COMP219:
Artificial Intelligence

Lecture 15: Prolog – Negation as Failure and Additional Programming Features

Overview

- Last time
  - n-queens problem; preventing backtracking with ‘cut’; green cuts and red cuts.

- Today:
  - ‘Negation as failure’ (using cut-fail combination)
  - The Closed World Assumption
  - Input/Output
  - Built-in predicates
  - Sample questions for the first class test

- Learning outcome covered today:

  Understand and write Prolog code to solve simple knowledge-based problems.

Singleton Variables

- Whilst completing your lab exercises, you will likely have come across Prolog warnings about singleton variables

- This is an indication that your program contains a statement in which a variable appears only once
  - This is to alert you that there may be a spelling mistake or that you forgot to use the variable elsewhere

- You can get rid of the warning by replacing the variable with the anonymous variable '_', or prefixing the variable's name with '_'
  - e.g. '_Y1'

Negation as Failure (and necessary for results)

- fail is a built in goal that always fails
- true is a built in goal that always succeeds
- Cut and fail are often used in combination

```
different(X,X):- !,fail.
different(_,_).
```

Where X in both, don't backtrack, and result fails. Otherwise, try other instantiations

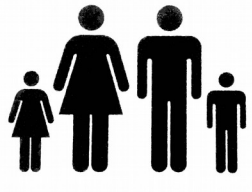
- If the two inputs to 'different' match, the first clause is matched
- The cut means the program is committed to this choice and cannot subsequently match with the second 'different' clause even if this can be true
- The goal 'fail' itself results in backtracking. However, in combination with cut in this order, no further choices

Negation as Failure


```
?- different(a,b).
true.
?- different(a,a).
false.
```

- If two inputs that do not match are supplied to `different` they cannot be matched with the first clause
- They can be matched with the second clause, which succeeds
- In summary, different(X,Y) means if X and Y match, then different(X,Y) fails, otherwise different(X,Y) succeeds

Family Tree Revisited



- Previously have encountered the issue of someone being their own sibling
- Can remedy this by defining sibling using *different* and *negation as failure*:

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

different(X,X):- !,fail.
different(_,_).
```

- This will cease to report someone being their own sibling (- this issue cropped up in lab exercise 1)

Not Member

- The cut-fail combination is useful for “not” queries
- One way to have `notmember` defined in terms of `member`:

```
member(H,[H|Tail]).
member(X,[_|Tail]):- member(X,Tail).

notmember(X,List):- member(X,List),!,fail.
notmember(_,_).
```

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

If member(X,List) is true, apply cut, and report false; notmember(X,List) is false.
If member(X,List) is false, don't get to cut or fail. Go to next clause, notmember(_,_), which is true on any terms.
Like *not(P)* construction, where:
not (member(X,List)) is true if member(X,List) is false; not (member(X,List)) is false if member(X,List) is true.

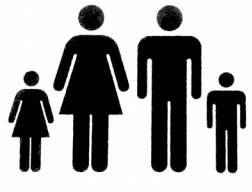
Not Member

- Another version of `notmember`
- Most versions of Prolog have a built in operator ***not*** that is defined using cut and fail where **`not Goal`** is true if **`Goal`** is not true

```
notmember(X,List):- not(member(X,List)).
```

- Be careful using both *cut* and the *cut fail* combination as they sometimes produce counter-intuitive results
- We have seen 'negation as failure' with explicit expressions. And previously we have seen it implicitly…

Family Tree Again and NAF



- Consider our original family tree program

```
female(cathy). % Cathy is female.
female(lucy).
female(pauline).
female(lou).
```

- and the following queries

```
?- female(cathy).
true.
?- female(peggy).
false.
```

- This means it is not possible to show in the program that peggy is female; where it fails to show it is true, the program returns false. In the 'real world', peggy could be male, or it could be that we do not know

Closed World Assumption

- The world is closed in the sense that everything that exists is stated in the program or can be derived from the program
- If something is not in the program (or cannot be derived from it) then it is presumed to be not true (false) and consequently its negation is true
- Whether what is presumed to be not true is "actually" true or not is irrelevant to the program
- Think of the basic facts of the program as a "toy" or "artificial" model world
- The program has all the facts of this toy model
  - All students have a record on Spider
- The rules provide a complete procedure, and there is no other way of showing the required fact

Closed World Assumption

- If we added

```
notfemale(X):- female(X),!,fail.
notfemale(_).
```

- and tried the following queries

```
?- notfemale(cathy).
```

`false.` (because cathy is a female)

```
?- notfemale(peggy).
```

`true.` (because peggy is not found to be female)

- The latter is counter-intuitive since we don't expect to have every female in our database
- It should not be understood as *peggy is not female*, but as *there is no information in the program to prove that peggy is female, so we conclude, relative to the program, that she is not*

Closed World Assumption

- The Closed World Assumption can sometimes be valid:

```
suit(clubs).
suit(diamonds).
suit(hearts).
suit(spades).
```

- We have all the suits, so we can accept a `false` to `suit(wands)` as it is really false
- In other words, the world as it "is" corresponds to our model of it in the program since decks of cards only have four suits

CWA for Rules

```
student(X):- undergraduate(X).
student(X):- postgraduate(X).
```

- There is no other way for a person to be a student so a false to student(trevor) means that *trevor* is not a student (Spider records all UGs and PGs)
- We (implicitly) complete the database with

```
notstudent(X) :- not(postgraduate(X)),
                 not(undergraduate(X)).
```

  i.e. student(X) if and only if undergraduate(X) or postgraduate(X)
- If this completion is acceptable (e.g. we are happy with this definition of student) then we can make the CWA

Input/Output

- write writes out a prolog term `write(katie)`
- read reads in a prolog term `read(Name)`

```
?- write(katie).
katie
true.
?- read(Name).
|: john.
Name = john.
```

Writes output

Variable gets value

- Always evaluate to true, so no effect on the logic

Input/Output - Interaction

```
cube :-
write('Next item, please:'),
   read(X),
   process(X).

process(stop) :- !.

process(N) :-
   C is N*N*N,
   write('Cube of '),
   write(N), write(' is '),
   write(C), nl,
   cube.
```

`cube` is a procedure that reads in numbers, processes them (cubed), writes the result, calls `cube` again, unless the process is stopped (with the `stop` atom).
Some textual prompts added.

```
?- cube.
Next item, please:5.
Cube of 5 is 125
Next item, please:12.
Cube of 12 is 1728
Next item, please:stop.
true.
```

Writing a List

```
writelist([]).
writelist([X|L]) :-
   write(X), nl,
   writelist(L).
```

Writes a list, one item of the list at a time

?- writelist([]).
true.

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

Exercise

- Consider the following program

```
writers([],[]).
writers([H1|T1],[H2|T2]) :-
write([H1, writes, H2]),nl,
writers(T1,T2).
```

- What will be the output to the following query?

```
writers([will,carol,ruth],[plays,poems,novels]).
```

Built-in Predicates: Lists

- Most of the list manipulation predicates are built into SWI-Prolog:
  - append(List1, List2, List3)
  - member(Elem, List)
  - length(List, Int)
- No need to define them for each new program; can just assume they are there and use them

Built-in Predicates: Sorts

- We sometimes need to know the type of different terms (variable, integer, atom, etc.)
- Suppose we want to add the values of two variables: `Z is X + Y.`

- X and Y are required to be instantiated as integers before the arithmetic can be calculated. If unsure that they are, can perform a check using the built-in predicate `integer`.

```
-? integer(X), integer(Y), Z is X+Y.
false.

-? X=5, integer(X).
X=5.
```

- `integer(+Term)` succeeds if Term is bound to an integer

More Sorts to Test a Term

- `nonvar(+Term)` succeeds if Term is not a variable, or Term is already instantiated
- `atom(+Term)` succeeds if Term is bound to an atom
- `float(+Term)`
- `atomic(+Term)` succeeds if Term is bound to an atom, string, integer or floating point number
- `compound(+Term)` succeeds if Term is bound to a compound term (a structure)

- More details about facilities provided in Prolog for advanced programming can be found in the recommended textbook

Class Test

- Questions on
  - Querying a Prolog program
  - Extending a Prolog program
  - Lists
  - Recursion
  - Cut, Fail, and their effects

Arrangements

- Tuesday 8th November (Week 7), 13:00-14:00, LIFS-LT2 and LIFS-LT3.

- Students whose surnames start with letters between A and K should report to LIFS-LT2, while students whose surnames start with letters between L and Z should report to LIFS-LT3.

- The test will start promptly at 13:00 and last 50 minutes.

Test Structure

- There will be multiple labelled sections
  - e.g. Section A, Section B etc.
- Some sections will have two (or more) parts according to the style of questions
  - e.g. part 1 multiple choice; part 2 writing code
- The marks will be stated for each section, or per question if they vary
  - e.g. every question in this section is worth 5 marks

Questions from a previous test

```
parent(pete,ian).      % Pete is a parent of Ian
parent(ian,peter).
parent(ian,lucy).
parent(lou,pete).
parent(lou,pauline).
parent(cathy,ian).

female(cathy).         % Cathy is female.
female(lucy).
female(pauline).
female(lou).

male(ian).             % Ian is male.
male(pete).
male(peter).
```

- Formulate a query to find a person who is a parent of *both* `pete` and `pauline`.

```
?- parent(X,pete), parent(X,pauline).
```

Questions from a previous test

```
parent(pete,ian).      % Pete is a parent of Ian
parent(ian,peter).
parent(ian,lucy).
parent(lou,pete).
parent(lou,pauline).
parent(cathy,ian).

female(cathy).         % Cathy is female.
female(lucy).
female(pauline).
female(lou).

male(ian).             % Ian is male.
male(pete).
male(peter).
```

- Define the relation `grandchild` using the `parent` relation.

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

Questions from a previous test

```
parent(pete,ian).      % Pete is a parent of Ian
parent(ian,peter).
parent(ian,lucy).
parent(lou,pete).
parent(lou,pauline).
parent(cathy,ian).

female(cathy).         % Cathy is female.
female(lucy).
female(pauline).
female(lou).

male(ian).             % Ian is male.
male(pete).
male(peter).
```

- What is the first solution found for the following query?

```
?- parent(X, Y), male(Y).

X = pete
Y = ian
```

Questions from a previous test

Consider the following append predicate considered in the course:

```
append([], L, L).
append([H|T], L, [H|L2]) :- append(T, L, L2).
```

What is the result of execution of the following query?

```
?- append([1,3], X, [1, 3, 4, 5, 7]).
```

```
X = [4, 5, 7]
```

Questions from a previous test

Define the predicate `ordered(List)` which is true if numbers in the list `List` are ordered. For example, `ordered([1,2,6,19])`.

```
ordered([]).
ordered([X]).
ordered([H1|[H2|T]]) :- H1 < H2, ordered([H2|T]).
```

Other Questions

- The example questions presented in these notes are from previous years. The current test may have differences in format, for example, some questions will be multiple choice

- As mentioned earlier, the questions on the test could cover topics additional to those given in these examples questions (e.g. there might be questions on cut, negation as failure, procedural and declarative meaning, etc.)

Summary

- ‘Negation as failure’
  - Cut makes it possible to introduce *negation as failure* via the cut-fail combination
  - Negation as failure corresponds to “real” negation only if the closed world assumption is valid. Otherwise it only means cannot be shown

- We have also explored further aspects of Prolog
  - Input/Output
  - Built in predicates
    - These two aspects will not be covered explicitly in the Prolog class test

- This lecture concludes the Prolog part of the module; from next week, Friday lectures will continue with the general AI material of the module