Part B

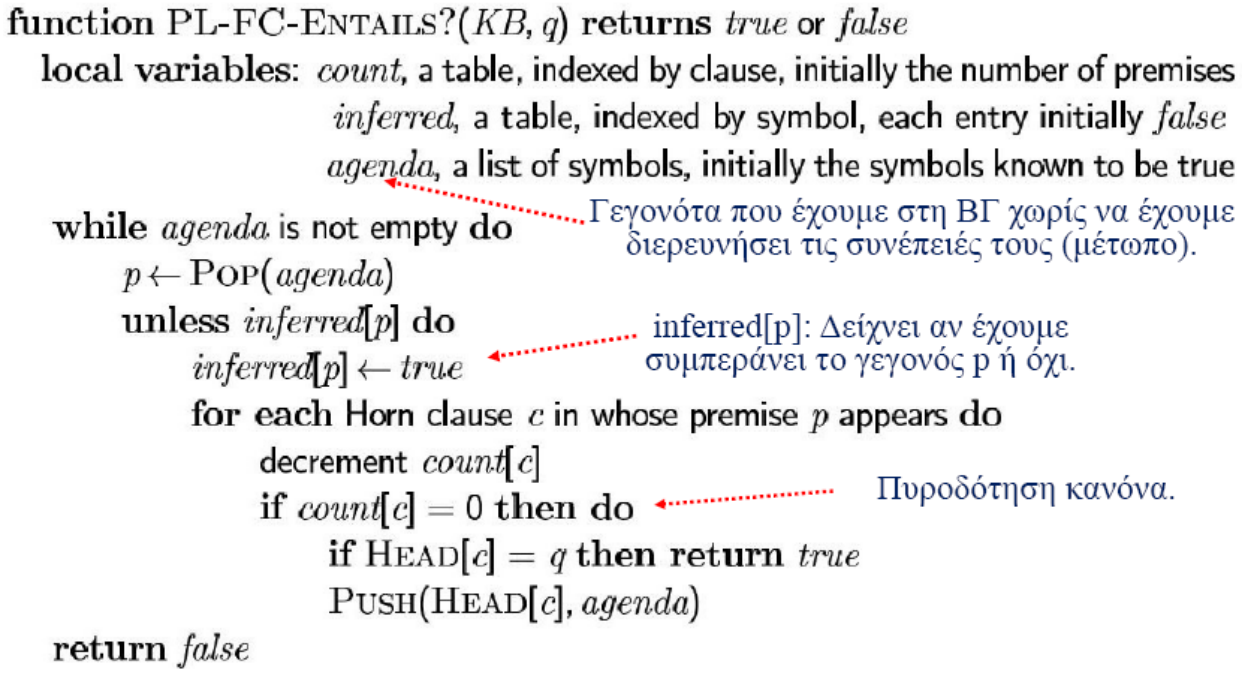
The following inference methods were selected:

1. **Forward chaining** for **Horn clauses** (more precisely, definite clauses) in **propositional logic**.
2. **Forward chaining** for **Horn clauses** (more precisely, definite clauses) in **first-order predicate logic**.

Propositional Logic Forward Chaining.

Program Basis

The program was implemented based on the PL_FC_Entails algorithm from Lecture 9 of our course.



Implementation File

The algorithm is implemented in the file PL_FC_Entails.java

Helper classes

ImplicationForm.java

ReadHornFromTxt.java

Implementation Analysis

In the section below, collections are created and initialized with the correct values.

After reading from the file, the knowledge base (KB) is converted into **Implication** form. Due to the use of definite clauses and the simplicity of the specific problem, only String types were used for the **premises** and **conclusions** of each implication.

Any implication with currentPremise size > 0 (line 25) is considered a **rule**; otherwise, it is a **fact** (line 34).

```
13      public static boolean doesPL_FC_Entails(Set<ImplicationForm> KB, String q)
14      {
15          Map<ImplicationForm, Integer> count = new HashMap<>();
16          Map<String, Boolean> inferred = new HashMap<>();
17          Queue<String> agenda = new LinkedList<>();
18
19          // Initializate Collections
20          for (ImplicationForm implication : KB)
21          {
22              List<String> currentPremise = implication.getPremise();
23              String conclusion = implication.getConclusion();
24
25              if (currentPremise.size() > 0)
26              {
27                  count.put(implication, currentPremise.size());
28
29                  for (String symbol : currentPremise)
30                  {
31                      inferred.put(symbol, value:false);
32                  }
33              }
34              else // symbols known to be true because of zero premise symbols
35              {
36                  agenda.add(conclusion);
37              }
38
39              inferred.put(conclusion, value:false);
40
41          }
```

In the main part, an element is extracted from the **agenda** collection, compared to the given element and it returns true if they are the same (lines 46-48).

Otherwise, the code continues and checks the boolean value of that element (line 52).

If the element hasn't already been inferred, then:

1. It is inserted into the corresponding collection (line 54)
2. It decreases the **count** for all rules that include this element as a premise (lines 61-64)
3. If a rule's count becomes zero after this decrease, then its **conclusion**, since it is now a new fact, is compared with the query q; if it's not the same, it is inserted into the **agenda** collection.

The code terminates if the **agenda** is emptied and no true value was returned during the comparisons in lines 47-48, 69-72.

```
42          // main part
43          while (!agenda.isEmpty())
44          {
45
46              String p = agenda.remove();
47              if (p.equals(q))
48                  return true;
49
50              Boolean isPInferred = inferred.get(p);
51
52              if (!isPInferred)
53              {
54                  inferred.put(p, value:true);
55
56                  for (Map.Entry<ImplicationForm, Integer> countElement : count.entrySet())
57                  {
58                      int numberOfPremises = countElement.getValue();
59
60                      // if the symbol is in the premise of current implication
61                      if (countElement.getKey().getPremise().contains(p))
62                      {
63                          count.put(countElement.getKey(), numberOfPremises - 1);
64                      }
65
66                      if (countElement.getValue().equals(0))
67                      {
68                          String conclusion = countElement.getKey().getConclusion();
69                          if (conclusion.equals(q))
70                          {
71                              return true;
72                          }
73                          agenda.add(conclusion);
74                      }
75                  }
76              }
77          }
78          return false;
```

Results with the given KB from the slides

KB

(¬PVQ)

(¬LV¬MVP)

(¬BV¬LVM)

(¬AV¬PVL)

(¬AV¬BVL)

(A)

(B)

Given Queries and Results

```
Enter the Symbol (accepted format:e.g Q):
A
True
```

```
Enter the Symbol (accepted format:e.g Q):
B
True
```

```
Enter the Symbol (accepted format:e.g Q):
Q
True
```

```
Enter the Symbol (accepted format:e.g Q):
C
False
```

ForwardChaining_FirstOrder

Program Basis

The program was implemented based on the fol-fc-ask algorithm from Lecture 12 of the course.

συνάρτηση fol-fc-ask($B\Gamma$, α) **επιστρέφει** ενοποιητή ή αποτυχία
είσοδοι: $B\Gamma$: η βάση γνώσης, σύνολο από **οριστικές προτάσεις ΠΚΛ**
α: η ερώτηση, ατομικός τύπος ΠΚΛ
βρόχος
  $νέο \leftarrow \{\}$
  για κάθε τύπο $\tau \in B\Gamma$ με μορφή
    $((\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n) \Rightarrow \beta) \leftarrow$ new-vars(τ)
    για κάθε θ με unify($\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n$, $\alpha_1' \wedge \alpha_2' \wedge ... \wedge \alpha_n'$) $= \theta \neq$ αποτυχία
    όπου $\alpha_1', \alpha_2', ..., \alpha_n' \in B\Gamma$
      $\beta' \leftarrow$ subst(θ, β)
      αν το β' δεν είναι «αντίγραφο» τύπου της $B\Gamma$ ή του $νέο$ **τότε**
        $νέο \leftarrow νέο \cup \{\beta'\}$
        $\theta' \leftarrow$ unify(β', α)
        αν το θ' δεν είναι αποτυχία **τότε επίστρεψε** θ'
  $B\Gamma \leftarrow B\Gamma \cup νέο$
μέχρι το $νέο$ να είναι κενό
επίστρεψε αποτυχία

Κάθε φορά νέες μεταβλητές. Π.χ. (Missile(x_1) ⇒ Weapon(x_1)), (Missile(x_2) ⇒ Weapon(x_2)), ...

Αλλάζουμε και στα α_i' μεταβλητές.

Π.χ. το Likes(x, Mary) είναι «αντίγραφο» του Likes(y, Mary): σε όλους αρέσει η Μαρία.

Αν ενοποιείται με το στόχο, τελειώσαμε.

Implementation Files

The algorithm is implemented in the file FOL.java

Methods

1. `fc_Ask(Set<Implication> KB, Implication a)`
2. `findImplicationMatch(Implication implication, Set<Implication> KB)`
3. `findPredicateMatch(Predicate predicate, Set<Implication> listToSearchIn)`
4. `main(String[] args)`
5. `printMenu()`

Helper classes:

FOLLoader.java

Methods

1. initialize(String filename)
2. convertInput(String input)

Implication.java

Methods

1. equals(Object obj)
2. getPremisePredicates()
3. isFact()
4. getConclusionPredicate()
5. setConclusionPredicate
6. isEmpty()
7. hashCode()
8. toString()

Predicate.java

Methods

1. getTerms()
2. getName()
3. negate()
4. increaseVariableName(int counter)
5. equals(Object obj)
6. hashCode()

Standarize.java

Term.java

Unifier.java

Implementation Analysis

In each while loop, the **counter** that renames variables from x to x1, x2, etc., is reset to zero to prevent the creation of large numbers (line 17).

Note: **Facts** are implications where the premisePredicates list is empty**.**

Lines 23-47: For each rule in the KB, using the helper method findImplicationMatch, it searches for substitutions for all **premises** by comparing them to the **facts** in the KB. If a substitution exists for the entire premise, a new **implication** is formed as a **fact**.

```java
public Map<String, String> fc_Ask(Set<Implication> KB, Implication a)
{
    while (true)
    {
        // resetCounter
        standarize.resetCounter();

        Set<Implication> newSentences = new HashSet<>();

        for (Implication currentImplication : KB)
        {
            standarize.newVars(currentImplication);

            Map<String, String> substitutions = findImplicationMatch(currentImplication, KB);

            // there is an available substitution for current implication
            if (substitutions.size() > 0)
            {
                Predicate newFact = new Predicate(currentImplication.getConclusionPredicate());

                // replace variables in conclusion
                List<Term> conclusionTerms = newFact.getTerms();
                for (Map.Entry<String, String> entry : substitutions.entrySet())
                {
                    for (Term term : conclusionTerms)
                    {

                        if (term.name.equals(entry.getKey()))
                        {
                            term.isVariable = false;
                            term.name = entry.getValue();
                        }
                    }
                }

                Implication newSentence = new Implication(newFact);
```

Next, using the findPredicateMatch method, it checks whether the new fact **cannot be unified** with any existing element in the KB or in the newSentences collection.

If this is the case, it is added to newSentences, and it is checked whether it can be unified with the **user's query**, returning the variable substitution if unification is successful.

The code terminates when the entire KB has been examined and no new sentences have been generated.

```
48                  if (!findPredicateMatch(newFact, KB) && !findPredicateMatch(newFact, newSentences))
49                  {
50                      newSentences.add(newSentence);
51
52                      // if currentConclusion matches input from user
53                      if (newFact.getName().equals(a.getConclusionPredicate().getName()))
54                      {
55
56                          // check if their terms can be unified
57                          List<Term> inputTerms = a.getConclusionPredicate().getTerms();
58
59                          Unifier unifier = new Unifier();
60                          boolean canUnify = true;
61
62                          for (int i = 0; i < conclusionTerms.size(); i++)
63                          {
64                              Term currentTerm = conclusionTerms.get(i);
65                              Term otherTerm = inputTerms.get(i);
66
67                              if (!unifier.unify(currentTerm, otherTerm))
68                              {
69                                  canUnify = false;
70                                  System.out.println(x:"cant unifiy");
71                                  break;
72                              }
73                          }
74
75                          if (canUnify)
76                          {
77                              return unifier.substitutions;
78                          }
79                      }
80                  }
81              }
82          }
83
84          if (newSentences.isEmpty())
85          {
86              return null;
87          }
88
89          KB.addAll(newSentences);
```

Results with the given KB from the lecture

KB

NOTAmerican(x) OR NOTWeapon(y) OR NOTSells(x, y, z) OR NOTHostile(z) OR Criminal(x)

NOTMissile(x) OR NOTOwns(Nono, x) OR Sells(West, x, Nono)

NOTMissile(x) OR Weapon(x)

NOTEnemy(x, America) OR Hostile(x)

American(West)

Enemy(Nono, America)

Missile(M1)

Owns(Nono, M1)

Given Queries and Results

```
Enter the sentence(accepted format:e.g Criminal(x) ) :
Criminal(West)
True answer: {}
```

```
Enter the sentence(accepted format:e.g Criminal(x) ) :
Criminal(x)
True answer: {x=West}
```

```
Enter the sentence(accepted format:e.g Criminal(x) ) :
Sells(West, x, Nono)
True answer: {x=M1}
```

```
Enter the sentence(accepted format:e.g Criminal(x) ) :
Weapon(M1)
True answer: {}
```

```
Enter the sentence(accepted format:e.g Criminal(x) ) :
Hostile(Nono)
True answer: {}
```

```
Enter the sentence(accepted format:e.g Criminal(x) ) :
Criminal(John)
cant unifiy
False
```