P2: Programming with Prolog

Overview

In this practical I had to develop and demonstrate experience in using an automated reasoning engine such as Prolog, which in turn allowed me to model logic problems and come up with solutions for them. This practical is split into three exercises out of which I was able to complete the first two exercises and had an attempt at doing exercise three.

Functionality

Below are the functionalities that I was able to implement for each exercise.

Exercise 1:

- Clause haveSameObj(R1, R2) which checks whether two different requirements are part of the same objective.
- Clauses belongsTo(A, B) which whether A depends on B and that A and B can either be an objective, a requirement, a component or a functionality.
- Clause shareDevelop(F1, F2) which checks whether two different functionalities share a common objective, requirement or component.
- Clause printAllComp(C) which prints all the components for a given objective regardless of whether it is an atomic or a compound component.
- Clause isCompoound(C) which checks if a component is a compound component and has other components depending on it.
- Clause printAllAtomic(R) which prints all the atomic components that depend on a given requirement.
- Clause hasPriorityCom(C2, C2) which checks whether the functionality that depends on C2 has a higher priority than the functionality that depends on C2.
- Clause hasHigherPriority(P, N) checks whether the priority functionality F is higher or equal to priority in number N.

Exercise 2:

- Clause validFormula(ID) which checks whether the formula given by the ID is has the correct syntax.
- Clause printFormula(ID) prints the formula given by the ID.
- Clause calculateValues(ID) which calculates the value of each node in the tree and the tree itself by creating truth values and assigning it to their IDs.
- isValue(ID, V) which checks whether the given truth value matches the truth value of the node given by its ID.

Exercise 3:

- Clause valid(A) which checks whether the configuration of the given block in the initial state is valid.

There are a few clauses which have only been implemented partially, and these are the clauses: shareDevelop(F1, F2), validFormula(), calculateValues(ID) and isValue(ID, V). This will be discussed in more detail in the evaluation and conclusion section.

Design

Exercise 1

The clause haveSamebject() simply calls the clause belongsTo() for each of the functionalities passed into it. Then it extracts the object for each of the requirement that it depends on and checks to see if they both are equal. Clause belongsTo() has three separate types of facts, one for each type of relationship, and checks to see if there are any facts that satisfy a relationship between the two give elements. For example, the fact req_depends_on_comp(A, B) checks to see if there is a requirement A that depends on component B. The shareDevelop() clause calls the shareElement() clause which does recursion. It goes down the hypothetical tree for both of the elements given and checks to see if the element that they depend on matches. Clause printAllCom() simply goes down one level to the component that handles the given requirement and then prints it. If this component is a compound component, recursion is used in the same way as the clause shareDevelop(). This is done using the printDependentComponents() clause and then the above steps are simply repeated until there aren't any compound components left. At the end of the clause printAllComp() there is a false statement which is used so that backtracking is done and other components that are under the given requirement are also printed. The clause isCompound() simply uses the fact compound() to see if the given component satisfies it. Clause printAllAtomic() works in the same way as the clause printAllComp() except a condition is included before printing a component. This is done so that the print is only made if the component is atomic which is checked using the atomic() fact. The clause hasPriorityComp() obtains the functionality for both the components and then compares them to see if the first requirement is greater than or equal to the second requirement in terms of their priority. The priority for each functionality is retrieved using the fact hasPriority(). Lastly, the clause hasHigherPriority() gets the priority number for the given functionality and compares it with the value given

Exercise 2

For the clause validFormula(), recursion is used to traverse the truth tree. This is done using a recursive clause when an inner node is visited, a leaf clause when a leaf node is visited and then a clause for when the node is not. At the end of the leaf and not clause, there is a

check that is done to see if it is a valid proposition, which is what determines the validity. Clause printFormula() uses similar recursive clauses as the first one, except there are write() statements placed at appropriate places so that the output is as required. The clause calculateValue() also uses similar recursive clauses, except at the beginning of the main recursive clause, a check is done to see if the current node has propositions as its children which can be calculated. If so, this calculation is done and asserta() is used to create a fact for the given node. This is done for the left and right node from the current parent node. Then at the end, for each of the child nodes of each of the parent nodes, their truth values are obtained and the appropriate Boolean operation is done to get the final result. The program goes through the usual execution path which is recursive, and at the end the final value is automatically calculated. The final clause isValue() is implemented by access the asserted fact from the previous clause and then checking to see if the truth value in the matches the truth value given for a particular node.

Exercise 3

The clause valid() uses the given object and then checks which on() fact has that object in it. Here, there are two cases which need to be checked for validity. First is when the given object is on something. This something needs to be either a table or another object which is checked using a simple conditional. Second is when the given object has something on it or nothing. This something can only be another object, so this is also checked.

Testing

Tests for exercise 1:

- Test for shareDevelop():

```
?- shareDevelop(func1, func2).
true
```

Tests for exercise 2:

- Test for validFormula():

```
?- validFormula(1).
true
```

- Test for calculateFormula():

```
?- calculateValues(1).
true .
```

- Test for isValue():

```
?- isValue(1, true).
true.
```

Tests for exercise 3:

- Test for valid():

```
[debug]  ?- valid(a).
true.
```

Evaluation and Conclusion

My implementation has three main clauses which are limited in its functionality as stated above. The first is the shareDevelop(F1, F2). For this clause there is an assumption that is made, which is that the length of the path from the root in the hypothetical tree for both given functionalities are the same. So it doesn't deal with situation where for example one functionality is attached to one component which in turn is attached to another component, but when the other functionality is attached to another component which doesn't have anymore components attached to it. The other clause which has limitations is validFormula(ID). Here, my solution is able to identify whether a formula is invalid by checking if a lead node is incorrect. But this doesn't cause false to be returned. So a valid formula will be returned to and so will an invalid formula. In addition, the clause calculateValues() only calculates for a limited number of formulas. It doesn't work well with cases such as 'Not(A And B)'. Lastly, The isValue() clause is limited mainly due to that fact that it isn't implemented properly and so is bugged. Another minor limitation is in the printFormula() clause. When a sub-tree is passed to be printed, the brackets are not printed exactly as they should.