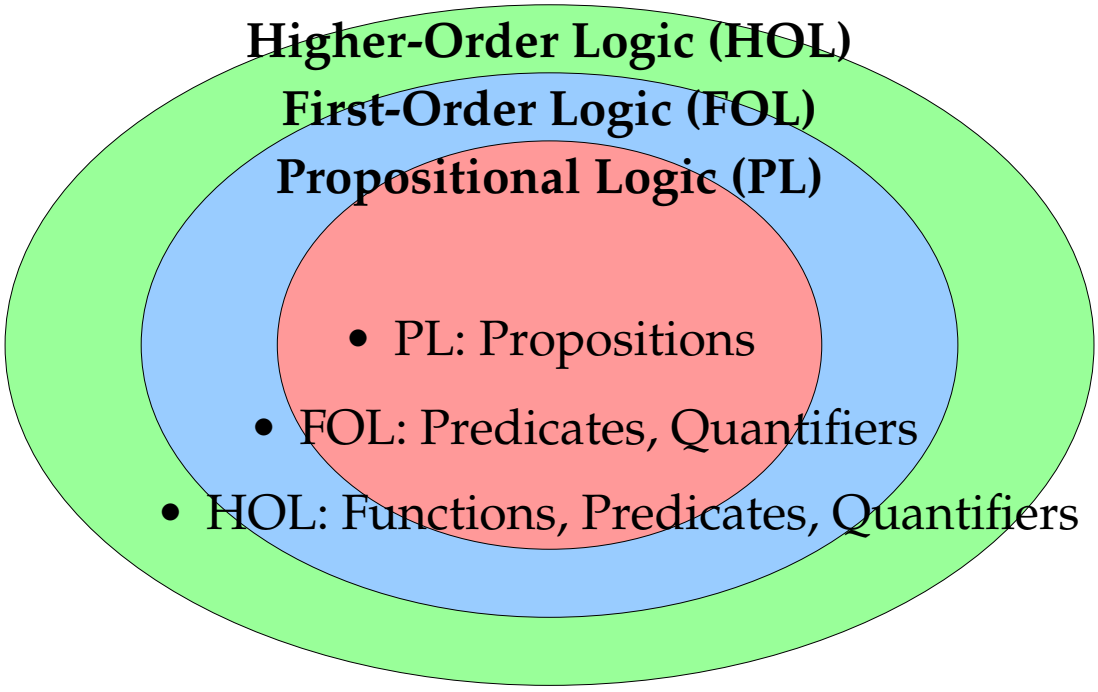
Software Verification

- Logic-based Program Verification -

Ji, Yong-Hyeon



A document presented for
the Software Verification

Department of Information Security, Cryptology, and Mathematics
College of Science and Technology
Kookmin University

July 4, 2024

Contents

Symbols

In this paper, symbols are defined as follows.

$I \models P$ I satisfies P
$I \not\models P$ I does not satisfies P

Chapter 1

Introduction

[1] [2] [9]

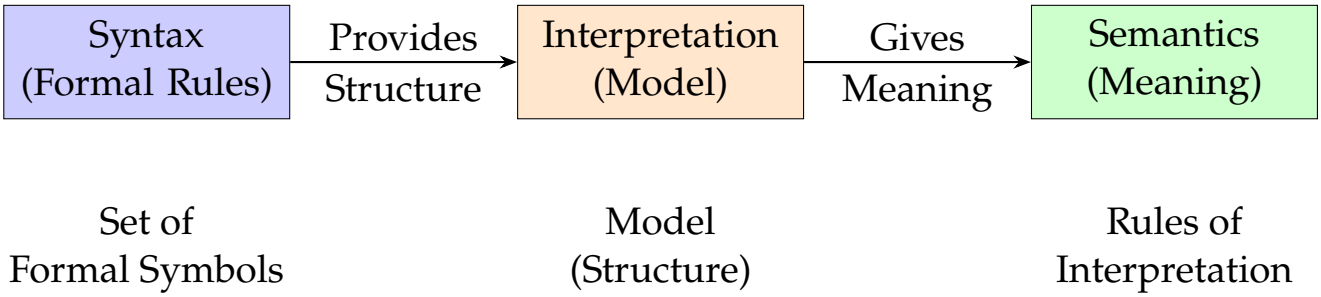
Chapter 2

Propositional Logic I

References:[3]

Propositional logic is a branch of logic that deals with propositions which can be either true or false. The basic components and operations in propositional logic are as follows:

2.1 Syntax and Semantic of Propositional Logic



2.1.1 Syntax

- **Atom**: basic elements
  - truth symbols $\bot$("false") and $\top$("true")
  - propositional variables $P, Q, R, \ldots$
- **Literal**: an atom α or its negation $\neg\alpha$.
- **Formula**: a literal or the application of a logical connective (boolean connectives) to formulas

$$
\begin{array}{rcll}
F & \to & \bot & \\
 & | & \top & \\
 & | & P & \\
 & | & \neg F & \text{negation (“not”)} \\
 & | & F_1 \wedge F_2 & \text{conjunction (“and”)} \\
 & | & F_1 \vee F_2 & \text{disjunction (“or”)} \\
 & | & F_1 \to F_2 & \text{implication (“implies”)} \\
 & | & F_1 \leftrightarrow F_2 & \text{iff (“if and only if”)}
\end{array}
$$

- Formula G is a **subformula** of formula F if it occurs syntactically within G.

$$\begin{aligned} \mathsf{sub}(\bot) &= \{\bot\} \\ \mathsf{sub}(\top) &= \{\top\} \\ \mathsf{sub}(P) &= \{P\} \\ \mathsf{sub}(\neg F) &= \{\neg F\} \cup \mathsf{sub}(F) \\ \mathsf{sub}(F_1 \wedge F_2) &= \{F_1 \wedge F_2\} \cup \mathsf{sub}(F_1) \cup \mathsf{sub}(F_2) \end{aligned} \quad \vdots$$

- Consider $F : (P \wedge Q) \rightarrow (P \vee \neg Q)$. Then

$$\mathsf{sub}(F) = \{F, P \wedge Q, P \vee \neg Q, P, Q, \neg Q\}.$$

- The strict subformulas of a formula are all its subformulas except itself.

- To minimally use parentheses, we define the relative precedence of the logical connectives from highest to lowest as follows:

$$\neg \quad \wedge \quad \vee \quad \rightarrow \quad \leftrightarrow$$

- (Currying) Additionally, $\rightarrow$ and $\leftrightarrow$ associate to the right, e.g.,

$$P \rightarrow Q \rightarrow R \iff P \rightarrow (Q \rightarrow R).$$

- Example:
  - $(P \wedge Q) \rightarrow (P \vee \neg Q) \iff P \wedge Q \rightarrow P \vee \neg Q$
  - $(P_1 \wedge ((\neg P_2) \wedge \top)) \vee ((\neg P_1) \wedge P_2) \iff P_1 \wedge P_2 \top \vee \neg P_1 \wedge P_2$

2.1.2 Semantics

- The semantics of a logic provides its meaning. The meaning of a PL formula is either true or false.
- The semantics of a formula is defined with an **interpretation** (or assignment) that assigns truth values to propositional variables.
- For example, $F : P \wedge Q \rightarrow P \vee \wedge Q$ evaluates to true under the interpretation $I : \{P \mapsto \text{true}, Q \mapsto \text{false}\}$:

P	Q	$\neg Q$	$P \wedge Q$	$P \vee \neg Q$	F
1	0	1	0	1	1

- The tabular notation is unsuitable for predicate logic. Instead, we define the semantics inductively.

2.1.3 Inductive Definition of Semantics

In an inductive definition, the meaning of basic elements is defined first. The meaning of complex elements is defined in terms of subcomponents.

- We write $I \models F$ if F evaluates to **true** under I.
- We write $I \not\models F$ if F evaluates to **false** under I.

Note that

$$
\begin{array}{ll}
I \models \top, \quad I \not\models \bot, & \\
I \models P & \text{iff } I[P] = \text{true} \\
I \models \neg F & \text{iff } I[P] = \text{false} \\
I \models \neg F & \text{iff } I \not\models F \\
I \models F_1 \wedge F_2 & \text{iff } I \models F_1 \text{ and } I \models F_2 \\
I \models F_1 \vee F_2 & \text{iff } I \models F_1 \text{ or } I \models F_2 \\
I \models F_1 \rightarrow F_2 & \text{iff } I \not\models F_1 \text{ or } I \models F_2 \\
I \models F_1 \leftrightarrow F_2 & \text{iff } (I \models F_1 \text{ and } I \models F_2) \text{ or } (I \not\models F_1 \text{ and } I \not\models F_2)
\end{array}
$$

Example 2.1.1. Consider the formula

$$F : P \wedge Q \rightarrow P \vee \neg Q$$

and the interpretation

$$I : \{P \mapsto \text{true}, Q \mapsto \text{false}\}.$$

The truth value of F is computed as follows:

1.	$I \models P$	since $I[P] = \text{true}$
2.	$I \not\models Q$	since $I[P] = \text{true}$
3.	$I \models \neg Q$	by 2 and semantics of $\neg$
4.	$I \not\models P \wedge Q$	by 2 and semantics of $\wedge$
5.	$I \models P \vee \neg Q$	by 1 and semantics of $\vee$
6.	$I \models F$	by 4 and semantics of $\rightarrow$

2.2 Satisfiability and Validity

- A formula F is **satisfiable** iff there exists an interpretation I such that $I \models F$.
- A formula F is **valid** iff for all interpretations I, $I \models F$.
- Satisfiability and validity are dual:

$$F \text{ is valid} \iff \neg F \text{ is unsatisfiable}$$

Proof. content... □

- We can check satisfiability by deciding validity, and vice versa.

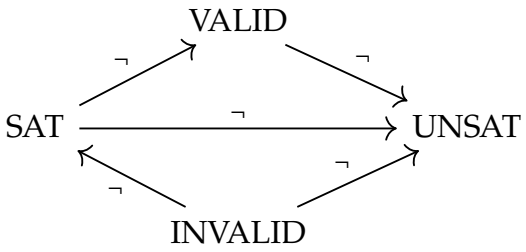
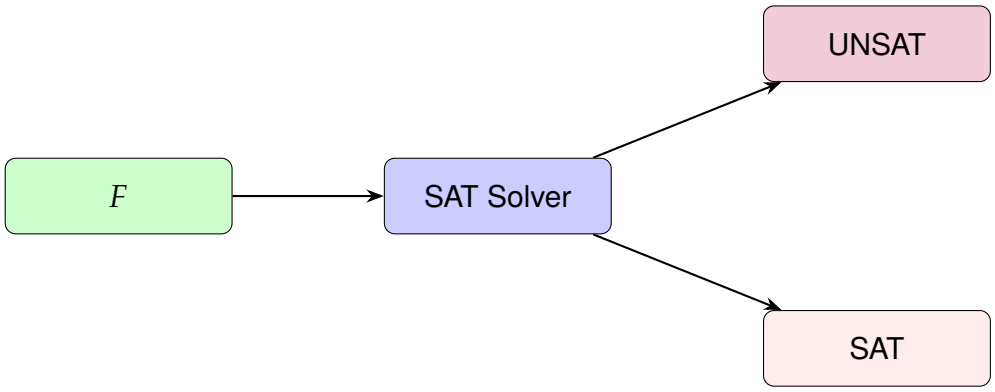


Figure 2.1: Relation of SAT, UNSAT, VALID and INVALID



2.2.1 Deciding Validity and Satisfiability

Two approaches to show F is valid:

- **Truth Table Method** performs exhaustive search: e.g.,

$$F : P \wedge Q \rightarrow P \vee \neg Q.$$

P	Q	$P \wedge Q$	$\neg Q$	$P \vee \neg Q$	F
0	0	0	1	1	1
0	1	0	0	0	1
1	0	0	1	1	1
1	1	1	0	1	1

  Non-applicable to logic with infinite domain (e.g., first-order logic).

- **Semantic Argument Method** uses deduction:
  - Assume F is valid: $I \not\models F$ for some I (falsifying interpretation).
  - Apply deduction rules (proof rules) to derive a contradiction.
  - If every branch of the proof derives a contradiction, then F is valid.
  - If some branch of the proof never derives a contradiction, then F is invalid.

2.2.2 Deduction Rules for Propositional Logic

Negation Elimination

$$\frac{I \models \neg F}{I \not\models F}$$

This rule shows how to derive F from $\neg F$. It eliminates the negation by asserting that if $\neg F$ holds, then F cannot hold.

Negation Introduction

$$\frac{I \not\models \neg F}{I \models F}$$

This rule introduces F from the fact that $\neg F$ does not hold. It asserts that if the negation of F does not hold, then F must hold.

Conjunction Elimination[a]

$$\frac{I \models F \land G}{I \models F, I \models G}$$

This rule breaks down a conjunction into its individual components, showing that if $F \land G$ is true, then both F and G are true.

De Morgan's Law for Conjunction[a]

$$\frac{I \not\models F \land G}{I \not\models F \mid I \not\models G}$$

This rule states that if $F \land G$ is not true, then at least one of F or G is not true. This is an application of De Morgan's laws.

Disjunction Elimination[b]

$$\frac{I \models F \lor G}{I \models F \mid I \models G}$$

This rule asserts that if $F \lor G$ is true in an interpretation, then either F is true, or G is true, or both are true. This is also sometimes referred to as the rule of cases.

De Morgan's Law for Disjunction[b]

$$\frac{I \not\models F \lor G}{I \not\models F, I \not\models G}$$

This rule states that if $F \lor G$ is not true, then both F is not true and G is not true. This is directly derived from De Morgan's laws in classical logic.

Implication Elimination[c]

$$\frac{I \models F \to G}{I \not\models F \mid I \models G}$$

This rule states that if $F \to G$ is true in an interpretation, then either F is false or G is true. This corresponds to the definition of material implication in classical logic.

Denial of Implication[c]

$$\frac{I \not\models F \to G}{I \models F, I \not\models G}$$

This rule states that if $F \to G$ is not true in an interpretation, then F must be true and G must be false. This is the contrapositive form of material implication.

Biconditional Elimination[d]

$$\frac{I \models F \leftrightarrow G}{I \models F \land G \mid I \models \neg F \land \neg G}$$

This rule states that if $F \leftrightarrow G$ is true in an interpretation, then either both F and G are true, or both F and G are false. This corresponds to the definition of a biconditional statement in classical logic.

Negation of Biconditional[d]

$$\frac{I \not\models F \leftrightarrow G}{I \models F \land \neg G \mid I \models \neg F \land G}$$

This rule states that if $F \leftrightarrow G$ is not true in an interpretation, then F and G have opposite truth values, i.e., either F is true and G is false, or F is false and G is true. This aligns with the concept of exclusive or (XOR).

[a] And-Elimination
[b] Or-Elimination
[c] Material Implication
[d] If and Only If Elimination

[a] Contrapositive of Conjunction Introduction
[b] Contrapositive of Disjunction Introduction
[c] Contrapositive of Implication
[d] Exclusive Or Elimination

Contradiction Introduction (Proof by Contradiction)

$$\frac{I \models F \quad I \not\models F}{I \models \bot}$$

Example 2.2.1. To prove that the formula

$$F : P \wedge Q \rightarrow P \vee \neg Q$$

is valid, assume that it is invalid and derive a contradiction:

1.	$I \not\models P\wedge \rightarrow P \vee \neg Q$	assumption
2.	$I \models P \wedge Q$	by 1 and semantics of $\rightarrow$
3.	$I \not\models P \vee \neg Q$	by 1 and semantics of $\rightarrow$
4.	$I \models P$	by 2 and semantics of $\wedge$
5.	$I\neg \models P$	by 3 and semantics of $\vee$
6.	$I \models \bot$	4 and 5 are contradictory

Example 2.2.2. To prove that the formula

$$F : (P \rightarrow Q) \wedge (Q \rightarrow R) \rightarrow (P \rightarrow R)$$

is valid, assume that it is invalid and derive a contradiction:

2.2.3 Proof Tree

A proof evolves as a tree.

- A branch is a sequence descending from the root.
- A branch is *closed* it contains a contradiction. Otherwise, the branch is *open*.
- It is a proof of the validity of F if every branch is closed; otherwise, each open branch describes a falsifying interpretation of F.

2.2.4 Derived Rules

The proof rules are sufficient, but **derived rules** can make proofs more concise. E.g., the rule of modus ponens:

$$\frac{I \models F \quad I \models F \rightarrow G}{I \models G}$$

The proof of the validity of the formula:

$$F : (P \rightarrow Q) \wedge (Q \rightarrow R) \rightarrow (P \rightarrow R)$$

1.	$I \not\models F$	assumption
2.	$I \models (P \rightarrow Q) \wedge (Q \rightarrow R)$	by 1 and semantics of $\rightarrow$
3.	$I \models P \rightarrow R$	by 1 and semantics of $\rightarrow$
4.	$I \models P$	by 3 and semantics of $\rightarrow$
5.	$I \not\models R$	by 3 and semantics of $\rightarrow$
6.	$I \not\models P \rightarrow Q$	2 and semantics of $\wedge$
7.	$I \not\models Q \rightarrow R$	2 and semantics of $\wedge$
8.	$I \not\models Q$	by 4, 6, and modus ponens
9.	$I \not\models R$	by 8, 7, and modus ponens
10.	$I \models \bot$	5 and 9 are contradictory

Chapter 3

Propositional Logic II

[3] [4] [5]

3.1 Equivalence and Implication, and Equisatisfiability

3.1.1 Equivalence and Implication

- Two formulas F_1 and F_2 are equivalent

$$F_1 \iff F_2$$

  iff $F_1 \leftrightarrow F_2$ is valid, i. e., for all interpretations I, $I \models F_1 \leftrightarrow F_2$.

- Formula F_1 implies formula F_2

$$F_1 \implies F_2$$

  iff $F_1 \rightarrow F_2$ is valid, i. e., for all interpretations I, $I \models F_1 \rightarrow F_2$.

- $F_1 \iff F_2$ and $F_1 \implies F_2$ are not formulas. They are semantic assertions.
- We can check equivalence and implication by checking satisfiability.

Example 3.1.1.

- $P \iff \neg\neg P$ means that $P \leftrightarrow \neg\neg p$ is valid.
- $P \rightarrow Q \iff \neg P \vee Q$ means that $P \rightarrow Q \leftrightarrow \neg P \vee Q$ is valid.

Exercise 3.1.1. Prove that

$$R \wedge (\neg R \vee P) \implies P.$$

Sol. content... □

3.2 Substitution

3.2.1 Substitution

Example 3.2.1. content...

Chapter 4

Problem Solving using SMT Solver

[6] [7] [8]

Chapter 5

First-Order Logic

Appendix A

Boolean Functions

A.1 Unary and Binary Boolean Function

Propositional Variable

Definition A.1.1. A propositional variable is an element of the set $\{0, 1\}$.

Example A.1.1. $P, Q, R, \ldots, \in \{0, 1\}$ and so on.

Truth Function

Definition A.1.2. Let $\mathbb{B} = \{0, 1\}$ be the *boolean domain*. Let $k \in \mathbb{N}$. A mapping

$$f : \mathbb{B}^k \to \mathbb{B}$$

is called a **truth function**.

Count of Truth Functions

Proposition A.1.1 *There are* $2^{(2^k)}$ *distinct truth functions on* k *variables.*

Proof. Let $f : \mathbb{B}^k \to \mathbb{B}$ be a truth function for $k \in \mathbb{N}$. Then

(Cardinality of Cartesian Product of Finite Sets)

$$\#(\mathbb{B}^k) = \#(\overbrace{\mathbb{B} \times \cdots \times \mathbb{B}}^{k \text{ times}}) = \overbrace{\#\mathbb{B}\#\mathbb{B} \cdots \#\mathbb{B}}^{k \text{ times}} = \overbrace{2 \cdot 2 \cdots 2}^{k \text{ times}} = 2^k.$$

(Cardinality of Set of All Mappings.)

$$\#(T^S) = \#\left\{f \subseteq S \times T : f \text{ is a mapping}\right\} = (\#T)^{(\#S)} \implies \#(\mathbb{B})^{\#(\mathbb{B}^k)} = 2^{(2^k)}$$

□

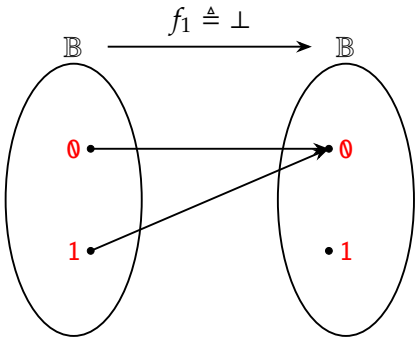
Unary Truth Functions

Corollary A.1.1 *There are 4 distinct unary truth functions:*

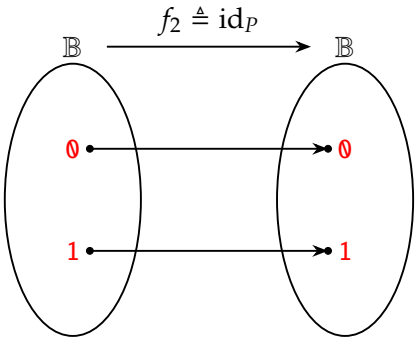
- *The contradiction function* $f(P) = \emptyset$
- *The tautology function* $f(P) = 1$
- *The identity function* $f(P) = P$
- *The logical not function* $f(P) = \neg P$

Proof. By Count of Truth Functions, there are $2^{(2^1)} = 4$ distinct truth functions on single variable. These can be depicted in a truth table as follows:

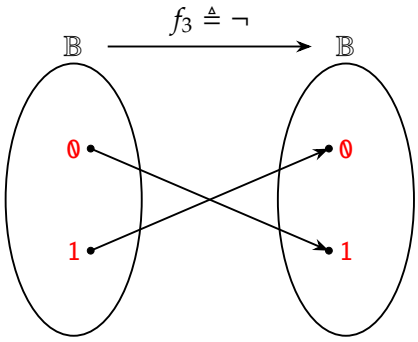
$P \in \mathbb{B}$	$\emptyset$	1
f_1	$\emptyset$	$\emptyset$
f_2	$\emptyset$	1
f_3	1	$\emptyset$
f_4	1	1



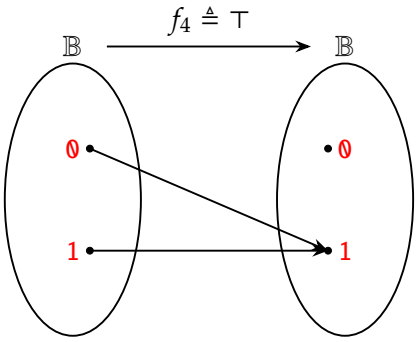
f_1 is the *contradiction function*.



f_2 is the *identity function*



f_3 is the *logical not function*



f_4 is the *tautology function*

□

Remark A.1.1. The structure is a non-associative magma, also known as a groupoid.

$\circ$	$\bot$	P	$\neg$	$\top$
$\bot$	$\bot$	$\bot$	$\top$	$\top$
P	$\bot$	P	$\neg$	$\top$
$\neg$	$\bot$	$\neg$	P	$\top$
$\top$	$\bot$	$\top$	$\bot$	$\top$

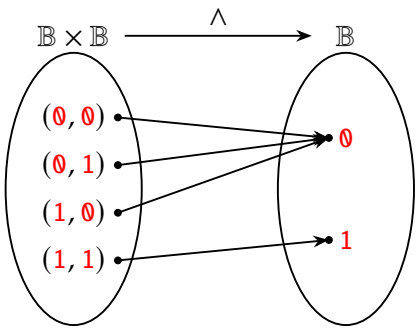
Binary Truth Functions

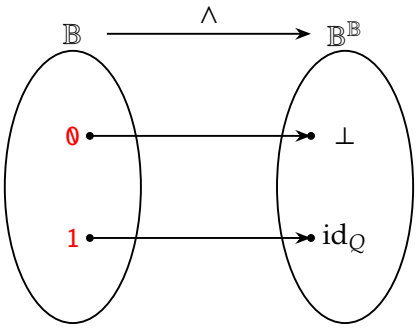
Corollary A.1.2 *There are 16 distinct binary truth functions:*

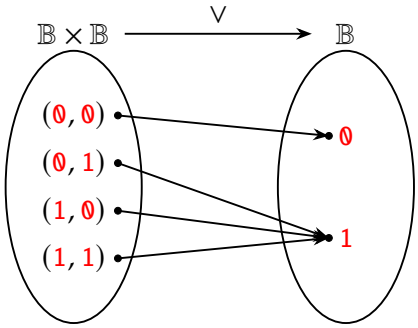
- *Two constant operations:*
  - $f_1(p,q) = 1$
  - $f_0(p,q) = 0$
- *Two projections:*
  - $\mathsf{Proj}_1(p,q) = p$
  - $\mathsf{Proj}_2(p,q) = q$
- *Two negated projections:*
  - $\overline{\mathsf{Proj}_1}(p,q) = \neg p$
  - $\overline{\mathsf{Proj}_2}(p,q) = \neg q$
- *The conjunction:* $p \wedge q$
- *The disjunction:* $p \vee q$
- *Two conditionals:*
  - $p \implies q$
  - $q \implies p$
- *The biconditional (iff):* $p \iff q$
- *The exclusive or (xor):* $\neg(p \iff q)$
- *Two negated conditionals:*
  - $\neg(p \implies q)$
  - $\neg(q \implies p)$
- *The NAND* $p \uparrow q$
- *The NOR* $p \downarrow q$

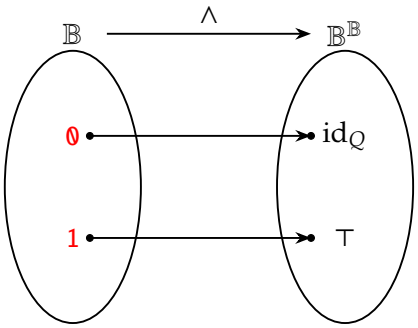
Proof. From Count of Truth Functions there are $2^{(2^2)} = 16$ distinct truth functions on 2 variable. These can be depicted in a truth table as follows:

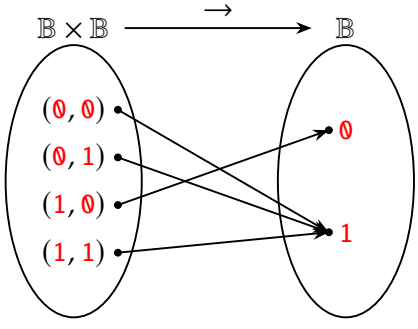
p	1	1	0	0
q	1	0	1	0
$f_0(p,q)$	0	0	0	0
$p \downarrow q$	0	0	0	1
$\neg(p \impliedby q)$	0	0	1	0
$\overline{\mathsf{Proj}_1}(p,q)$	0	0	1	1
$\neg(p \implies q)$	0	1	0	0
$\overline{\mathsf{Proj}_2}(p,q)$	0	1	0	1
$\neg(p \iff q)$	0	1	1	0
$p \uparrow q$	0	1	1	1
$p \wedge q$	1	0	0	0
$p \iff q$	1	0	0	1
$\mathsf{Proj}_2(p,q)$	1	0	1	0
$p \implies q$	1	0	1	1
$\mathsf{Proj}_1(p,q)$	1	1	0	0
$p \impliedby q$	1	1	0	1
$p \vee q$	1	1	1	0
$f_1(p,q)$	1	1	1	1

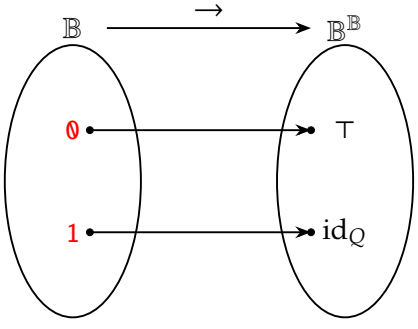


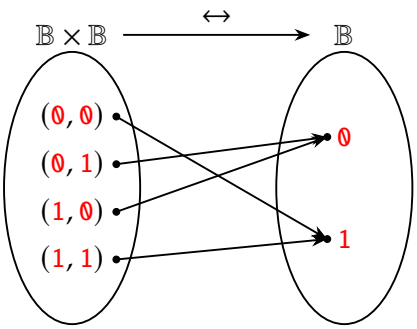


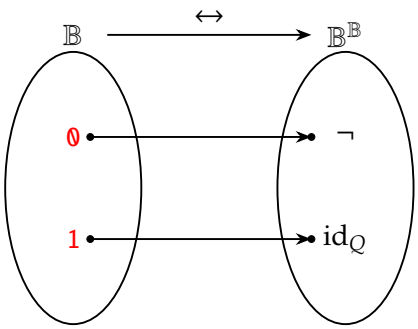









$\mathbb{B} \times \mathbb{B}$
$\leftrightarrow$
$\mathbb{B}$
$(0,0)$
$(0,1)$
$(1,0)$
$(1,1)$
0
1


$\mathbb{B}$
$\leftrightarrow$
$\mathbb{B}^{\mathbb{B}}$
0
1
$\neg$
id_Q

□

A.2 Backus-Naur Form (BNF)

Backus-Naur Form (BNF) is a notation technique for context-free grammars, often used to describe the syntax of languages used in computing. Introduced by John Backus and Peter Naur in the 1960s, BNF was initially developed to describe the syntax of the ALGOL 60 programming language.

A grammar defines a language by providing a set of production rules that describe how sentences in the language can be formed. BNF uses non-terminal symbols, terminal symbols, and production rules to define these grammars.

Non-terminal symbols are placeholders for patterns of terminal symbols that can be generated by applying production rules.

Terminal symbols are the actual symbols of the language's alphabet, and they appear in the strings generated by the grammar.

Production rules define how non-terminal symbols can be replaced with combinations of non-terminal and terminal symbols. BNF uses a specific syntax to describe these rules:

```
<rule-name> ::= <expression>
```

Extended BNF (EBNF) provides additional notation to simplify grammar definitions, such as optional elements and repetitions.

BNF is a powerful tool for defining the syntax of programming languages and has been fundamental in the development of many language specifications. For more information, refer to resources such as the *ALGOL 60 Report*, or textbooks on formal language theory and compiler design.

Example A.2.1. Here is an example of BNF describing simple arithmetic expressions:

```
<expression> ::= <term> | <term> "+" <expression>
<term> ::= <factor> | <factor> "*" <term>
<factor> ::= <number> | "(" <expression> ")"
<number> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Bibliography

[1] 오학주[교수 / 컴퓨터학과]. Cose419 lecture 1: Introduction to software analysis (1). YouTube, 2024. Accessed: 2024-06-28.

[2] 오학주[교수 / 컴퓨터학과]. Cose419 lecture 1: Introduction to software analysis (2). YouTube, 2024. Accessed: 2024-06-28.

[3] 오학주[교수 / 컴퓨터학과]. Cose419 lecture 4: Propositional logic (1). YouTube, 2024. Accessed: 2024-06-28.

[4] 오학주[교수 / 컴퓨터학과]. Cose419 lecture 4: Propositional logic (2). YouTube, 2024. Accessed: 2024-06-28.

[5] 오학주[교수 / 컴퓨터학과]. Cose419 lecture 4: Propositional logic (3). YouTube, 2024. Accessed: 2024-06-28.

[6] 오학주[교수 / 컴퓨터학과]. Cose419 lecture 5: Problem solving using smt solver (1). YouTube, 2024. Accessed: 2024-06-28.

[7] 오학주[교수 / 컴퓨터학과]. Cose419 lecture 5: Problem solving using smt solver (2). YouTube, 2024. Accessed: 2024-06-28.

[8] 오학주[교수 / 컴퓨터학과]. Cose419 lecture 5: Problem solving using smt solver (3). YouTube, 2024. Accessed: 2024-06-28.

[9] bycho211. Course overview. Naver Blog, 2017. Accessed: 2024-06-28.