Prolog Supervision Work

Andrew Rice[0],
Department of Computer Science and Technology,
Cambridge University.
acr31@cam.ac.uk

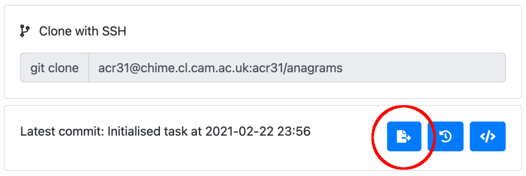
Academic Year 2020–21

Introduction

These questions form the suggested supervision material for the Prolog course. Supervisors are encouraged to select an appropriate subset of these for their students to complete.

Prolog contains a number of features and facilities not covered in the lectures such as: `assert`, `findall` and `retract`. Students should limit themselves to using only the features covered in the lecture course and are not expected to know about anything further. All questions can be successfully answered using only the lectured features.

Many of the questions here are practical exercises to be completed on the Chime service. These are also linked at appropriate points between the lecture videos. Chime provides an export page to summarise your work on a task. Save or print this page as PDF for handing in.



```
git clone acr31@chime.cl.cam.ac.uk:acr31/anagrams
```

Latest commit: Initialised task at 2021-02-22 23:56

Questions are classified into the following types:

Bookwork questions that require the students to review the lectured material and locate the relevant information

Shallow questions that require recall of lectured material and its direct application in a formulaic manner

Deeper questions that require students to apply the lectured material in a new context or to relate material to each other but still with a clear right or wrong answer

Open ended questions requiring students to form their own viewpoints with various ways to interpret the answer

[0]With grateful thanks to Ian Lewis and Nik Sultana who also lectured this course and contributed to this question set.

Supervision 1

Prolog Basics

(1.1) Deeper Different implementations of Prolog produce different behaviour when you attempt to unify `a(A)` with `A`. Describe the various possibilities which might arise.

(1.2) Open How does unification relate to ML type inference? What is the ML equivalent of unifying `a(A)` with `A`? What behaviour is desirable in this case?

Zebra Puzzle

(2.1) Shallow Complete Part 1 and 2 of the Zebra task on Chime.

Rules

(3.1) Bookwork Build a glossary of the important Prolog terminology so far

(3.2) Shallow What are the FOL formula for the `valuable` rules in the slides? How would you join the rules together in FOL?

(3.3) Deeper Complete Part 3 of the Zebra task on Chime: use rules to rewrite the puzzle. How short can you make it without adding more terms to the query?

Lists

(4.1) Shallow Consider the implementation of append given below. Explain how it should be used and draw out a search tree for a representative example.

append([],A,A).
append([H|T],A,[H|R]) :- **append**(T,A,R).

(4.2) Deep You are given two implementations of member

member(H,[H|_]).
member(H,[_|T]) :- **member**(H,T).

% or, alternatively

member(X,Y) :- **append**(_,[X|_],Y).

1. Discuss how the second implementation compares with the use of 'partial application' from functional programming, and how it differs.

2. If the two implementations are equivalent, should a programmer simply inline all calls to member with a call to append?

3. If the two implementations are equivalent, what advantages does one form of expression have over the other?

4. Prove (informally) why the two are logically equivalent.

(4.3) Deeper What it the purpose of the following clauses:

a([]).
a([H|T]) :- **a**(T,H).
a([],_).
a([H|T],Prev) :- H >= Prev, **a**(T,H).

(4.4) Deeper What does the following do and how does it work?:

b(X,X) :- **a**(X).
b(X,Y) :- **append**(A,[H1,H2|B],X), H1 > H2, **append**(A,[H2,H1|B],X1), **b**(X1,Y).

Arithmetic

(5.1) Shallow Complete the Last Call Optimisation task on Chime.

(5.2) Deeper Complete the Implementing Arithmetic task on Chime.

(5.3) Open If we ask Prolog to solve algebra then the `is` operator throws an error. For example the query `3 is A+2` fails. Give an example of how this is not the case with our `prim` and `plus` rules and explain what the difference is.

Backtracking

These questions ask you to consider the use of predicates 'backwards' - i.e. swapping the inputs and outputs. Its easy to find out what they do by just trying it in the interpreter. However, the real question is to see whether you can reason it out from your knowledge of how the search and backtracking algorithms work.

(6.1) Shallow What happens if the accumulator version of len is used 'backwards'?

(6.2) Shallow What happens if take is used 'backwards'?

(6.3) Shallow What happens if append is used 'backwards'?

Generate and Test

(7.1) Shallow Draw the Prolog search tree for perm([1,2],A).

(7.2) Shallow Complete Part 1 of the Dutch National Flag task on Chime.

(7.3) Deeper Complete the 8-Queens task on Chime.

(7.4) Deeper Complete the Anagram task on Chime.

(7.5) Open In what situations is it more efficient to Test-and-Generate rather than Generate-and-Test?

Supervision 2

Negation

(1.1) Shallow State and explain Prolog's response to the following queries:

```
X=1.
```

```
not(X=1).
not(not(X=1)).

not(not(not(X=1))).
```

In those cases where Prolog says 'yes' your answer should include the unified result for X.

Databases

(2.1) Deeper Complete the Databases task on Chime.

Countdown

(3.1) Deeper Complete the Countdown task on Chime.

Graph search

(4.1) Shallow Complete the Missionaries and Cannibals task on Chime.

(4.2) Deeper Complete Part 1 of the Towers of Hanoi task on Chime.

(4.3) Deeper Complet ethe Umbrella task on Chime

Difference lists

(5.1) Deeper Complete the Quicksort task on Chime.

(5.2) Deeper Complete Part 2 of the Towers on Hanoi task on Chime.

(5.3) Deeper Complete Parts 2 and 3 of the Dutch National Flag task on Chime.

Constraints

(6.1) Shallow Complete the Sudoku task on Chime.

(6.2) Deeper Complete the Cryptarithmetic task on Chime.

Extra-fun

The findall predicate is an extra-logical predicate for backtracking and collecting the results into a list. The implementation within Prolog is something along the lines of:

findall(Template,Goal,Solutions) :- **call**(Goal), **assertz**(**findallsol**(Template)), **fail**.
findall(Template,Goal,Solutions) :- **collect**(Solutions).

collect([Template|RestSols]) :- **retract**(**findallsol**(Template)), !, **collect**(RestSols).
collect([]).

(7.1) Deeper Consult the Prolog documentation and work out what the above is doing. The predicate assertz adds a new clause to the end of the running Prolog program and the predicate retract removes a clause which unifies with its argument.