Assignment 2 - Propositional Logic Inference Engine

TEAM MEMBERS:

Chethan Bhaskar – 104069283

Meezan Hussain - 104330015

ESP Group ID – COS30019_A02_T018

Table of Contents

Instructions on running the program

The program is implemented in Python Programming language. There are 3 inference methods implemented in our program:

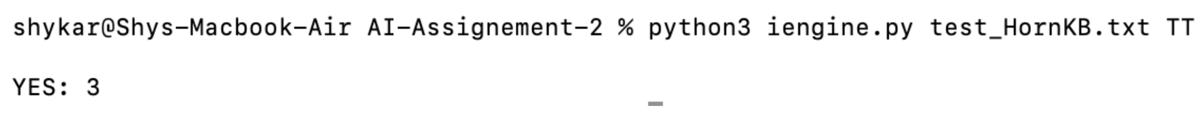
- Truth Table Method
- Forward Chaining
- Backward Chaining

The file *iengine.py* has been implemented in the program, so the program can be run using command **python3 iengine.py {file_name} {method}**

Below are the instructions on running each of the inference methods:

1. Truth Table Method (**Can handle Generic KBs and Horn KBs**)
   Command: **python3 iengine.py {file_name} TT**

   For the test case test_HornKB.txt -
   Command: **python3 iengine.py test_HornKB.txt TT**



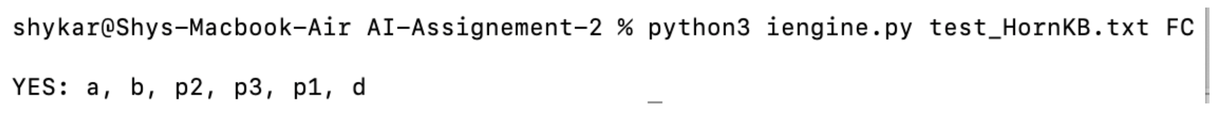
```
shykar@Shys-Macbook-Air AI-Assignement-2 % python3 iengine.py test_HornKB.txt TT

YES: 3
```

Fig.1 – Truth Table Method Running on Terminal

2. Forward Chaining Method (**Can handle Horn KBs only**)
   Command: **python3 iengine.py {file_name} FC**

   For the test case test_HornKB.txt -
   Command: **python3 iengine.py test_HornKB.txt FC**



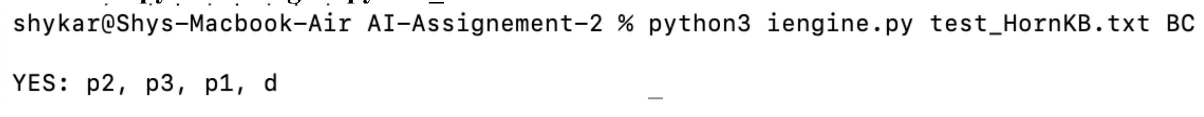
```
shykar@Shys-Macbook-Air AI-Assignement-2 % python3 iengine.py test_HornKB.txt FC

YES: a, b, p2, p3, p1, d
```

Fig.2 – Forward Chaining Method Running on Terminal

3. Backward Chaining Method (**Can handle Horn KBs only)**
   Command: **python3 iengine.py {file_name} BC**

   For the test case test_HornKB.txt -
   Command: **python3 iengine.py test_HornKB.txt BC**



```
shykar@Shys-Macbook-Air AI-Assignement-2 % python3 iengine.py test_HornKB.txt BC

YES: p2, p3, p1, d
```

Fig.3 – Backward Chaining Method Running on Terminal

Introduction to Propositional Logic Inference Engine

In the realm of artificial intelligence (AI), the Inference Engine serves as a fundamental component for making logical deductions and drawing conclusions from available information. Its primary

purpose is to apply logical reasoning to a given set of facts or knowledge, allowing AI systems to make informed decisions, solve problems, and perform tasks intelligently.
Propositional logic is a branch of logic that deals with propositions or statements expressing truth values. Propositional logic provides a formal framework for representing and reasoning about knowledge in AI systems. It enables the representation of facts, rules, and relationships between entities in a concise and unambiguous manner.

Key Terms:

1. **Logical Connectives:** Logical connectives are fundamental symbols used to form compound propositions from simpler propositions. Common logical connectives include AND (∧), OR (∨), NOT (¬), IMPLICATION (=>), and BICONDITIONAL (⇔). These connectives allow for the expression of complex relationships and conditions within a knowledge base.
2. **Truth Table:** A truth table is a tabular representation of all possible combinations of truth values for a given set of propositional symbols and their corresponding truth values under different scenarios. Truth tables provide a systematic way to evaluate the truth value of compound propositions and determine their validity.
3. **Models of a Sentence:** In the context of propositional logic, a model of a sentence refers to an assignment of truth values to propositional symbols that satisfies the truth value of the entire sentence. A sentence is considered true in a model if it evaluates to true under that assignment of truth values. Models play a crucial role in determining the validity of logical statements and inferences.

Significance of Propositional Logic in AI:

Propositional logic serves as the foundation for many AI applications and reasoning tasks. Its significance lies in its ability to represent knowledge in a formal, structured manner that is amenable to automated reasoning. By employing propositional logic, AI systems can:

- **Make Inferences:** AI systems use propositional logic to derive new knowledge from existing facts and rules, enabling them to draw conclusions and make informed decisions.
- **Problem Solving:** Propositional logic provides a framework for formulating and solving problems in various domains, such as planning, scheduling, and diagnostics.
- **Knowledge Representation:** Propositional logic allows for the compact and precise representation of knowledge, facilitating efficient storage, retrieval, and manipulation of information in AI systems.

In summary, the Inference Engine, built upon the principles of propositional logic, plays a central role in AI by enabling automated reasoning, problem-solving, and knowledge representation. Understanding key concepts such as logical connectives, truth tables, and models of sentences is essential for effective utilization of the Inference Engine in AI applications.

Inference Methods Introduction

1. Truth Table (TT) Method: The Truth Table method is a brute-force approach to determining the validity of a logical statement. It works by exhaustively enumerating all possible combinations of truth values for the propositional symbols in the knowledge base (KB) and evaluating the truth value of the query (q) under each combination. If the query evaluates to true for every row where the KB is true, then q is considered entailed by the KB.

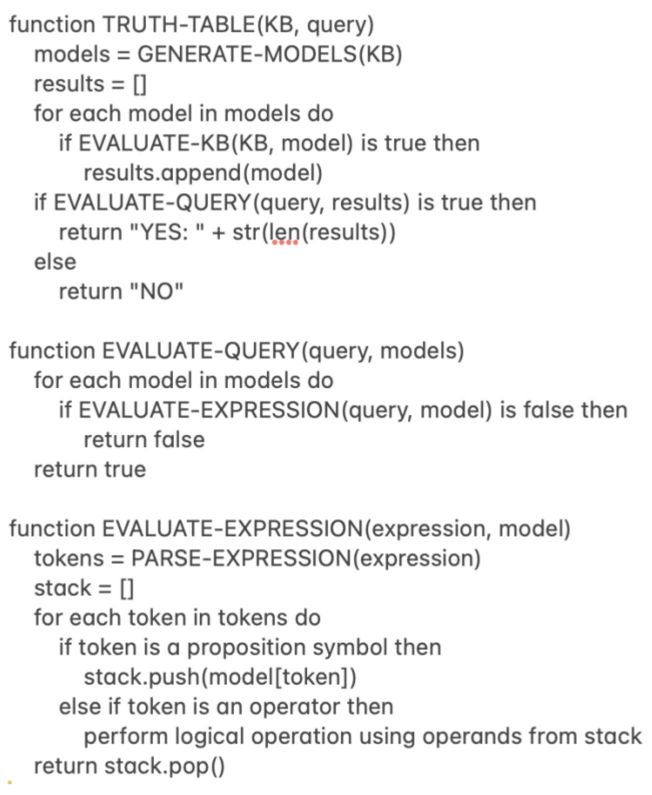
2. Forward Chaining (FC) Method: Forward Chaining is a heuristic-based inference method that starts with the known facts in the knowledge base and iteratively applies inference rules to derive new conclusions until the query is either proven true or no further inferences can be made. It operates by repeatedly applying modus ponens to deduce new facts from existing ones, propagating information forward through the knowledge base.

3. Backward Chaining (BC) Method: Backward Chaining is a goal-driven inference method that starts with the query and recursively works backward through the knowledge base to find evidence that supports the query. It begins by attempting to prove the query directly from the known facts, and if that fails, it recursively attempts to prove the premises of rules that entail the query.

Implementation of Inference Methods

Truth Table Method:

Pseudocode:



```
function TRUTH-TABLE(KB, query)
    models = GENERATE-MODELS(KB)
    results = []
    for each model in models do
        if EVALUATE-KB(KB, model) is true then
            results.append(model)
    if EVALUATE-QUERY(query, results) is true then
        return "YES: " + str(len(results))
    else
        return "NO"

function EVALUATE-QUERY(query, models)
    for each model in models do
        if EVALUATE-EXPRESSION(query, model) is false then
            return false
    return true

function EVALUATE-EXPRESSION(expression, model)
    tokens = PARSE-EXPRESSION(expression)
    stack = []
    for each token in tokens do
        if token is a proposition symbol then
            stack.push(model[token])
        else if token is an operator then
            perform logical operation using operands from stack
    return stack.pop()
```

Fig.4 Truth Table Method Pseudocode

Implementation:

The TRUTH-TABLE algorithm is designed to determine whether a given query is entailed by a knowledge base (KB) using truth table enumeration. Here's how the implementation works:

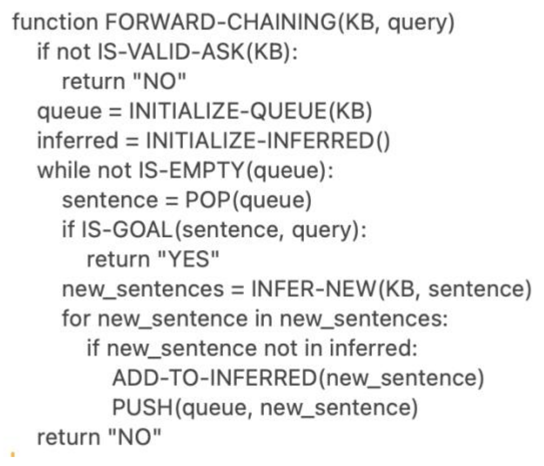
1. **Generating Models**: Initially, the algorithm generates all possible models based on the propositional symbols present in the knowledge base. Each model represents a truth assignment to these symbols, either true or false.
2. **Evaluating Knowledge Base**: For each generated model, the algorithm evaluates whether it satisfies the KB. This involves substituting the truth values of the model into the sentences of the KB and checking if each sentence evaluates to true under the model.
3. **Building Results**: Models that satisfy the KB are stored in a list called 'results'. This list represents all the possible worlds where the KB holds true.

4. **Query Evaluation**: Once all models are evaluated against the KB, the algorithm proceeds to evaluate the query. It checks if the query holds true in all models stored in the 'results' list. If it does, the algorithm returns "YES" along with the count of models satisfying the KB. Otherwise, it returns "NO".
5. **Expression Evaluation**: To evaluate expressions within the KB and query, the algorithm uses a separate EVALUATE-EXPRESSION function. This function parses the expressions into tokens, including proposition symbols and logical operators. It then utilizes a stack-based approach to compute the result of the expression based on the truth values provided by the model.

Overall, the TRUTH-TABLE algorithm systematically explores all possible interpretations of the KB by exhaustively evaluating truth assignments to its symbols. By using this approach, it can accurately determine whether the query logically follows from the given KB. This method ensures completeness in determining entailment.

Forward Chaining Method:

Pseudocode:



```
function FORWARD-CHAINING(KB, query)
    if not IS-VALID-ASK(KB):
        return "NO"
    queue = INITIALIZE-QUEUE(KB)
    inferred = INITIALIZE-INFERRED()
    while not IS-EMPTY(queue):
        sentence = POP(queue)
        if IS-GOAL(sentence, query):
            return "YES"
        new_sentences = INFER-NEW(KB, sentence)
        for new_sentence in new_sentences:
            if new_sentence not in inferred:
                ADD-TO-INFERRED(new_sentence)
                PUSH(queue, new_sentence)
    return "NO"
```

Fig.5 Forward Chaining Method Pseudocode

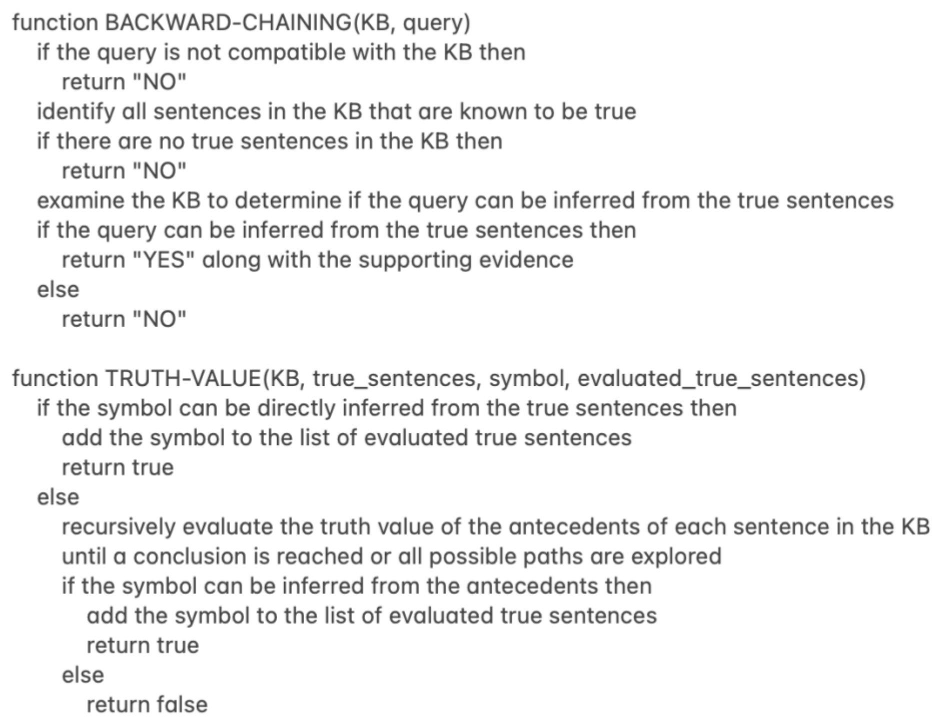
Implementation:

Initially, the method ensures the validity of the query against the KB by checking if all symbols in the query are present in the KB. If not, it returns "NO", indicating that the query cannot be answered based on the provided information.
Upon validation, the algorithm initializes a queue (queue) to store sentences known to be true based on the KB. Additionally, it creates a dictionary (separated_sentence_dict) to store the implications of sentences in the KB. This dictionary organizes sentences based on their antecedents and consequents. The queue is then populated by iterating over each sentence in the KB. If a sentence is determined to be true (is_sentence_true), it is added to the queue. Otherwise, its implications are stored in the separated_sentence_dict, associating each antecedent with its consequent.
The algorithm enters a main loop, continuing until a termination condition is met. Within this loop, it iterates over each implication stored in separated_sentence_dict. For each implication, it checks if the antecedent is satisfied by the current contents of the queue. If so, it adds the consequent to the queue if it's not already present. This process repeats until no new sentences can be inferred.
Termination occurs when either no new implications can be satisfied or when the query is reached. If the query is reached, the loop terminates, and the function returns "YES", indicating that the query can be inferred from the KB. Along with "YES", the function returns the contents of the queue that led to the query. If the query is not reached, the function returns "NO".

Finally, based on whether the query was reached, the function constructs the solution string. If the query is present in the queue, it returns "YES" along with the contents of the queue that led to the query. Otherwise, it returns "NO", signifying that the query cannot be logically inferred from the provided information.

Backward Chaining Method:

Pseudocode:



```
function BACKWARD-CHAINING(KB, query)
    if the query is not compatible with the KB then
        return "NO"
    identify all sentences in the KB that are known to be true
    if there are no true sentences in the KB then
        return "NO"
    examine the KB to determine if the query can be inferred from the true sentences
    if the query can be inferred from the true sentences then
        return "YES" along with the supporting evidence
    else
        return "NO"

function TRUTH-VALUE(KB, true_sentences, symbol, evaluated_true_sentences)
    if the symbol can be directly inferred from the true sentences then
        add the symbol to the list of evaluated true sentences
        return true
    else
        recursively evaluate the truth value of the antecedents of each sentence in the KB
        until a conclusion is reached or all possible paths are explored
        if the symbol can be inferred from the antecedents then
            add the symbol to the list of evaluated true sentences
            return true
        else
            return false
```

Fig.6 Backward Chaining Method Pseudocode

Implementation:

The backward chaining method begins by validating the query against the knowledge base (KB) to ensure that all symbols in the query exist within the KB. If any symbol in the query is not found in the KB, the method returns "NO".
After validation, the algorithm initializes two lists: true_sentences and evaluated_true_sentences. The former stores sentences known to be true based on the KB, while the latter contains evaluated true sentences contributing to the query inference. The algorithm then iterates through each sentence in the KB, determining their truth value. If a sentence is deemed true, it is appended to the true_sentences list.
The truth_value function recursively evaluates the truth value of the query symbol by iterating through each sentence in the KB and checking if the consequent of the sentence matches the query symbol. If so, it recursively evaluates the truth value of each antecedent. If the query symbol can be inferred from the antecedents or directly from the true sentences, it is added to the evaluated_true_sentences list.
Based on the evaluation result, the backward_chaining method constructs the solution string. If the query symbol can be inferred from the true sentences, the method returns "YES", along with the supporting evidence stored in evaluated_true_sentences. Otherwise, it returns "NO", indicating that the query cannot be logically deduced from the provided information.

Testing interface methods

In this section, we have classified our 20 test cases in two parts: 1. Generic test Cases which focus in on TT (Truth table) method and 2. Horn KB Test Cases for all TT, FC, and BC.

1. Generic Test Cases (Generic KBs are only handled by the truth table method):
The generic test cases evaluate the inference engine with knowledge bases that include various logical operators and complex expressions. These test cases help verify the correctness of the engine's ability to handle diverse logical scenarios beyond simple Horn clauses.

Test Case 3:
Description: Logical expressions with conjunctions and disjunctions.

```
TELL
(~p || q) & (r => s);
ASK
p
```

Truth Table Output: NO

Output Explanation: The output is NO as the given knowledge base does not provide sufficient information to derive p.

Test Case 4:
Description: Nested logical expressions.

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

Output: NO

Output Explanation: The output is NO as the nested logical expressions do not support the derivation of d.

Test Case 5:
Description: Multiple logical operations in ASK value.

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

Output: 'YES: 3'

Output Explanation: The output is YES with a count of 3, indicating that the ASK statement is satisfied with the given knowledge base.

Test Case 6:
Description: Complex logical expressions with multiple operations.

```
TELL
(a || b) & (c & d) & (e || f) & (g => h); i || j || k; l & m;
```

```
ASK
h & (i || j) & (~k || l)
```

Output: ‘NO’

Output Explanation: The output is NO as the complex logical expressions do not collectively lead to the derivation of the ASK statement.

Test Case 7:
Description: Implications and conjunctions.

```
TELL
(p => q) & (q => r); s & t & u;
ASK
r & (p | t) & (~u => s)
```

Output: ‘YES: 4’

Output Explanation: The output is YES with a count of 4, indicating the ASK statement is satisfied under the given knowledge base.

Test Case 8:
Description: Multiple logical expressions with conjunctions and disjunctions.

```
TELL
(a & b) => c; c <=> (d || e); ~e => f; f & g;
ASK
g
```

Output: 'YES: 15'

Output Explanation: The output is YES with a count of 15 for TT, indicating the ASK statement is derivable in TT.
Test Case 9:
Description: Simple chain of implications.

```
TELL
(x => y); (y => z); x;
ASK
z
```

Output: 'YES: 1'

Output Explanation: The output is YES with a count of 1 for TT, indicating the ASK statement is derivable in TT.

Test Case 10:
Description: Chain of implications with conjunction.

```
TELL
(a => (b & c)); a; b; (c => d);
ASK
d
```

Output: ‘YES: 1’

Output Explanation: The output is YES with a count of 1 for TT, indicating the ASK statement is derivable in TT.

Test Case 11:
Description: Logical expressions with negations and conjunctions.

```
TELL
(~a || b); (b => (c & d)); a; e & (d => f);
ASK
f
```

Output: ‘YES: 1’

Output Explanation: The output is YES with a count of 1 for TT, indicating the ASK statement is derivable in TT.

Test Case 12:
Description: Equivalence and implications with additional facts.

```
TELL
(p <=> (q || r)); (q => s); (r => t); p; t;
ASK
s
```

Output: NO

Output Explanation: The output is NO for TT indicating the ASK statement is not derivable in TT.

Test Case 13:
Description: Equivalence with conjunction and disjunction.

```
TELL
(x & y <=> (u || v)); (u => w); (v => z); x; y; w || z;
ASK
w
```

Output: NO

Output Explanation: The output is NO for TT indicating the ASK statement is not derivable in TT.

Test Case 14:
Description: Chain of implications with conjunctions.

```
TELL
(p => q); (r => s); p & r;
ASK
q & s
```

Output: ‘YES: 1’

Output Explanation: The output is YES with a count of 1 for TT, indicating the ASK statement is derivable in TT.

Test Case 15:

Description: Simple chain of implications.

```
TELL
(a => b); (b => c); a;
ASK
c
```

Output: ‘YES: 1’

Output Explanation: The output is YES with a count of 1 for TT, indicating the ASK statement is derivable in TT.

2. HornKB Test Cases (These are handles by all algorithms (FC, BC, TT)):
Test Case 1

Description: A series of implications forming a chain.

```
TELL
a => b; b => c; c => d; d => e; e => f;
ASK
f
```

Output: NO for TT, FC AND BC

Output Explanation: This test case is expected to return NO for all reasoning methods (TT, FC, BC) as there might be missing or incorrect information preventing the derivation of ‘f’.

Test Case 2:
Description: Chain of implications with an additional fact.

```
TELL
p => q; q => r; r => s; s => t; t => u; s;
ASK
u
```

Output: 'YES: 4' for TT
Output: 'YES: s, t, u' for FC
Output: 'YES: s, t, u' for BC

Output Explanation: This test case is expected to return YES for all reasoning methods (TT, FC, BC) because the fact s enables the derivation of u through the chain of implications.

Test Case 16:
Description: Series of implications forming a complex chain.

```
TELL
p1&p2 => q1; q1&p3 => q2; q2&p4 => r; p1; p2; p3; p4;
ASK
r
```

OUTPUT: 'YES: 1' for TT
OUTPUT: 'YES: p1, p2, p3, p4, q1, q2, r' for FC
OUTPUT: 'YES: p1, p2, q1, p3, q2, p4, r' for BC

Output Explanation: The output is YES with a count of 1 for TT, and the derivation sequence (p1, p2, p3, p4, q1, q2, r) for FC and (p1, p2, q1, p3, q2, p4, r) for BC, indicating r can be derived.

Test Case 17:
Description: Multiple implications with conjunctions.

```
TELL
a&b => c; c&d => e; e&f => g; g&h => i; a; b; d; f; h;
ASK
i
```

OUTPUT: 'YES: 1' for TT
OUTPUT: 'YES: a, b, d, f, h, c, e, g, i' for FC
OUTPUT: 'YES: a, b, c, d, e, f, g, h, i' for BC

Output Explanation: The output is YES with a count of 1 for TT, and the derivation sequence (a, b, d, f, h, c, e, g, i) for FC and (a, b, c, d, e, f, g, h, i) for BC, indicating i can be derived.

Test Case 18:
Description: Long chain of implications with conjunctions.

```
TELL
a1&a2 => b; b&a3 => c; c&a4 => d; d&a5 => e; e&a6 => f; f&a7 => g; a1; a2; a3; a4; a5; a6; a7;
ASK
g
```

Output:

OUTPUT: 'YES: 1' for TT
OUTPUT: 'YES: a1, a2, a3, a4, a5, a6, a7, b, c, d, e, f, g' for FC
OUTPUT: 'YES: a1, a2, b, a3, c, a4, d, a5, e, a6, f, a7, g' for BC

Output Explanation: The output is YES with a count of 1 for TT, and the derivation sequence (a1, a2, a3, a4, a5, a6, a7, b, c, d, e, f, g) for FC and (a1, a2, b, a3, c, a4, d, a5, e, a6, f, a7, g) for BC, indicating g can be derived.

Test Case 19:
Description: Multiple implications with conjunctions.

```
TELL
m&n => o; n => p; p&q => r; r&s => t; m; n; q; s;
ASK
t
```

OUTPUT: 'YES: 1' for TT
OUTPUT: 'YES: m, n, q, s, o, p, r, t' for FC
OUTPUT: 'YES: n, p, q, r, s, t' for BC

Output Explanation: The output is YES with a count of 1 for TT, and the derivation sequence (m, n, q, s, o, p, r, t) for FC and (n, p, q, r, s, t) for BC, indicating t can be derived.

Test Case 20:
Description: Series of implications with conjunctions.

```
TELL
p&q => r; p => s; s&t => u; u&v => w; p; q; t; v;
```

```
ASK
w
```

OUTPUT: 'YES: 1' for TT
OUTPUT: 'YES: p, q, t, v, r, s, u, w' for FC
OUTPUT: 'YES: p, s, t, u, v, w' for BC

Output Explanation: The output is YES with a count of 1 for TT, and the derivation sequence (p, q, t, v, r, s, u, w) for FC and (p, s, t, u, v, w) for BC, indicating w can be derived.

Research

Shunting Yard Algorithm:

The shunting yard algorithm has been used for the Truth Table Method for the truth table method *to handle generic KBs.* This method of parsing enables the algorithm to parse all the propositional symbols and operators in any given KB (Generic KB and Horn KB)

Our Truth Table algorithm can now handle complex generic KBs like:

(~a || b); (b => (c & d)); a; e & (d => f);

(a || b) & (c & d) & (e || f) & (g => h); i || j || k; l & m;

Implementation of Shunting Yard Algorithm:

```
def eval_expression(tokens, model):
    # Convert infix to postfix (Reverse Polish Notation) using the Shunting Yard Algorithm
    prec = {'~': 3, '&': 2, '||': 2, '=>': 1, '<=>': 1}
    assoc = {'~': 'R', '&': 'L', '||': 'L', '=>': 'L', '<=>': 'L'}
    stack = []
    output = []
    for token in tokens:
        if token in prec:  # Operator token
            while (stack and stack[-1] != '(' and
                   (prec[stack[-1]] > prec[token] or
                    (prec[stack[-1]] == prec[token] and assoc[token] == 'L'))):
                output.append(stack.pop())
            stack.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()
        else:
            output.append(token)  # Propositional symbol or constant

    while stack:
        output.append(stack.pop())

    # Evaluate the postfix expression using stack for logical operations
    stack = []
    for token in output:
```

```
    if token in prec:  # Handling operators with logical operations
        if token == '~':
            arg = stack.pop()
            stack.append(int(not arg))
        else:
            arg2 = stack.pop()
            arg1 = stack.pop()
            if token == '&':
                stack.append(int(arg1 and arg2))
            elif token == '||':
                stack.append(int(arg1 or arg2))
            elif token == '=>':
                stack.append(int(not arg1 or arg2))
            elif token == '<=>':
                stack.append(int((not arg1 or arg2) and (not arg2 or arg1)))
    else:
        stack.append(model[token])
```

About the Shunting yard algorithm:

The Shunting Yard algorithm, created by Edsger Dijkstra, converts infix expressions (e.g., "A & B || C") to postfix notation (RPN) for efficient stack-based evaluation. This is vital in computational logic and programming languages.

Key Concepts:

Operators and Precedence:
Operators have defined precedence and associativity (left or right).
Stacks and Queues:
A stack handles operators, and a queue (or list) holds the output.
Token Processing:
Operands go directly to the output queue.
Operators are managed based on precedence and associativity.
Parentheses ensure proper grouping.
Finalizing:
Remaining stack operators are moved to the output queue.

Evaluating Postfix Expressions:
Push operands onto the stack.
Apply operators by popping operands, processing them, and pushing the result back.
Continue until one value remains on the stack.

Example:

For (A & B) || C:
Conversion: A B & C ||
Evaluation:
Push A, B, apply &, push result, push C, apply ||.

Team summary report

This assignment was done in a team of 2. All the algorithms and the operations of handing the knowledge bases was equally divided among us.

Below is a report of how we contributed in making these algorithm individually:

Chethan Bhaskar – 104069283:

- Implemented the Knowledge Base class along with 2 functions which extract operators and extract symbols respectively.
- Implemented the truth table method which handles all Horn KBs
- Implemented forward chaining method which effectively handles all Horn KBs
- Implemented the functions to get the LHS and RHS of any horn KBs which are crucial in forward chaining and backward chaining algorithms.
- Made effective changes to handle the issues and potential bugs like if the asked value (alpha) is not in KB.
- Made 10 testcases of which 4 are horn KBs and 6 are generic KBs.
- Contributed in implemented the shunting yard algorithm (research)
- **Overall contribution percentage: 50%**

Meezan Hussain – 104330015:

- Implemented the creation of all models needed for truth table using the *itertools* module of python.
- Implemented parsers for evaluating all the sentences in the string format to usable format in KBs using the ***re*** module of python
- Implemented the conversion of all the sentences in KB and stored them in usable format using *re.findall* function and made patterns like r'\b[a-zA-Z]+\d*\b' to extract the correct symbols and operators from a sentence.
- Implemented the backward chaining method along with the *truth_value* method which is used in backward chaining.
- Made 10 testcases of which 4 are horn KBs and 6 are generic KBs.
- Contributed in implemented the shunting yard algorithm (research)
- **Overall contribution percentage: 50%**

As a team, we finished all the designated work by setting a personal deadline for each task and we were able to achieve the desired output which was to implement all three algorithms and test the algorithms thoroughly to find any bugs or implementation errors. Our algorithms have been made to perfection because of the teamwork and mutual understanding we possess between the both of us.

Conclusion

In conclusion, this report details the implementation of three inference methods for a propositional logic inference engine: Truth Table, Forward Chaining, and Backward Chaining. The report also covers the Shunting Yard algorithm used for parsing expressions in the Truth Table method.

While all three methods can determine the entailment of a query from a knowledge base, their efficiency varies. Truth Table is exhaustive and can be slow for large knowledge bases. Forward Chaining is generally faster but may not terminate for certain knowledge bases. Backward Chaining is goal-directed but might require more iterations depending on the query and knowledge base structure.

For improved performance, the choice of algorithm depends on the specific problem characteristics. For smaller knowledge bases or when completeness is essential, Truth Table might be suitable. For larger knowledge bases with simpler queries, Forward Chaining could be a better option. Backward Chaining might be preferable when the query is specific, and the knowledge base is vast.

Acknowledgements

1. GeeksForGeeks.com – The geeks for geeks website helped us find the 'itertools' module which has a function 'product' which helped us create all the models for truth table inference method. (GeeksForGeeks, 2021)
2. Codecademy – The codecademy website clearly guided us on the 're' module in python which helped us implement the parse function using the imported *re.findall* function. This was required for all the inference methods. (codecademy, 2023)
3. Brilliant.org – Brilliant.org website helped us by giving a step by step explanation and use cases of the shunting yard algorithm which formed a base for our research and eventually helped us enable the truth table method to handle all complex generic KBs. (brilliant.org, 2024)

Bibliography

brilliant.org. (2024). *Shunting Yard Algorithm*. Retrieved from brilliant.org: https://brilliant.org/wiki/shunting-yard-algorithm/

codecademy. (2023). *re.findall()*. Retrieved from codecademy.com: https://www.codecademy.com/resources/docs/python/regex/findall

GeeksForGeeks. (2021). *Python – Itertools.Product()*. Retrieved from geeksforgeeks.com: https://www.geeksforgeeks.org/python-itertools-product/