A First Look at Propositional Logic

Andreas Klappenecker

There are many reasons why a computer scientist should study logic. For example, logic underlies the reasoning in mathematical statements. In particular, proving that a program meets its specification is argued with the help of logical deductions. We begin by studying propositional logic, which is one of the simplest logical theories. Nevertheless, propositional logic has a large number of applications; in particular, it provides the foundation for combinatorial circuits which implement the arithmetic logic unit of a processor. A nice property of propositional logic is that the truth of the statements formulated in this logic can be checked in a simple way by a computer. Surprisingly, propositional logic leads to the most intriguing open problem in computer science.

1 Motivation

A proposition is a statement that is either true or false. Propositional logic describes ways to combine true statements by means of connectives to produce other true statements. Let us have a look at an example.

If it is asserted that 'Jack is taller than Jill' and 'Jill can run faster than Jack' are true, then we can combine them to the true statement 'Jack is taller than Jill and Jill can run faster than Jack'. However, if Jill is actually taller than Jack, then the first statement is false and the combined statement is false as well.

Propositional logic allows us to formalize such statements. Additionally, the expressions become more concise. For example, if we abbreviate the proposition 'Jack is taller than Jill' by A, and the proposition 'Jill can run faster than Jack' by B, then the combined statement A and B is expressed in propositional logic in the form $A \wedge B$, where $\wedge$ is a connective formalizing the word 'and'.

The connectives provided by propositional logic are

Name of Connective	Symbol for Connective
negation	$\neg$
and	$\wedge$
or	$\vee$
exclusive or	$\oplus$
conditional	$\rightarrow$
biconditional	$\leftrightarrow$

The precise meaning of these connectives will be explained below.

In the next section, we introduce a tool that is widely used in computer science to specify formal languages. The Backus Naur form allows us to present the syntax of propositional logic in a concise way. The meaning of the connectives will be explained in Section 4.

2 The Backus Naur Form

A common way to specify a formal language is to use the so-called **Backus Naur form**. The Backus Naur form consist of a set of derivation rules of the form

$$\langle \textit{symbol} \rangle ::= \langle \textit{symbol}_1 \rangle \langle \textit{symbol}_2 \rangle \cdots \langle \textit{symbol}_n \rangle$$

The left hand side of a derivation consists of a single symbol, called a **nonterminal symbol**. The right hand side consists of a sequence of symbols. If a symbol occurs only on the right hand side of derivations, then it is called a **terminal symbol** or a **token**. A grammar consists of a finite number of such derivation rules.

One nonterminal symbol is designated as the start symbol. Given the start symbol, one applies a derivation rule that has the start symbol on the left hand side and replaces it with the right hand side of the derivation rule. For example, if $\langle \textit{symbol} \rangle$ is the start symbol, then the above rule would yield

$$\langle \textit{symbol}_1 \rangle \langle \textit{symbol}_2 \rangle \cdots \langle \textit{symbol}_n \rangle$$

Subsequently, derivation rules can be applied to replace any nonterminal symbol. If one obtains after a finite number of steps a sequence consisting only of terminal symbols, then the resulting sequence is said to belong to the language generated by the grammar.

This is really quite a simple process. Let us have a look at a few examples illustrating this concept.

Example 1. A single digit binary number can be specified in Backus Naur form by

$$\begin{array}{lcl} \langle \textit{binary digit} \rangle & ::= & 0 \\ \langle \textit{binary digit} \rangle & ::= & 1 \end{array}$$

The start symbol is $\langle \textit{binary digit} \rangle$ and it is the only nonterminal symbol. There are two terminal symbols, namely 0 and 1. We can apply either the first or the second derivation rule, and respectively obtain 0 or 1. The right hand side of both derivation rules consists of terminal symbols alone, so one cannot apply further derivation rules. Thus, the language generated by this grammar is $\{0, 1\}$.

One frequently has several choices of productions rules that have the same nonterminal symbol on the left hand side. Since such choices are common, the Backus Naur form allows one to combine the different derivation rules into a single derivation rule, where the alternatives are separated by $|$. For instance, the grammar of the previous example can be formulated more succinctly as

$$\langle \textit{binary digit} \rangle ::= 0 \mid 1$$

This simply means that the nonterminal $\langle \textit{binary digit} \rangle$ can be replaced by either 0 or 1.

The next example is slightly more elaborate.

Example 2. One can specify multidigit binary numbers in the form

$$\begin{array}{lcl}
\langle \textit{binary number} \rangle & ::= & \langle \textit{binary number} \rangle \langle \textit{binary digit} \rangle \\
\langle \textit{binary number} \rangle & ::= & \langle \textit{binary digit} \rangle \\
\langle \textit{binary digit} \rangle & ::= & 0 \mid 1
\end{array}$$

The start symbol is $\langle \textit{binary number} \rangle$. According to this grammar, 001 is a valid binary number. Indeed, given the start symbol $\langle \textit{binary number} \rangle$, one applies the first derivation rule twice to obtain $\langle \textit{binary number} \rangle \langle \textit{binary digit} \rangle \langle \textit{binary digit} \rangle$. One can apply the second derivation rule to the first nonterminal symbol to obtain

$$\langle \textit{binary digit} \rangle \langle \textit{binary digit} \rangle \langle \textit{binary digit} \rangle.$$

Applying the last derivation rule to each of these three nonterminal symbols, we can obtain 001 or any other three digit binary number. The language generated by this grammar consists of all binary numbers $\{0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$.

One caveat is that there might be several sequence of derivation rules that produce the same sequence of terminal symbols.

Remark. The Backus Naur form is one of the most common ways to specify formal languages, including programming languages, network protocols, and more. The languages that one can specify with the Backus Naur form are context-free, as the derivation rules are applied without regard to the context in which a nonterminal appears. A good source for everything related to formal languages is [A.V. Aho, J.D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Volume I: Parsing, Prentice-Hall, 1972].

3 The Language of Propositional Logic

The language **Prop** of propositional logic is defined by the following grammar.

$$\begin{array}{lcl}
\langle \textit{formula} \rangle & ::= & \neg \langle \textit{formula} \rangle \\
& \mid & (\langle \textit{formula} \rangle \wedge \langle \textit{formula} \rangle) \\
& \mid & (\langle \textit{formula} \rangle \vee \langle \textit{formula} \rangle) \\
& \mid & (\langle \textit{formula} \rangle \oplus \langle \textit{formula} \rangle) \\
& \mid & (\langle \textit{formula} \rangle \rightarrow \langle \textit{formula} \rangle) \\
& \mid & (\langle \textit{formula} \rangle \leftrightarrow \langle \textit{formula} \rangle) \\
& \mid & \langle \textit{symbol} \rangle
\end{array}$$

The nonterminal $\langle \textit{symbol} \rangle$ can be any letter, possibly with a nonnegative integer subscript. In other words, a $\langle \textit{symbol} \rangle$ is an element of the set

$$\mathbf{S} = \{a, a_0, a_1, \ldots, b, b_0, b_1, \ldots, z, z_0, z_1, \ldots\}.$$

The start symbol is $\langle \textit{formula} \rangle$. The terminal symbols are the elements in $\mathbf{S}$, the connectives, and the parentheses. We will refer to the elements in $\mathbf{S}$ as **propositional variables** or **variables** for short.

Remark. We could have used any other countable set $\mathbf{S}$ to name the variables, as long as it does not contain the parentheses and the connectives.

Example 3. The formula

$$((a \wedge b) \vee \neg c)$$

belongs to the language of propositional logic. Indeed, this can be seen as follows:

$$\begin{array}{lll} \langle \textit{formula} \rangle & \text{yields} & (\langle \textit{formula} \rangle \vee \langle \textit{formula} \rangle) \\ & \text{yields} & ((\langle \textit{formula} \rangle \wedge \langle \textit{formula} \rangle) \vee \langle \textit{formula} \rangle) \\ & \text{yields} & ((\langle \textit{formula} \rangle \wedge \langle \textit{formula} \rangle) \vee \neg \langle \textit{formula} \rangle). \end{array}$$

By applying the last rule three times, we get

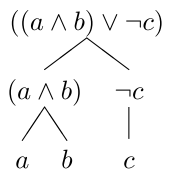
$$((\langle \textit{symbol} \rangle \wedge \langle \textit{symbol} \rangle) \vee \neg \langle \textit{symbol} \rangle),$$

and the symbols can be chosen to be a, b and c, respectively.

You might have wondered why each logical connective is enclosed in parentheses, except for the negation connective $\neg$. This ensures that we can associate a *unique* binary tree to each proposition, called the **formation tree**. The formation tree contains all subformulas of a proposition. A formation tree of a proposition p has a root labeled with p and satisfies the following rules:

T1. Each leaf is an occurrence of a propositional variable in p.
T2. Each internal node with a single successor is labeled by a subformula $\neg q$ of p and has q as a successor.
T3. Each internal node with two successors is labeled by a subformula aXb of p with X in $\{\wedge, \vee, \oplus, \rightarrow, \leftrightarrow\}$ and has a as a left successor and b as a right successor.

Example 4. The formation tree of the formula $((a \wedge b) \vee \neg c)$ is given by



Remark. In a course on compiler construction, you will learn how to write a parser for languages such as the one that we have specified for propositional logic. You can check out lex and yacc if you want to write a parser for propositional logic in C or C++ now. In Haskell, you can use for example the parser generator Happy.

4 The Semantics of Propositional Logic

Let $\mathbf{B} = \{\mathbf{t}, \mathbf{f}\}$ denote the set of truth values, where $\mathbf{t}$ and $\mathbf{f}$ represent *true* and *false*, respectively. We associate to the logical connective $\neg$ the function $M_\neg \colon \mathbf{B} \to \mathbf{B}$ given by

P	$M_\neg(P)$
$\mathbf{f}$	$\mathbf{t}$
$\mathbf{t}$	$\mathbf{f}$

Thus, $M_\neg(P)$ is true if and only if P is false. This justifies the name negation for this connective. The graph of the function $M_\neg$ given above is called the **truth table** of the negation connective. Similarly, we associate to a connective X in the set $\{\wedge, \vee, \oplus, \to, \leftrightarrow\}$ a binary function $M_X \colon \mathbf{B} \times \mathbf{B} \to \mathbf{B}$. The truth tables of these connectives are as follows:

P	Q	$M_\wedge(P,Q)$	$M_\vee(P,Q)$	$M_\oplus(P,Q)$	$M_\to(P,Q)$	$M_\leftrightarrow(P,Q)$
$\mathbf{f}$	$\mathbf{f}$	$\mathbf{f}$	$\mathbf{f}$	$\mathbf{f}$	$\mathbf{t}$	$\mathbf{t}$
$\mathbf{f}$	$\mathbf{t}$	$\mathbf{f}$	$\mathbf{t}$	$\mathbf{t}$	$\mathbf{t}$	$\mathbf{f}$
$\mathbf{t}$	$\mathbf{f}$	$\mathbf{f}$	$\mathbf{t}$	$\mathbf{t}$	$\mathbf{f}$	$\mathbf{f}$
$\mathbf{t}$	$\mathbf{t}$	$\mathbf{t}$	$\mathbf{t}$	$\mathbf{f}$	$\mathbf{t}$	$\mathbf{t}$

You should very carefully inspect this table! It is critical that you memorize and fully understand the meaning of each connective.

The semantics of the language **Prop** is given by assigning truth values to each proposition in **Prop**. Clearly, an arbitrary assignment of truth values is not interesting, since we would like everything to be consistent with the meaning of the connectives that we have just learned. For example, if the propositions a and b have been assigned the value $\mathbf{t}$, then it is reasonable to insist that $a \wedge b$ be assigned the value $\mathbf{t}$ as well. Therefore, we will introduce the concept of a valuation, which models the semantics of **Prop** in an appropriate way.

A **valuation** $v \colon \mathbf{Prop} \to \mathbf{B}$ is a function that assigns a truth value to each proposition in **Prop** such that

V1. $v[\![\neg a]\!] = M_\neg(v[\![a]\!])$
V2. $v[\![(a \wedge b)]\!] = M_\wedge(v[\![a]\!], v[\![b]\!])$
V3. $v[\![(a \vee b)]\!] = M_\vee(v[\![a]\!], v[\![b]\!])$
V4. $v[\![(a \oplus b)]\!] = M_\oplus(v[\![a]\!], v[\![b]\!])$
V5. $v[\![(a \to b)]\!] = M_\to(v[\![a]\!], v[\![b]\!])$
V6. $v[\![(a \leftrightarrow b)]\!] = M_\leftrightarrow(v[\![a]\!], v[\![b]\!])$

holds for all propositions a and b in **Prop**. The properties **V1**–**V6** ensure that the valuation respects the meaning of the connectives. We can restrict a valuation v to a subset of the set of proposition. If A and B are subsets of **Prop** such that $A \subseteq B$, and $v_A \colon A \to \mathbf{B}$ and $v_B \colon B \to \mathbf{B}$ are valuations, then v_B is called an **extension** of the valuation v_A if and only if v_B coincides with v_A when restricted to A.

The consistency conditions **V1-V6** are quite stringent, as the next theorem shows.

Theorem 1. *If two valuations v and v' coincide on the set $\mathbf{S}$ of symbols, then they coincide on the set $\mathbf{Prop}$ of all propositions.*

Proof. Seeking a contradiction, we assume that there exist two valuations v and v' that coincide on $\mathbf{S}$, but do not coincide on $\mathbf{Prop}$. Thus, the set

$$C = \{a \in \mathbf{Prop} \mid v[\![a]\!] \neq v'[\![a]\!]\}$$

of counter examples is not empty. Choose a counter example a in C of minimal length, where the length of the proposition is defined as the number of terminal symbols. Then a cannot be of the form $a = \neg b$, since the minimality of the counter example implies that $v[\![b]\!] = v'[\![b]\!]$, which implies

$$v[\![a]\!] = M_\neg(v[\![b]\!]) = M_\neg(v'[\![b]\!]) = v'[\![a]\!]\,.$$

Similarly, a cannot be of the form bXc for some propositions b and c in $\mathbf{Prop}$ and some connective X in $\{\wedge, \vee, \oplus, \rightarrow, \leftrightarrow\}$. Indeed, by the minimality of the counter example $v[\![b]\!] = v'[\![b]\!]$ and $v[\![c]\!] = v'[\![c]\!]$, which implies

$$v[\![a]\!] = M_X(v[\![b]\!]\,, v[\![c]\!]) = M_X(v'[\![b]\!]\,, v'[\![c]\!]) = v'[\![a]\!]\,.$$

Therefore, a must be a symbol in $\mathbf{S}$, but both valuations coincide on the set $\mathbf{S}$ of symbols, so a cannot be an element of C, which is a contradiction. $\square$

An **interpretation** of a proposition p in $\mathbf{Prop}$ is an assignment of truth values to all variables that occur in p. More generally, an interpretation of a set Y of propositions is an assignment of truth values to all variables that occur in formulas in Y. The previous theorem states that an interpretation of $\mathbf{Prop}$ has *at most* one extension to a valuation on $\mathbf{Prop}$.

It remains to show that each interpretation of $\mathbf{Prop}$ has an extension to a valuation. For this purpose, we define the **degree** of a proposition p in $\mathbf{Prop}$, denote $\deg p$, as the number of occurrences of logical connectives in p. In other words, the degree function satisfies the following properties:

D1. An element in $\mathbf{S}$ has degree 0.
D2. If a in $\mathbf{Prop}$ has degree n, then $\neg a$ has degree $n + 1$.
D3. If a and b in $\mathbf{Prop}$ are respectively of degree n_a and n_b, then aXb is of degree $n_a + n_b + 1$ for all connectives X in $\{\wedge, \vee, \oplus, \rightarrow, \leftrightarrow\}$.

Example 5. The proposition $((a \wedge b) \vee \neg c)$ is of degree 3.

Theorem 2. *Each interpretation of **Prop** has a unique extension to a valuation.*

Proof. We will show by induction on the degree of a proposition that an interpretation $v_0 \colon \mathbf{S} \to \mathbf{B}$ has an extension to a valuation $v \colon \mathbf{Prop} \to \mathbf{B}$. The uniqueness of this extension is obvious from Theorem 1.

We set $v(a) = v_0(a)$ for all a of degree 0. Then v is certainly a valuation on the set of degree 0 propositions.

Suppose that v is a valuation for all propositions of degree less than n extending v_0. If a is a proposition of degree n, then it has a unique formation tree.

The immediate successors of a in the formation tree are labeled by subformulas of a of degree less than n; hence, these successors have a valuation assigned. Therefore, v has a unique extension to a using the consistency rules **V1**–**V6**. We can conclude that v is a valuation on the set of all proposition of degree n extending v_0. Therefore, the claim follows by induction. $\square$

The key reason that the previous argument by induction works is that the formation tree is unique. If there would exist several different trees for a single formula, then such a recursive definition of a valuation would be ambiguous, and the definition of the valuation v might not be well-defined.

Remark. Perhaps you would like to see a more direct argument based on the grammar rather than on the degree of the formulas. One can use structural induction, a generalization of induction to so-called freely generated recursively defined sets. For a proof of the existence of valuations using structural induction, see [J.H. Gallier, *Logic for Computer Science – Foundations of Automatic Theorem Proving*, John Wiley & Sons, 1987].

In this section, we have been a little bit pedantic by distinguishing the purely syntactical form of a proposition such as $(a \to b)$ from its meaning $M_{\to}(a,b)$. Of course, it is a good idea to clearly distinguish between syntax and semantics until the semantics of the connectives is clearly understood. From now on, we will abuse notation and freely interpret $(a \to b)$ as the function $M_{\to}(a,b)$.

Summary. Informally, we can summarize the meaning of the connectives as follows:

1) The and connective $(a \land b)$ is true if and only if both a and b are true.
2) The or connective $(a \lor b)$ is true if and only if at least one of a, b is true.
3) The exclusive or $(a \oplus b)$ is true if and only if precisely one of a, b is true.
4) The implication $(a \to b)$ is false if and only if the premise a is true and the conclusion b is false.
5) The biconditional connective $(a \leftrightarrow b)$ is true if and only if the truth values of a and b are the same.

An interpretation of a subset S of **Prop** is an assignment of truth values to all variables that occur in the propositions contained in S. We showed that there exist a unique valuation extending an interpretation of all propositions.

5 Tautologies and Satisfiability

In the previous two sections, we have introduced the language of propositional logic and gave the propositions a meaning using valuations. In this section, we will see propositional logic "at work".

A proposition p is called a **tautology** if and only if $v[\![p]\!] = \mathbf{t}$ for all valuations v on **Prop**.

A proposition p is a tautology if and only if p evaluates to $\mathbf{t}$ under each interpretation of the variables in p. If the proposition p contains n variables, then we have to check all 2^n possible interpretations of p.

Example 6. Let us verify that $((a \to b) \leftrightarrow (\neg a \vee b))$ is a tautology. This proposition contains 2 variables, so we have to check $2^2 = 4$ cases. The truth table of this proposition can be obtained as follows:

a	b	$(a \to b)$	$(\neg a \vee b)$	$((a \to b) \leftrightarrow (\neg a \vee b))$
f	**f**	**t**	**t**	**t**
f	**t**	**t**	**t**	**t**
t	**f**	**f**	**f**	**t**
t	**t**	**t**	**t**	**t**

Thus, the above proposition is indeed a tautology.

A proposition p is called **satisfiable** if and only if there exists a valuation v such that $v[\![p]\!] = \mathbf{t}$. A proposition that is not sastisfiable is called **unsatisfiable**.

Example 7. We claim that $((a \oplus b) \to \neg(a \vee b))$ is satisfiable. Indeed, if we choose a valuation v such that $v[\![a]\!] = \mathbf{f}$ and $v[\![b]\!] = \mathbf{f}$, then $v[\![a \oplus b]\!] = \mathbf{f}$; hence, $v[\![((a \oplus b) \to \neg(a \vee b))]\!] = \mathbf{t}$.

Example 8. The proposition $(a \wedge \neg a)$ is unsatisfiable. Indeed, if we choose a valuation v such that $v[\![a]\!] = \mathbf{t}$, then $v[\![\neg a]\!] = \mathbf{f}$, so $v[\![(a \wedge \neg a)]\!] = \mathbf{f}$. The only other interpretation is $v'[\![a]\!] = \mathbf{f}$, which implies $v'[\![(a \wedge \neg a)]\!] = \mathbf{f}$. Therefore, the proposition $(a \wedge \neg a)$ is unsatisfiable, as claimed.

Discussion. The development of mathematics and computer science is driven by famous open problems. We digress a little bit and informally discuss some fundamental problems of computational complexity theory.

You might not be able to fully appreciate this discussion until you have taken a course on analysis of algorithms. However, knowing the significance of these questions will provide some additional motivation to study propositional logic carefully.

We say that an algorithm is **polynomial time** if and only if there exists some positive integer k such that for all inputs of length n the algorithm can compute the result in at most $n^k + k$ steps. We assume here that the algorithm is executed on a so-called Turing machine, a formal model of computation.

A decision problem is a computational problem that has a *yes* or *no* answer. The complexity class **P** consists of all computational decision problems that can be solved with a polynomial time algorithm.

Problem 1. *Does there exist a polynomial time algorithm to determine whether a proposition is satisfiable?*

This problem baffles computer scientist already for more than three decades! You might wonder why this problem is so challenging.

Let us digress to introduce some terminology. The complexity class **NP** consists, loosely speaking, of all decision problems such that someone can convince you of a *yes* answer with a short proof that can be checked by a program in polynomial time. Of course, **NP** contains the complexity class **P**.

The problem to decide whether a given proposition p is satisfiable is in **NP**, since a satisfying interpretation can serve as a proof that can be checked in polynomial time by evaluating the proposition under this interpretation. It is not difficult to show that Problem 1 has an affirmative answer if and only if **NP**=**P**. However, the question whether **NP** is equal to **P** is considered the most important open problem of computer science.

Another interesting problem concerns the tautology of propositions.

Problem 2. *Does there exist a polynomial time algorithm to determine whether a propositions is a tautology?*

The complexity class **coNP** consists, loosely speaking, of all decision problems such that someone can convince you of a *no* answer with a short proof that can be checked by a program in polynomial time.

The question whether a given proposition is a tautology is a typical example of a decision problem in **coNP**. Indeed, it suffices to produce a non-satisfying interpretation to prove that a proposition is *not* a tautology, and evaluating the proposition under this interpretation can be done by a program in polynomial time. Problem 2 has an affirmative answer if and only if **coNP**=**P**.

Problem 3. *Is the problem to decide whether a proposition is a tautology contained in the complexity class **NP**?*

If Problem 3 has an affirmative answer, then **NP**=**coNP**. On the other hand, if Problem 3 does not have an affirmative answer, then **NP**≠**coNP**, which implies **P**≠**NP**.

Remark. My guess is that **NP**≠**coNP**, whence **NP**≠**P**.

Remark. If you are interested in the precise definitions of the complexity classes **P**, **NP**, and **coNP**, then the survey paper [S. Cook, *The P versus NP problem*, `http://www.claymath.org/millennium/P_vs_NP/pvsnp.pdf`] is a good start. By the way, the Clay Mathematics Institute gives you one million reasons to study propositional logic.

6 Conclusion

We have seen that even propositional logic gives rise to interesting problems. If you are interested in learning more about propositional and predicate logic, then I can highly recommend the book [R. Smullyan, *First-Order Logic*, Dover, 1995]. This book served as a reference when writing these notes.

Remark. If you feel overwhelmed, then read again, but this time read the definitions *very carefully.* Whenever some term is defined, then try to understand its meaning with the help of some examples. Our textbook contains numerous examples (but lacks the development of the theory).