Name: **Jamie Grech**
ID: **123499M**
Course: **B.Sc. Information Technology (Hons.) (Artificial Intelligence)**

Faculty of Information and Communication Technology
Department of Artificial Intelligence

Study-unit: **Knowledge Representation and Reasoning**
Code: **ICS1019**
Lecturer: **Dr. Joel Azzopardi**

Contents

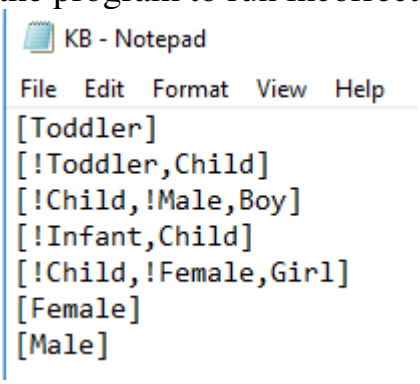
Part 1: Parsing of Horn Clauses, and Reasoning using Back-Chaining

Running the Program

1. Open your command prompt
2. Insert the following command: java -jar “mydirectory” (replace mydirectory with the directory where your Part1.jar file is stored, as shown in the screenshot below)

```
C:\Users\Jamie>java -jar "C:\Users\Jamie\Desktop\Uni\ICS1019 - Knowledge Representation and Reasoning\Part1\dist\Part1.jar"
Please insert a valid text file address to read from
```

3. Insert the address of the text file containing the input clauses for your knowledge base. The file should be formatted as shown in the screenshot below, using an exclamation mark to represent negative clauses and with no spaces in the file and no empty lines. Input not following this format is likely to cause the program to run incorrectly.



4. Insert your query simply as a positive string, i.e. no negation. The program will negate it automatically. The program will return SOLVED if it has managed to resolve the query and NOT SOLVED otherwise.

```
Please insert a valid text file address to read from
C:\Users\Jamie\Desktop\Uni\kb.txt
Insert your query (will be automatically negated)
Boy
Solved

Please insert a valid text file address to read from
C:\Users\Jamie\Desktop\Uni\kb.txt
Insert your query (will be automatically negated)
Infant
Not Solved
```

Structure of the Program

The program saves every literal as an object of type ‘Atom’. This type consists of a String representing the value of the literal, ex. ‘Toddler’, and two Boolean flags. The first Boolean, ‘negative’, is used to display the Atom’s polarity, i.e. true for negative literals and false for positive literals. The second Boolean, resolved, marks true if the Atom has already been resolved and false otherwise.

Each input clause is then saved as linked list of these Atoms using the StringTokenizer and BufferedReader classes to read and process the input, and finally these linked lists are all saved in one array of type LinkedList<Atom>, which represents our knowledge base. The size of the array is determined by a method countLines which returns the amount of lines in the inputted text file.

The user’s query is resolved as follows. Firstly, the program checks the size of the knowledge base. If the knowledge base is empty, the query is automatically marked as solved. Otherwise, the program goes through the knowledge base to find clauses with a positive literal matching the user’s negated query. If no such clause is found, the query is marked as not solved. If a clause containing only a positive literal matching the user’s query is found, the query is marked as solved. Otherwise, the program recursively calls itself for each negative literal in the clause until either all of them are solved or one is not solved. If all of them are solved, the query is marked as solved. If one of the negative literals is not solved, the program then refers to the next clause with a positive literal matching the query and repeats the process of checking its negative literals. If the program goes through every valid clause without resolving any of them, the query is marked as not solved.

Satisfaction of Goals

Goal	None	Partial	Full
Construction of knowledge base according to user inputs			X
Resolution by backchaining			X

It has been examined through testing that all goals requested for this part of the assignment have been satisfied.

Part 2: Construction and Reasoning with Inheritance Networks

Running the program

1. Open your command prompt
2. Insert the following command: java -jar “mydirectory” (replace mydirectory with the directory where your Part2.jar file is stored, as shown in the screenshot below)

```
C:\Users\Jamie>java -jar "C:\Users\Jamie\Desktop\Uni\ICS1019 - Knowledge Representation and Reasoning\Part2\dist\Part2.jar"
Please insert a valid text file address to read from
```

3. Insert the address of the text file containing the input clauses for your knowledge base. It should be in the following format, with no spaces in the subconcepts and superconcepts. The program assumes each line has 3 words divided by spaces and if the middle word is not “IS-A”, the program automatically assumes it is IS-NOT-A.

IN - Notepad

File Edit Format View Help

```
Clyde IS-A FatRoyalElephant
FatRoyalElephant IS-A RoyalElephant
Clyde IS-A Elephant
RoyalElephant IS-A Elephant
RoyalElephant IS-NOT-A Gray
Elephant IS-A Gray
```

4. Insert the subconcept followed by the superconcept of the relationship you wish to query the database with. This relationship is automatically treated as an IS-A relationship.

```
Please insert a valid text file address to read from
C:\Users\Jamie\Desktop\Uni\in.txt
Insert query subconcept
RoyalElephant
Insert query superconcept
Gray
Printing all paths:
RoyalElephant IS-A Elephant IS-A Gray
RoyalElephant IS-NOT-A Gray
Shortest path: RoyalElephant IS-NOT-A Gray
```

Structure of the program

Each clause inputted to the program is stored as an object of type ‘relationship’ which consists of two Strings, one for the subconcept and one for the superconcept, and one Boolean for the polarity, which is true for IS-A and false otherwise. These relationships are all stored in an array of type relationship. The input is read and processed using BufferedReader and StringTokenizer, similarly to what is done in part 1.

The program uses a recursive method to find possible paths for a given query. The method first checks if the current path is empty. If so, it goes through the knowledge base and checks every element whose subconcept is the same as the subconcept in the query. If the superconcept also matches the superconcept, it is converted to a string and added to the list of possible paths. Otherwise, the relationship is added as the first element to a linked list representing the current path and the method is called recursively with this new path. It goes through the knowledge base again, but this time since the path is no longer empty, it first checks if the element being examined is already in the current path. If it isn’t, it compares the element’s subconcept to the superconcept of the last element in the current path to check for a match. If these conditions are both satisfied, the new relationship is added to the end of the path. If the superconcept of the new relationship matches that of the given query, it is added to the list of possible paths. If the superconcepts do not match and the polarity of the newly added relationship is negative, that path is abandoned. Otherwise, the method recursively calls itself again.

Satisfaction of Goals

Goal	None	Partial	Full
Construction of inheritance network according to user inputs			X
Output of possible paths according to user queries		X	
Determination of shortest path			X
Determination of preferred path according to inferential distance	X		

While for some queries all possible paths are successfully displayed, it has been observed in testing that some queries return only the first path found. Despite rigorous attempts at debugging, the cause of this was not found. Searching for the preferred path in terms of inferential difference was also not achieved due to repeatedly encountering problems when attempting to remove redundant paths.