School of Computer Science, University of St Andrews, CS3050, 2022-23

CS3050: Practical 2, Programming with Prolog

Assignment: P2 - Practical 2

Deadline: 25th November 2022

Weighting: 45% of coursework component

Please note that MMS is the definitive source for deadline and credit details. You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturer regarding any queries well in advance of the deadline.

---

1 Objective

This practical is aimed at designing and implementing automated reasoning solutions.

1.1 Competencies

- Develop experience in programming in Prolog
- Develop experience in modelling a logic problem
- Develop experience in using an automated reasoning engine

2 Practical Requirements

You are required to implement the following exercises using GNU Prolog[1]. Please see Section 3.1 for information on how to structure the submission.
Additional notes:

- Each exercise specifies a problem and asks you to write a set of clauses with a specific head, please adhere to these instructions.
- Please also note you are free to add any additional subclause to achieve the requirements, and you can reuse already written clauses as subclauses for other requirements.
- Each solution should be implemented in a different file called respectively **‘ex1.pl’,‘ex2.pl’, ‘ex3.pl’,** etc.
- Two queries are expected to be written per required clause. These queries should be placed by completing a correspondent test file **‘ex1test.pl, ‘ex2test.pl’,‘ex3test.pl’,** etc. See instructions in Section 3.2 on how to prepare these files.

[1] GNU Prolog, http://www.gprolog.org, also available on the lab machines.

2.1 Exercise 1 - Software Requirements Audit Trails

We are tasked to design a system that holds information regarding software development, from objectives, requirements, development of components, to functionalities, so that it is easier for a software engineer to query requirements and dependencies when developing a piece of software. The system will facilitate analysis of audit trails and should be able to respond to explanations as to why a specific functionality was developed and what objectives it serves. You are required to model a simplified version of the problem as follows:

The development of a piece of software is motivated by a set of objectives. Each objective can be broken down into software requirements, where each requirement may depend on one or more objectives. Each requirement may be addressed by a number of components. Each component may depend on one or more requirements. Components can be atomic (a single component) or may be compound, which means that a component may have one or more subcomponents, which in turn may be atomic or compound. No component A can be a subcomponent of a component B, if B is already a subcomponent of A; in other words there are no cycles in the component relationships. A single functionality depends on a single atomic component in the system. Each functionality has also an associated priority number, which is a positive integer where the lower values indicate the higher priority for development (eg. 1 has absolute high priority, 100 has low priority).

For example, assume the following objectives, requirements, components and functionalities:

- Objective `obj`: develop a calculator app
- Requirement `rq`: the user must be able to enter the formula to be calculated through an interface
- Component `cp1`: calculator interface
- Component `cp2`: plus button
- Component `cp3`: addition calculator
- Functionality `ft1`: visualise a plus on the interface when pressed, with priority 1
- Functionality `ft2`: calculates the result of the addition, with priority 2

With the above definitions, `rq` is a requirement depending on `obj`, which in turn is fulfilled by (at least) two components: a compound component `cp1`, and an atomic component `cp3`. `cp2` is a subcomponent of `cp1`. The functionalities `ft1` and `ft2` depend on the atomic components `cp2` and `cp3` respectively.

For this task you are required to:

A. Provide sufficient facts to illustrate elements and relationships in this scenario
B. Complete the following Prolog clauses to fulfil each of following requirements
C. Provide two queries per requirement to demonstrate their usage

Clauses:

1. `haveSameObj(R1,R2):-...` checks whether two *different* requirements R1 and R2 have been formulated for the same objective.

2. `belongsTo(A,B):-...` checks whether the relationship where A depends on B is true, where each variable A,B can be either an objective, a requirement, a component or a functionality.

3. `shareDevelop(F1,F2):-...` checks whether two functionalities F1,F2 have a common objective, a common component or a common requirement.

4. `printAllCom(O):-...` prints all components fulfilling an objective O, one per line, regardless of whether they are compound or atomic. It is ok to print duplicates.

5. `isCompound(C):-...` determines whether a component C is compound.

6. `printAllAtomic(R):-...` prints all atomic components that are dependent on a requirement R.

7. `hasPriorityCom(C1,C2):-...` checks whether an atomic component C1 has higher priority over another atomic component C2 on the basis of the functionalities they implement.

8. `hasHigherPriority(F,N):-...` checks whether a functionality F has higher or equal priority than a given number N.

You are free to use hypothetical values for this exercise (e.g. `obj`,`rq`, ...).

2.2 Exercise 2 - Logic Trees & Truth Values

In this exercise, we intend to develop an automated truth calculator for propositional logic that helps compute a truth value of a propositional formula. We assume that our input formula only contains operators $\wedge, \vee, \rightarrow, \neg$ which we refer to with the following constants `and, or, imp, not` respectively. Our system assumes that a formula has already been parsed, and a binary tree representing it has been stored in the knowledge base. In particular, nodes of the tree are facts with the structure `node(ID,P,LC,RC)` representing the following information: ID identifies the node, e.g. i_x; P is an element representing either a i) proposition if the node is a leaf or ii) an operator $\wedge, \vee, \rightarrow, \neg$ if this is an internal node; LC and RC are pointers to its two children, e.g., i_l,i_r. Note that 'void' can be used if the child of the node is not present, and for operator $\neg$ we assume that the left-hand side child is void.

An example tree is provided in Figure 1 representing the formula $((R \vee Q) \wedge (\neg P))$.

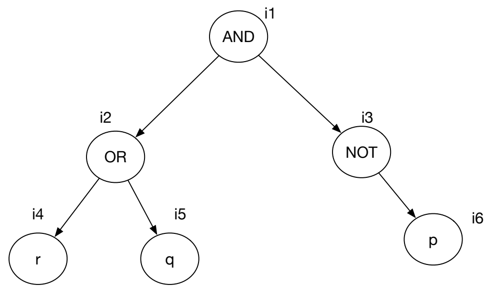

Figure 1: Tree of Formula $((R \vee Q) \wedge (\neg P))$

For this task you are required to:

A. Provide sufficient facts to demonstrate functioning of the queries below

B. Complete the following Prolog clauses to fulfil each of following requirements

C. Provide two queries per requirement to demonstrate their usage

Clauses:

1. `validFormula(ID):-...` checks whether the formula associated to the tree or subtree given by node ID has correct syntax. In other words, this clause should check two conditions: whether the tree is connected and whether it terminates with appropriate leaves. We assume that all nodes have one or two children at most.

2. `printFormula(ID):-...` prints the formula associated to the tree or subtree given a node ID. For example querying `printFormula(i1).` should return *and(or(r,q),not(p))* but no text should be printed if the formula has incorrect syntax.

Assume we introduce a new set of facts of type `truth(P,V)` indicating a truth value for a proposition, where P is the proposition, and V is a constant among two options `true` or `false`. For example if we would like to assign to the three propositions in the example a truth value we could add facts `truth(r,true),truth(q,true),truth(p,false)`. With this assignment, write the following clauses:

3. `calculateValues(ID):-...` a clause that creates a new truth fact for a formula associated to the tree given by node ID as well as one for each of the subformulas. This can be done by applying the appropriate operators for propositional logic as shown in the lectures. For example, querying `calculateValues(ID).` should create a fact `truth(i1,true)` and additional facts for all the node ids in this formula.

4. `isValue(ID,V):-...` a clause that checks whether the truth value of a formula associated to the tree given by node ID is equal to V (`true/false`), by using facts generated in the previous step. For example, querying `isValue(i1,true).` should respond yes in the example above.

For this exercise, you may need to assert facts dynamically, see lectures on how to use `asserta(_)`. If you run two queries sequentially on different trees or assignments, please make sure that tree ids are different, or you can use `retractall(_)` to clear older calculations. See GNU prolog manual for further information on this instruction[2].

2.3 Exercise 3 - Plan Reachability

A plan is a sequence of steps starting from an initial situation and ending in a different situation or state of the world. Here we may represent a plan as a sequence of states, where each state represents the result of executing an action in the previous state. For example, a plan created by a route planner will start from the departure place, and will indicate the list of places encountered to arrive to a destination place. A reachability problem is that of determining whether a goal state of a system is reachable from the initial state of the system.

Inspired by a famous planning problem, *blocks world*[3], in this exercise we will use a simple block stacking problem. Assume we have four blocks (e.g. `a,b,c,d`) which can be stacked on top of each other or placed on top of a table. A predicate `on(A,B,S)` indicates that a block A is on top of block B at state X, where X is a positive integer number. The initial configuration at state 1 is represented by the initial facts of type `on(A,B,1)`. When a block B has no other blocks on top — i.e. is clear — we indicate this with a constant `none`, e.g. `on(none,B,1)` and when a block A is on the table we also use a constant `table`, e,g. `on(A,table,1)`. You may also provide a list of available objects (four blocks and table) as part of the initial facts.

There are two actions that can be executed in this environment:

[2] `http://www.gprolog.org/manual/html_node/gprolog031.html#sec105`
[3] `https://en.wikipedia.org/wiki/Blocks_world`

- PutOn where a block A can be put on top of a block B. The action is possible in a specific state only if block A is on the table, and both blocks A and B are clear.
- PutBack where a block A can be put back on the table. The action is possible in a specific state only if block A is clear.

In this exercise, we are tasked to write clauses that check different validity and reachability conditions of a plan. In particular, you are required to:

A. Provide sufficient facts to demonstrate functioning of the queries below

B. Complete the following Prolog clauses to fulfil each of following requirements

C. Provide two queries per requirement to demonstrate their usage

Clauses:

- `valid(A):-...` determines whether the configuration for a block A at the initial state 1 is a valid configuration, such that it is placed on top of something, and it either has a block on top or it is clear.
- `isOnPossible(A,B,SX):-...` this clause implements a partial reachability problem. It checks whether a specific configuration, where A is on B, can be reached at state SX from state 1. One way to do so is to work back from the end state SX to the initial state, and check whether it is possible for a block A to be placed on another block B, or on the table at step SX, by only looking at the action conditions in the previous state SX-1. For example, block A can be on block B at SX, if the conditions for a PutOn action at SX-1 are satisfied. Block A can be on the table at SX, if the conditions for a PutBack action at SX-1 are satisfied. We also need to say that A can be on B at SX, if A is already on B at SX-1. Finally we must consider whether it is possible for B to be clear (have none of the blocks on top) at SX.
- `printAllPossible(SX):-...` for any two available blocks (or table) A and B, this clause checks whether it is possible to have A on B at state SX, and if so it prints out 'A on B', one possible option per line.

Consider this example for query 2: assume that the initial configuration is the one in Figure 2 at state 1 (asserted with six initial facts of type `on(A,B,1)`). We want to test whether block `a` can be on top of `d` as in the configuration in state 3. By querying the system with `isOnPossible(a,d,3)` we expect the system to answer yes, since `a` both blocks can be clear in state 2, and `a` can be on the table in state 2 (for PutOn) because `a` is clear in state 1 (for PutBack). However, if we had queried `isOnPossible(a,d,2)`, the system should answer no.

2.4 Additional Functionalities

It is strongly recommended that you ensure you have completed the Requirements in Exercise 1, 2, and 3 before attempting any of these requirements. You may consider at most two additional requirements for this assignment. Please note that the focus will be on the quality of your extension(s) rather than on the volume. Some suggestions are presented here, but you may also present your own requirements. Additional requirements should aim to demonstrate additional interesting uses of Prolog to extend the functionalities of the systems provided in the Exercises above. Please describe well the purpose of your additional requirements in the report and place your additional requirements in separate files, (**ex4.pl** or **ex5.pl**) and correspondent query files (**ex4test.pl** or **ex5test.pl**).

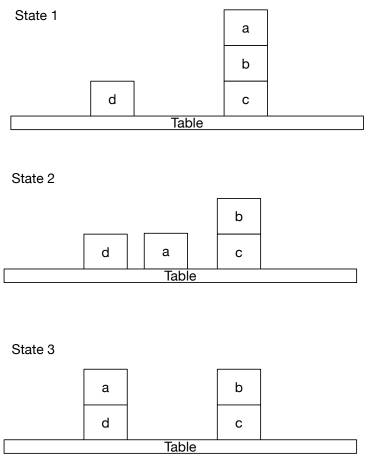

Figure 2: An example of block configurations, state 1 is the initial configuration, state 3 is the queried state

.

- Design an interactive system to examine the audit trail of the requirement formulation on the basis of Exercise 1, taking into consideration the developer preferences. For example consider requirements already executed or different types of requirements (functional, non functional), using a priority system for deciding the next functionality to implement, etc.
- Explore the use of lists and apply those to Exercise 2 or 3 to make the system more flexible or the code more compact.
- Extend Exercise 2 to perform equivalence transformations as seen in the lectures, and check whether the truth values are indeed equivalent.
- Extend Exercise 3 with additional actions such as using a robot which navigates to pick up a block.

3 Code and Report Requirements

3.1 Code and Libraries

Please develop your programs using GNU Prolog. The solution for each exercise should be saved on an appropriate file **'ex?.pl'**. These files should be placed in a folder called "src". Starter code containing the head of the clauses to be completed is provided. For each exercise complete the testing file with your queries in a separate and appropriately named file **'ex?test.pl'**. Your queries will be used to test your code, but please do not change the clause heads given, as we will also perform additional automated tests to check the basic functionalities of your programs. The submission should only make use of standard Prolog instructions and GNU Prolog built-in predicates.

3.2 Test files

Each test file **'ex?test.pl'** must be loaded in Prolog together with their correspondent knowledge base **'ex?.pl'** and is used to simulate external queries. Please do not delete the initial part of these files, but you are free to add more if you wish.

For each exercise, complete the test cases at the end of the file by uncommenting the correspondent test case. This is indicated by a sequence of three numbers ⟨ *exercise number,* *clause number, query number* ⟩. For example to test the second query for clause 3 of exercise 1, uncomment and complete the clause with goal `test123`.

Each clause must be completed by placing it within the appropriate test predicate depending on your expected output. For example if you expect a query Q to return `yes` please use predicate `test_yes(Q)`, if you expect to return `no` please use predicate `test_no(Q)`.

If you have more than one goal required to be called per query, please combine them in the body of the test case, eg `test123:-test_yes(Q1),test_no(Q2).` or `test123:-Q1,test_no(Q2).` if Q1 is an auxiliary goal.

The full list of test queries can then be called with a single goal, for example `?-testex1.` for exercise 1.

3.3 Report

You are required to submit a report describing your submission. The report should include:

- A brief checklist of the functionalities implemented
- If any of the functionalities is only partially working, ensure that this is discussed in your report
- Design: A description and justification for the functionalities implemented as part of your solutions
- Examples and Testing: Selected examples or queries of the functionalities implemented. Please do not include all the queries in the report or their results as they are available in the submission, use this part of the report to highlight any interesting characteristic of your design, or any key example that demonstrates your achievements.
- Evaluation and Conclusions: A critical evaluation of your solutions and what can be improved
- Please cite any source consulted.

The suggested length for the report is between 1000–2000 words.

4 Deliverables

You should submit a ZIP file via MMS by the deadline containing:

- A PDF report as discussed in Section 3.3.
- Your implementation of the exercises (see Section 3.1).

5 Assessment Criteria

Marking will follow the guidelines given in the school student handbook (see link in the next section). The following issues will be considered:

- Achieved requirements
- Quality of the code and the solution provided
- Insights demonstrated in the report

Some guideline descriptors for this assignment are given below:

8-10	The submission implements fully the functionalities of exercise 1 with appropriate queries, adequately documented or reported.
11-13	The submission implements fully the functionalities of exercise 1 & some attempt to exercise 2 with appropriate queries. The code submitted is of an acceptable standard, and the report describes clearly what was done, with good style.
14-16	The submission implements fully the functionalities of exercise 1, and exercise 2 with appropriate queries. The code submitted is clear and well-structured, and the report is clear showing a good level of understanding. For the upper part of this band some attempt to exercise 3 should be provided.
17-18	The submission implements fully the functionalities of exercise 1, exercise 2 and exercise 3 with appropriate queries. It contains clear, well-designed code, together with a clear, insightful and well-written report, showing in-depth understanding of the solutions presented.
18 or above	The submission implements fully the functionalities of exercise 1, exercise 2 and exercise 3, with one or two (at most) additional functionalities completed and appropriate queries included. The submission demonstrates unusual clarity of design and implementation, together with an outstandingly well-written report showing evidence of extensive background reading, a full knowledge of the subject and insight into the problem.

6 Policies and Guidelines

Marking: See the standard mark descriptors in the School Student Handbook

`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors`

Lateness Penalty: The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):

`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

Good Academic Practice: The University policy on Good Academic Practice applies:

`https://www.st-andrews.ac.uk/students/rules/academicpractice/`

Alice Toniolo

cs3050.staff@st-andrews.ac.uk

November 11, 2022