INST0072 Logic and Knowledge Representation

Lecture 8

Prolog: Meta-level programming

In the Last Lecture

In the last lecture we saw that

- In Prolog, 'not' is regarded as 'not provable', and the symbol for this in SWI Prolog is '`\+`'. For example, a definition of 'cousin' might be:

  ```
  cousin(X,Y) :-
      grandparent(Z,X),
      grandparent(Z,Y),
      \+ brother(X,Y),
      \+ sister(X,Y),
      \+ X == Y.
  ```

- Prolog uses a rule called *negation as failure* to evaluate a negative sub-goal. If it encounters the query

  ```
  ?- \+ p(...)
  ```

  during some execution, it attempts to evaluate the query

  ```
  ?- p(...)
  ```

  If '`?- p(...)`' succeeds then '`?- \+ p(...)`' fails, but if '`?-p(...)`' fails then '`?- \+ p(...)`' succeeds.

- Command line output can be achieved in Prolog with the inbuilt predicate 'write/1'. The goal

  ```
  write(X)
  ```

  always succeeds and as a side effect displays the term `X` on the screen.

- Command line input can be achieved in Prolog with the inbuilt predicate 'read/1'. The goal

  ```
  read(X)
  ```

  always succeeds and as a side effect instantiates (i.e. assigns) `X` with whatever is subsequently typed at the keyboard.

An Example from the Last Lecture

- Suppose we have the following program:

```
happy(X) :-
    rich(X),
    \+ stressed(X).

stressed(X) :-
    overworked(X).
stressed(X) :-
    tired(X).

rich(sally).
rich(jim).
rich(ann).

overworked(sally).

tired(jim).
```

- If we query this program with

```
?- happy(X)
```

  Prolog will answer with the single solution

```
X = ann.
```

[happyNotStressed.pl]

Meta-level Programming

- A Prolog program can include statements (i.e. clauses) about itself. Such statements are sometimes called *meta-level* or *meta-logical* statements.
- It can even include statements which change the program itself when they are evaluated. This is useful (for example) when maintaining or altering a database of Prolog facts and/or rules (such a database is sometimes called an *object-level program*).
- Three commonly used predefined meta-level predicates are 'clause/2', 'assertz/1' and 'retract/1':
    - '`clause(`*`Head`*`, `*`Body`*`)`' is true if there is a clause '*`Head`*` :- `*`Body`*`.`' in the version of the program currently in working memory.
    - '`assertz((`*`Clause`*`))`' always succeeds and as a side effect adds '*`Clause`*' to the version of the program currently in working memory.
    - '`retract((`*`Clause`*`))`' succeeds if there is a clause of the form '*`Clause`*' in the version of the program currently in working memory, but as a side effect removes that clause.
- The arguments *`Head`* and *`Clause`* in 'clause/2', 'assertz/1' and 'retract/1' described above must be *partially bound* when they are called (i.e. they can't just be uninstantiated variables). Additionally, 'assertz/1' and 'retract/1' will only work with predicates previously declared as dynamic, by a line in the program such as:

```
:- dynamic likes/2.
```

Some Example Queries with a Dynamic Predicate and Built-in Predicates

- Suppose we have the following Prolog program saved in the file 'likes.pl':

```
:- dynamic likes/2.

likes(mary, ice_cream).
likes(sue, apple_pie).
```

- We can change Prolog's working directory to the directory containing 'likes.pl' with the inbuilt predicate 'working_directory/2':

```
?- working_directory(_, 'n:/prolog/inst0072').
```

- We can then load 'likes.pl' into Prolog's working memory with a 'consult/1' query:

```
?- consult(likes).
```

- We can alter the definition of 'likes/2' with 'assertz/1' and 'retract/1' queries:

```
?- assertz((likes(eric, bananas))).
?- retract((likes(mary, ice_cream))).
```

- We can check the current definition of 'likes/2' in working memory with a 'listing/1' query:

```
?- listing(likes/2).
```

- We can direct the output of 'listing/1' back the file 'likes.pl' with 'tell/1' and 'told/0' queries:

```
?- tell('likes.pl'), listing(likes/2), told.
```

[likes.pl]

Example SWI Prolog Session

```
1 ?- working_directory(_, 'n:/prolog/inst0072').
true.

2 ?- consult(likes).
% likes compiled 0.00 sec, 3 clauses
true.

3 ?- listing(likes/2).
:- dynamic likes/2.

likes(mary, ice_cream).
likes(sue, apple_pie).

true.

4 ?- assertz((likes(eric, X) :- likes(sue, X))).
true.

5 ?- listing(likes/2).
:- dynamic likes/2.

likes(mary, ice_cream).
likes(sue, apple_pie).
likes(eric, A) :-
        likes(sue, A).

true.

6 ?- clause(likes(P, T), Body).
P = mary,
T = ice_cream,
Body = true ;
P = sue,
T = apple_pie,
Body = true ;
P = eric,
Body = likes(sue, T).

7 ?- retract((likes(P1, T1) :- likes(P2, T2))).
P1 = eric,
T1 = T2,
P2 = sue.

8 ?- listing(likes/2).
:- dynamic likes/2.

likes(mary, ice_cream).
likes(sue, apple_pie).

true.

9 ?- assertz((likes(eric, cheese))).
true.

10 ?- listing(likes/2).
:- dynamic likes/2.

likes(mary, ice_cream).
likes(sue, apple_pie).
likes(eric, cheese).

true.

11 ?- tell('likes.pl'), listing(likes/2), told.
true.
```

A Meta-level User Interface

- A meta-level program saved in the file 'addLikes.pl' that allows the user to add 'likes/2' facts:

```
add_likes :-
    write('Who likes something? '),
    read(Person),
    write('What do they like? '),
    read(Thing),
    assertz((likes(Person, Thing))),
    tell('likes.pl'),
    listing(likes/2),
    told,
    write('\'likes('),
    write(Person),
    write(', '),
    write(Thing),
    write(')\' has been added to the program and saved to the file \'likes.pl\'.\n').
```

- Example input/output:

```
1 ?- [likes,addLikes].
% likes compiled 0.00 sec, 4 clauses
% addLikes compiled 0.00 sec, 2 clauses
true.

2 ?- add_likes.
Who likes something? rob.
What do they like? coffee.
'likes(rob, coffee)' has been added to the program and saved to the file 'likes.pl'.
true.

3 ?- listing(likes/2).
:- dynamic likes/2.

likes(mary, ice_cream).
likes(sue, apple_pie).
likes(rob, coffee).

true.
```

[addLikes.pl | likes.pl]

Meta-interpreters

- The 'clause/2' predicate can be used in programs which describe Prolog's execution strategy (Lecture 2) or similar strategies. Such programs are called *meta-interpreters*.

- An example is the program

```
provable(true) :- !.          /* cut to make sure only this clause used for 'true' */

provable((Goal1, Rest)) :-    /*  for conjunctions of goals  */
    provable(Goal1),
    provable(Rest).

provable(Goal) :-             /* for goals which match with the  */
    clause(Goal,Body),        /* head of a clause in the program */
    provable(Body).
```

which describes Prolog's strategy for solving queries to programs which don't include negation-as-failure.

- The definition of 'provable/1' could be modified or extended in several ways, for example to alter Prolog's search strategy, or query the user about goals that can't be satisfied within the program.

[provable.pl]

Logic & Prolog: The Meaning of Prolog Programs

- When programs do not incorporate negation-as-failure, Prolog's strategy for finding a solution to a query is analagous to the *resolution* proof procedure (see lecture 3).
- However, unlike Classical Logic, Prolog incorporates two assumptions about the knowledge represented in programs:
    1. *The Closed World Assumption:*
       Prolog assumes that any statement it can't prove is false (i.e. it assumes it knows everything!).
    2. *The Uniqueness-of-names Assumption:*
       Prolog assumes that different names (i.e. constants and terms without variables) refer to different objects. For example, it assumes that '`mary`' and '`mother_of(eric)`' are two different people.
- So the 'meaning' of the program consisting of the two clauses

  ```
  likes(mary, ice_cream).
  likes(sue, apple_pie).
  ```

  might be written in classical logic as the theory

  ```
  (∀x)(∀y)(Likes(x,y) ↔ ((x = Mary ∧ y = Ice_cream)
                          ∨ (x = Sue ∧ y = Apple_pie)).
  Mary ≠ Ice_cream ∧ Mary ≠ Sue ∧ Mary ≠ Apple_pie
  ∧ Ice_cream ≠ Sue ∧ Ice_cream ≠ Apple_pie ∧ Sue ≠ Apple_pie.
  ```

- This is a simple example of *Clark Completion*, which we will look at in the next lecture.

Other Logic Programming Systems

- **Constraint Logic Programming (CLP)**. CLP systems allow the programmer to include (usually mathematical) *constraints* using relationships such as "less than", "greater than" etc. in the body of clauses. Constraints are gathered together in a *constraint store* and checked for consistency during the execution of a query. SWI Prolog has constraint handling capability.
- **Abductive Logic Programming (ALP)**. ALP systems let the program make and store *assumptions* about certain predicates (called *abducibles*) while answering a query. This makes such programs good for generating possible explanations for observed phenomena, e.g. for fault diagnosis problems.
- **Inductive Logic Programming (ILP)**. ILP systems let the program *learn general rules* from a series of (positive and negative) examples.
- **Answer Set Programming (ASP)**. ASP programs look similar to Prolog programs, but the computation is *model-theoretic* rather than proof-theoretic. An ASP system shows a goal is true by constructing a model of the program that also includes the goal, and by failing to construct a model of the program in which the goal is not included.