Artificial Intelligence
Inference Engine Implementation (unification)

(rule based system using the forward chaining strategy)

Table of Contents

Part 1 – Requirements

Implement an **inference engine** for a rule-based system using a first-order representation language that operates according to the **forward chaining strategy**, using two data structures: a rule base and a working memory. Having a first-order language means that the rules should have variables (that is identifiers which can be bound to any constant).

The inference engine should filter all working memory elements through the rule base according to the following three steps: match, select and act. The match procedure should try to find a rule from the rule base whose antecedents are all matched by facts from the working memory. The first discovered rule with such a characteristic should be taken, and its consequents should be added to the working memory. The inference engine mechanism should be **re-invoked recursively** until no rule produces a new assertion, or until proving a certain goal.

Design a prototype of the inference engine, operating with the **depth-first search algorithm**, using an imperative language. Demonstrate its performance using two different sets of examples, as sown below:

<u>Case 1</u>

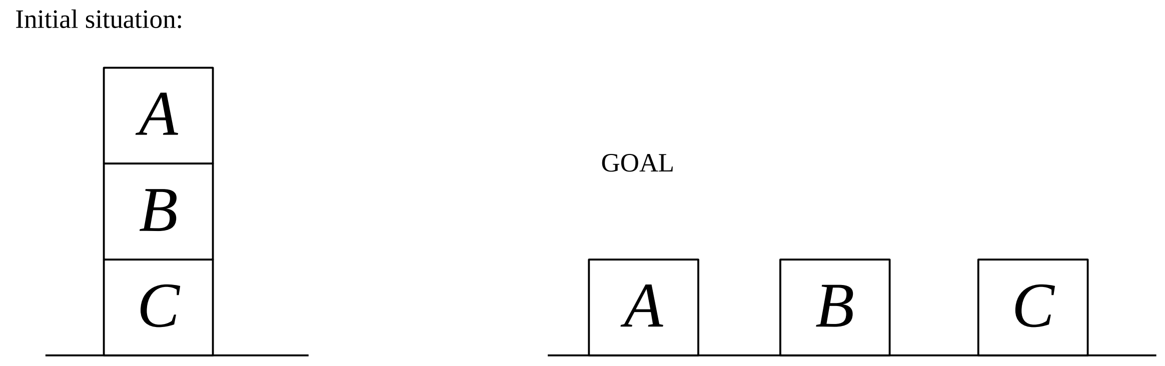
Rule base:

```
(RULE 1   (IF    ( ? h ) & ( a c ))
          (THEN  ( b g )))

(RULE 2   (IF    ( n s ))
          (THEN  ( e m )))

(RULE 3   (IF    ( r ? ))
          (THEN  ( p q )))

(RULE 4   (IF    ( ? j ) & ( e m ) & ( k i ))
          (THEN  ( a c )))

(RULE 5   (IF    ( p ? ))
          (THEN  ( n s )))

(RULE 6   (IF    ( ? v ))
          (THEN  ( k i )))
```

Working memory:

$WM =$ `( ( f h ) ( d j )( u v ) ( r t ) )`

Case 2 (a.k.a Block 3)



First-order rules contain variables that are bound to values through **unification**.

Unification is a matching procedure that takes a variable and a substitution list with variable-value (or variable-variable) pairs and produces a binding of the variable with a value if it is not already available in the substitution list .

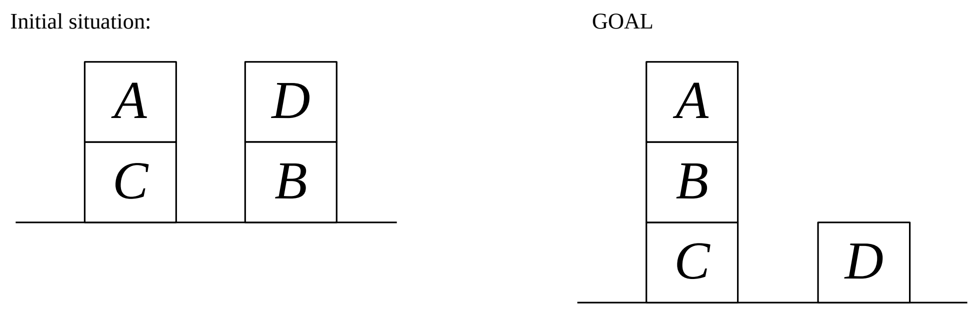
Example: Consider the following rules from the three-blocks world with variables:

```
(RULE 1   (IF    ( ?x on ?y ) & ( clear ?x ) & ( clear ?z ))
          (THEN (DELETE ( ?x on ?y )) & (DELETE ( clear ?z )) &
                     (ADD ( ?x on ?z )) & (ADD ( clear ?y ))))

(RULE 2   (IF    ( ?x on ?y ) & ( clear ?x ))
          (THEN (DELETE ( ?x on ?y )) &
                     (ADD ( ?x on table )) & (ADD ( clear ?y ))))

(RULE 3   (IF    ( ?x on table ) & ( clear ?x ) & ( clear ?z ))
          (THEN (DELETE ( ?x on table )) & (DELETE ( clear ?z )) &
                     (ADD ( ?x on ?z ))))
```

Assume the following working memory for block unstacking:

WM = ((clear A)(A on B)(B on C)(C on table))

Case 3 (a.k.a Block 4)



Use the same rules as Case 2 and assume the following working memory:

```
WM = ((A on C)(C on table)(clear A)(clear D)(D on B)(B on table) )
```

Part 2 – Algorithm

```
//*****************
//***************** Forward Chaining Inference Algorithm
//***************** Created By Frederic Marechal on 11/01/2017
//*****************
package AI;
import java.util.*;

public class FwdChainingInferenceEngine1 {

		final int ruleIdNotFound =-1;
		static Boolean goalIsReached = false;
		static boolean countCheck = true;
		static int count = 0;

		//Contains the list of rules with all variable replacement when a this is required,
		//i.e. when List<RuleVariableValue> contains a non empty list of elements.
		//When a rule is mapped against an empty List<RuleVariableValue>, then no variable replacement is required.
		//Key= RuleName, Value is a RuleVariableValue tuple => a variable against a value
		//For a given rule, more than one variable could have more than one value
		static Hashtable<Integer, ArrayList<ArrayList<RuleVariableValue>>>  htRulesValues;
		static List<Integer>  visitedRuleIds;

		private static void print(String message)
		{
				System.out.println(message);
		}

		@SuppressWarnings({ "serial", "unused" })
		private static InferenceEngineInputTuple simpleCaseExample (final FwdChainingInferenceEngine1 fcie) throws Exception
		{
				print ("");
				print ("************");
				print ("Case 1");
				print ("************");
				print ("");

				countCheck = false;

				final Item item0 = fcie.new Item(0,"f");
				final Item item1 = fcie.new Item(1,"?");
				final Item item2 = fcie.new Item(2,"h");
				final Item item3 = fcie.new Item(3,"a");
				final Item item4 = fcie.new Item(4,"c");
				final Item item5 = fcie.new Item(5,"b");
				final Item item6 = fcie.new Item(6,"g");
				final Item item7 = fcie.new Item(7,"n");
				final Item item8 = fcie.new Item(8,"s");
				final Item item9 = fcie.new Item(9,"e");
				final Item item10 = fcie.new Item(10,"m");
				final Item item11 = fcie.new Item(11,"r");
				final Item item12 = fcie.new Item(12,"t");
				final Item item13 = fcie.new Item(13,"d");
				final Item item14 = fcie.new Item(14,"j");
				final Item item15 = fcie.new Item(15,"k");
				final Item item16 = fcie.new Item(16,"i");
				final Item item17 = fcie.new Item(17,"p");
				final Item item18 = fcie.new Item(18,"q");
				final Item item19 = fcie.new Item(19,"u");
				final Item item20 = fcie.new Item(20,"v");
				//final Item item21 = fcie.new Item(21,"z");
				print("");
				print("***Create the fact list");
				final Fact fact1 = new Fact(1, new ArrayList<Item>(){{add(item1.deepCopy());add(item2);/*add(item21);*/}}); //? h
				final Fact fact2 = new Fact(2, new ArrayList<Item>(){{add(item3);add(item4);}}); // a c
				final Fact fact3 = new Fact(3, new ArrayList<Item>(){{add(item7);add(item8);}}); // n s
				final Fact fact4 = new Fact(4, new ArrayList<Item>(){{add(item11);add(item1.deepCopy());}}); // r ?
		final Fact fact5 = new Fact(5, new ArrayList<Item>(){{add(item1.deepCopy());add(item14);/*add(item21);*/}}); // ? j


		final Fact fact6 = new Fact(6, new ArrayList<Item>(){{add(item9);add(item10);}}); // e m


		final Fact fact7 = new Fact(7, new ArrayList<Item>(){{add(item15);add(item16);}}); // k i
		final Fact fact8 = new Fact(8, new ArrayList<Item>(){{add(item17);add(item1.deepCopy());}}); // p ?
		final Fact fact9 = new Fact(9, new ArrayList<Item>(){{add(item1.deepCopy());add(item20);}}); // ? v
		print("");
		print("***Create the consequent list");
		final Consequent consequent1 = fcie.new Consequent(100, new ArrayList<Item>(){{add(item5);add(item6);}}); //b g
		final Consequent consequent2 = fcie.new Consequent(6, new ArrayList<Item>(){{add(item9);add(item10);}}); //e m
		final Consequent consequent3 = fcie.new Consequent(8, new ArrayList<Item>(){{add(item17);add(item18);}}); // p q
		final Consequent consequent4 = fcie.new Consequent(2, new ArrayList<Item>(){{add(item3);add(item4);}}); //a c
		final Consequent consequent5 = fcie.new Consequent(3, new ArrayList<Item>(){{add(item7);add(item8);}}); // n s
		final Consequent consequent6 = fcie.new Consequent(7, new ArrayList<Item>(){{add(item15);add(item16);}}); // k i
		print("");
		print("***Create the RuleBase data structure and add rules");
		RuleBase ruleBase = fcie.new RuleBase();
		ruleBase.addRule(fcie.new Rule(1, new ArrayList<Fact>(){{add(fact1);add(fact2);}}, consequent1));
		ruleBase.addRule(fcie.new Rule(2, new ArrayList<Fact>(){{add(fact3);}}, consequent2));
		ruleBase.addRule(fcie.new Rule(3, new ArrayList<Fact>(){{add(fact4);}}, consequent3));
		ruleBase.addRule(fcie.new Rule(4, new ArrayList<Fact>(){{add(fact5);add(fact6);add(fact7);}}, consequent4));
		ruleBase.addRule(fcie.new Rule(5, new ArrayList<Fact>(){{add(fact8);}}, consequent5));
		ruleBase.addRule(fcie.new Rule(6, new ArrayList<Fact>(){{add(fact9);}}, consequent6));
		print("");
		print("***Create the WorkingMemory and add facts");
		WorkingMemory workingMemory = fcie.new WorkingMemory();
		workingMemory.addFact(new Fact(1, new ArrayList<Item>(){{add(item0);add(item2);}})); // f h
		workingMemory.addFact(new Fact(5, new ArrayList<Item>(){{add(item13);add(item14);/*add(item21);*/}})); // d j
		workingMemory.addFact(new Fact(9, new ArrayList<Item>(){{add(item19);add(item20);}})); // u v
		workingMemory.addFact(new Fact(4, new ArrayList<Item>(){{add(item11);add(item12);}})); // r t

		return new InferenceEngineInputTuple(ruleBase, workingMemory);
	}
```

```
@SuppressWarnings({ "serial", "unused" })
private static InferenceEngineInputTuple block3Example (final FwdChainingInferenceEngine1 fcie) throws Exception
{
		print ("");
		print ("************");
		print ("Block3 Case");
		print ("************");
		print ("");

		countCheck = true;

		print("***Create the item list");
		final Item item1 = fcie.new Item(1,"clear A");
		final Item item2 = fcie.new Item(2,"A on B");
		final Item item3 = fcie.new Item(3,"B on C");
		final Item item4 = fcie.new Item(4,"C on table");
		final Item item5 = fcie.new Item(5, "?x on ?y");
		final Item item6 = fcie.new Item(6,"clear ?x");
		final Item item7 = fcie.new Item(7,"clear ?z");
		final Item item8 = fcie.new Item(8,"clear ?y");
		final Item item9 = fcie.new Item(9,"?x on table");
		final Item item10 = fcie.new Item(10,"?x on ?z") ;
		final Item item11 = fcie.new Item(11,"A on table");
		final Item item12 = fcie.new Item(12,"clear B");
		final Item item13 = fcie.new Item(13,"B on table");
		final Item item14 = fcie.new Item(14,"clear C");
		final Item item15 = fcie.new Item(15,"C on table");
		print("");
		print("***Create the fact list");
		final Fact fact1 = new Fact(1, new ArrayList<Item>(){{add(item5.deepCopy());}}); //(?x on ?y)
		final Fact fact2 = new Fact(2, new ArrayList<Item>(){{add(item6.deepCopy());}}); //(clear ?x)
		final Fact fact3 = new Fact(3, new ArrayList<Item>(){{add(item7.deepCopy());}}); //(clear ?z)
		final Fact fact4 = new Fact(4, new ArrayList<Item>(){{add(item9.deepCopy());}}); //(?x on table)
		final Fact fact5 = new Fact(5, new ArrayList<Item>(){{add(item10.deepCopy());}}); //(?x on ?z)
		print("");
		print("***Create the consequent list");
		final Consequent consequent1 = fcie.new Consequent(1, new ArrayList<Item>(){{
				add(fcie.new Item(Action.DELETE,item5.getId(), item5.getLabel())); //DELETE (?x on ?y)
				add(fcie.new Item(Action.DELETE,item7.getId(), item7.getLabel())); //DELETE  (clear ?z)
				add(fcie.new Item(Action.ADD,item10.getId(), item10.getLabel()));  //ADD (?x on ?z)
				add(fcie.new Item(Action.ADD,item8.getId(), item8.getLabel()));    //ADD  (clear ?y) }});

		final Consequent consequent2 = fcie.new Consequent(2, new ArrayList<Item>(){{
				add(fcie.new Item(Action.DELETE,item5.getId(), item5.getLabel())); //DELETE (?x on ?y)
				add(fcie.new Item(Action.ADD,item9.getId(), item9.getLabel())); //ADD (?x on table)
				add(fcie.new Item(Action.ADD,item8.getId(), item8.getLabel())); //ADD (clear ?y)}});

		final Consequent consequent3 = fcie.new Consequent(3, new ArrayList<Item>(){{
				add(fcie.new Item(Action.DELETE,item9.getId(), item9.getLabel())); //DELETE (?x on table)
				add(fcie.new Item(Action.DELETE,item7.getId(), item7.getLabel())); //DELETE (clear ?z)
				add(fcie.new Item(Action.ADD,item10.getId(), item10.getLabel())); //ADD (?x on ?z) }});
		print("");
		print("***Create the RuleBase data structure and add rules");
		RuleBase ruleBase = fcie.new RuleBase();
		ruleBase.addRule(fcie.new Rule(1, new ArrayList<Fact>(){{add(fact1);add(fact2);add(fact3);}}, consequent1));
		ruleBase.addRule(fcie.new Rule(2, new ArrayList<Fact>(){{add(fact1);add(fact2);}}, consequent2));
		ruleBase.addRule(fcie.new Rule(3, new ArrayList<Fact>(){{add(fact4);add(fact2);add(fact3);}}, consequent3));
		print("");
		print("***Create the WorkingMemory and add facts");
		WorkingMemory workingMemory = fcie.new WorkingMemory();
		workingMemory.addFact(new Fact(1, new ArrayList<Item>(){{add(item1);}})); //clear A
		workingMemory.addFact(new Fact(2, new ArrayList<Item>(){{add(item2);;}})); //A on B
		workingMemory.addFact(new Fact(3, new ArrayList<Item>(){{add(item3);}})); //B on C
		workingMemory.addFact(new Fact(4, new ArrayList<Item>(){{add(item4);}})); //C on table

		return new InferenceEngineInputTuple(ruleBase, workingMemory);
}
```

```
@SuppressWarnings({ "serial", "unused" })
private static InferenceEngineInputTuple block4Example (final FwdChainingInferenceEngine1 fcie) throws Exception
{
        print ("");
        print ("***********");
        print ("Block4 Case");
        print ("***********");
        print ("");

        countCheck = true;

        print("***Create the item list");
        final Item item1 = fcie.new Item(1,"clear A");
        final Item item2 = fcie.new Item(2,"A on B");
        final Item item3 = fcie.new Item(3,"B on C");
        final Item item4 = fcie.new Item(4,"C on table");
        final Item item5 = fcie.new Item(5, "?x on ?y");
        final Item item6 = fcie.new Item(6,"clear ?x");
        final Item item7 = fcie.new Item(7,"clear ?z");
        final Item item8 = fcie.new Item(8,"clear ?y");
        final Item item9 = fcie.new Item(9,"?x on table");
        final Item item10 = fcie.new Item(10,"?x on ?z") ;
        final Item item11 = fcie.new Item(11,"A on table");
        final Item item12 = fcie.new Item(12,"clear B");
        final Item item13 = fcie.new Item(13,"B on table");
        final Item item14 = fcie.new Item(14,"clear C");
        final Item item15 = fcie.new Item(15,"C on table");
        final Item item16 = fcie.new Item(16,"A on C");
        final Item item17 = fcie.new Item(17,"clear D");
        final Item item18 = fcie.new Item(18,"D on B");
        print("");
        print("***Create the fact list");
        final Fact fact1 = new Fact(1, new ArrayList<Item>(){{add(item5.deepCopy());}}); //(?x on ?y)
        final Fact fact2 = new Fact(2, new ArrayList<Item>(){{add(item6.deepCopy());}}); //(clear ?x)
        final Fact fact3 = new Fact(3, new ArrayList<Item>(){{add(item7.deepCopy());}}); //(clear ?z)
        final Fact fact4 = new Fact(4, new ArrayList<Item>(){{add(item9.deepCopy());}}); //(?x on table)
        final Fact fact5 = new Fact(5, new ArrayList<Item>(){{add(item10.deepCopy());}}); //(?x on ?z)
        print("");
        print("***Create the consequent list");
        final Consequent consequent1 = fcie.new Consequent(1, new ArrayList<Item>(){{
                add(fcie.new Item(Action.DELETE,item5.getId(), item5.getLabel())); //DELETE (?x on ?y)
                add(fcie.new Item(Action.DELETE,item7.getId(), item7.getLabel())); //DELETE  (clear ?z)
                add(fcie.new Item(Action.ADD,item10.getId(), item10.getLabel()));  //ADD (?x on ?z)
                add(fcie.new Item(Action.ADD,item8.getId(), item8.getLabel()));    //ADD  (clear ?y) }});
        final Consequent consequent2 = fcie.new Consequent(2, new ArrayList<Item>(){{
                add(fcie.new Item(Action.DELETE,item5.getId(), item5.getLabel())); //DELETE (?x on ?y)
                add(fcie.new Item(Action.ADD,item9.getId(), item9.getLabel())); //ADD (?x on table)
                add(fcie.new Item(Action.ADD,item8.getId(), item8.getLabel())); //ADD (clear ?y) }});
        final Consequent consequent3 = fcie.new Consequent(3, new ArrayList<Item>(){{
                add(fcie.new Item(Action.DELETE,item9.getId(), item9.getLabel())); //DELETE (?x on table)
                add(fcie.new Item(Action.DELETE,item7.getId(), item7.getLabel())); //DELETE (clear ?z)
                add(fcie.new Item(Action.ADD,item10.getId(), item10.getLabel())); //ADD (?x on ?z)}});
        print("");
        print("***Create the RuleBase data structure and add rules");
        RuleBase ruleBase = fcie.new RuleBase();
        ruleBase.addRule(fcie.new Rule(1, new ArrayList<Fact>(){{add(fact1);add(fact2);add(fact3);}}, consequent1));
        ruleBase.addRule(fcie.new Rule(2, new ArrayList<Fact>(){{add(fact1);add(fact2);}}, consequent2));
        ruleBase.addRule(fcie.new Rule(3, new ArrayList<Fact>(){{add(fact4);add(fact2);add(fact3);}}, consequent3));
        print("");
        print("***Create the WorkingMemory and add facts");
        WorkingMemory workingMemory = fcie.new WorkingMemory();
        workingMemory.addFact(new Fact(1, new ArrayList<Item>(){{add(item16);}})); //A on C
        workingMemory.addFact(new Fact(2, new ArrayList<Item>(){{add(item4);}})); //C on table
        workingMemory.addFact(new Fact(3, new ArrayList<Item>(){{add(item1);}})); //clear A
        workingMemory.addFact(new Fact(4, new ArrayList<Item>(){{add(item17);}})); //clear D
        workingMemory.addFact(new Fact(5, new ArrayList<Item>(){{add(item18);}})); //D on B
        workingMemory.addFact(new Fact(5, new ArrayList<Item>(){{add(item13);}})); //B on table

        return new InferenceEngineInputTuple(ruleBase, workingMemory);
}
```

```
public static void main(String[] args) {
        try {
                final FwdChainingInferenceEngine1 fcie = new FwdChainingInferenceEngine1();

                InferenceEngineInputTuple inferenceEngineInputTuple = null;

                inferenceEngineInputTuple = simpleCaseExample(fcie);
                //inferenceEngineInputTuple = block3Example(fcie);
                //inferenceEngineInputTuple = block4Example(fcie);

                print("");
                print("***Create InferenceEngine and start the inference process");
                visitedRuleIds =  new ArrayList<Integer>();

                InferenceEngine inferenceEngine = fcie.new InferenceEngine(inferenceEngineInputTuple.getRuleBase(),
                        inferenceEngineInputTuple.getWorkingMemory() );
                boolean res = inferenceEngine.execute();
                //Stop, as inference could not be proven!
                if (res == false){
                        print("FAILURE");
                        return;
                }
                //Success!
                print("SUCCESS / ALL RULES HAVE BEEN PROVEN!");

        } catch (Exception e) {
                e.printStackTrace();
        }
}

public class InferenceEngine {

        private RuleBase ruleBase;
        private WorkingMemory workingMemory;

        public InferenceEngine(RuleBase ruleBase, WorkingMemory workingMemory ) throws Exception{
                this.ruleBase = ruleBase;
                this.workingMemory = workingMemory;
                if (this.ruleBase == null) {throw new Exception ("InferenceEngine -> The ruleBase cannot be null" );}
        }

        //Starts the engine
        public boolean execute()
        {
                try {
                        print("***The 'infer' method is executed");
                        return infer();

                } catch (Exception e) {
                        e.printStackTrace();
                }

                return false;
        }
```

```
//Infer does the match, select and act recursively
private boolean infer() throws Exception {

		htRulesValues = new Hashtable<Integer, ArrayList<ArrayList<RuleVariableValue>>>();

		print("***MATCH Rule with antecedent");
		int ruleId =  this.getRuleId(this.workingMemory);
		if (ruleId == ruleIdNotFound){
			return false;
		}

		print("***SELECT a rule");
		Rule rule = this.select (ruleId);
		print("Rule selected: " + rule.id );

		print("***Unify rule variables (binds variables with constants)");
		Rule unifiedRule = this.unify(rule, 0);
		print("Rule Unified: " + rule.id );

		print ("***ACT - Attempt to add Fact to working memory");
		act(unifiedRule.getConsequent());

		if (count < this.ruleBase.getRulesCount()-1){
			count++;
			Boolean res = infer();
			if (res == false){
				return false;
			}
		}
		return true;
}

public int getRuleId(WorkingMemory workingMemory){
		int ruleId = this.ruleBase.getRuleId(workingMemory);
		if (ruleId <0){
			print ("No Rule id cannot be found that matches the working memory.");
		}
		return ruleId;
}

public List<Fact> matchASubGoalWithAntecedents(Rule rule) throws Exception{
		List<Fact> antecedents = rule.getAntecedents();
		if (rule.getAntecedents() ==null){
			throw new Exception ("There is no antecedant for rule id: " + rule.getId() );
		}
		return antecedents;
}

public Rule select(int ruleId) throws Exception{
		Rule rule = this.ruleBase.getRule(ruleId);
		return rule;
}

public Rule unify(Rule rule, int index) throws Exception{
		return ruleBase.unify (rule, 0, htRulesValues);
}
```

```
        public void act(Consequent consequent) throws Exception{
            for (int i =0; i<consequent.getItems().size(); i++){
                Item itemConsequent  = consequent.getItems().get(i);
                if (itemConsequent.getAction() != Action.NONE){
                    int randFactId = (int) (System.currentTimeMillis() & 0xfffffff);
                    List<Item> items =new ArrayList<Item>();
                    items.add(itemConsequent);
                    if (itemConsequent.getAction() == Action.ADD){
                        this.workingMemory.addFact(new Fact(randFactId, items));
                    }else if (itemConsequent.getAction() == Action.DELETE){
                        this.workingMemory.removeFact(new Fact(randFactId, items));
                    }
                    else {
                        throw new Exception ("InferenceEngine -> act -  consequent key: " + consequent.getId() +"
                        'action' not supported (only ADD and DELETE supported)");
                    }
                }
                else{
                    this.workingMemory.addFact(consequent);
                    break;
                }
            }
        }
    }

    public class WorkingMemory{
        private ArrayList<Fact> facts;

        public WorkingMemory (){
            this.facts = new ArrayList<Fact>();
        }

        public ArrayList<Fact> getFacts(){ return this.facts;}

        public void addFact(Fact fact) throws Exception{
            WorkingMemoryFactInfo workingMemoryFactInfo = this.getFactInfo(fact);
            if (workingMemoryFactInfo.getExistsFlag()){
                print ("WorkingMemory -> addFact - labels: " + fact.getLabelsNoAction() +" already contained in the working
memory.");
            }

            facts.add(fact);
            print("WorkingMemory -> addFact - key: " + fact.id + " - Fact:" + fact.getLabelsNoAction());
            display();
        }

        public void removeFact(Fact fact) throws Exception{
            WorkingMemoryFactInfo workingMemoryFactInfo = this.getFactInfo(fact);
            if ( workingMemoryFactInfo.getExistsFlag()){
                facts.remove( workingMemoryFactInfo.getIndex());
                print("WorkingMemory -> removeFact - key: " + fact.id + " - Fact:" + fact.getLabels());
                display();
                return;
            }

            throw new Exception ("WorkingMemory -> removeFact - labels: " + fact.getLabels() +" cannot be removed as it does not exists
            in the working memory.");
        }

        public WorkingMemoryFactInfo getFactInfo(Fact fact){
            for (int i = 0; i< this.facts.size(); i++){
                if (Fact.areEqual(this.facts.get(i), fact)){
                    return new WorkingMemoryFactInfo(true, i);
                }
            }
            return new WorkingMemoryFactInfo(false, -1);
        }
```

```
                public class WorkingMemoryFactInfo {
                    public boolean exists; // indicates where the fact exists in the working memory
                    public int index; //position in the working memory

                    public WorkingMemoryFactInfo(boolean exists, int index) {
                        this.exists = exists;
                        this.index = index;
                    }

                    public boolean getExistsFlag() {return this.exists;}
                    public int getIndex() {return this.index;}
                }

                private void display(){
                                StringBuilder sb = new StringBuilder();
                                String factLabel = null;
                                sb.append("Working Memory State: ( ");
                                for (int i = 0; i< facts.size(); i++){
                                                factLabel = facts.get(i).getLabelsNoAction();
                                                sb.append(factLabel);
                                }
                                print(sb.toString() + " )");
                }
}

public class RuleBase{

                private Hashtable<Integer, Rule> rules;

                public RuleBase (){
                                this.rules = new Hashtable<Integer, Rule>();
                }

                public void addRule(Rule rule ) throws Exception{
                                if (this.rules.containsKey(rule.getId())){
                                                throw new Exception ("RuleBase -> addRules - key: " + rule.getId() +" already contained in hashtable.");
                                }

                                rules.put(rule.getId(), rule);
                                print("RuleBase -> addRules - key: " + rule.getId());
                }

                public Rule getRule(int ruleId){
                                return rules.get(ruleId);
                }

                public int getRulesCount(){
                                return rules.size();
                }
```

```
public Rule unify(Rule rule, int index, Hashtable<Integer,ArrayList<ArrayList<RuleVariableValue>>> htRulesValues){
        Rule copyRule = rule.deepCopy();
        ArrayList<RuleVariableValue> unifiedlist = new ArrayList<RuleVariableValue>();
        ArrayList<ArrayList<RuleVariableValue>> list = htRulesValues.get(copyRule.getId());
        if (list!=null){
                Hashtable<String, RuleVariableValueCount> ht = new Hashtable<String, RuleVariableValueCount>();
                int countVariables = list.get(index).size();
                int m = 0;
                while (m < countVariables)
                {
                        for (int k=0; k<list.size(); k++){
                                ArrayList<RuleVariableValue> currentRuleVariableValues  = list.get(k);
                                if (currentRuleVariableValues.size() == countVariables){
                                        String variableName = currentRuleVariableValues.get(m).getVariableName();
                                        RuleVariableValueCount ruleVariableValueCount;
                                        if (!ht.containsKey(variableName)){
                                                ruleVariableValueCount = new RuleVariableValueCount
                                                (currentRuleVariableValues.get(m), 1);
                                                ht.put(variableName, ruleVariableValueCount);
                                        }else{
                                                ruleVariableValueCount = ht.get(variableName);
                                                ht.remove(variableName);
                                                ruleVariableValueCount = new RuleVariableValueCount
(ruleVariableValueCount.getRuleVariableValue(), ruleVariableValueCount.getCount()+1);
                                                ht.put(variableName, ruleVariableValueCount);
                                        }
                                }
                        }
                        //Find the variableName Vs variableValue that occurs the most often
                        int maxFreq = 0;
                        String maxKey = "";
                        if (ht.size() > 0){
                                for (String key: ht.keySet()){
                                        RuleVariableValueCount currentRuleVariableValue = ht.get(key);
                                        int currentCount = currentRuleVariableValue.getCount();
                                        if (maxFreq < currentCount){
                                                maxFreq = currentCount;
                                                maxKey = key;
                                        }
                                }
                                unifiedlist.add(ht.get(maxKey).getRuleVariableValue());
                        }
                        m++;
                }

                ArrayList<RuleVariableValue> ruleVariableValues = unifiedlist;
                //ruleVariableValues = list.get(index);
                if (ruleVariableValues.size() > 0){
                        //case where one or more variable(s) replacement is needed
                        for (int i=0; i<ruleVariableValues.size(); i++){
                                RuleVariableValue ruleVariableValue = ruleVariableValues.get(i).deepCopy();
                                replaceVariableByConstant(ruleVariableValue.variableName,
                                ruleVariableValue.variableValue,copyRule);
                        }
                }
        }
        return copyRule;
}

private int expectedVariableCount(String[] splits){
        int count = 0;
        for (String str : splits){
           if (str.contains("?"))
              count++;
        }
        return count;
}
```

```
private int workingMemoryValueCount(String[] splits){
        int count = 0;
        for (String str : splits){
                str = str.trim().toLowerCase();
            if (str.equals("on") || str.equals("table") || str.equals("clear"))
                count++;
        }
        return splits.length - count;
}


private int getBranchIndex(ArrayList<ArrayList<RuleVariableValue>> list,String antecedentElem ){
        int index = 0;
        for (ArrayList<RuleVariableValue> rvvs :list){
                for (RuleVariableValue rvv :rvvs){
                        if (rvv.getVariableName().equals(antecedentElem)){
                                index++;
                                break;
                        }
                }
        }
        return index;
}

private Boolean isVariableAlreadyMappedToGivenValue(ArrayList<ArrayList<RuleVariableValue>> list,String antecedentElem, String
wmElem ){
        if (list !=null){
                for (ArrayList<RuleVariableValue> rvvs :list){
                        for (RuleVariableValue rvv :rvvs){
                                if (rvv.getVariableName().equals(antecedentElem) &&
                                        rvv.getVariableValue().equals(wmElem)       ){
                                        return true;
                                }
                        }
                }
        }
        return false;
}

@SuppressWarnings({ "unchecked","rawtypes" })
private Boolean isVariablesValueUnique(ArrayList<RuleVariableValue> ruleVariableValues){
        Set set = new HashSet();
        for (RuleVariableValue ruleVariableValue:ruleVariableValues){
                set.add(ruleVariableValue.getVariableValue());
        }
        return (ruleVariableValues.size() == set.size());
}

private Boolean canContinue(String[] antecedentSplits, String[] wmSplits, Boolean countCheck){

        int countExpectedVariables =  expectedVariableCount(antecedentSplits);
        int countExpectedValues = workingMemoryValueCount(wmSplits);
        if(countCheck){
                return (wmSplits.length == antecedentSplits.length &&
                                countExpectedVariables == countExpectedValues);
        }
        return (wmSplits.length == antecedentSplits.length);
}

private Boolean isValueInWM(String antecedentLabels, WorkingMemory workingMemory){
        for(Fact fact: workingMemory.getFacts()){
                if (antecedentLabels.equals(fact.getRawLabels()))
                        return true;
        }
        return false;
}
```

```
public int getRuleId(WorkingMemory workingMemory){
        for ( int i =1; i <= this.rules.size(); i++){
                Rule rule = this.rules.get(i);
                Integer ruleId = rule.getId();
                //Check to avoid infinite loop
                if (visitedRuleIds.contains(ruleId)){continue;}
                //If no variable in antecedent list, the check all items are present in the WM.
                //if 'yes' return the rule, else 'continue'
                Integer foundRuleId = matchWorkingMemoryWithAntecedent(rule, workingMemory);
                if (foundRuleId != ruleIdNotFound){
                        return foundRuleId;
                }
                List<Fact> antecedents= rule.getAntecedents();
                for ( int f =0; f < antecedents.size(); f++){
                        int branchId = 0;
                        String antecedentLabels  = antecedents.get(f).getRawLabels();
                        if (!antecedentLabels.contains("?")){
                                if (!isValueInWM(antecedentLabels, workingMemory)){
                                        if(htRulesValues.containsKey(ruleId)){
                                                htRulesValues.remove(ruleId);
                                                break;
                                        }
                                }
                        }
                        //This section relates to going through the list of antecedents (for a given rule)
                        //and map each variable with a constant. A rule may have more than one mapping.
                        //The mapping is stored in the htRulesValues table.
                        String[] antecedentSplits = antecedentLabels.split(" ");
                        for (int k=0; k<workingMemory.getFacts().size(); k++){
                                String wmLabels = workingMemory.getFacts().get(k).getRawLabels();
                                String[] wmSplits = wmLabels.split(" ");
                                if (canContinue(antecedentSplits,wmSplits, countCheck )){
                                        for (int idx=0; idx<wmSplits.length; idx++){
                                                String antecedentElem = antecedentSplits[idx];
                                                final String wmElem = wmSplits[idx];
                                                if (antecedentElem.contains("?")){
                                                        RuleVariableValue ruleVariableValue = new
                                                RuleVariableValue(antecedentElem,wmElem);
                                                        ArrayList<ArrayList<RuleVariableValue>> list =
                                                htRulesValues.get(ruleId);
                                                        //only analyse unmapped variable with a different value from
                                                        previous cases
                                                        if (!isVariableAlreadyMappedToGivenValue(list,
                                                        antecedentElem,wmElem)){
                                                                ArrayList<RuleVariableValue>
                                                                ruleVariableValues;
                                                                if (list!=null){
                                                                        //get the next branch
                                                                        branchId = getBranchIndex(list,
                                                                                antecedentElem);
                                                                        if (list.size()-1 < branchId){
                                                                                //add branching for a rule
                                                                                ruleVariableValues = new
                                                                                ArrayList<RuleVariableValu
                                                                        e>();

                                                                        ruleVariableValues.add(ruleVariableVa
                                                                lue);

                                                                        list.add((ArrayList<RuleVariableValue>
                                                                )
                                                                        ruleVariableValues);
                                                                        }
                                                                        else{
                                                                                //update an existing branch
                                                                                of a rule
                                                                                ruleVariableValues =
                                                                                list.get(branchId);
                                                                                if
                                                                                (ruleVariableValues!=null){
```

```
                                                                                          //There is at
                                                                            least one variable set in this rule...
                                                                            update


                                                                            ruleVariableValues.add(ruleVariableVa
                                                                            lue);
                                                                            htRulesValues.remove(ruleId);
                                                                                    }
                                                                            }
                                                                    }
                                                                    else{

                                                                            //Create a new entry in the current rule
                                                                            ruleVariableValues = new
                                                                            ArrayList<RuleVariableValue>();


                                                                    ruleVariableValues.add(ruleVariableValue);
                                                                            list = new
                                                                    ArrayList<ArrayList<RuleVariableValue>>();
                                                                            list.add((ArrayList<RuleVariableValue>
                                                                            ) ruleVariableValues);
                                                                    }
                                                                    htRulesValues.put(ruleId,list);
                                                            }
                                                    }
                                            }
                                    }
                            }
                    }
                    if (htRulesValues.size() >0){
                            //All variables have been unified, so we can use this use
                            if (htRulesValues.get(ruleId).get(0).size() == rule.getAntecedentUniqueVariablesCount() &&
                                    isVariablesValueUnique(htRulesValues.get(ruleId).get(0))){
                                    //Furthermore this rule has not been triggered in the past... so use it...else we could have an
                                    //infinite loop
                                    if (!visitedRuleIds.contains(ruleId)){
                                            visitedRuleIds.add(ruleId);
                                            return ruleId;
                                    }
                            }
                            else{
                                    //Unification has failed for this rule, remove the rule
                                    htRulesValues.remove(ruleId);
                            }
                    }
            }
            return ruleIdNotFound;
    }
```

```
	private int matchWorkingMemoryWithAntecedent(Rule rule, WorkingMemory workingMemory){
		Integer ruleId = rule.getId();
		//If no variable in antecedent list, then
		//check all items are present in the WM.
		//if 'yes' return the rule, else 'continue'
		if (rule.getAntecedentUniqueVariablesCount() == 0){
			for (Fact antecedent : rule.getAntecedents()){
				for (Fact fact : workingMemory.getFacts()){
					if (antecedent.getRawLabels().equals(fact.getRawLabels())){
						if (!visitedRuleIds.contains(ruleId)){
							visitedRuleIds.add(ruleId);
							return ruleId;
						}
						else
							return ruleIdNotFound;
					}
				}
			}
			//the antecedent is not in the WM... continue
			return ruleIdNotFound;
		}
		return ruleIdNotFound;
	}

	//Replace all instance of variable name in an item list by a given value

	private void replaceVariableByConstant (	String variableName, String constantValue, Rule rule){

		//replace the antecedent variable with corresponding value
		for (int i=0; i<rule.getAntecedents().size(); i++){
			Fact antecedent = rule.getAntecedents().get(i).deepCopy();
			for (int j=0; j<antecedent.items.size(); j++){
				Item item = antecedent.items.get(j).deepCopy();
				String label = item.getLabel().replace(variableName, constantValue);
				rule.getAntecedents().get(i).items.get(j).setLabel(label);
				print( "Rule Id: " + rule.getId() + " - " +
					 "Antecedent " + label + " - " +
						variableName + " set to " + constantValue);
			}
		}

		//replace the consequent variable with corresponding value
		for (int i=0; i<rule.getConsequent().getItems().size();i++){
			Item item = rule.getConsequent().getItems().get(i).deepCopy();
			String label = item.getLabel();
			if (label.contains(variableName)){
				label = label.replace(variableName, constantValue);
				item.setLabel(label);
				rule.getConsequent().getItems().get(i).setLabel(label);
				print( "Rule Id: " + rule.getId() + " - " +
					 "Consequent " + label + " - " +
						variableName + " set to " + constantValue);
			}
		}
	}
}
```

```
public class Rule{
        private Integer id;
        private List<Fact> antecedents;
        private Consequent consequent;

        public Rule (Integer id, List<Fact> antecedents, Consequent consequent){
                this.id = id;
                this.antecedents = antecedents;
                this.consequent = consequent;
        }

        public Integer getId() {return this.id;}
        public List<Fact> getAntecedents() {return this.antecedents;}
        public Consequent getConsequent() {return this.consequent;}

        public void setAntecedents(List<Fact> antecedents){this.antecedents=antecedents;}

        public String getLabels()
        {
                StringBuilder sb = new StringBuilder();
                sb.append("RuleId :  ");
                sb.append(this.id);
                sb.append(" ");
                sb.append("Antecedent List:  ");
                for (int i = 0; i< this.antecedents.size();i++){
                        sb.append(this.antecedents.get(i).getLabels());
                }
                sb.append("Consequent List:  ");
                for (int i = 0; i< this.consequent.getItems().size();i++){
                        sb.append(this.consequent.getItems().get(i).getLabel());
                        sb.append(" ");
                }
                return sb.toString();
        }

        public int getAntecedentUniqueVariablesCount(){
                return getAntecedentUniqueVariables().size();
        }

        @SuppressWarnings({ "rawtypes", "unchecked" })


        public Set getAntecedentUniqueVariables(){
                Set finalSet = new HashSet();
                for (int j=0; j< this.antecedents.size(); j++){
                        Fact fact = this.antecedents.get(j);
                        String[] splits = fact.getLabels().split(" ");
                        for (int k=0; k<splits.length; k++){
                                String split = splits[k].toString();
                                if (split.contains("?")){
                                        finalSet.add(split);
                                }
                        }
                }
                return finalSet;
        }

        public Rule deepCopy(){

                List<Fact> antecedentsDeepCopy = new ArrayList<Fact>();
                for (Fact a : antecedents){
                        antecedentsDeepCopy.add(a.deepCopy());
                }

                return new Rule(this.id, antecedentsDeepCopy, this.consequent.deepCopy());
        }
}
```

```
public static class Fact{
        private Integer id;
        private List<Item> items;

        public static Boolean areEqual(Fact fact1, Fact fact2){
                List<Item> fact1Items = fact1.getItems();
                int fact1Size = fact1Items.size();
                //check the items have the same id
                for (int k = 0; k<fact1Size;k++){
                        //the
                        if (fact1Items.get(k).getLabel().equals(fact2.getItems().get(k).getLabel())){
                                return true;
                        }
                }
                return false;
        }

        public Fact (Integer id, List<Item> Items){
                this(id, Items, false);
        }

        public Fact (Integer id, List<Item> items, Boolean proven){
                this.id = id;
                this.items = items;
                display();
        }

        public Integer getId() {return this.id;}
        public List<Item> getItems() {return this.items;}

        public String getLabels()
        {
                StringBuilder sb = new StringBuilder();
                sb.append("(");
                for (int i=0; i < this.items.size() ;i++){
                        sb.append(" ");
                        sb.append(items.get(i).getLabel());
                        sb.append(" ");
                }
                sb.append(") ");

                return sb.toString();
        }

        public String getLabelsNoAction()
        {
                return this.getLabels();


        }

        public String getRawLabels()
        {
                StringBuilder sb = new StringBuilder();
                for (int i=0; i < this.items.size() ;i++)
                {
                        sb.append(this.items.get(i).getLabel());
                        sb.append(" ");
                }
                return sb.toString();
        }

        protected void display() {
                print("Fact - key: " + id +" - label: " + this.getLabels());
        }

        public Fact deepCopy(){
                List<Item> itemsDeepCopy = new ArrayList<Item>();
                for (Item a : this.items){
                        itemsDeepCopy.add(a.deepCopy());
                }
```

```
				return new Fact(this.id, itemsDeepCopy);
			}
		}

		public class Consequent extends Fact {

			public Consequent (Integer id, List<Item> items){
				super(id, items, false);
			}

			public String getLabels()
			{
				StringBuilder sb = new StringBuilder();
				for (int i=0; i < super.items.size();i++){
					Item item =  super.items.get(i);
					sb.append(item.getAction());
					sb.append(" ( ");
					sb.append(item.getLabel());
					sb.append(" ) ");
					if (i<super.items.size()-1){
						sb.append("& ");
					}
				}
				return sb.toString();
			}

			public String getLabelsNoAction()
			{
				return super.getLabels();
			}

			protected void display() {
				print("Consequent - key: " + super.getId() +" - label: " + this.getLabels());
			}

			public Consequent deepCopy(){
				List<Item> itemsDeepCopy = new ArrayList<Item>();
				for (Item i : super.items){
					itemsDeepCopy.add(i.deepCopy());
				}
				return new Consequent(super.id, itemsDeepCopy);
			}
		}


		public class Goal extends Fact{

			public Goal (Integer id, List<Item> items){
				super(id, items);
			}



			protected void display() {
				print("Goal (or Sub Goal) - key: " + super.getId() +" - label: " + super.getLabels());
			}
		}

		public class Item{

			private Integer id;
			private String label;
			private Action action;

			public Item (Action action, Integer id, String label){

				this.id = id;
				this.label = label;
				this.action = action;
			}
		public Item (Integer id, String label){
			this (Action.NONE, id, label);
		}

		public Integer getId() {return this.id;}
		public String getLabel(){return this.label;}
		public void setLabel(String label){this.label = label;}
		public Action getAction(){return this.action;}

		public Item deepCopy(){
			return new Item(this.action, this.id, this.label);
		}
	}

	public class RuleVariableValue {

		String variableName;
		String variableValue;

		public RuleVariableValue(String variableName, String variableValue){
			this.variableName = variableName;
			this.variableValue = variableValue;
		}

		public String getVariableName(){
			return this.variableName;
		}

		public String getVariableValue(){
			return this.variableValue;
		}

		public RuleVariableValue deepCopy(){
			return new RuleVariableValue(this.variableName, this.variableValue);
		}
	}
	public class RuleVariableValueCount {

		RuleVariableValue ruleVariableValue;
		int count;

		public RuleVariableValueCount(RuleVariableValue ruleVariableValue, int count){
			this.ruleVariableValue = ruleVariableValue;
			this.count = count;
		}

		public RuleVariableValue getRuleVariableValue(){
			return this.ruleVariableValue;
		}

		public Integer getCount(){
			return this.count;
		}

		public String toString(){
			return this.ruleVariableValue.getVariableName() + "=" + this.ruleVariableValue.getVariableValue();
		}


	}

	public static class InferenceEngineInputTuple {

		Goal goal;
		RuleBase ruleBase;
		WorkingMemory workingMemory;

		public InferenceEngineInputTuple(RuleBase ruleBase, WorkingMemory workingMemory){
			this.ruleBase = ruleBase;
			this.workingMemory = workingMemory;
		}

		public RuleBase getRuleBase(){
			return this.ruleBase;
		}

		public WorkingMemory getWorkingMemory(){
			return this.workingMemory;
		}
	}

	public enum Action {
		NONE,
	    ADD,
	    DELETE
	}
}
```

Part 3 – Detailed Output

```
***********
```

Case 1

```
***********


***Create the fact list
Fact - key: 1 - label: ( ?  h )
Fact - key: 2 - label: ( a  c )
Fact - key: 3 - label: ( n  s )
Fact - key: 4 - label: ( r  ? )
Fact - key: 5 - label: ( ?  j )
Fact - key: 6 - label: ( e  m )
Fact - key: 7 - label: ( k  i )
Fact - key: 8 - label: ( p  ? )
Fact - key: 9 - label: ( ?  v )

***Create the consequent list
Consequent - key: 100 - label: NONE ( b ) & NONE ( g )
Consequent - key: 6 - label: NONE ( e ) & NONE ( m )
Consequent - key: 8 - label: NONE ( p ) & NONE ( q )
Consequent - key: 2 - label: NONE ( a ) & NONE ( c )
Consequent - key: 3 - label: NONE ( n ) & NONE ( s )
Consequent - key: 7 - label: NONE ( k ) & NONE ( i )

***Create the RuleBase data structure and add rules
RuleBase -> addRules - key: 1
RuleBase -> addRules - key: 2
RuleBase -> addRules - key: 3
RuleBase -> addRules - key: 4
RuleBase -> addRules - key: 5
RuleBase -> addRules - key: 6

***Create the WorkingMemory and add facts
Fact - key: 1 - label: ( f  h )
WorkingMemory -> addFact - key: 1 - Fact:( f  h )
Working Memory State: ( ( f  h )  )
Fact - key: 5 - label: ( d  j )
WorkingMemory -> addFact - key: 5 - Fact:( d  j )
Working Memory State: ( ( f  h ) ( d  j )  )
Fact - key: 9 - label: ( u  v )
WorkingMemory -> addFact - key: 9 - Fact:( u  v )
Working Memory State: ( ( f  h ) ( d  j ) ( u  v )  )
Fact - key: 4 - label: ( r  t )
WorkingMemory -> addFact - key: 4 - Fact:( r  t )
Working Memory State: ( ( f  h ) ( d  j ) ( u  v ) ( r  t )  )

***Create InferenceEngine and start the inference process
***The 'infer' method is executed
***MATCH Rule with antecedent
***SELECT a rule
Rule selected: 3
***Unify rule variables (binds variables with constants)
Fact - key: 4 - label: ( r  ? )
Consequent - key: 8 - label: NONE ( p ) & NONE ( q )
Fact - key: 4 - label: ( r  ? )
Rule Id: 3 - Antecedent r - ? set to h
Rule Id: 3 - Antecedent h - ? set to h
Rule Unified: 3
***ACT - Attempt to add Fact to working memory
```

```
WorkingMemory -> addFact - key: 8 - Fact:( p  q )
Working Memory State: ( ( f  h ) ( d  j ) ( u  v ) ( r  t ) ( p  q )  )
***MATCH Rule with antecedent


***SELECT a rule
Rule selected: 5
***Unify rule variables (binds variables with constants)
Fact - key: 8 - label: ( p  ? )
Consequent - key: 3 - label: NONE ( n ) & NONE ( s )
Fact - key: 8 - label: ( p  ? )
Rule Id: 5 - Antecedent p - ? set to h
Rule Id: 5 - Antecedent h - ? set to h
Rule Unified: 5
***ACT - Attempt to add Fact to working memory
WorkingMemory -> addFact - key: 3 - Fact:( n  s )
Working Memory State: ( ( f  h ) ( d  j ) ( u  v ) ( r  t ) ( p  q ) ( n  s )  )
***MATCH Rule with antecedent
***SELECT a rule
Rule selected: 2
***Unify rule variables (binds variables with constants)
Fact - key: 3 - label: ( n  s )
Consequent - key: 6 - label: NONE ( e ) & NONE ( m )
Rule Unified: 2
***ACT - Attempt to add Fact to working memory
WorkingMemory -> addFact - key: 6 - Fact:( e  m )
Working Memory State: ( ( f  h ) ( d  j ) ( u  v ) ( r  t ) ( p  q ) ( n  s ) ( e  m )
)
***MATCH Rule with antecedent
***SELECT a rule
Rule selected: 6
***Unify rule variables (binds variables with constants)
Fact - key: 9 - label: ( ?  v )
Consequent - key: 7 - label: NONE ( k ) & NONE ( i )
Fact - key: 9 - label: ( ?  v )
Rule Id: 6 - Antecedent f - ? set to f
Rule Id: 6 - Antecedent v - ? set to f
Rule Unified: 6
***ACT - Attempt to add Fact to working memory
WorkingMemory -> addFact - key: 7 - Fact:( k  i )
Working Memory State: ( ( f  h ) ( d  j ) ( u  v ) ( r  t ) ( p  q ) ( n  s ) ( e  m )
( k  i )  )
***MATCH Rule with antecedent
***SELECT a rule
Rule selected: 4
***Unify rule variables (binds variables with constants)
Fact - key: 5 - label: ( ?  j )
Fact - key: 6 - label: ( e  m )
Fact - key: 7 - label: ( k  i )
Consequent - key: 2 - label: NONE ( a ) & NONE ( c )
Fact - key: 5 - label: ( ?  j )
Rule Id: 4 - Antecedent f - ? set to f
Rule Id: 4 - Antecedent j - ? set to f
Fact - key: 6 - label: ( e  m )
Rule Id: 4 - Antecedent e - ? set to f
Rule Id: 4 - Antecedent m - ? set to f
Fact - key: 7 - label: ( k  i )
Rule Id: 4 - Antecedent k - ? set to f
Rule Id: 4 - Antecedent i - ? set to f
Rule Unified: 4
***ACT - Attempt to add Fact to working memory
WorkingMemory -> addFact - key: 2 - Fact:( a  c )
Working Memory State: ( ( f  h ) ( d  j ) ( u  v ) ( r  t ) ( p  q ) ( n  s ) ( e  m )
( k  i ) ( a  c )  )
***MATCH Rule with antecedent
```

```
***SELECT a rule
Rule selected: 1
***Unify rule variables (binds variables with constants)
Fact - key: 1 - label: ( ?  h )
Fact - key: 2 - label: ( a  c )
Consequent - key: 100 - label: NONE ( b ) & NONE ( g )
Fact - key: 1 - label: ( ?  h )


Rule Id: 1 - Antecedent f - ? set to f
Rule Id: 1 - Antecedent h - ? set to f
Fact - key: 2 - label: ( a  c )
Rule Id: 1 - Antecedent a - ? set to f
Rule Id: 1 - Antecedent c - ? set to f
Rule Unified: 1
***ACT - Attempt to add Fact to working memory
WorkingMemory -> addFact - key: 100 - Fact:( b  g )
Working Memory State: ( ( f  h ) ( d  j ) ( u  v ) ( r  t ) ( p  q ) ( n  s ) ( e  m )
( k  i ) ( a  c ) ( b  g )  )
SUCCESS / ALL RULES HAVE BEEN PROVEN!
```

```
***********
```

Case 2

```
***********

***Create the item list

***Create the fact list
Fact - key: 1 - label: ( ?x on ?y )
Fact - key: 2 - label: ( clear ?x )
Fact - key: 3 - label: ( clear ?z )
Fact - key: 4 - label: ( ?x on table )
Fact - key: 5 - label: ( ?x on ?z )

***Create the consequent list
Consequent - key: 1 - label: DELETE ( ?x on ?y ) & DELETE ( clear ?z ) & ADD ( ?x on ?z
) & ADD ( clear ?y )
Consequent - key: 2 - label: DELETE ( ?x on ?y ) & ADD ( ?x on table ) & ADD ( clear ?y
)
Consequent - key: 3 - label: DELETE ( ?x on table ) & DELETE ( clear ?z ) & ADD ( ?x on
?z )

***Create the RuleBase data structure and add rules
RuleBase -> addRules - key: 1
RuleBase -> addRules - key: 2
RuleBase -> addRules - key: 3

***Create the WorkingMemory and add facts
Fact - key: 1 - label: ( clear A )
WorkingMemory -> addFact - key: 1 - Fact:( clear A )
Working Memory State: ( ( clear A )  )
Fact - key: 2 - label: ( A on B )
WorkingMemory -> addFact - key: 2 - Fact:( A on B )
Working Memory State: ( ( clear A ) ( A on B )  )
Fact - key: 3 - label: ( B on C )
WorkingMemory -> addFact - key: 3 - Fact:( B on C )
Working Memory State: ( ( clear A ) ( A on B ) ( B on C )  )
Fact - key: 4 - label: ( C on table )
WorkingMemory -> addFact - key: 4 - Fact:( C on table )
Working Memory State: ( ( clear A ) ( A on B ) ( B on C ) ( C on table )  )

***Create InferenceEngine and start the inference process
***The 'infer' method is executed
***MATCH Rule with antecedent
***SELECT a rule
Rule selected: 2
***Unify rule variables (binds variables with constants)
Fact - key: 1 - label: ( ?x on ?y )
Fact - key: 2 - label: ( clear ?x )
Consequent - key: 2 - label: DELETE ( ?x on ?y ) & ADD ( ?x on table ) & ADD ( clear ?y
)
Fact - key: 1 - label: ( ?x on ?y )
Rule Id: 2 - Antecedent A on ?y - ?x set to A
Fact - key: 2 - label: ( clear ?x )
Rule Id: 2 - Antecedent clear A - ?x set to A
Rule Id: 2 - Consequent A on ?y - ?x set to A
Rule Id: 2 - Consequent A on table - ?x set to A
Fact - key: 1 - label: ( A on ?y )
Rule Id: 2 - Antecedent A on B - ?y set to B
Fact - key: 2 - label: ( clear A )
Rule Id: 2 - Antecedent clear A - ?y set to B
Rule Id: 2 - Consequent A on B - ?y set to B
Rule Id: 2 - Consequent clear B - ?y set to B
Rule Unified: 2
```

```
***ACT - Attempt to add Fact to working memory
Fact - key: 222378475 - label: ( A on B )


WorkingMemory -> removeFact - key: 222378475 - Fact:( A on B )
Working Memory State: ( ( clear A ) ( B on C ) ( C on table )  )
Fact - key: 222378475 - label: ( A on table )
WorkingMemory -> addFact - key: 222378475 - Fact:( A on table )
Working Memory State: ( ( clear A ) ( B on C ) ( C on table ) ( A on table )  )
Fact - key: 222378475 - label: ( clear B )
WorkingMemory -> addFact - key: 222378475 - Fact:( clear B )
Working Memory State: ( ( clear A ) ( B on C ) ( C on table ) ( A on table ) ( clear B
)  )
***MATCH Rule with antecedent
***SELECT a rule
Rule selected: 1
***Unify rule variables (binds variables with constants)
Fact - key: 1 - label: ( ?x on ?y )
Fact - key: 2 - label: ( clear ?x )
Fact - key: 3 - label: ( clear ?z )
Consequent - key: 1 - label: DELETE ( ?x on ?y ) & DELETE ( clear ?z ) & ADD ( ?x on ?z
) & ADD ( clear ?y )
Fact - key: 1 - label: ( ?x on ?y )
Rule Id: 1 - Antecedent B on ?y - ?x set to B
Fact - key: 2 - label: ( clear ?x )
Rule Id: 1 - Antecedent clear B - ?x set to B
Fact - key: 3 - label: ( clear ?z )
Rule Id: 1 - Antecedent clear ?z - ?x set to B
Rule Id: 1 - Consequent B on ?y - ?x set to B
Rule Id: 1 - Consequent B on ?z - ?x set to B
Fact - key: 1 - label: ( B on ?y )
Rule Id: 1 - Antecedent B on C - ?y set to C
Fact - key: 2 - label: ( clear B )
Rule Id: 1 - Antecedent clear B - ?y set to C
Fact - key: 3 - label: ( clear ?z )
Rule Id: 1 - Antecedent clear ?z - ?y set to C
Rule Id: 1 - Consequent B on C - ?y set to C
Rule Id: 1 - Consequent clear C - ?y set to C
Fact - key: 1 - label: ( B on C )
Rule Id: 1 - Antecedent B on C - ?z set to A
Fact - key: 2 - label: ( clear B )
Rule Id: 1 - Antecedent clear B - ?z set to A
Fact - key: 3 - label: ( clear ?z )
Rule Id: 1 - Antecedent clear A - ?z set to A
Rule Id: 1 - Consequent clear A - ?z set to A
Rule Id: 1 - Consequent B on A - ?z set to A
Rule Unified: 1
***ACT - Attempt to add Fact to working memory
Fact - key: 222378475 - label: ( B on C )
WorkingMemory -> removeFact - key: 222378475 - Fact:( B on C )
Working Memory State: ( ( clear A ) ( C on table ) ( A on table ) ( clear B )  )
Fact - key: 222378475 - label: ( clear A )
WorkingMemory -> removeFact - key: 222378475 - Fact:( clear A )
Working Memory State: ( ( C on table ) ( A on table ) ( clear B )  )
Fact - key: 222378475 - label: ( B on A )
WorkingMemory -> addFact - key: 222378475 - Fact:( B on A )
Working Memory State: ( ( C on table ) ( A on table ) ( clear B ) ( B on A )  )
Fact - key: 222378475 - label: ( clear C )
WorkingMemory -> addFact - key: 222378475 - Fact:( clear C )
Working Memory State: ( ( C on table ) ( A on table ) ( clear B ) ( B on A ) ( clear C
)  )
***MATCH Rule with antecedent
***SELECT a rule
Rule selected: 3
***Unify rule variables (binds variables with constants)
```

```
Fact - key: 4 - label: ( ?x on table )
Fact - key: 2 - label: ( clear ?x )
Fact - key: 3 - label: ( clear ?z )
Consequent - key: 3 - label: DELETE ( ?x on table ) & DELETE ( clear ?z ) & ADD ( ?x on
?z )
Fact - key: 4 - label: ( ?x on table )


Rule Id: 3 - Antecedent C on table - ?x set to C
Fact - key: 2 - label: ( clear ?x )
Rule Id: 3 - Antecedent clear C - ?x set to C
Fact - key: 3 - label: ( clear ?z )
Rule Id: 3 - Antecedent clear ?z - ?x set to C
Rule Id: 3 - Consequent C on table - ?x set to C
Rule Id: 3 - Consequent C on ?z - ?x set to C
Fact - key: 4 - label: ( C on table )
Rule Id: 3 - Antecedent C on table - ?z set to B
Fact - key: 2 - label: ( clear C )
Rule Id: 3 - Antecedent clear C - ?z set to B
Fact - key: 3 - label: ( clear ?z )
Rule Id: 3 - Antecedent clear B - ?z set to B
Rule Id: 3 - Consequent clear B - ?z set to B
Rule Id: 3 - Consequent C on B - ?z set to B
Rule Unified: 3
***ACT - Attempt to add Fact to working memory
Fact - key: 222378475 - label: ( C on table )
WorkingMemory -> removeFact - key: 222378475 - Fact:( C on table )
Working Memory State: ( ( A on table ) ( clear B ) ( B on A ) ( clear C )  )
Fact - key: 222378475 - label: ( clear B )
WorkingMemory -> removeFact - key: 222378475 - Fact:( clear B )
Working Memory State: ( ( A on table ) ( B on A ) ( clear C )  )
Fact - key: 222378475 - label: ( C on B )
WorkingMemory -> addFact - key: 222378475 - Fact:( C on B )
Working Memory State: ( ( A on table ) ( B on A ) ( clear C ) ( C on B )  )
SUCCESS / ALL RULES HAVE BEEN PROVEN!
```

```
***********
```

Case 3

```
***********

***Create the item list

***Create the fact list
Fact - key: 1 - label: ( ?x on ?y )
Fact - key: 2 - label: ( clear ?x )
Fact - key: 3 - label: ( clear ?z )
Fact - key: 4 - label: ( ?x on table )
Fact - key: 5 - label: ( ?x on ?z )

***Create the consequent list
Consequent - key: 1 - label: DELETE ( ?x on ?y ) & DELETE ( clear ?z ) & ADD ( ?x on ?z
) & ADD ( clear ?y )
Consequent - key: 2 - label: DELETE ( ?x on ?y ) & ADD ( ?x on table ) & ADD ( clear ?y
)
Consequent - key: 3 - label: DELETE ( ?x on table ) & DELETE ( clear ?z ) & ADD ( ?x on
?z )

***Create the RuleBase data structure and add rules
RuleBase -> addRules - key: 1
RuleBase -> addRules - key: 2
RuleBase -> addRules - key: 3

***Create the WorkingMemory and add facts
Fact - key: 1 - label: ( A on C )
WorkingMemory -> addFact - key: 1 - Fact:( A on C )
Working Memory State: ( ( A on C )  )
Fact - key: 2 - label: ( C on table )
WorkingMemory -> addFact - key: 2 - Fact:( C on table )
Working Memory State: ( ( A on C ) ( C on table )  )
Fact - key: 3 - label: ( clear A )
WorkingMemory -> addFact - key: 3 - Fact:( clear A )
Working Memory State: ( ( A on C ) ( C on table ) ( clear A )  )
Fact - key: 4 - label: ( clear D )
WorkingMemory -> addFact - key: 4 - Fact:( clear D )
Working Memory State: ( ( A on C ) ( C on table ) ( clear A ) ( clear D )  )
Fact - key: 5 - label: ( D on B )
WorkingMemory -> addFact - key: 5 - Fact:( D on B )
Working Memory State: ( ( A on C ) ( C on table ) ( clear A ) ( clear D ) ( D on B )  )
Fact - key: 5 - label: ( B on table )
WorkingMemory -> addFact - key: 5 - Fact:( B on table )
Working Memory State: ( ( A on C ) ( C on table ) ( clear A ) ( clear D ) ( D on B ) (
B on table )  )

***Create InferenceEngine and start the inference process
***The 'infer' method is executed
***MATCH Rule with antecedent
***SELECT a rule
Rule selected: 2
***Unify rule variables (binds variables with constants)
Fact - key: 1 - label: ( ?x on ?y )
Fact - key: 2 - label: ( clear ?x )
Consequent - key: 2 - label: DELETE ( ?x on ?y ) & ADD ( ?x on table ) & ADD ( clear ?y
)
Fact - key: 1 - label: ( ?x on ?y )
Rule Id: 2 - Antecedent A on ?y - ?x set to A
Fact - key: 2 - label: ( clear ?x )
Rule Id: 2 - Antecedent clear A - ?x set to A
Rule Id: 2 - Consequent A on ?y - ?x set to A
Rule Id: 2 - Consequent A on table - ?x set to A
```

```
Fact - key: 1 - label: ( A on ?y )
Rule Id: 2 - Antecedent A on C - ?y set to C


Fact - key: 2 - label: ( clear A )
Rule Id: 2 - Antecedent clear A - ?y set to C
Rule Id: 2 - Consequent A on C - ?y set to C
Rule Id: 2 - Consequent clear C - ?y set to C
Rule Unified: 2
***ACT - Attempt to add Fact to working memory
Fact - key: 222444985 - label: ( A on C )
WorkingMemory -> removeFact - key: 222444985 - Fact:( A on C )
Working Memory State: ( ( C on table ) ( clear A ) ( clear D ) ( D on B ) ( B on table
)  )
Fact - key: 222444985 - label: ( A on table )
WorkingMemory -> addFact - key: 222444985 - Fact:( A on table )
Working Memory State: ( ( C on table ) ( clear A ) ( clear D ) ( D on B ) ( B on table
) ( A on table )  )
Fact - key: 222444985 - label: ( clear C )
WorkingMemory -> addFact - key: 222444985 - Fact:( clear C )
Working Memory State: ( ( C on table ) ( clear A ) ( clear D ) ( D on B ) ( B on table
) ( A on table ) ( clear C )  )
***MATCH Rule with antecedent
***SELECT a rule
Rule selected: 1
***Unify rule variables (binds variables with constants)
Fact - key: 1 - label: ( ?x on ?y )
Fact - key: 2 - label: ( clear ?x )
Fact - key: 3 - label: ( clear ?z )
Consequent - key: 1 - label: DELETE ( ?x on ?y ) & DELETE ( clear ?z ) & ADD ( ?x on ?z
) & ADD ( clear ?y )
Fact - key: 1 - label: ( ?x on ?y )
Rule Id: 1 - Antecedent D on ?y - ?x set to D
Fact - key: 2 - label: ( clear ?x )
Rule Id: 1 - Antecedent clear D - ?x set to D
Fact - key: 3 - label: ( clear ?z )
Rule Id: 1 - Antecedent clear ?z - ?x set to D
Rule Id: 1 - Consequent D on ?y - ?x set to D
Rule Id: 1 - Consequent D on ?z - ?x set to D
Fact - key: 1 - label: ( D on ?y )
Rule Id: 1 - Antecedent D on B - ?y set to B
Fact - key: 2 - label: ( clear D )
Rule Id: 1 - Antecedent clear D - ?y set to B
Fact - key: 3 - label: ( clear ?z )
Rule Id: 1 - Antecedent clear ?z - ?y set to B
Rule Id: 1 - Consequent D on B - ?y set to B
Rule Id: 1 - Consequent clear B - ?y set to B
Fact - key: 1 - label: ( D on B )
Rule Id: 1 - Antecedent D on B - ?z set to A
Fact - key: 2 - label: ( clear D )
Rule Id: 1 - Antecedent clear D - ?z set to A
Fact - key: 3 - label: ( clear ?z )
Rule Id: 1 - Antecedent clear A - ?z set to A
Rule Id: 1 - Consequent clear A - ?z set to A
Rule Id: 1 - Consequent D on A - ?z set to A
Rule Unified: 1
***ACT - Attempt to add Fact to working memory
Fact - key: 222444985 - label: ( D on B )
WorkingMemory -> removeFact - key: 222444985 - Fact:( D on B )
Working Memory State: ( ( C on table ) ( clear A ) ( clear D ) ( B on table ) ( A on
table ) ( clear C )  )
Fact - key: 222444985 - label: ( clear A )
WorkingMemory -> removeFact - key: 222444985 - Fact:( clear A )
Working Memory State: ( ( C on table ) ( clear D ) ( B on table ) ( A on table ) (
clear C )  )
```

```
Fact - key: 222444985 - label: ( D on A )
WorkingMemory -> addFact - key: 222444985 - Fact:( D on A )
Working Memory State: ( ( C on table ) ( clear D ) ( B on table ) ( A on table ) (
clear C ) ( D on A )  )
Fact - key: 222444985 - label: ( clear B )
WorkingMemory -> addFact - key: 222444985 - Fact:( clear B )
Working Memory State: ( ( C on table ) ( clear D ) ( B on table ) ( A on table ) (
clear C ) ( D on A ) ( clear B )  )
***MATCH Rule with antecedent
***SELECT a rule
Rule selected: 3
***Unify rule variables (binds variables with constants)
Fact - key: 4 - label: ( ?x on table )
Fact - key: 2 - label: ( clear ?x )
Fact - key: 3 - label: ( clear ?z )
Consequent - key: 3 - label: DELETE ( ?x on table ) & DELETE ( clear ?z ) & ADD ( ?x on
?z )
Fact - key: 4 - label: ( ?x on table )
Rule Id: 3 - Antecedent C on table - ?x set to C
Fact - key: 2 - label: ( clear ?x )
Rule Id: 3 - Antecedent clear C - ?x set to C
Fact - key: 3 - label: ( clear ?z )
Rule Id: 3 - Antecedent clear ?z - ?x set to C
Rule Id: 3 - Consequent C on table - ?x set to C
Rule Id: 3 - Consequent C on ?z - ?x set to C
Fact - key: 4 - label: ( C on table )
Rule Id: 3 - Antecedent C on table - ?z set to D
Fact - key: 2 - label: ( clear C )
Rule Id: 3 - Antecedent clear C - ?z set to D
Fact - key: 3 - label: ( clear ?z )
Rule Id: 3 - Antecedent clear D - ?z set to D
Rule Id: 3 - Consequent clear D - ?z set to D
Rule Id: 3 - Consequent C on D - ?z set to D
Rule Unified: 3
***ACT - Attempt to add Fact to working memory
Fact - key: 222444985 - label: ( C on table )
WorkingMemory -> removeFact - key: 222444985 - Fact:( C on table )
Working Memory State: ( ( clear D ) ( B on table ) ( A on table ) ( clear C ) ( D on A
) ( clear B )  )
Fact - key: 222444985 - label: ( clear D )
WorkingMemory -> removeFact - key: 222444985 - Fact:( clear D )
Working Memory State: ( ( B on table ) ( A on table ) ( clear C ) ( D on A ) ( clear B
)  )
Fact - key: 222444985 - label: ( C on D )
WorkingMemory -> addFact - key: 222444985 - Fact:( C on D )
Working Memory State: ( ( B on table ) ( A on table ) ( clear C ) ( D on A ) ( clear B
) ( C on D )  )
SUCCESS / ALL RULES HAVE BEEN PROVEN!
```