

## COS30019 - Introduction to Artificial Intelligence

### Assignment 2

---

## Contents

<b>1</b>	<b>Program Instruction</b>	<b>2</b>
1.1	File Organization . . . . .	2
1.2	MacOS User . . . . .	2
1.3	Microsoft Windows 11 User . . . . .	2
1.4	Input Requirements . . . . .	2
1.4.1	Horn Clause Knowledge Base . . . . .	3
1.4.2	Generic Knowledge Base . . . . .	3
<b>2</b>	<b>Implementation For Horn Clause Knowledge Base Checking</b>	<b>3</b>
2.1	Horn Clause . . . . .	3
2.2	Knowledge Base . . . . .	4
2.3	Truth Table Checking . . . . .	5
2.3.1	PLTrueAlpha() Function . . . . .	5
2.3.2	ExtendModel() Function . . . . .	5
2.3.3	PLTrueKB() function . . . . .	6
2.3.4	TTCheckAll() Function . . . . .	7
2.3.5	TT_Entail() Function and Solve() Function . . . . .	7
2.3.6	Result . . . . .	7
2.4	Forward Chaining . . . . .	8
2.4.1	Count Table . . . . .	8
2.4.2	Inferred . . . . .	9
2.4.3	Agenda . . . . .	9
2.4.4	Executing . . . . .	9
2.4.5	Result . . . . .	10
2.5	Backward Chaining . . . . .	10
2.5.1	Proposed Algorithm . . . . .	11
2.5.2	Result . . . . .	12
<b>3</b>	<b>Research: Applying Recursion to Check Generic Knowledge Base</b>	<b>13</b>
3.1	Problem Define . . . . .	13
3.2	Methodology and Implementation . . . . .	13
3.2.1	Logical Expression Coding Presentation . . . . .	13
3.2.2	Advanced Knowledge Base . . . . .	14
3.2.3	Truth Table Checking for generic knowledge base . . . . .	16
3.2.4	Result . . . . .	17

# 1 Program Instruction

## 1.1 File Organization

The submission contains a **.exe** file for Microsoft Windows OS, a **.exec** file for MacOS, and a list of test case **.txt** file and its result. Source code can be found by visiting the **SourceCode** folder.

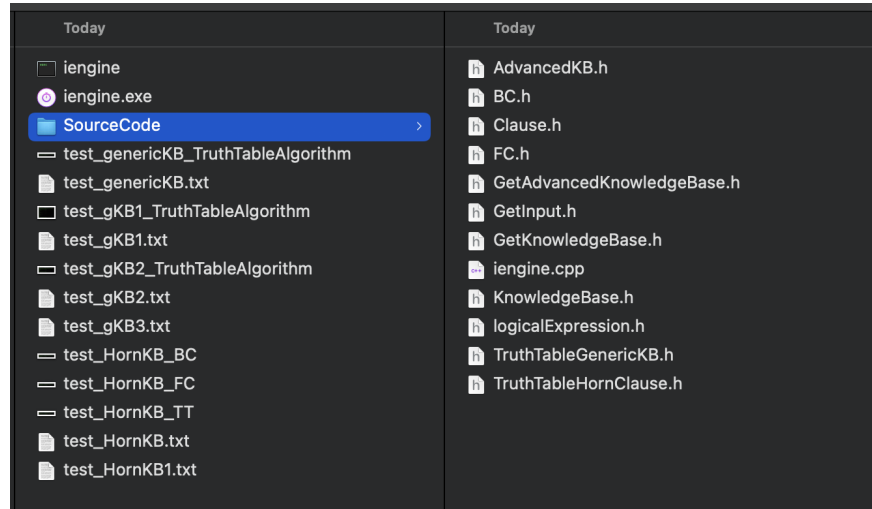


Figure 1: File Organization

## 1.2 MacOS User

For those who use MacOS systems, here is the command line to run the program on the terminal

```
Assignment2 % ./iengine FC test_HornKB.txt
```

The searching method **FC** can be replaced by **BC**, **TT** to deal with knowledge base containing only Horn Clauses.

To deal with generic knowledge, the method is **TTGeneric**:

```
Assignment2 % ./iengine TTGeneric test_genericKB.txt
```

## 1.3 Microsoft Windows 11 User

For those who use the Microsoft Windows Operating Systems, especially Windows 11, here is the command line to run the program in the **cmd**

```
PS D:\Desktop\Assignment2> ./iengine FC test_HornKB.txt
```

```
PS D:\Desktop\Assignment2> ./iengine TTGeneric test_genericKB.txt
```

## 1.4 Input Requirements

The program requires some requirements in order to execute.

### 1.4.1 Horn Clause Knowledge Base

- Each symbol must be separated by a space
- The premise must not contain the bracket
- Every clause in the knowledge base should be followed by a semicolon

Here is a valid input for Horn Clause Knowledge Base:

```
1 TELL
2 p2 => p3; p3 => p1; c => e; b & e => f; f & g => h; p1 => d; p1 & p3 => c; a; b; p2;
3 ASK
4 d
```

Listing 1: Valid Input For Horn Clause

### 1.4.2 Generic Knowledge Base

For generic knowledge base, here are some requirements to be fulfilled:

- Each symbol must be separated by a space
- It should contain the brackets allowing the program recognizes each logical expression precisely.
- Every clause in the knowledge base should be followed by a semicolon

Here is the demonstration of valid and invalid generic knowledge base:

```
1 TELL
2 (a <=> (c => ~d)) & b & (b => a); c; ~f || g;
3 ASK
4 d
```

Listing 2: Valid Input For Generic Knowledge Base

```
1 TELL
2 a <=> c => ~d & b & (b => a); c; ~f || g;
3 ASK
4 d
```

Listing 3: Invalid Input For Generic Knowledge Base

## 2 Implementation For Horn Clause Knowledge Base Checking

### 2.1 Horn Clause

A Horn Clause contains the premise and conclusion part. For example, here is a list of example of Horn Clause:  $(A \wedge B) \Rightarrow C$ ,  $(A \wedge B \wedge C) \Rightarrow D, E$ .

It is easy to conclude that the premise contains a list of symbols and the conclusion is a symbol. For the Horn Clause as propositional symbol (eg.  $E$ ), I will treat this expression as a clause with only conclusion. Figure 2 is my implementation in coding session.

The Clause class has two attribute: a string of conclusion and a list of symbols in the premise.

**Function Explanation:**

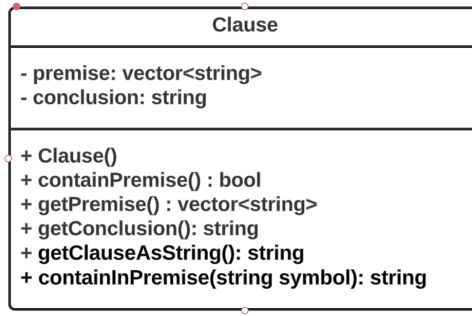


Figure 2: Implementation of Clause in Programming

- `containPremise()`: this function will return true if the clause contains premise
- `getPremise()`: Function returns a list of symbols in the premise
- `getConclusion()`: Function returns the conclusion as a string
- `getClauseAsString()`: Function returns the entire clause as a string
- `containedInPremise(string symbol)`: Function checks whether the given symbol is contained in the premise

## 2.2 Knowledge Base

A Knowledge Base simply contains a list of Horn Clause. Figure 3 is my implementation for knowledge base in programming.

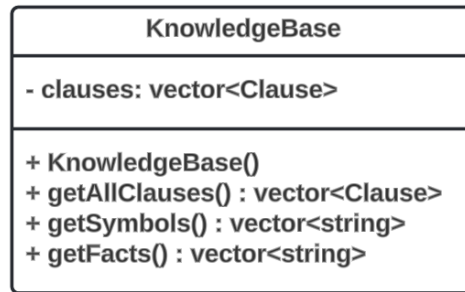


Figure 3: Implementation of Knowledge Base in Programming

The KnowledgeBase class has only one attribute: a list of Horn Clause

### Function Explanation:

- `getAllClauses()`: Function returns the list of Horn Clauses in the knowledge base
- `getSymbols()`: Function returns a list of symbols in the knowledge base.
- `getFacts()`: Function returns the list of propositional symbols. For example, given this knowledge base:  $(A \wedge B) \Rightarrow C$ ,  $(A \wedge B \wedge C) \Rightarrow D$ ,  $E$ ,  $M$  the function will return a list that contains  $E$  and  $M$ . This function will be called by backward chaining algorithm.

## 2.3 Truth Table Checking

The algorithm is provided in the lecture, so, this part does not include the theory. Instead, this section focus on my implementation. Figure 4 is an overview of the Truth Table Checking Agent.

TruthTableHornClause
- kbCount;
+ TruthTableHornClause() - PL_TrueAlpha(string alpha) : bool - PL_TrueKB(KnowledgeBase kb, map<string, bool> model) : bool - ExtendModel(string key, bool value, map<string, bool>) : map<string, bool> - TTCheckAll(KnowledgeBase kb, string alp, vector<string> symbols, map<string, bool> model) : bool - TTEntails(KnowledgeBase kb, string alpha) : bool + solve(KnowledgeBase kb, string alpha) : void

Figure 4: Implementation of Truth Table Checking in Programming

Given Knowledge Base and a query, we need to check whether the query can be entailed from Knowledge Base. By using Truth Table Checking, we will find all models where KB is true, then if "for **ALL** models making our KB true, the value of the query is **always** true", we can conclude that the query can be entailed from the Knowledge Base.

To present a model, I used the data structure hash-map table `map<string, bool>`, for example, supposed that here is our model: A true, B True, C False; then, this is the presented model (hash-map table) in coding: `model = {'A': 1, 'B': 1, 'C': 0}`.

### 2.3.1 PLTrueAlpha() Function

I will begin with the simplest function: `PLTrueAlpha()`. The function returns the value of the query in a given model.

```

1 bool PL_TrueAlpha(string _query, map<string, bool> model){
2     return model[_query];
3 }

```

Listing 4: `PLTrueAlpha()` function

### 2.3.2 ExtendModel() Function

This function takes a key, a value, and a current model, it adds a new pair key-value to that model, then it returns the new model with new key-value pair.

Here is my implementation code

```

1 map<string, bool> extendModel(string key, bool value, map<string, bool> &model){
2     // Adding a new key-value pair using the insert() method
3     model.insert(make_pair(key, value));
4     return model;
5 }

```

Listing 5: `ExtendModel()` function

### 2.3.3 PLTrueKB() function

This function will return True under one condition: with the given model, our KB is true. The lecture does not provide how to check the Horn Clause Knowledge base, so I would like to show my pseudocode. My implementation for this function can be found at the file **TruthTableHorn-Clause.h**, in this report, I just provide my pseudocode for this function.

---

**Algorithm 1** PL\_TrueKB(knowledgeBase, model)

---

```
1: function PL_TRUEKB(knowledgeBase, model)
2:   listOfClause  $\leftarrow$  getting from knowledgeBase
3:   output  $\leftarrow$  True
4:   for clause in listOfClause do
5:     valueOfClauseWithThisModel  $\leftarrow$  True
6:     if the clause contains only conclusion then
7:       valueOfClauseWithThisModel  $\leftarrow$  model[conclusionOfClause]
8:     else
9:       listOfSymbolsInPremise  $\leftarrow$  getting from clause
10:      conclusionOfClause  $\leftarrow$  getting from clause
11:      if model[conclusionOfClause] is True then
12:        valueOfClauseWithThisModel  $\leftarrow$  True
13:      else
14:        conjunction  $\leftarrow$  True
15:        for symbol in listOfSymbolsInPremise do
16:          conjunction  $\leftarrow$  conjunction And model[symbol]
17:        end for
18:        valueOfClauseWithThisModel  $\leftarrow$  valueOfClauseWithThisModel And !(conjunction)
19:      end if
20:    end if
21:    output  $\leftarrow$  output And valueOfClauseWithThisModel
22:  end for
23:  return output
24: end function
```

---

This Idea is inspired by the fact  $(A \rightarrow B)$  is equivalent to  $(\neg A \vee B)$

The algorithm traverses every clause in the knowledge base, if the clause is only a propositional symbol, then, the value of this clause is the value of the symbol in the model. Or else: if the clause contains premise and conclusion, we have 2 cases to be considered:

- **Case 1:** if the value of the conclusion is true, then definitely the value of the clause is true.
- **Case 2:** if the value of the conclusion is false, we will get the conjunction of all symbols in the premise, if the conjunction value is false, the value of the clause is true.

Above is the explanation for pseudocode from lines 4-22.

In my coding, to get the list of clause from knowledge base (line 2 in the pseudocode above), I called the `getAllClauses()` function as below:

```
1 vector<Clause> listOfClause = kb.getAllClauses();
```

Listing 6: Getting the list of clauses from Knowledge Base

In line 6, to check whether the current clause contains only conclusion, the program called the `containPremise()` from the **Clause** class as below:

```

1 if (currentClause.containPremise() == false){
2     valueOfCurrentClauseWithThisModel = model[currentClause.getConclusion()];
3 }

```

Listing 7: Coding for lines 6-7

To get the list of symbols in the premise and the conclusion of the clause, I used `getPremise()` method and `getConclusion()` method from the **Clause** class.

#### 2.3.4 TTCheckAll() Function

My full implementation for this function can be found at the file **TruthTableHornClause.h**, here is the pseudocode demonstrating my idea for this function.

---

#### Algorithm 2 TT\_CheckAll(knowledgeBase, alpha, listOfSymbols, model)

---

```

1: function TT_CHECKALL(knowledgeBase, alpha, listOfSymbols, model)
2:   if listOfSymbols is Empty then
3:     Call PL_TrueKB(knowledgeBase, model)
4:     if the output from PL_TrueKB is True then
5:       Call PL_TrueAlpha(alpha, model)
6:     else
7:       return True
8:     end if
9:   else
10:     $P \leftarrow$  popping the first element from listOfSymbols
11:     $newModelWithPTrue \leftarrow$  Call ExtendModel( $P$ , True, model)
12:     $newModelWithPFalse \leftarrow$  Call ExtendModel( $P$ , False, model)
13:    return TT_CheckAll(knowledgeBase, alpha, listOfSymbols, newModelWithPTrue)
14:  And TT_CheckAll(knowledgeBase, alpha, listOfSymbols, newModelWithPFalse)
15:  end if
16: end function

```

---

This algorithm is provided from the lecture, the `newModelWithPTrue` makes me confused at first, so I will show my implementation as below:

```

1 map<string, bool> restUnionWithPTrue = extendModel(P, true, model);

```

Listing 8: Extend model

The `ExtendModel()` function will return a new model with the value of **P** and its boolean value, where:

- **P** is the first element in the symbols list
- `model` is the model but without the value of **P**

#### 2.3.5 TT\_Entail() Function and Solve() Function

`TT_Entails()` simply calls the `TT_CheckAll()` and `Solve()` calls `TT_Entails()`. So I will not discuss it right here.

#### 2.3.6 Result

```

Assignment2 — -zsh — 116x22
(base) tamvo@Tams-MacBook-Air Assignment2 % ./iengine TT test_HornKB.txt
KB: p2 => p3; p3 => p1; c => e; b & e => f; f & g => h; p1 => d; p1 & p3 => c; a; b; p2;
Query: d
YES: 3
(base) tamvo@Tams-MacBook-Air Assignment2 %

```

Figure 5: Result for Truth Table Checking Algorithm

## 2.4 Forward Chaining

The pseudocode is provided in the lecture as Figure 6 below, this section will show my implementation for each line of pseudocodes.

```

function PL-FC-ENTAILS?(KB, q) returns true or false
    count ← a table, where count[c] is the number of symbols in c's premise
    inferred ← a table, where inferred[s] is initially false for all s
    agenda ← a queue of symbols, initially symbols known to be true in KB
    while agenda is not empty do
        p ← Pop(agenda)
        if p = q then return true
        if inferred[p] = false then
            inferred[p] ← true
            for each clause c in KB where p is in c.premise do
                decrement count[c]
                if count[c] = 0 then add c.conclusion to agenda
    return false

```

Figure 6: Pseudocode for Forward Chaining Algorithm

### 2.4.1 Count Table

Again, I used the data structure hash-map table for this Count Table. The key is a string of propositional logic and the value is an integer.

```

1 // define count
2 map<string, int> count;

```

Listing 9: Define Count Table

Then I traversed every clause and called the `getClauseAsString()` function to get the key for the clause. For the number of symbols in the premise, I called `getPremise()` function and count the number of element to get the value for this key:pair value.

```

1 for (int i = 0; i < listOfAllClause.size(); i++){
2     Clause currentClause = listOfAllClause[i];
3     string stringClause = currentClause.getClauseAsString();
4     count.insert(make_pair(stringClause, currentClause.getPremise().size()));
5 }

```

Listing 10: Initialize Count Table



## 2.4.2 Inferred

The `map<string, bool>` data structure is applied again. All I did is traversing the list taken from **Knowledge Base** function: `getSymbols()` and inserting key-value pair to inferred hash map table.

```
1  map<string, bool> inferred;
2  // adding to inferred
3  for (int i = 0; i < listOfAllSymbol.size(); i++){
4      string currentSymbol = listOfAllSymbol[i];
5      inferred.insert(make_pair(currentSymbol, false));
6  }
```

Listing 11: Define Inferred

## 2.4.3 Agenda

For Agenda, I applied the queue data structure as below:

```
1  // define ageneda
2  queue<string> agenda;
```

Listing 12: Define Agenda

## 2.4.4 Executing

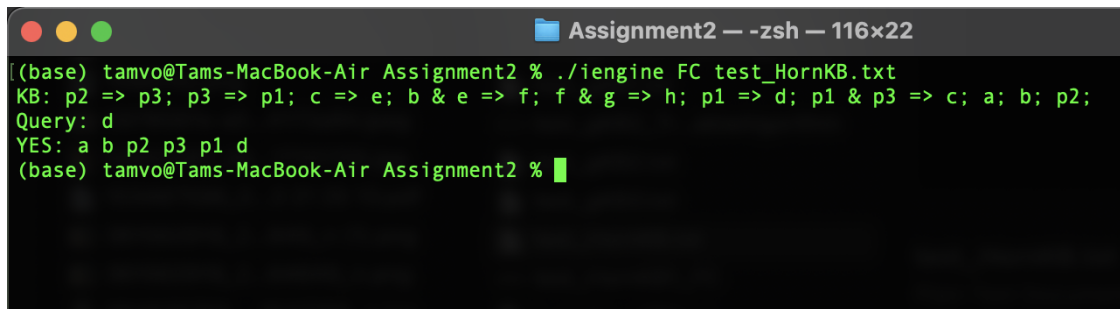
Here is my implementation in coding for the rest of the pseudocode:

```
1  while (agenda.empty() == false){
2      string currentSymbol = agenda.front();
3      agenda.pop();
4      propositionalSymbol.push(currentSymbol);
5
6      if (currentSymbol == query){
7          return true;
8      }
9
10     if (inferred[currentSymbol] == false){
11         inferred[currentSymbol] = true;
12
13         // traverse every clause
14         for (int i = 0; i < listOfAllClause.size(); i++){
15             Clause currentClause = listOfAllClause[i];
16
17             /// checking "currentSymbol" is in the premise of the current Clause or not
18             if (currentClause.containedInPremise(currentSymbol)){
19                 count[currentClause.getClauseAsString()]--;
20                 if (count[currentClause.getClauseAsString()] == 0){
21                     agenda.push(currentClause.getConclusion());
22                 }
23             }
24         }
25     }
26 }
27
28 return false;
```

Listing 13: Checking the queue/agenda

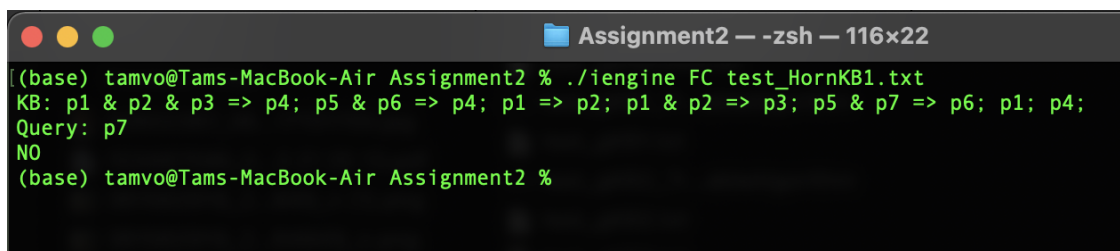
### 2.4.5 Result

Here is some outputs using Forwarding Chaining Algorithm:



```
Assignment2 — -zsh — 116x22
(base) tamvo@Tams-MacBook-Air Assignment2 % ./iengine FC test_HornKB.txt
KB: p2 => p3; p3 => p1; c => e; b & e => f; f & g => h; p1 => d; p1 & p3 => c; a; b; p2;
Query: d
YES: a b p2 p3 p1 d
(base) tamvo@Tams-MacBook-Air Assignment2 %
```

Figure 7: Result 1



```
Assignment2 — -zsh — 116x22
(base) tamvo@Tams-MacBook-Air Assignment2 % ./iengine FC test_HornKB1.txt
KB: p1 & p2 & p3 => p4; p5 & p6 => p4; p1 => p2; p1 & p2 => p3; p5 & p7 => p6; p1; p4;
Query: p7
NO
(base) tamvo@Tams-MacBook-Air Assignment2 %
```

Figure 8: Result 2

## 2.5 Backward Chaining

The lecture does not provide the pseudocode of backward chaining algorithm, so I will propose my pseudocode to solve the problem. The algorithm applied the recursion concept to find out the solution. You can find the coding implementation in the file **BC.h**.

The data structure hash-map is used again in backward chaining. But first we need to figure out what is fact in a given knowledge base. Supposed that we have this knowledge base:  $(A \wedge B) \Rightarrow C$ ,  $(A \wedge B \wedge C) \Rightarrow D$ ;  $E$ ;  $M$ . We all know the "FACT" that  $E$  and  $M$  are always **True** in any models satisfying our Knowledge Base. That is, I will use a hash-map to store the Fact lists. In the given example, our fact list is `map<string, bool> fact = { 'E' : True, 'M' : True }`. Here is my implementation:

```
1 map<string, bool> fact;
2 vector<string> factList = kb.getFact();
3 for (int i = 0; i < factList.size(); i++){
4     fact.insert(make_pair(factList[i], true));
5 }
```

Listing 14: Define Fact

### 2.5.1 Proposed Algorithm

---

**Algorithm 3** BC\_Entail(knowledgeBase, query)

---

```
1: function BC_ENTAIL(knowledgeBase, query)
2:   if query in fact then
3:     return True
4:   end if
5:   listOfClauses  $\leftarrow$  getting from knowledgeBase
6:   output  $\leftarrow$  True
7:   for clause in listOfClauses do
8:     if query is a conclusion of clause then
9:       premiseList  $\leftarrow$  getting from clause
10:      for symbol in premiseList do
11:        output  $\leftarrow$  output And BC_Entail(knowledgeBase, query)
12:        if output is True then
13:          fact  $\leftarrow$  insert new key:value (query: True)
14:        end if
15:      end for
16:    end if
17:  end for
18:  return fact[query]
19: end function
```

---

#### Explanation

In this algorithm, we recursively check the premise until we reach the fact. Given this Knowledge Base:  $A; B; (A \wedge B) \Rightarrow L; (A \wedge P) \Rightarrow L; (P \wedge L) \Rightarrow M; (L \wedge M) \Rightarrow P; P \Rightarrow Q$ . We are interested in whether  $Q$  can be entailed from KB, then we need to check whether  $P$  can be proved. But to check whether  $P$  can be proved, we need to check whether  $L$  and  $M$  can be proved, doing so until we reach our fact, which contains  $A$  and  $B$ . By applying the concept of recursion, I solved this problem.

In line 2, which is the base case of recursion, to check whether the fact contains the query, I used the built-in function by C++, here is my implementation:

```
1 if (fact.count(query) > 0){
2   return true;
3 }
```

Listing 15: Checking whether fact contains query

In line 5, to get the list of clauses from the knowledge base, I called the method `getAllClauses()` from `KnowledgeBase` class. This method is demonstrated before, so you can check again. Then, in line 8, to check whether the query is a conclusion of the given clause, I used the `getConclusion()` method as below:

```
1 if (currentClause.getConclusion() == query){
2   //coding
3 }
```

Listing 16: Checking whether the query is the conclusion of the given clause

If the query is the conclusion of the current clause, then, in the conditional statement above, I created a list of symbols in the premise by calling the method `getPremise()` from the **Clause** class and recursively checked until I reached the fact as below:

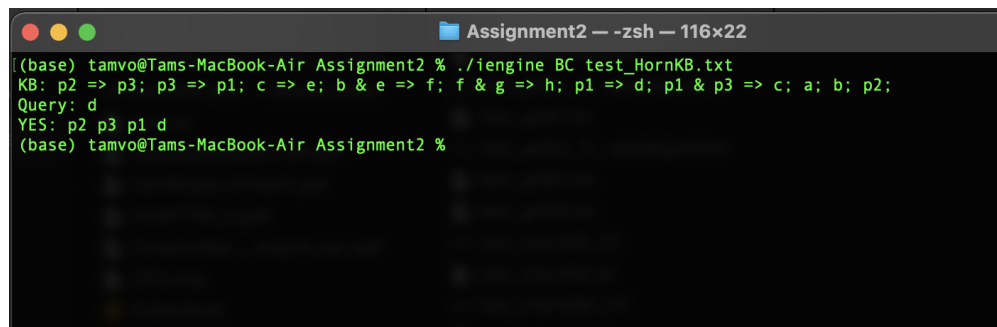
```

1 bool output = true;
2   for (int i = 0; i < listOfClauses.size(); i++){
3       Clause currentClause = listOfClauses[i];
4       if (currentClause.getConclusion() == query){
5
6           // traverse all symbols in the premise
7           vector<string> premiseList = currentClause.getPremise();
8           for (int j = 0; j < premiseList.size(); j++){
9               output = output && BC_Entail(kb, premiseList[j]);
10              if (output){
11                  fact.insert(make_pair(query, true));
12              }
13          }
14      }
15  }
16  return fact[query];

```

Listing 17: Implementation from lines 7 to 18

## 2.5.2 Result

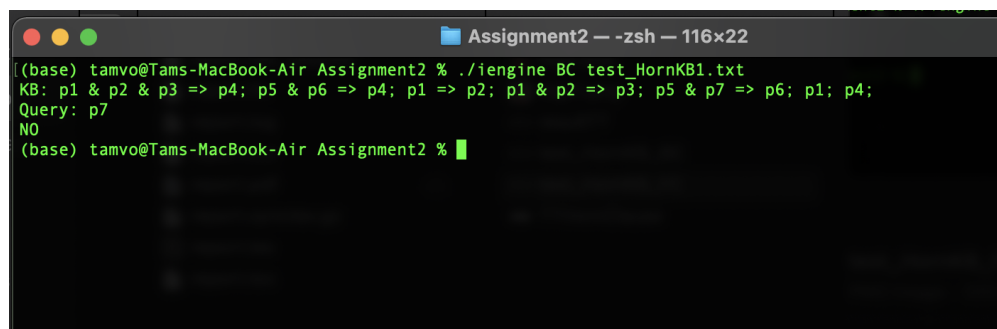


```

Assignment2 - -zsh - 116x22
(base) tamvo@Tams-MacBook-Air Assignment2 % ./engine BC test_HornKB.txt
KB: p2 => p3; p3 => p1; c => e; b & e => f; f & g => h; p1 => d; p1 & p3 => c; a; b; p2;
Query: d
YES: p2 p3 p1 d
(base) tamvo@Tams-MacBook-Air Assignment2 %

```

Figure 9: Result 1



```

Assignment2 - -zsh - 116x22
(base) tamvo@Tams-MacBook-Air Assignment2 % ./engine BC test_HornKB1.txt
KB: p1 & p2 & p3 => p4; p5 & p6 => p4; p1 => p2; p1 & p2 => p3; p5 & p7 => p6; p1; p4;
Query: p7
NO
(base) tamvo@Tams-MacBook-Air Assignment2 %

```

Figure 10: Result 2

## 3 Research: Applying Recursion to Check Generic Knowledge Base

### 3.1 Problem Define

The lecture provides algorithms: Truth Table Checking, Forward Chaining, Backward Chaining to check whether a propositional symbol can be entailed from a given knowledge base. However, these algorithms are used for the problem with Horn Clause, if the knowledge base is a general form, the problem can not be solved.

For example, given this knowledge base:

$$(a \iff (c \Rightarrow \neg d)) \wedge b \wedge (b \Rightarrow a); c; \neg f \vee g;$$

The question is whether  $D$  can be entailed from the given Knowledge Base. As human being, we will generate all possible combination, there are 6 unique symbols, so there are  $2^6 = 64$  models in total and check each model respectively. But for a model with 10 different symbols, it is impossible to traverse  $2^{10} = 1024$  models. This research will cover the explanation of a program checking whether a query can be entailed from a given generic knowledge base.

### 3.2 Methodology and Implementation

#### 3.2.1 Logical Expression Coding Presentation

The atomic element in a knowledge base is a logical expression, for example, here is the list of logical expression:  $(a \iff (c \Rightarrow \neg d))$ ,  $a$ ,  $(b \Rightarrow a)$ , etc. Here is the implementation of class `logicalExpression`:

logicalExpression
<ul style="list-style-type: none"><li>- originalString : string</li><li>- connective : string</li><li>- listOfChildren : vector&lt;logicalExpression&gt;</li><li>- onlySymbol : bool</li><li>- listOfSymbols : string</li></ul>
<ul style="list-style-type: none"><li>+ logicalExpression()</li><li>- removeWhiteSpace() : string</li><li>- split() : void</li><li>+ getOriginalExpression() : string</li><li>+ getChildren() : vector&lt;logicalExpression&gt;</li><li>+ containOnlySymbol() : bool</li><li>+ gettingConnective() : string</li></ul>

Figure 11: `logicalExpression` class in programming

The class has 5 attribute:

- `originalString` is the logical expression as a string
- `connective` is the main connective of the logical expression (Eg.  $\Rightarrow$ ,  $\iff$ , etc.)
- `listOfChildren` is a list of logical expression children (Eg.  $(a \iff (c \Rightarrow \neg d))$  has 2 children, which are:  $a$  and  $(c \Rightarrow \neg d)$ )
- `onlySymbol` is true if the current logical expression is a propositional symbol

- `listOfSymbols` is a list that contains all symbols in the logical expression

### Function explanation

- `split()` functions finds the children and connective of current logical expression, then adds logical expression child to `listOfChildren`
- `getOriginalExpression()` function returns the logical expression as a string.
- `getChildren()` function returns `listOfChildren`
- `containOnlySymbol()` function returns true if the the logical expression is a propositional symbol
- `gettingConnective()` function returns the main connective of the logical expression

For example, the expression  $(a \iff (c \Rightarrow \neg d))$  has these properties:

- `originalString` is  $(a \iff (c \Rightarrow \neg d))$
- `connective` is  $\iff$
- `listOfChildren` contains  $a$  and  $(c \Rightarrow \neg d)$ . Both of them are `logicalExpression` datatype
- `onlySymbol` is `False`
- `listOfSymbols` is a list that contains:  $a, c, d$

### 3.2.2 Advanced Knowledge Base

Advanced knowledge base contains a list of logical expressions and a function to return all symbols in the knowledge base.

AdvancedKB
- <b>vector&lt;logicalExpression&gt; logicalExpressions</b>
<b>+ AdvancedKB()</b> <b>+ getAllLogicalExpression() : vector&lt;logicalExpression&gt;</b> <b>- getSymbolsInEachLE() : vector&lt;string&gt;</b> <b>+ getSymbolsOfAdvanceKB() : vector&lt;string&gt;</b>

Figure 12: **AdvancedKB** class in programming

This section will discuss how to get every symbol in the knowledge base, but first, the pseudocode to get all symbol in a logical expression will be shown:

---

**Algorithm 4** `getSymbolsInEachLE()` getting all symbols in a logical expression

---

```

1: function GETSYMBOLSINEACHLE(logicalExpression)
2:   output  $\leftarrow \emptyset$ 
3:   if logicalExpression is a propositional symbol then
4:     Add logicalExpression to output
5:     return output
6:   end if
7:   for child in logicalExpression do
8:     listOfSymbols  $\leftarrow$  getSymbolsInEachLE(child)
9:     for symbol in listOfSymbols do
10:      if symbol not in output then
11:        add symbol to output
12:      end if
13:    end for
14:  end for
15:  return output
16: end function

```

---

For lines 3-5, the method `containOnlySymbol()` and `getOriginalExpression()` in the `logicalExpression` class will be called as below:

```

1 if (_logicalExpression.containOnlySymbol()){
2   output.push_back(_logicalExpression.getOriginalExpression());
3   return output;
4 }

```

Listing 18: Coding for lines 3-5

Then, the algorithm will traverse every child of the logical expression and recursively called the `getSymbolsInEachLE()` until reaching the base case: the logical expression is a propositional symbol.

Here is the coding implementation:

```

1 for (int i = 0; i < _logicalExpression.getChildren().size(); i++){
2   logicalExpression currentChild = _logicalExpression.getChildren()[i];
3   vector<string> temp = getSymbolsInEachLE(currentChild);
4
5   for (int j = 0; j < temp.size(); j++){
6
7     // ensure the result is already in the output
8     if (count(output.begin(), output.end(), temp[j]) == 0){
9       output.push_back(temp[j]);
10    }
11  }
12 }
13 return output;

```

Listing 19: Coding for lines 7-15

The method `getSymbolsOfAdvancedKB()` will traverse every logical expression in the knowledge base and call the `getSymbolsInEachLE()` method to extract a list of symbols.

### 3.2.3 Truth Table Checking for generic knowledge base

The algorithm will be the same as Truth Table Checking for Horn Clause. However, the function `PL_TrueKB()` is different and there will be a another function handling the task model checking. Below is the pseudocode for `LogicalExpressionChecking()` function:

---

**Algorithm 5** `LogicalExpressionChecking(expression, model)`

---

```
1: function LOGICALEXPRESSIONCHECKING(expression, model)
2:   if logicalExpression is a propositional symbol then
3:     return model[logicalExpression]
4:   end if
5:   listOfChildren  $\leftarrow$  getting from logicalExpression
6:   connective  $\leftarrow$  getting from logicalExpression
7:   if listOfChildren has 2 children then
8:     leftChild  $\leftarrow$  listOfChildren[0]
9:     rightChild  $\leftarrow$  listOfChildren[1]
10:    valueOfLeftChild  $\leftarrow$  LogicalExpressionChecking(leftChild, model)
11:    valueOfRightChild  $\leftarrow$  LogicalExpressionChecking(rightChild, model)
12:    if connective is biconditional then
13:      if valueOfLeftChild is True and valueOfRightChild is True then
14:        return True
15:      else if valueOfLeftChild is False And valueOfRightChild is False then
16:        return True
17:      end if
18:      return False
19:    else if connective is implication then
20:      return not valueOfLeftChild Or valueOfRightChild
21:    else if connective is disjunction then
22:      return valueOfLeftChild Or valueOfRightChild
23:    else if connective is conjunction then
24:      return valueOfLeftChild And valueOfRightChild
25:    end if
26:  else
27:    return not LogicalExpressionChecking(listOfChildren[0], model)
28:  end if
29: end function
```

---

The coding implementation can be found in the file **TruthTableGenericKB.h**. Look at the algorithm, there is a question: Why does a logical expression have 2 children? To answer this question, if there is a connective in the given expression, my implementation will consider 2 cases:

- the connective is **negation**: the whole part after the negation sign is the only child of the original expression
- the connective is **biconditional, implication, conjunction, disjunction**: there are always 2 children. The first child is the expression before the connective, the second child is the expression after the connective.

For example, given this logical expression:  $(a \Rightarrow b) \wedge b \wedge (c \Longleftrightarrow \neg d)$ , the given logical expression has these properties:

- First Child:  $(a \Rightarrow b)$



- Second Child:  $b \wedge (c \iff \neg d)$
- Connective: Conjunction

Then the function `PL_TrueKB(advancedKB, model)` will traverse every clause in the knowledge base and call the `LogicalExpressionChecking()` function.

---

**Algorithm 6** `PL_TrueKB(advancedKB, model)`

---

```

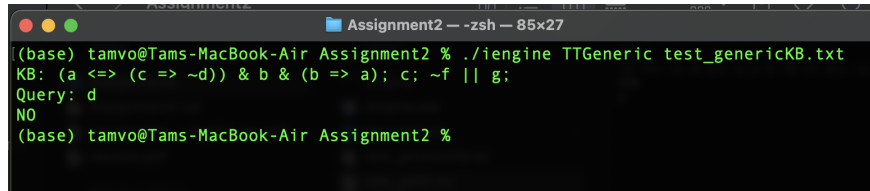
1: function PL_TrueKB(advancedKB, model)
2:   listOfLogicalExpression  $\leftarrow$  getting from advancedKB
3:   output  $\leftarrow$  True
4:   for expression in listOfLogicalExpression do
5:     valueOfExpression  $\leftarrow$  Call LogicalExpressionChecking(expression, model)
6:     output  $\leftarrow$  output And valueOfExpression
7:   end for
8:   return output
9: end function

```

---

### 3.2.4 Result

Here are some results after applying my algorithm:

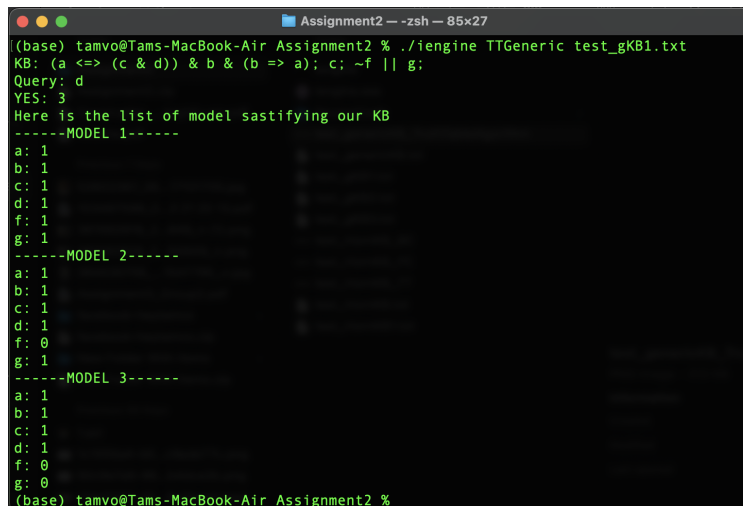


```

Assignment2 - zsh - 85x27
(base) tamvo@Tams-MacBook-Air Assignment2 % ./engine TTGeneric test_genericKB.txt
KB: (a <=> (c => ~d)) & b & (b => a); c; ~f || g;
Query: d
NO
(base) tamvo@Tams-MacBook-Air Assignment2 %

```

Figure 13: Result 1



```

Assignment2 - zsh - 85x27
(base) tamvo@Tams-MacBook-Air Assignment2 % ./engine TTGeneric test_gKB1.txt
KB: (a <=> (c & d)) & b & (b => a); c; ~f || g;
Query: d
YES: 3
Here is the list of model sastifying our KB
-----MODEL 1-----
a: 1
b: 1
c: 1
d: 1
f: 1
g: 1
-----MODEL 2-----
a: 1
b: 1
c: 1
d: 1
f: 0
g: 1
-----MODEL 3-----
a: 1
b: 1
c: 1
d: 1
f: 0
g: 0
(base) tamvo@Tams-MacBook-Air Assignment2 %

```

Figure 14: Result 2