Propositional Logic Application In Prolog Programming Language

Rainaldi Pratama F Sembiring - 13524117
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: raynaldismb@gmail.com , 13524117@std.stei.itb.ac.id

***Abstract*— The Prolog programming language is widely recognized for its foundation in the declarative paradigm, setting it apart from common imperative and object-oriented languages. This study aims to investigate the practical application of propositional logic within Prolog to determine the depth of this relationship. While logic is known to be the foundation of computer science and reasoning, the direct, functional mapping between formal logical concepts and Prolog's syntax is often not explicitly detailed. Adopting a practical methodology, this research utilized Visual Studio Code as the primary text editor and the GNU Prolog compiler to translate theoretical logical constructs into executable code. The analysis involved creating a knowledge base of facts and rules and executing queries to test logical arguments. The findings reveal that propositional logic is not merely applicable to Prolog, but constitutes its fundamental operational core. It was found that atomic propositions directly translate to Prolog facts, which serve as the axiomatic truths of a program. Furthermore, logical operators such as conjunction (AND) and disjunction (OR) are functionally implemented by Prolog's operators, respectively. Most significantly, the logical implication, or the conditional statement "if p, then q", is the foundational structure of every Prolog rule (conclusion :- premise.). The query-execution mechanism is a direct automation of logical inference, mirroring argument forms like Modus Ponens. This study concludes that a thorough understanding of these logical underpinnings is essential for programmers to leverage the full power of Prolog, shifting from a purely syntactic approach to a truly logical and declarative mindset.**

***Keywords*—prolog, propositional logic**

I. INTRODUCTION

Logic is a field of knowledge that functions as a foundational pillar in computer science, serving as an essential framework for thinking and reasoning. The process of reasoning itself is defined as the method of reaching a conclusion based on a series of multiple statements. One of the main branches of this study is propositional logic, which focuses specifically on the relationships between declarative statements, or propositions. A proposition is formally defined as a sentence that has a definitive truth value—it is either true or false, but not both simultaneously. The most basic of these are atomic propositions, which are singular statements of truth. These can then be combined using logical constructs, or operators, such as conjunction (AND), disjunction (OR), and negation (NOT), to form more complex compound propositions that allow for sophisticated logical expressions.

Nowadays, a vast array of programming languages exists, each designed for specific purposes; for instance, Dart is typically used for creating mobile applications with its framework, Flutter. While the majority of these languages are built on imperative or object-oriented paradigms, which require step-by-step instructions, Prolog is a notable exception that utilizes the declarative paradigm. As its name suggests, Prolog stands for "Programming in Logic" and is a logical and declarative programming language. This paradigm is rooted in a different approach: program statements express facts and rules about a problem rather than defining an explicit procedure for its solution. This makes Prolog particularly suitable for programs that involve symbolic or non-numeric computation. For this reason, Prolog was famously adopted by the Japanese Fifth-Generation Computer Project and remains a primary language in the field of Artificial Intelligence, where fundamental tasks like symbol manipulation and inference manipulation are paramount.

This study focuses on analysing the practical application of propositional logic within the Prolog programming language to demonstrate the depth and functionality of their integration. The discussion will begin by showing how the fundamental building blocks of logic—atomic and compound propositions—are translated into a Prolog knowledge base composed of facts and rules. Following this, the paper will analyse how the logical implication form, "if p, then q", becomes the very backbone of Prolog's inference engine, forming the structure of its rules (conclusion :- premise.) and enabling its capacity for reasoning. Finally, it will illustrate how Prolog automates the validation of logical arguments by treating the program's code as a series of premises and a user's query as a conclusion to be proven, a process that mirrors valid logical inference and makes Prolog a system designed for executing logic.

II. Literature Review

A. Propositional Logic

Logic is knowledge that helps people with thinking and reasoning, reasoning is a way to achieve conclusions from multiple statements.

Logic is based on relation between sentences or statements, only a sentence that has a truth value that's considered a proposition, proposition is a statement/sentence that's either true or false, but not both.

Propositional Logic is a branch of discrete mathematical logic, propositional logic based on the relation between sentence or statement, only sentences with a truth value (true or false but not both) are considered. In terms of prolog logic, propositional logic used is logical inference and resolution, it's also used to state facts and rules, and implement logical connectives using operator such as:

1. Conjunction AND (∧)

   This operator is true only if both connected propositions are true.

   p∧q = true if p = true and q = true.

2. Disjunction OR (∨)

   This operator is true if at least one or both connected propositions are true.

   p∨q = ture if p = true or q = true.

3. Negation NOT (¬)

   This operator reverses the truth value of an argument.

   If p = true, then ¬p = false.

4. Exclusive Disjunction (⊕)

   This operator is true if one proposition is true but not both.

   1. p = True dan q = False

   2. p = False dan q = True

These are the propositional operators.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

p	$\sim p$
T	F
F	T

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

Pict 2.1 Truth table for the 4 propositional operations

Source:

https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/01-Logika-2024.pdf

Implication that also also called conditional proposition, proposition form "if p, then q", and the notation is p → q or the equivalents is ~p ∨ q, p is premise, and q is conclusion, implication is the representation of rules in prolog programming languages

p	q	$p \to q$
T	T	T
T	F	F
F	T	T
F	F	T

Pict 2.2 Implication truth table

Source:

https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/01-Logika-2024.pdf

Two compound proposition is called equivalent if both have identical truth table, this equivalent symbolized byP⇔Q. One of the most important equivalents is De Morgan Law, that is :

~(p∧q)⇔~p∨~q.

These concepts form the foundation for constructing and analyzing arguments. An argument is a series of propositions consisting of premises and a conclusion. An argument is said to be valid if its conclusion is true when all of its hypotheses are true. Otherwise, the argument is considered false or invalid. The validity of an argument can be demonstrated by showing that the implication (p1∧p2∧...∧pn)→q is a tautology, that is, a proposition that is always true in all cases. Well-established rules of inference, such as Modus Ponens (p→q,p⇒q) and Modus Tollens (p→q,~q⇒~p), are often used to validate arguments.

B. Prolog

The heritage of prolog includes research on theorem provers and some other automated deduction systems that were developed in 1960s and 1970s. The Inference mechanism of the Prolog is based on Robinson's Resolution Principle, that was proposed in 1965, and Answer extracting mechanism by Green (1968). These ideas came together forcefully with the advent of linear resolution procedures.

The explicit goal-directed linear resolution procedures gave impetus to the development of a general-purpose logic programming system. The first Prolog was the Marseille Prolog based on the work by Colmer Auer in the year 1970. The manual of this Marseille Prolog interpreter (Roussel, 1975) was the first detailed description of the Prolog language.

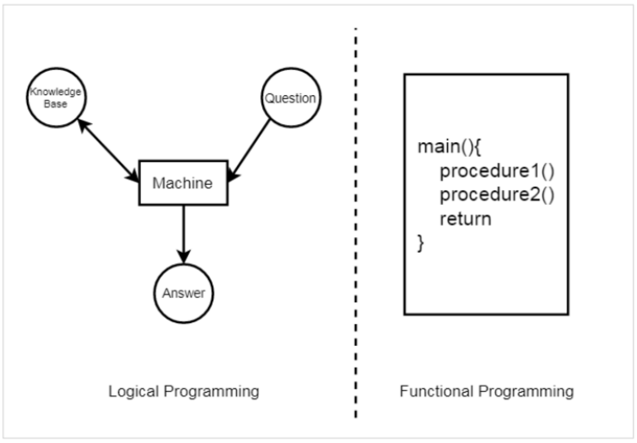
Prolog is also considered as a fourth-generation programming language supporting the declarative programming paradigm. The well-known Japanese Fifth-Generation Computer Project, that was announced in 1981, adopted Prolog as a development language, and thereby grabbed considerable attention on the language and its capabilities.

Prolog as the name itself suggests, is the short form of LOGical PROgramming. It is a logical and declarative programming language. Before diving deep into the concepts of Prolog, let us first understand what exactly logical programming is.

Logic Programming is one of the Computer Programming Paradigm, in which the program statements express the facts and rules about different problems within a system of formal logic. Here, the rules are written in the form of logical clauses, where head and body are present. For example, H is head and B1, B2, B3 are the elements of the body. Now if we state that "H is true, when B1, B2, B3 all are true", this is a rule. On the other hand, facts are like the rules, but without anybody. So, an example of fact is "H is true".

Some logic programming languages like Datalog or ASP (Answer Set Programming) are known as purely declarative languages. These languages allow statements about what the program should accomplish. There is no such step-by-step instruction on how to perform the task. However, other languages like Prolog, have declarative and also imperative properties. This may also include procedural statements like "To solve the problem H, perform B1, B2 and B3"

The difference between logical programming and functional programming.



```
main(){
   procedure1()
   procedure2()
   return
}
```

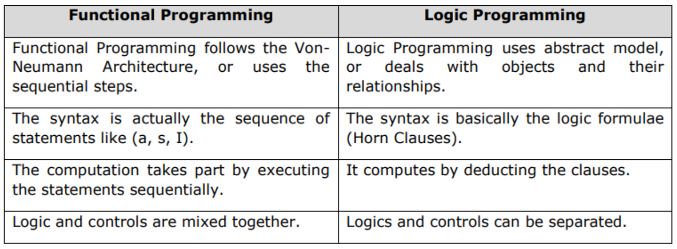
Pict 2.3 Difference between logical programming and functional shown with a picture.

Source:

https://www.tutorialspoint.com/prolog/prolog_tutorial.pdf

From this illustration, we can see that in Functional Programming, we have to define the procedures, and the rules of how the procedures work. These procedures work step by step to solve one specific problem based on the algorithm. On the other hand, for Logic Programming, we will provide a knowledge base. Using this knowledge base, the machine can find answers to the given questions, which is totally different from functional programming.

In functional programming, we have to mention how one problem can be solved, but in logic programming we have to specify for which problem we actually want the solution. Then logic programming automatically finds a suitable solution that will help us solve that specific problem.



Functional Programming	Logic Programming
Functional Programming follows the Von-Neumann Architecture, or uses the sequential steps.	Logic Programming uses abstract model, or deals with objects and their relationships.
The syntax is actually the sequence of statements like (a, s, I).	The syntax is basically the logic formulae (Horn Clauses).
The computation takes part by executing the statements sequentially.	It computes by deducting the clauses.
Logic and controls are mixed together.	Logics and controls can be separated.

Table 2.1 Difference between functional programming and logic programming

Source:

https://www.tutorialspoint.com/prolog/prolog_tutorial.pdf

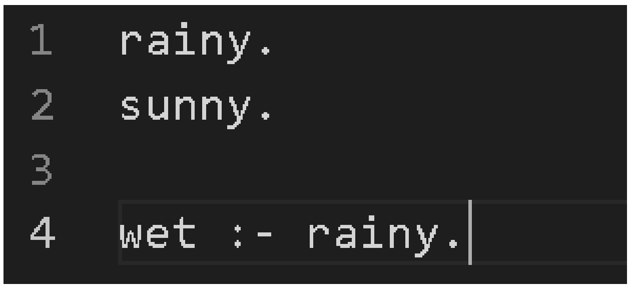
Prolog or PROgramming in LOGics is a logical and declarative programming language. It is one major example of the fourth-generation language that supports the declarative programming paradigm. This is particularly suitable for programs that involve symbolic or non-numeric computation. This is the main reason to use Prolog as the programming language in Artificial Intelligence, where symbol manipulation and inference manipulation are the fundamental tasks.

In Prolog, we need not mention the way one problem can be solved, we just need to mention what the problem is, so that Prolog automatically solves it. However, in Prolog we are supposed to give clues as the solution method.

Prolog language basically has three different elements:

Facts: The fact is predicate that is true, for example, if we say, "Tom is the son of Jack", then this is a fact.

Rules: Rules are extinctions of facts that contain conditional clauses. To satisfy a rule these conditions should be met. For example, if we define a rule as:



```
rainy.
sunny.

wet :- rainy.
```

Pict 2.4 rule example.

This implies that if its rainy, then its wet, since the rainy fact is stated above, then wet function value is true.

Questions: And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules.

Prolog's internal mechanisms automatically use inference rules, as described in the theory of logic, to deduce answers to given queries based on existing facts and rules. This makes Prolog an ideal subject for analyzing direct applications of the theory of propositional logic.

III. Methodology and Tools

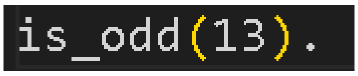
This chapter outlines the practical methodology and the specific software tools employed to conduct the analysis for this paper. The research approach adopted for this study is practical and analytical, designed to bridge the gap between the theoretical principles of propositional logic and their functional implementation within the Prolog programming language. The core of this approach involved creating a knowledge base of facts and rules and executing queries to test logical arguments. This hands-on method of translating theoretical logical constructs into executable code ensures that the findings are not merely theoretical but are grounded in the actual, observable behavior of a Prolog environment, making the conclusions both practical and verifiable.

To facilitate this implementation and testing process, a standard set of development tools was used, specifically Visual Studio Code as the primary text editor and the GNU Prolog compiler. All Prolog source code was written using Visual Studio Code, a modern and extensible code editor whose features, such as syntax highlighting and integrated terminal support, provided an efficient environment for creating the knowledge bases. The compilation and execution of the code were handled by GNU Prolog, a free and widely used implementation of the language. This tool served the dual functions of a compiler, used to parse source files and check for syntactical errors, and an interactive interpreter. The interactive top-level shell was critical to the methodology, as it allowed for the knowledge base to be loaded directly into memory and for queries to be posed in real-time, providing immediate feedback for testing the validity of the logical arguments.

IV. Analysis and Discussion

According to the established theoretical foundations of propositional logic and Prolog programming languages detailed in the previous chapter, this chapter will now begin the deep and practical analysis required to demonstrate how facts and rules are stated to represent propositional logic within the Prolog programming language. This exploration starts by examining the very heart of logical systems. Logic itself is a field of knowledge that provides the essential tools for thinking and reasoning, with reasoning defined as the process of achieving conclusions from multiple statements. The entire framework of reasoning is built upon its most fundamental component: the proposition.

The analysis, therefore, commences with this foundational building block. A proposition is formally defined as a statement or sentence that is definitively either true or false, but not both. The simplest form, a single or atomic proposition, is a declarative statement asserting a singular, indivisible truth, such as the example p : 13 is an odd number. In the Prolog logic programming paradigm, this concept finds its direct and functional equivalent in a **fact**. A fact in Prolog is defined as a predicate that is asserted to be true and serves as the most basic component for building a program by stating an unconditional truth. It can also be understood as a special type of rule that has no body or conditions. Therefore, the logical proposition p can be represented within the Prolog knowledge base with the following syntax, forming the ground truth from which the program will reason:



```
is_odd(13).
```

Pict 4.1 fact example.

While atomic propositional is the fundamental, most logical reasoning involves compound propositions, which are formed by combining 2 or more propositions with connective operators, such as:

Logic operator in prolog:

1. Conjunction (AND)

In prolog ‘,’ represents logical conjunction. So, a :- b, c. means “a is true if b and c is true” in propositional logic.

c, it is usually written b∧c⟶a.

2. Disjunction (OR)

The logical representation of disjunction is a∨b ⟶c, there is no direct representation of disjunction in prolog like conjunction does, but you can derive the logical notation a∨b⟶c into a⟶c∨b⟶c, so it finally can be represented as:

c:-a.

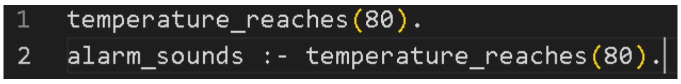
c:-b.

means “if a or b is true then c is true”

3. Negation (NOT)

Prolog provides “\+” for negation as failure, while not strictly classical negation, it often serves a similar purpose in practical applications. \+a. means “it’s not provable that it’s a”

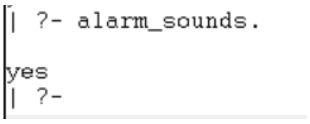
The most significant application of propositional logic within Prolog is the implementation of logical implication. A conditional proposition, expressed as “if p, then q”, is an essential component of logical deduction and forms the very structure of a prolog rule.In logic, implication writen with p⟶q, where ‘p’ is premise and ‘q’ is conclusion, on the other hand prolog has the syntax q:-p. Therefore, the rule q:-p is the computational equivalent of the logical expression p⟶q. For instance, the classic implication "if the temperature reach 80°C, then the alarm will ring" directly translates into the Prolog rule alarm_sounds :- temperature_reaches(80). This structure allows Prolog to perform deductions; if the program can prove the premise of a rule is true, it can then deduce that the conclusion is also true.



```
temperature_reaches(80).
alarm_sounds :- temperature_reaches(80).
```

Pict 4.2 logical implication in prolog.

Since we it is already stated that temperature_reaches(80), so the value is true, then the alarm_sounds must be true too



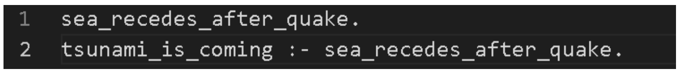
```
| ?- alarm_sounds.

yes
| ?-
```

Pict 4.3 query result from Pict 4.2

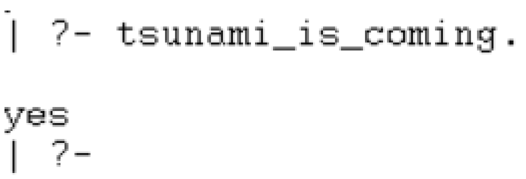
As shown on the picture above, the query alarm_sounds return the value true.

This deductive capability allows Prolog to automate logical arguments. An argument is a sequence of propositions, called premises, that lead to a final proposition, the conclusion. An argument is considered valid if the conclusion is necessarily true whenever all premises are true. In Prolog, the entire collection of facts and rules in a program forms the set of premises, while a user's **query** represents the conclusion that needs to be proven. The primary rule of inference that Prolog uses is **Modus Ponens**, an argument form stated as: if (p→q) is true and (p) is true, then (q) must be true. The provided source material illustrates this with a clear example: the premise "If the sea water recedes after an earthquake at sea, then a tsunami will come" combined with the premise "The sea water recedes after an earthquake at sea" leads to the valid conclusion "tsunami will come". This is modeled in a Prolog knowledge base with the rule tsunami_is_coming :- sea_recedes_after_quake. and the fact sea_recedes_after_quake. When a user poses the query ?- tsunami_is_coming., Prolog's inference engine checks if the conclusion logically follows from the premises, finds that it does through a process identical to Modus Ponens, and confirms the argument's validity by responding with true.



```
sea_recedes_after_quake.
tsunami_is_coming :- sea_recedes_after_quake.
```

Pict 4.4 logical implication in prolog



```
| ?- tsunami_is_coming.

yes
| ?-
```

Pict 4.5 query result from Pict 4.4

As shown in the picture above, the query resulted a true.

Finally, the concept of logical equivalence, where two compound propositions are considered equivalent if they possess identical truth tables, holds significant practical value in Prolog programming. Beyond well-known equivalences like De Morgan's Law, which states that $\sim(p \land q) \Leftrightarrow \sim p \lor \sim q$, a particularly useful example in programming is the equivalence between a logical implication (p→q) and its contrapositive (∼q→∼p). The source material effectively demonstrates this with the example of two shopkeeper mottos: the first stating "if an item is good, it is not cheap" (p→∼q), and the second stating "if an item is cheap, it is not good" (q→∼p). Although these statements are worded differently, they are logically identical, a fact verifiable through truth table analysis. This equivalence provides flexibility to the programmer, who can implement either is_not_cheap(Item) :- is_good(Item). or is_not_good(Item) :- is_cheap(Item).. This choice can be based on which formulation is more intuitive or leads to a more computationally efficient search by Prolog's inference engine, all while ensuring the logical integrity of the program remains intact. This chapter has systematically demonstrated the direct application of such core tenets of propositional logic within the Prolog language. It has been established that atomic propositions directly translate to Prolog facts, which serve as the axiomatic truths of a program. Furthermore, it was shown that logical operators like conjunction and disjunction are functionally implemented by Prolog's syntax within its rules, and that the logical implication is the foundational structure of every Prolog rule (conclusion :- premise.). The query-execution mechanism was revealed to be a direct automation of logical inference, mirroring argument forms like Modus Ponens to derive new truths. Prolog is, therefore, more than a language inspired by logic; it is a system explicitly designed for executing logic. Having established this foundational mapping, the following chapter will apply these principles to a comprehensive case study, demonstrating how this logical framework can be used to solve a complex problem.

V. CONCLUSION

This study was conducted to investigate and analyze the practical application of propositional logic within the Prolog programming language, aiming to move beyond a general acknowledgment of their connection to a detailed demonstration of their functional relationship. The research confirmed that the link is not superficial; propositional logic constitutes the fundamental operational core of Prolog. The analysis systematically revealed that the core components of propositional logic have direct and tangible equivalents within Prolog's structure. It was found that atomic propositions, which are singular statements of truth such as "13 is an odd number", are represented as Prolog facts, forming the axiomatic foundation of a program's knowledge base. This initial mapping establishes the groundwork upon which all other logical operations are built, providing the unconditional truths from which Prolog's reasoning engine begins its work.

Furthermore, the study demonstrated how compound propositions are built using logical operators that are functionally implemented in Prolog's syntax. The logical conjunction (AND) is directly represented by the comma operator (,) to connect multiple necessary conditions within the premise of a rule. Disjunction (OR), while not having a single direct operator within a rule's body, is effectively expressed by defining multiple rules that share the same conclusion, stating that the conclusion is true if any of the separate premises are met. Most significantly, the analysis affirmed that logical implication, the conditional statement "if p, then q", is the foundational structure of every Prolog rule, expressed in the syntactically reversed format conclusion :- premise.. This structure allows Prolog to perform deductions, a capability that is fully realized through its query-execution mechanism. This mechanism was shown to be a direct automation of logical inference, with its process mirroring established argument forms like Modus Ponens to validate conclusions against a set of premises, as demonstrated with the tsunami example. Finally, the practical value of logical equivalence, such as the relationship between an implication and its contrapositive, was shown to allow for more intuitive and flexible program design, enabling programmers to choose the most efficient or readable representation of a logical truth.

In synthesizing these findings, this paper concludes that a thorough understanding of these logical underpinnings is essential for any programmer seeking to leverage the full power of Prolog. This is because Prolog supports a declarative paradigm where, unlike in functional or imperative programming, the programmer's role is not to provide step-by-step instructions on *how* to solve a problem, but rather to specify *what* the problem is within a knowledge base of facts and rules. The significance of this approach was recognized by initiatives like the Japanese Fifth-Generation Computer Project, which adopted Prolog due to its strengths in handling symbolic or non-numeric computation. This knowledge elevates a developer's approach from a purely syntactic one to a truly logical and declarative mindset, which is critical in fields like Artificial Intelligence where symbol manipulation and inference manipulation are fundamental tasks.

Therefore, while this study has demonstrated numerous applications of propositional logic in the Prolog programming language, the ultimate conclusion is that the relationship is far deeper. It is not just that propositional logic is *applicable* to Prolog, but that propositional logic serves as the very *foundation* on which the language is built. This research solidifies that Prolog is a system meticulously designed for executing logic, providing a computational framework for the very act of thinking and reasoning. This understanding demystifies the language, showing that its power comes not from complex, user-defined algorithms, but from its innate ability to perform inference manipulation over a well-defined logical system. Ultimately, the language truly embodies its name: Programming in Logic.

ACKNOWLEDGMENT

The author would like to express profound gratitude to almighty God for blessing me throughout making this paper, also the gratitude to Dr. Ir. Rinaldi Munir, M.T. for his invaluable guidance and insights as IF1220 Discrete Mathematics course lecturer, which greatly contributed to the development of this paper, and and also an honorable thanks to Yey's Cozy Campfire(discord server), for being a place for discussion and helping each other.

REFERENCES

[1] Munir, R. 2024. Logika. [online] Source : https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/01-Logika-2024.pdf [Acessed: 18 June. 2025]

[2] Tutorialspoint, prolog_tutorial. [online] Source : https://www.tutorialspoint.com/prolog/prolog_tutorial.pdf

[3] Paun, N. 2016. Propositional logic in prolog. [online] Source : https://www.cs.mcgill.ca/~npaun/articles/propositional-logic-in-prolog

[4] a bit of intelligence, Logic and Prolog. [online] Source : https://youtu.be/nDLrpT50vFE?si=m13FUZdU6s_LcJCi

STATEMENT

I hereby declare that this paper is my own work, not a paraphrase or a translation of someone else's paper, and definitely not plagiarism.

Bandung, 19 June 2025

[illegible]

Ttd
Rainaldi Pratama F Sembiring 13524117