INST0072 Logic and Knowledge Representation

Lecture 5

Prolog: Terminology, and How Prolog Works

Recap of Lecture 2

In Lecture 2 we saw that

- Prolog programs consist of *facts* and *rules*. An example of a fact from 'mothersAndFathers.pl' is

  ```
  father(rob, rebecca).
  ```

- Prolog *rules* provide a means of defining relationships in terms of other relationships. One of the rules in 'mothersAndFathers.pl' is

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

- The ':-' is read as an 'if', so that the above rule can be interpreted as

  'X is the grandfather of Z if X is the father of Y and Y is the father of Z'

- In general, rules have the form

  $\texttt{A :- B}_1\texttt{, ..., B}_n\texttt{.}$

  '`A`' is known as the *head* of the rule, the conjunction '$\texttt{B}_1\texttt{, ..., B}_n$' is known as the *body*.

- The definition of a *predicate* may be a mixture of facts and rules. Here is part of the defintion for the predicate 'male/1':

  ```
  male(X) :-
      father(X, Y).
  male(oscar).
  .....
  ```

- Prolog programs are run via queries:

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

[mothersAndFathers.pl]

Clauses, Variables and Variable Names

- The facts and rules in a Prolog program are often called *clauses*.
- Any term in a clause that begins with an uppercase letter or the underscore character ("_") is a (logical) *variable*.
- Variables in Prolog programs are completely different from variables in procedural languages such as Javascript. Prolog variables are **not** "pockets of computer memory".
- Each Prolog clause is a statement in its own right, so that variables with the same name in different clauses do not refer to the same thing. But within a clause, variables with the same name do refer to the same thing.
- Example: this two-clause program

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

  and this two-clause program

  ```
  grandfather(X, Z) :-
      father(X, Y), father(Y, Z).
  grandfather(A, C) :-
      father(A, B), mother(B, C).
  ```

  mean exactly the same thing, and behave in exactly the same way, in Prolog.

Variables and Logical Quantifiers

- We can regard variables that appear in the head of a clause as being *universally quantified*, i.e. as being in the scope of a ∀ quantifier. Variables which appear only in the body of a clause can be thought of as *existentially quantified*, i.e. as being in the scope of a ∃ quantifier.

- Example: the clause

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

  means "*for all X's, X is a parent if there exists a Y such that X is the mother of Y*". In classical logic:

  ```
  ∀x.(parent(x) ← ∃y.mother(x,y))
  ```

  In prenex normal form this is equivalent to:

  ```
  ∀x∀y.(parent(x) ← mother(x,y))
  ```

- Prolog makes sure that it won't get confused by variables with the same name in different clauses by continually renaming all variables (with names such as '`_G1078`') during the processing of a query. This can be seen when using a Prolog "trace" facility (more about this later in the lecture), and is somewhat similar to the idea of 'standardising variables apart' (see lecture 3, slide 30).

The Anonymous Variable

- Recall that Prolog variable names either begin with an upper-case letter or with the underscore character. For example, '`Teacher`', '`X`' and '`_x`' are all variable names.
- There is one special variable called the *anonymous variable*, denoted by '`_`', which is not subject to the usual interpretation or quantification of variables.
- The values of two or more anonymous variables in one clause may or may not be the same.
- For example, the clause

  ```
  p(_, _) :- q(_).
  ```

  should be thought of as

  ```
  p(X, Y) :- q(Z).
  ```

Clauses

- Recall that the facts and rules making up Prolog programs are known as *clauses*. The general form of a clause is as follows:

  $\texttt{p(...) :- q}_1\texttt{(...), q}_2\texttt{(...), ..., q}_n\texttt{(...).}$

  '`p(...)`' is called the *head* of the clause, '$\texttt{q}_1\texttt{(...), q}_2\texttt{(...), ..., q}_n\texttt{(...)}$' is called the *body* of the clause, and each individual '$\texttt{q}_i\texttt{(...)}$' is called a *sub-goal* of the clause.

- The set of clauses that have '`p(...)`' as their head is called the *definition* of the predicate 'P/n', where 'n' is the number of *arguments* in the '`(...)`' brackets, also known as the *arity* of P.

- Example:

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

  is a definition of the predicate 'grandfather/2', consisting of two clauses. The head of the first clause is '`grandfather(X, Z)`' and body is '`father(X, Y), father(Y, Z)`'. The head of the second clause is again '`grandfather(X, Z)`' and the body is '`father(X, Y), mother(Y, Z)`'.

Terms and Functors

- The individual arguments inside the '`(...)`' brackets of predicates can have more stucture than just constants or variables. In general they can be *terms* (Bratko refers to terms as *structures*).

- For example, we might write

```
    likes(friend_of(john), cooking_of(mary)).
    likes(X, cooking_of(father_of(X))).
```

  to express that john's friend likes mary's cooking, and everyone likes their own father's cooking.

- '`john`', '`friend_of(john)`', '`mary`', '`cooking_of(mary)`', '`X`', '`father_of(X)`' and '`cooking_of(father_of(X))`' are all terms. '`friend_of`', '`cooking_of`' and '`father_of`' are *functors*.

- In general
  - a constant is a term
  - a variable is a term
  - an expression '$f(t_1, \ldots\ldots, t_n)$' is a term, if '`f`' is a functor and 't_1, ..., t_n' are all terms.

Substitution and Matching

- A *substitution* is a set of *assignments* (or *bindings*) to a collection of variables. For example `{X=john, Y=big_mac}`, or `{X=mary, Y=Z}`.

- To indicate that a substitution 's' has been applied to the variables in a goal 'G' we will write 'Gs'. For example:

```
likes(X, Y){X=mary, Y=Z} = likes(mary, Z).
```

- We say the goal A is an *instance* of the goal B if there is a substitution s such that A = Bs.

- We say the goals A and B *match* or *unify* if there is a substitution s such that As = Bs.

- Examples:
    - `likes(mary, Z)` is an instance of `likes(X, Y)`, with substitution `{X=mary, Y=Z}`.
    - `likes(X, cooking_of(father_of(X)))` matches with `likes(eric, Y)`,
      with substitution `{X=eric, Y=cooking_of(father_of(eric))}`.

- All of the above definitions apply to terms as well as goals.

- More formally and precisely, a substitution is a set of the form $\{X_1=t_1, \ldots\ldots, X_n=t_n\}$ where each X_i is a distinct variable, each t_j is a term, and no X_i occurs in any t_j.

The Unification Algorithm

- As we saw in the last slide, two goals or terms A and B unify (i.e. match) if there is a substitution s such that As = Bs.
- There may be more than one such substitution, but there is always one that is more general than any other. This one is called the *most general unifier* of A and B.
- When answering a query, Prolog uses a routine called the *unification algorithm* which is able to find the most general unifier of any two terms or goals.
- Examples:
  - '`likes(john,X)`' unifies with '`likes(john, big_mac)`', with most general unifier `{X=big_mac}`.
  - '`equal(X,Y)`' unifies with '`equal(Z,Z)`', with most general unifier `{X=Z, Y=Z}`.
  - '`takes(john, ai)`' does not unify with '`takes(X,prolog)`'.

An Example Program - Happy Teacher

- An example program:

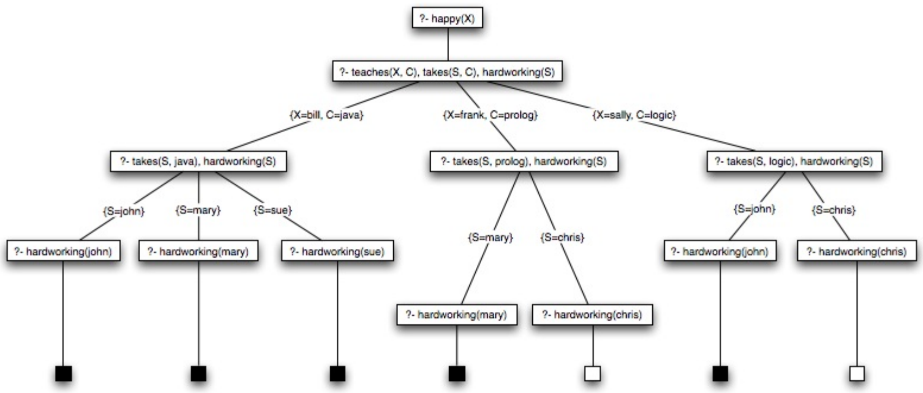
```
teaches(bill, java).
teaches(frank, prolog).
teaches(sally, logic).

takes(john, java).
takes(john, logic).
takes(mary, java).
takes(mary, prolog).
takes(sue, java).
takes(chris, logic).
takes(chris, prolog).

hardworking(chris).

/* A teacher is happy if he/she is teaching at least one ha
happy(Teacher) :-
    teaches(Teacher, Course),
    takes(Student, Course),
    hardworking(Student).
```

- An example query:

```
?- happy(X).
X = frank ;
X = sally.
```

[happyTeacher.pl]

The Search Tree for Happy Teacher



[happyTeacherSearchTree.pdf] [happyTeacherSearchTree in steps.pdf]

The SWI Prolog Command Line Tracer

- It is possible to use the command line interface of SWI Prolog to "trace" through the search tree of a query, using the command '`trace`':

```
?- trace.
true.

[trace]  ?- happy(X).
   Call: (6) happy(_G2787) ? creep
   Call: (7) teaches(_G2787, _G2859) ? creep
   Exit: (7) teaches(bill, java) ? creep
   Call: (7) takes(_G2858, java) ? creep
   Exit: (7) takes(john, java) ? creep
   Call: (7) hardworking(john) ? creep
   Fail: (7) hardworking(john) ? creep
   Redo: (7) takes(_G2858, java) ? creep
   Exit: (7) takes(mary, java) ? creep
   Call: (7) hardworking(mary) ? creep
   Fail: (7) hardworking(mary) ? creep
   Redo: (7) takes(_G2858, java) ? creep
   Exit: (7) takes(sue, java) ? creep
   Call: (7) hardworking(sue) ? creep
   Fail: (7) hardworking(sue) ? creep
   Redo: (7) teaches(_G2787, _G2859) ? creep
   Exit: (7) teaches(frank, prolog) ? creep
   Call: (7) takes(_G2858, prolog) ? creep
   Exit: (7) takes(mary, prolog) ? creep
   Call: (7) hardworking(mary) ? creep
   Fail: (7) hardworking(mary) ? creep
   Redo: (7) takes(_G2858, prolog) ? creep
   Exit: (7) takes(chris, prolog) ? creep
   Call: (7) hardworking(chris) ? creep
   Exit: (7) hardworking(chris) ? creep
   Exit: (6) happy(frank) ? creep
X = frank .
```

- To turn off the tracer, use the query '`notrace`' followed by the query '`nodebug`'.

Prolog's Execution Strategy

Given a program and a goal, Prolog's execution strategy proceeds as follows:

a. It selects the first sub-goal of the goal.

b. It looks for the clause in the program whose head unifies with this sub-goal

c. When it finds such a clause it makes a new goal from the original goal by replacing the first sub-goal by (all) the sub-goals of this clause, and applies the substitution resulting from the (most general) unification of the sub-goal and the clause head to this new goal.

d. It re-applies the above procedure to this new goal.

Succeeding, Failing and Backtracking

In Prolog's execution strategy, one of two things can happen:

Either

> Prolog reaches a state in which the goal is 'empty', i.e. it has solved all the sub-goals of the original goal. This is called *succeeding*, and the substitution that has been built up during the process of solving sub-goals (restricted to the variables of the original goal) is called the *computed answer substitution*.

Or

> There is no clause head that unifies with the sub-goal Prolog is attempting to solve. In such a situation, Prolog *backtracks*, that is, is attempts to find an alternative way of satisfying the most recently solved sub-goal. To do this, it goes back to the most recent point that a sub-goal was solved (i.e. matched with a clause head), undoes any bindings (i.e. assignments to variables) made by substitutions that arose from this match, returns the goal to the state it was in just prior to this match, and then continues to search down the program for another clause head that matches this sub-goal. If such a clause head is found, Prolog continues with its execution as before, with the new goal and substitution arising from this match. Otherwise, it backtracks again.

If Prolog has tried all possible alternatives arising from backtracking but still has been unable to reduce the original goal to the empty goal, then the original goal *fails* (and Prolog answers the goal with 'false').

Depth First Search and Looping

- As we have seen with the "happyTeacher.pl" example, the graph of all possible clauses for a goal, together with all possible bindings of variables, is called the *search tree* for the goal. Prolog's search strategy is sometimes called *depth first search*, because of the way it explores the search tree, one full branch at a time.

- Most Prolog systems have a trace facility similar to that of SWI Prolog, so that programmers can observe Prolog's progress through a particular search tree. This is often useful for debugging programs.

- In practice, Prolog's execution strategy means that it is possible to design programs and queries to them which make the system go into an infinite *loop* (i.e. never succeed or fail). Some care needs to be taken to avoid these. One simple example of a loop would be the program:

  ```
  happy(X) :- healthy(X).
  healthy(X) :- happy(X).
  ```

  with the query

  ```
  ?- happy(rob).
  ```

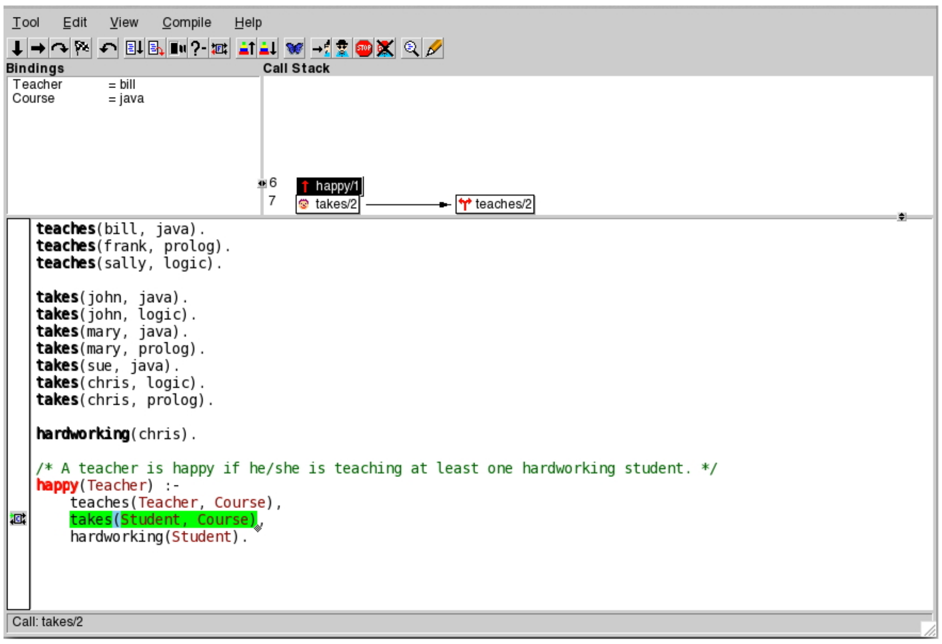
  (An even simpler example would be '`happy(X) :- happy(X).`'.)

[happyHealthy.pl]

The SWI Prolog Graphical Tracer

- SWI Prolog has a graphical (or 'GUI') facility for "tracing" through the search tree of a query, called with either the command (i.e. goal) '`gtrace`' or the command '`guitracer`':

```
?- gtrace.
% The graphical front-end will be used for subsequent traci
true.

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



```
Teacher     = bill
Course      = java
```

```
6  happy/1
7  takes/2 ——→ teaches/2
```

```
teaches(bill, java).
teaches(frank, prolog).
teaches(sally, logic).

takes(john, java).
takes(john, logic).
takes(mary, java).
takes(mary, prolog).
takes(sue, java).
takes(chris, logic).
takes(chris, prolog).

hardworking(chris).

/* A teacher is happy if he/she is teaching at least one hardworking student. */
happy(Teacher) :-
    teaches(Teacher, Course),
    takes(Student, Course),
    hardworking(Student).
```

```
Call: takes/2
```

- To swap from the GUI tracer back to the basic command line tracer, use the query '`noguitracer`'.

- As before, to turn off either the GUI or the command line tracer, use the query '`notrace`' followed by the query '`nodebug`'.