

More Examples

We may want to represent all triples of numbers.

Using a function symbol `tr` that takes 3 arguments we can build all triples

```
tr(0,0,0),  tr(s(0),0,s(s(0))),
tr(tr(0,0,0),0,0) ...
```

Note that `tr` need not have only constants as arguments

Example 3:

We may have structured information about songs like their: title, raga, tala and composer.

A function symbol `song` taking 4 arguments can be used for this.

```
song(vatapi, hamsadvani, aadi, dikshitar)
song(brocheva, khamas, aadi, vasudevachar)
...
```

Predicate Symbols (Relations)

A predicate symbol is somewhat like a function symbol. It has an arity (number of arguments) and takes that many **terms** as arguments to return a unit clause.

Example: `plus` has *arity* 3 (denoted `plus/3`).

Unit Clauses: plus(0,s(0),s(0)), plus(0,0,s(0))

Similarly `double(X,Y)` can mean `Y = 2 * X`.

Note: The difference is that while function symbols can be nested in terms (example – `s(s(0))`), predicate symbols can occur only at the outermost level and cannot be nested.

Question: We cannot write:

`plus(0,plus(0,0,s(0)),0)`, why?

Facts

A fact is a unit clause.

If `p` is a predicate symbol of arity n and `t1, ..., tn` are terms, then `p(t1, ... ,tn)` is an unit clause.

To input this in as a Prolog program the "." is also needed.

Examples:

```
cm_of(tn, jaya).
spoken_in(maha, marati).
title_of(song(Ttl,Raga,Tala,Comp), Ttl).
```

Rules

A **rule** for a Prolog program is

```
head :- Body.
```

where `head` is a unit clause and `Body` is unit clauses separated by ","
This is to be read as

IF Body THEN Head

Examples:

```
spoken_near(River, Lang) :-
   flows_through(River, State),
   spoken_in(State, Lang).


times(s(X),Y,Z) :-
   times(X,Y,Z1),
   add(Y,Z1,Z).
```

The intended meaning of the second rule above is:

IF (`X` times `Y` is `Z1` AND `Y` plus `Z1` is `Z`)
THEN `s(X)` times `Y` is `Z`.

How Prolog answers goals

Given a Prolog program `P` and a goal `G`, what is an **answer** to the goal?

1. `G` has no variables
   - `yes` (if `G` is true/provable from `P`)
   - `no` (otherwise)
2. `G` has variables
   - Values for the variables in `G` that make it true
   - `no` if `G` can never be true

Here "true" ("provable") are used informally. This is the **Declarative Semantics** (Meaning) of a Prolog program.

Many answers to a goal

What if G has many solutions?

A **complete** interpreter is one that produces them all (if they are finitely many), or enumerates them ($fairly$) if there are infinitely many.

But, in what order?

And how do Prolog interpreters actually do this?

This is the **Procedural Semantics**

Prolog's Solution Mechanism

How does Prolog answer a query?

- Searches the program from **Top To Bottom** looking for a fact or a rule that can be used.
- If a fact can be used, one answer can be generated.
- If the head of a rule fits, then the body gives new subgoals which are solved from **Left To Right**.

Note that each time we use a fact or rule we

- Use new copies with new varibles.
- Keep track of substitutions made in the variables of the goal to produce the final answer.

Example Program

Suppose program P had the following information about ragas and songs **in this order** .

```
mela_raga(kharaharapriya).
mela_raga(mayamalavagoula).

same_family(kharaharapriya,sahana).
same_family(kharaharapriya,khamas).
same_family(mayamalavagoula,saveri).
same_family(R,R).
same_family(R1, R2) :- same_family(R2, R1).

song(giripai, sahana)
song(chakkani, kharaharapriya)
song(brocheva, khamas)
song(karikalaba, saveri)
song(merusamana, mayamalavagoula)
song(devadeva, mayamalavagoula)
```

Example Program (ctd.)

And P has some facts and rules about which ragas and songs are liked by some person.

```
likes(siva, saveri).

likes_song(Person, Song) :- song(Song, Raga),
                            likes(Person, Raga).
likes_song(Person, Song) :-  likes(Person, Raga1),
     song(Song, Raga2), same_family(Raga1, Raga2).
```

Goals

```
:- song(Name, mayamalavagoula).
:- song(Name, Raga).

:- likes_song(siva,X).
```

Top-to-Bottom selection of facts/rules

`song(Name, mayamalavagoula)` has two answers.

```
Name = merusamana;
Name = devadeva;
```

and they will be produced in this order.

Similarly, `song(Name, Raga)` has 6 answers and will be produced in this order.

```
Name = giripai, Raga =  sahana;
Name = chakkani, Raga =  kharaharapriya;
Name = brocheva, Raga =  khamas;
Name = karikalaba, Raga =  saveri;
Name = merusamana, Raga =  mayamalavagoula;
Name = devadeva, Raga =  mayamalavagoula;
```

Left-to-Right for Subgoals

Consider the goal `likes_song(siva,X)`. From the *declarative* meaning we know that there are 3 answers. But, will they all be produced? In what order?

To solve this goal, two rules can be used. The first from top will be tried first and all answers using this produced.

Then, the program will try the second rule and try to produce more answers.

Using the first rule, we get two subgoals

`song(Song,Raga), likes(siva,Raga).`

These will be tried **left-to-right**

Procedural Semantics (ctd.)

The first answer for `song(Song,Raga)` is

`Song = giripai, Raga = sahana`

- Proceed to the remaining subgoal `likes(siva,Raga)` with the value `Raga = sahana`. That is, we try to solve the goal `likes(siva, sahana)`.
- *fails*, as no fact or rule matches this.
- Produce next answer for `song(Song, Raga)`

`Song = chakkani, Raga = kharaharapriya`

- *fail* on new subgoal `likes(siva,kharaharapriya)`

Continuing like this, we produce only one answer `X = karikalaba` for the goal `likes(siva,X)` using the first matching rule.

Solution Strategy (ctd.)

We now proceed to use the second rule for solving `likes(siva,X)`.
This gives subgoals

```
likes(siva, Raga1), song(Song, Raga2),
        same_family(Raga1, Raga2).
```

to be solved from left-to-right.
The first subgoal has only one answer `Raga1 = saveri`.
The first answer to second subgoal is again

`Song = giripai, Raga2 = sahana`

Now we proceed to the third subgoal

`same_family(saveri,sahana)`

Infinite Loops!

To solve `same_family(saveri, sahana)`,
the only rule usable is

```
same_family(R1, R2) :- same_family(R2, R1).
```

which produces the subgoal `same_family(sahana, saveri)`
To solve this, we use the same rule again and get back the goal `same_family(saveri, sahana)`.
And we are stuck in this loop!

1. How to fix the program to get all the answers for the query?
2. How to re-order goals in the body of rules?
3. How to re-order facts and rules to make the search better?
4. Draw in tree form the complete execution trace for sample goals.

It must be emphasized that the top-to-bottom and left-to-right choices are not a must. They are chosen only for efficiency.

Natural Numbers

As seen earlier:

```
Constant:   0
Function:   s   (unary)
```

The following Prolog program defines what is a natural number.

```
is_nat(0).
is_nat(s(X)) :-  is_nat(X).
```

What will be the solutions to the goal

```
is_nat(U).
```

Simple Predicates

How to test if a number is *even*?

```
is_even(0).
is_even(s(s(X))) :- is_even(X).
```

No need for division!

Similarly can you define

```
is_odd(X):- .....
is_multof_3(X):- ....
```

Comparison Functions

How to compare numbers? Let `is_less_equal(X,Y)` mean

"`X` is less than or equal to `Y`"

How does the following work?

```
is_less_equal(0,X) :- is_nat(X).
is_less_equal(s(X),s(Y)) :- is_less_equal(X,Y).
```

Consider queries with:

- No variables (`is_less_equal(s(s(0)), s(s(s(0))))`).
- One variable (`is_less_equal(s(s(0)),X)`).
- Both variables (`is_less_equal(U,V)`).

Question: Is Prolog *fair* when enumerating answers?

Simple Arithmetic

How to define **plus(X, Y, Z)** to mean

" Z is the sum of X and Y"

```
plus(0,Y,Y).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

Try sample goals:
- No variables.
- 1, 2, 3 variables.

Don't we need to say:

```
plus(0,Y,Y) :- is_nat(Y).
```

Remember: there is no explicit *typing* in Prolog!

Subtraction

How do we write the relation **diff(X, Y, Z)** to mean that:

"Z is X - Y."

Method 1:

```
diff(X,0,X).
diff(s(X),s(Y),Z) :- diff(X,Y,Z).
```

Method 2:

```
diff(X,Y,Z) :- plus(Y,Z,X).
```

Questions: Are these the same?

Are we testing if Y is less-than X anywhere?

Multiplication

How do we write **times(X, Y, Z)** to mean

"Z is the *product* of X and Y."

Check this solution:

```
times(0,Y,0).
times(s(X),Y,Z):- times(X,Y,Z1), plus(Y,Z1,Z).
```

Again try all types of goals.

- No variables.
- 1, 2, 3 variables.

Note about Prolog

The example of times shows how definitions written in Prolog which work well in the "testing" mode (that is, queries with all arguments as ground terms without varibles) do not behave as desired when some of the arguments are changed to variables.

That is, even though Prolog does not give any wrong answers, it has the following drawbacks:

- Answers not enumerated fairly.
- Prolog runs out of stack/heap and aborts.
- All answers enumerated but Prolog continues in an "infinte loop".

Using the **times** example all this behaviour can be observed

This "misbehaviour" of Prolog is because of the two "arbitrary" choices that Prolog made:

- Top To Bottom in choosing rules/facts.
- Left To Right when solving goals.

for efficiency reasons.

Division

divides(X, Y) means

"Y is a *factor* of X"

Method 1:

```
divides(X,Y):- times(Y, U, X)
```

Method 2:

```
divides(0,Y).
divides(Y,Y).
divides(X,Y):-  plus(Y,U,X), divides(U,Y).
```

Questions: Which is better?

- Do we need the second fact in Method 2?
- How do they work on different queries?

Remainder

mod(X,Y,Z) means

"On dividing X by Y we get Z as the remainder"

```
mod(X,Y,X):- is_less(X,Y).
mod(X,Y,Z):- plus(Y,U,X), mod(U,Y,Z).
```

Compare this with the "`divides`" predicate.
Similarly can you write "`quotient`" predicate?
How to do this using the "`times`" predicate?

GCD computation

gcd(X, Y, Z) means

"Z is the greatest common divisor of X and Y"

```
gcd(0,X,X) :- is_gt(X,0)  % To avoid gcd(0,0,0)
gcd(X,Y,Ans) :- mod(X,Y,Z), gcd(Y,Z,Ans).
```

Exercise: Trace some sample goals. How to do this using the "`times`" predicate?

Homework

Try the following.

```
factorial(0) = 1
factorial(n+1) = (n+1) * factorial(n)

ackermann(0,N) = N + 1.
ackermann(M,0) = ackermann(M-1, 1).
ackermann(M+1,N+1) =
   ackermann(M, ackermann(M+1, N))

fib(0) = 1.
fib(1) = 1.
fib(n+2) = fib(n+1) + fib(n).
```