Computational Proof Theory

A Computer Implementation of Fitch Calculus

Joohee Jeong

August 9, 2023

github.com/jhjeong314/proofmood

Contents

1 Formal Language

1.1 Symbols, strings and languages

Language is constructed using *symbols*, which are considered undefined terms. The collection of symbols used in a language is referred to as an *alphabet*. We assume that the alphabet is a nonempty finite set.[1]

Sequences of symbols that are finite in length are referred to as *strings*. A *formal language* can be defined as a collection of strings composed of symbols.

It is common to refer to a formal language simply as a language.

Example 1.1 Let $\Sigma = \{a, \ldots, z, A, \ldots, Z\}$ be an alphabet. Followings are some simple examples of languages on Σ.

(1) $\varnothing$

(2) $\{a, bc, Po, sEo\}$ (Strictly speaking, 'a', 'bc', and so on, are finite sequences of symbols and should be represented as $\langle a \rangle$, $\langle b, c \rangle$, and so forth, respectively. However, for the sake of convenience, we will not be overly pedantic about this notation.)

(3) $\{a^n b c^n \mid n \in \mathbb{N}\}$ ⊣

Concatenation is an operation between strings. We denote the concatenation of strings x and y by xy. So if

$$x = a_1 a_2 \cdots a_n, \text{ and } y = b_1 b_2 \cdots b_m, \text{ with } a_i, b_j \in \Sigma,$$

then

$$xy = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$$

In this case, we say that the *length* of x is n and the length of y is m. We denote the length of a string x by $|x|$.

[1] In model theory, it is occasionally necessary to employ an infinite alphabet (countable or uncountable); however, for our specific purpose, a finite alphabet will suffice.

The set of symbols occurring in a string x is denoted by $\mathrm{set}(x)$. If $x = a_1 a_2 \cdots a_n$, then $\mathrm{set}(x) := \{a_1, a_2, \ldots, a_n\}$ and $|\mathrm{set}(x)| \leq |x| = n$.

A string is called an *empty string* if its length is zero. It is often denoted by λ. So $|\lambda| = 0$ and $\lambda x = x\lambda = x$ for any string x.

The set of all strings over an alphabet Σ with length n is denoted by Σ^n. The set of all strings over Σ is denoted by Σ^*, which is called the *Kleene star*. Therefore we have

$$\begin{aligned}\Sigma^* &= \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots = \{\lambda\} \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \\ &= \textstyle\bigcup_{n\in\mathbb{N}} \Sigma^n\end{aligned}$$

We may say that a language is just a subset of Σ^*.

The set of all nonempty strings over Σ is denoted by Σ^+. So $\Sigma^+ = \Sigma^* - \{\lambda\}$.

The *substring* relation between strings is defined as follows.

$$x \textit{ is a substring of } y \overset{\text{def}}{\Leftrightarrow} (\exists u, v \in \Sigma^*)(y = uxv) \tag{1.1}$$

If $u = \lambda$ in (1.1), then x is called a *prefix* of y. If $v = \lambda$ in (1.1), then x is called a *suffix* of y.

1.2 Grammar of a language

The languages given in [Example 1.1] are certainly not languages in the usual sense. As an example of a language that makes sense, consider the following.

$$L_{ar} \overset{\text{def}}{=} \{a,\ b,\ a+b,\ a*b,\ b+a*c,\ (c+a)*c, \ldots\}$$

L_{ar} is the set of arithmetic expressions on variables a, b, c and operators $+$, $*$, and parentheses. The subscript ar in L_{ar} means arithmetic. The alphabet of L_{ar} consists of the following 7 symbols.

$$\Sigma = \{a,\ b,\ c,\ +,\ *,\ (,\ )\}$$

Consider a language $L \subseteq \Sigma^*$ generated with the following rules.

(1) $a, b, c \in L$

(2) If $x, y \in L$ then $x + y \in L$ and $x * y \in L$.

(3) If $x \in L$ then $(x) \in L$.

(4) All members of L are obtained by applying the rules (1), (2), (3) repeatedly.

It seems that we have $L_{ar} = L$. But let us be more precise on rule (4).

Let $\mathcal{L}$ be the set of all subsets of Σ^* that satisfy (1), (2) and (3). $\mathcal{L}$ is nonempty since $\Sigma^* \in \mathcal{L}$. Then we define L_{ar} as follows.

$$L_{ar} \overset{\text{def}}{=} \bigcap \mathcal{L} \tag{1.2}$$

This definition of L_{ar} is called the top down approach. If we want to use the bottom up approach, then we can define L_{ar} as follows.

$$\begin{aligned} L_0 &:= \{a,\ b,\ c\} \\ L_n &:= \{(x),\ x+y,\ x*y \mid x, y \in L_{n-1}\} \cup L_{n-1} \text{ for } n \geq 1 \\ L_{ar} &:= \bigcup_{n\in\mathbb{N}} L_n \end{aligned} \tag{1.3}$$

(1.3) is the mathematicians way of expressing bottom up approach. In computer science, the bottom-up approach is typically described using *formal grammar*. A (formal) grammar G is defined to be a quadruple

$$G = (V, T, S, P), \tag{1.4}$$

where V is the set of *variable symbols*, T is the set of *terminal symbols*. These two sets are disjoint: i.e., $V \cap T = \varnothing$. Members of V are sometimes called the *non-terminal symbols* and we use N in place of V in this case.

The alphabet of the language defined by this grammar G is T. $S \in V$ is called the *start symbol*. Finally, P is the set of *production rules*. A production rule is of the form

$$x \to y, \quad \text{where } x \in (V \cup T)^+,\ y \in (V \cup T)^*,\ \text{set}(x) \cap V \neq \varnothing,\ \text{and } x \neq y. \tag{1.5}$$

Before we present the grammar for L_{ar}, let us take a look at an example of a very simple language to show how a grammar determines a language. Consider the following language

$$L = \{a^n b^n \mid n \in \mathbb{N}\} = \{\lambda, ab, a^2b^2, a^3b^3, \ldots\}$$

The alphabet Σ for this language is $\{a, b\}$. A grammar G that defines this language is given below.

$$\begin{aligned} V &= \{S\} \\ T &= \{a,\ b\} \\ S &= S \\ P &= \{S \to aSb, \quad S \to \lambda\} \end{aligned}$$

Let us see how we *derive* elements of L with the following examples:

$$S \Rightarrow \lambda, \tag{1.6}$$

$$S \Rightarrow aSb \Rightarrow a\lambda b \equiv ab \tag{1.7}$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb \tag{1.8}$$

$$\vdots \qquad \vdots$$

We obtained 3 elements λ, ab, $aabb$ of L by deriving these strings using our grammar G. It is clear that we can obtain all elements of L by such derivations.

In the grammar G, a *derivation sequence*, or simply a *derivation*, is a finite sequence of elements from $(V \cup T)^*$ that satisfy specific conditions, which we will describe shortly. Examples of derivation sequences are given in (1.6)–(1.8).

A derivation can yield an element of T^*. When we say that we obtain $z \in (V \cup T)^*$

from $w \in (V \cup T)^+$ by applying the production rule $x \to y$ of G, it means that there exist $u, v \in (V \cup T)^*$ such that $w = uxv$ and $z = uyv$. In symbols, we write

$$\exists u, v \in (V \cup T)^* \text{ s.t. } w = uxv \text{ and } z = uyv.$$

We denote this one-step derivation by

$$w \Rightarrow_G z$$

We may omit the subscript G when it is clear from the context which grammar we are using in the derivation.

When we obtain z from w by applying the production rules of G repeatedly, we write

$$w \Rightarrow_G^* z$$

Again we may omit the subscript G when it is appropriate. Please note that $\Rightarrow_G^*$ includes the case where $\Rightarrow_G$ is applied zero times, resulting in $w = z$.

Definition 1.2 We define the language $L(G)$ determined by a grammar G as follows.

$$L(G) \stackrel{\text{def}}{=} \{w \in T^* \mid S \Rightarrow_G^* w\} \qquad \dashv$$

Now we see that L_{ar} is determined by the following grammar

$$V = \{E, I\}$$
$$T = \{a,\ b,\ c,\ +,\ *,\ (,\ )\}$$
$$S = E$$

with production rules

$$E \to I$$
$$E \to (E)$$
$$E \to E + E$$
$$E \to E * E$$
$$I \to a$$
$$I \to b$$
$$I \to c$$

It is cumbersome to write the production rules as above. We usually write them more succinctly as follows. Here the symbol '|' means 'or'.

$$E \to I \mid (E) \mid E + E \mid E * E$$
$$I \to a \mid b \mid c$$

We may view the variable E as *expression* and I as *identifier*. Now let us derive $a + b * c \in L_{ar}$ with this grammar.

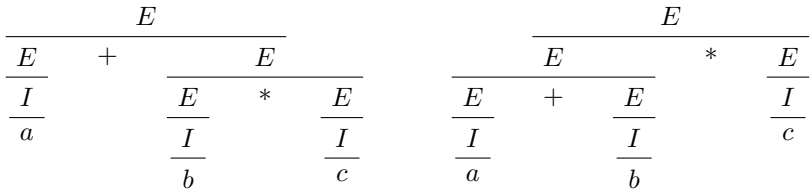
$$\begin{aligned} E &\Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + E * E \\ &\Rightarrow a + I * E \Rightarrow a + b * E \Rightarrow a + b * I \Rightarrow a + b * c \end{aligned} \tag{1.9}$$

Remark 1.3 We have defined L_{ar} in three different ways. The top down method $L_{ar} = \bigcap \mathcal{L}$, the bottom up method $L_{ar} = \bigcup_{n\in\mathbb{N}} L_n$, and the grammar method $L_{ar} = L(G)$. These are shown at (1.2), (1.3), and [Definition 1.2] respectively.

We omit the proof that these three methods indeed determine the same language. ⊣

The derivation sequence (1.9) is easily understood using the tree diagram shown in [Figure 1.1]. The one on the left shows the derivation (1.9). Such a tree is called the *derivation tree*, or *parse tree*.

Figure 1.1: Two different derivation trees for $a + b * c$



In this grammar L_{ar}, terminal symbols do not appear on the left of any production rule and hence cannot be replaced by any other strings. Therefore they appear in parse trees only at terminal nodes or *leaves*.

The tree on the right of [Figure 1.1] shows another derivation of the same string $a+b*c$ with the same grammar. This shows that the grammar is *ambiguous*.

The derivation sequence corresponding to this tree is

$$\begin{aligned} &E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow I + E * E \Rightarrow a + E * E \\ &\quad \Rightarrow a + I * E \Rightarrow a + b * E \Rightarrow a + b * I \Rightarrow a + b * c \end{aligned}$$

One way of avoiding ambiguity of grammars is to introduce the concept of *priority*, or *precedence* between operators.

Since multiplication is considered to have priority over addition in most cases, the tree on the right of [Figure 1.1] should be thought of as the correct derivation tree of $a+b*c$. We can introduce the priority of operators by introducing new variables T, F and modifying the grammar as follows.

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid I \\ I &\to a \mid b \mid c \end{aligned}$$

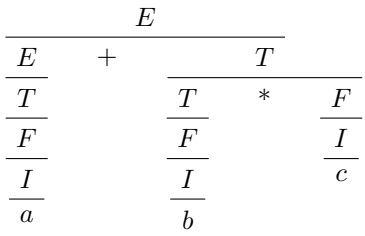
Here T stands for *term* and F stands for *factor*. We call the old grammar G_{ar1} and new grammar G_{ar2}. Be aware that T is also used to denote the set of terminal symbols. We hope that this will not cause any confusion.

Let us derive $a + b * c$ with G_{ar2}.

$$\begin{aligned} &E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow I + T \Rightarrow a + T \\ &\quad \Rightarrow a + T * F \Rightarrow a + F * F \Rightarrow a + I * F \Rightarrow a + b * F \\ &\quad \Rightarrow a + b * I \Rightarrow a + b * c \end{aligned}$$

This derivation tree is shown in [Figure 1.2].

Figure 1.2: Parse tree of $a + b * c$ in an unambiguous grammar



G_{ar2} is an unambiguous grammar. Proving the unambiguity of a formal language is a challenging task, and we will refrain from presenting the associated theory in this context.

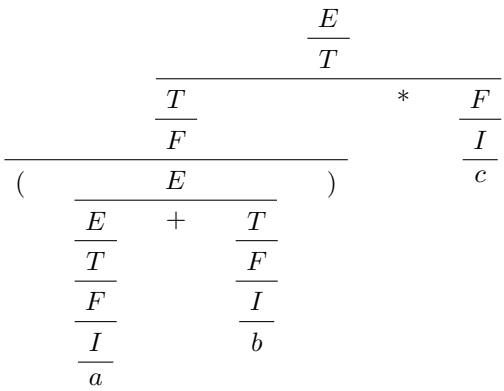
Be aware that every string in $L(G_{ar2})$ has a unique parse tree but may have several different derivation sequences. As an example, we show two different derivation sequences of $a + b$ in G_{ar2}.

(1) $E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + I \Rightarrow E + b \Rightarrow T + b \Rightarrow F + b \Rightarrow I + b \Rightarrow a + b$

(2) $E \Rightarrow E + T \Rightarrow T + T \Rightarrow T + F \Rightarrow T + I \Rightarrow T + b \Rightarrow F + b \Rightarrow I + b \Rightarrow a + b$

So far all expressions we have considered do not have parentheses. As an example of a string with parentheses we take $(a + b) * c$ and let us get a parse tree for this string with G_{ar2}.

Figure 1.3: Parse tree of $(a + b) * c$



Exercise 1.4 Draw parse trees for $(a + b) + c$ and $a + (b + c)$ in G_{ar2}. ⊣

1.3 Classes of formal languages

Definition 1.5 In a formal grammar $G := (V, T, S, P)$, the production rules P have elements of the form $x \to y$ where $x \in (V \cup T)+$.[2] If we restrict x to be a member of V (or

[2]This was defined at (1.5).

V^1 to be rigorous), we call this grammar a *context-free grammar*. A language determined by a context-free grammar is called a *context-free language*. ⊣

Most computer related languages are context-free. For instance, G_{ar1}, G_{ar2} are both context-free. If we weaken the condition $x \in V$ for context-free grammar on production rule $x \to y$ to $|x| \leq |y|$, then we obtain a *context-sensitive grammar*.[3] A language determined by a context-sensitive grammar is called a *context-sensitive language*.

If we denote the class of context-free languages by $\mathcal{L}_{CF}$ and the class of context-sensitive languages by $\mathcal{L}_{CS}$, the inclusion relation $\mathcal{L}_{CF} \subsetneqq \mathcal{L}_{CS}$ holds.

Some classes of languages are even bigger than $\mathcal{L}_{CS}$, namely $\mathcal{L}_{REC}$(recursive languages) and $\mathcal{L}_{RE}$(recursively enumerable languages). $\mathcal{L}_{RE}$ corresponds to grammars with no restriction on production rules.

For language classes smaller than $\mathcal{L}_{CF}$, we have $\mathcal{L}_{DCF}$(deterministic context-free languages) and $\mathcal{L}_{REG}$(regular languages). The latter is very important and used widely.

The inclusion relations among the classes of languages are as follows. This is called the *Chomsky hierarchy*.

$$\mathcal{L}_{REG} \subsetneqq \mathcal{L}_{DCF} \subsetneqq \mathcal{L}_{CF} \subsetneqq \mathcal{L}_{CS} \subsetneqq \mathcal{L}_{REC} \subsetneqq \mathcal{L}_{RE} \tag{1.10}$$

The grammar corresponding to $\mathcal{L}_{reg}$ is called the *regular grammar*. We will not discuss regular grammar here. Regular languages are often defined using *regular expressions*. In practice regular expressions are much more frequently used than regular grammars.

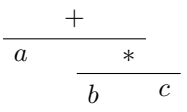
There is another way of defining formal languages other than grammars and regular expressions—we may define them using some kind of abstract machines.

Some examples of abstract machines are finite state accepter(deterministic and non-deterministic), pushdown automaton(deterministic and non-deterministic), and Turing machine. We will not go into details about these machines here.

1.4 Parse trees and Syntax trees

If we compare the left picture of [Figure 1.1] and [Figure 1.2], we see they are different. But if we construct a new tree from each tree by choosing the terminal nodes only, then we get the same tree as follows.

Figure 1.4: Abstract syntax tree of $a + b * c$



The diagram shown in [Figure 1.4] is called an *abstract syntax tree(AST)*, or simply *syntax tree*, which is different from parse tree. Please be cautioned that people often use these two terms, syntax tree and parse tree(or derivation tree) interchangeably. This is probably because parse trees are sometimes called the *concrete syntax trees(CST)*.

Certainly CST's have more information than AST's. But in many circumstances these extra information do not help in doing anything. For instance, evaluation, substitution, replacement, determining the type of an expression, etc. can be done with AST's. We will

[3] A different formulation for the definition of the context-sensitive grammar is possible.

need additional attributes (such as whether the symbol is an operator or an operand, the precedence between operators etc.) to the nodes of the AST's to do these things.

We will be mostly concerned with syntax trees(i.e., AST's) only in this book.

Obtaining syntax trees from parse trees for strings containing parentheses poses a problem. For instance, the parse tree for $(a+b)*c$ is shown in [Figure 1.3]. There are 7 terminal nodes in this tree. But how can we choose only the terminal nodes and build a tree?

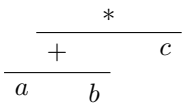
Come to think of it, syntax trees were never defined rigorously, while parse trees were. Syntax trees are called *abstract* because they do not represent every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure (i.e., partial order structure).

On the other hand, the syntax of CST is *concrete* in the sense that it represents every detail. For instance, grouping parentheses are explicit in the tree structure.

Traditionally, parentheses are commonly classified as a terminal symbol, but in reality, they do not function as such. Rather than being terminals, parentheses serve the purpose of establishing precedence among operators. They dictate the *order of association* in terms of syntactic sense or the *order of evaluation* in terms of semantic sense. Parentheses do not fall under the category of operands or operators themselves; instead, they act as delimiters for grouping expressions. Unlike operators (including functions and predicates), which can be prefixed, infixed, or postfixed, parentheses can be considered as both-fixed in a sense.

So, for instance, the syntax tree of $(a + b) * c$ should be drawn without parentheses as follows. This tree structure overrides the usual precedence between operators.

Figure 1.5: Syntax tree of $(a + b) * c$



1.5 BNF and EBNF

We may use the following grammar for the language $L(G_{ar2})$. We will call this grammar G_{ar3}.

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid a \mid b \mid c$$

Describing a grammar in this manner is usually done in textbooks on formal language theory. But in practice, we use a more practical notation called BNF(Bachus-Naur Form) or EBNF(Extended BNF). The grammar for the grammar above in BNF is as follows. We use ⟨expr⟩, ⟨term⟩ and ⟨factor⟩ for E, T and F respectively.

```
⟨expr⟩ ::= ⟨expr⟩ + ⟨term⟩ ¦ ⟨term⟩
⟨term⟩ ::= ⟨term⟩ * ⟨factor⟩ ¦ ⟨factor⟩
⟨factor⟩ ::= ( ⟨expr⟩ ) ¦ a ¦ b ¦ c
```

In BNF, we use a meaningful string instead of a single character for the name of a variable symbol, and each string is enclosed in angle brackets. The terminal symbols are written as

they are. The symbol ::= is used to denote the production rule. Some people enclose the terminal symbols with quotes but this practice is in fact introduced in EBNF.

EBNF can be considered as an extension of BNF. In EBNF, we can use the following symbols. The symbol [ ] is used to denote that the symbol inside it is optional. The symbol { } is used to denote that the symbol inside it can be repeated zero or more times. The symbol ( ) is used to group symbols and in most cases used with the choice symbol ¦.

Below we show some production rules in both BNF and EBNF

```
BNF
<if statement> ::= if <expr> then <statement> else <statement> ¦
                   if <expr> then <statement>
EBNF
<if statement> ::= "if" <expr> "then" <statement> ["else" <statement>]
BNF
<unsigned integer> ::= <digit> ¦ <unsigned integer><digit>
EBNF
<unsigned integer> ::= <digit> { <digit> }
BNF
<expr> ::= <expr> + <expr> ¦ <expr> - <expr> ¦ <expr> * <expr> ¦
           <expr> - <expr>
EBNF
<expr> ::= <expr> ( "+" ¦ "-" ¦ "*" ¦ "/" ) <expr>
```

In EBNF, we may use regular expressions to increase the expressive power. For instance, we may use [0-9] instead of <0¦1¦2¦3¦4¦5¦6¦7¦8¦9>. We may also use [a-zA-Z] instead of a¦b¦..¦z¦A¦B¦..¦Z.

2 Parsing

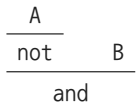
In this book, parsing will mean the process of obtaining a syntax tree(i.e., AST) from input text. Text may consist of a single line or multiple lines. We will consider the former case only for the time being.

The main programming language for our implementation will be Python. Later, we will be using HTML, CSS and JavaScript for the implementation of a web-based system.

2.1 Token

As an example of an input text, we take the logical formula '`not A and B`', which is parsed to the following syntax tree. This tree has root at the bottom, and we will continue to do so from now on.

Figure 2.1: Syntax tree of a logical formula



In this example, we see that not only '`A`' and '`B`' but also '`not`' and '`and`' are terminal symbols. Instead of calling '`not`' and '`and`' symbols, we call them *tokens*.

When a text to be parsed is input, we must first transform the text into a list of *tokens*, which refers to the smallest unit or element of a language. A token is a sequence of characters or symbols that represents a meaningful entity in a language. The process of transforming the input text into a list of tokens is called *tokenization*. For instance, '`not A and B`' is tokenized to produce the list `[not, A, and, B]`.

Token is just a string in a narrow sense. If we are to extend the meaning of token it is possible to view it as an object with several attributes. For instance, '`not`' and '`and`' are operators, and '`A`' and '`B`' are operands. Also '`not`' is a unary prefix operator and '`and`' is a binary infix operator. The precedence of '`not`' is higher than that of '`and`'. And '`and`' is left associative. Note that most binary infix operators are left associative but the logical connective '`imp`'(short for 'implication', or '$\rightarrow$') is right associative.

The attributes associated with each token help the parser understand the structure and

semantics of the code, allowing it to perform tasks like syntax validation, code generation, or interpretation. By extending the meaning of a token to include its attributes, the parser gains a more comprehensive understanding of the code being parsed, enabling it to make informed decisions and facilitate accurate analysis of the input text.

We can think of other kinds of tokens, such as function symbols, predicate symbols, and quantifiers, but for the time being, we assume that there are only two kinds of tokens: operators and operands.

The process of categorizing tokens in this manner is known as *tokenizing* or *lexical analysis*. The role of performing this task falls upon functions known as *tokenizers* or *lexers*. These tokenizers are responsible for analyzing the input stream and identifying the individual tokens, assigning them to their respective categories based on their characteristics.

2.2 Lexer

Let's further develop the grammar G_{ar3} we discussed earlier in Extended Backus-Naur Form (EBNF) by introducing additional rules and expanding it into an extended grammar, which we will refer to as G_{ar4}.

```
⟨expr⟩ ::= ⟨term⟩ { ("+" ¦ "-") ⟨term⟩ }
⟨term⟩ ::= ⟨factor⟩ { ("*" ¦ "/") ⟨factor⟩ }
⟨factor⟩ ::= "(" ⟨expr⟩ ")" ¦ ⟨atom⟩
⟨atom⟩ ::= ⟨identifier⟩ ¦ ⟨numeral⟩
⟨identifier⟩ ::= ⟨letter⟩ { ⟨letter⟩ ¦ ⟨digit⟩ }
⟨letter⟩ ::= [a-z]
⟨numeral⟩ ::= ⟨positive_digit⟩ { ⟨digit⟩ }
⟨digit⟩ ::= [0-9]
⟨positive_digit⟩ :: = [1-9]
```

We implement a class called `Token` and define a function named `tokenizer()`. Additionally, we create a function called `testTokenizer()` specifically designed to test the functionality of the tokenizer. The code is shown below.

```
1  # "value" argument is input through tokenizer().
2  # So a certain degree of validity check is done already.
3
4  class Token():
5    def __init__(self, value):
6    self.value = value
7      if value in ("+", "-"):
8        self.token_type = 'op_type1' # precedence 1
9      elif value in ("*", "/"):
10       self.token_type = 'op_type2' # precedence 2
11     elif value == "(":
12       self.token_type = 'lparen'
13     elif value == ")":
14       self.token_type = 'rparen'
15     elif value.isdecimal():
16       self.token_type = 'numeral'
17     elif value.isalnum() and value[0].isalpha():
18       self.token_type = 'identifier'
```

```
    else:
      raise ValueError(f"'{value}' is invalid (Token)")

  def __str__(self):
    return f'{self.value} ({self.token_type})'
```

Then we define `tokenizer()` as follows. Note that we allow only ASCII printable characters in our language. We also assume that the input text is a single line for the time being.

```
import re

def tokenizer(input_text):
  tokens = []
  # Split the input text into a list of tokens at word boundaries and
  # whitespaces, then remove empty strings and strip off leading and
  # trailing whitespaces.
  li = [s.strip() for s in re.split(r"\b|\s", input_text, re.ASCII)
                   if s.strip()]
  for s in li: # s is a string
    if not s.isascii():
      raise ValueError(f"'{s}' is invalid (non-ASCII)")
    if not (set(s).issubset("+-*/()") or      # operator or parenthesis
            (s.isdecimal() and s[0]!='0') or  # numeral
            (s.isalnum() and s[0].isalpha() and s.islower())):
                                              # identifier
      raise ValueError(f"'{s}' is invalid (non-token)")
    if set(s).issubset("+-*/()") and len(s) > 1:
      # split string of consecutive operators into individual characters
      for c in s: # c is an operator character
        tokens.append(Token(c))
    else:
      tokens.append(Token(s))

  return tokens
```

Finally, the function `testTokenizer()` is defined as follows.

```
def testTokenizer(input_text):
  try:
    tokens = tokenizer(input_text)
  except ValueError as e:
    print(f"Tokenizer: {e}")
  else:
    for t in tokens:
      print(t)
```

Some sample runs of `testTokenizer()` are shown below.

```
testTokenizer("first + second* (hello + c1)*a23")
```

```
first (identifier)
+ (op_type1)
```

```
second (identifier)
* (op_type2)
( (lparen)
hello (identifier)
+ (op_type1)
c1 (identifier)
) (rparen)
+ (op_type1)
a23 (identifier)
```

```
testTokenizer("first + second* +Hello + 23+2")
testTokenizer("first + second*-hello + 023+2")
```

```
Tokenizer: 'Hello' is invalid (non-token)
Tokenizer: '023' is invalid (non-token)
```

2.3 Parser for Language of Arithmetic

2.3.1 Addition, Subtraction, Multiplication, and Division

We will now proceed to construct a parser that can handle the grammar G_{ar4}. We will use the following class to represent a node of a syntax tree.

```
class Node:
  def __init__(self, token, children=None):
    self.token = token # the node is labeled with a Token object
    self.children = children if children else [] # list of Node objects

    def __str__(self):
    return self.build_polish_notation()

  def build_polish_notation(self):
    ret_str = f"{self.token.value}"
    if self.children:
      ret_str += ' '
    ret_str += ' '.join(child.build_polish_notation()
                        for child in self.children)
    return ret_str
```

As evident in the code, the resulting AST is displayed using Polish notation. For instance, the AST for the expression `1 * 2 + 3` is displayed as `+ * 1 2 3`, while the AST for the expression `1 + 2 * 3` is represented as `+ 1 * 2 3`.

In our parser, we create a separate method for each variable, such as `<expr>`, `<term>`, `<factor>`, and so on. This approach is known as *recursive descent parsing*.[1]

We define our `Parser` class as follows.

```
class Parser:
  def __init__(self, tokens):
    self.tokens = tokens
```

[1]The recursive descent parsing is a top-down parsing technique that constructs a parse tree from the top and the input is read from left to right. The recursive descent parsing is a special case of LL parsing, where the first L stands for left-to-right scan of the input, and the second L stands for leftmost derivation.

```
4       self.current_token = None
5       self.index = -1
6       self.advance()  # set self.current_token to
7                       # the first(i.e. self.index=0) element of tokens
8
9     def advance(self): # increment self.index and set self.current_token
10      self.index += 1
11      if self.index < len(self.tokens):
12        self.current_token = self.tokens[self.index]
13      else:
14        self.current_token = None
15
16    def parse(self):
17      return self.expr() # expr() corresponds to the starting symbol <expr>
18
19    def expr(self):
20      node = self.term()
21
22      while(self.current_token is not None and
23            self.current_token.token_type in ('op_type1')):
24        # If we are at '+' in "a + b * c - ..." then the next token is '-'
25        # because we will consume tokens by self.advance() and self.term().
26        token = self.current_token
27        self.advance()
28        right_term = self.term()
29        node = Node(token, [node, right_term]) # left associative
30
31      return node
32
33    def term(self):
34      node = self.factor()
35
36      while(self.current_token is not None and
37            self.current_token.token_type in ('op_type2')):
38        token = self.current_token
39        self.advance()
40        right = self.factor()
41        node = Node(token, [node, right])
42
43      return node
44
45    def factor(self):
46      if(self.current_token is not None and
47         self.current_token.token_type == 'lparen'):
48        self.advance()
49        node = self.expr()
50        if(self.current_token is not None and
51           self.current_token.token_type == 'rparen'):
52          self.advance()
53        else:
54          raise SyntaxError("Expected ')' after expression, in factor()")
55      else:
```

```
56        node = self.atom()
57
58      return node
59
60    def atom(self):
61      if self.current_token is not None:
62        token = self.current_token
63        if token.token_type in ('numeral', 'identifier'):
64          self.advance()
65          return Node(token)
66        else:
67          raise SyntaxError(f"Expected numeral or identifier, in atom(): token")
68      else:
69        raise SyntaxError("Unexpected end of input, in atom()")
```

We obtain the AST of `input_text` by the following function.

```
def parse_input(input_text):
  tokens = tokenizer(input_text)
  parser = Parser(tokens)
  ast = parser.parse() # ast = Abstract Syntax Tree
  if parser.current_token is not None:
    raise SyntaxError(f"Unexpected token parser.current_token at " +
      f"parser.index, in parse_input(). Expected end of input.")
  return ast
```

We can test the parser with the following function.

```
def testParser(input_text):
  try:
    tree = parse_input(input_text)
  except ValueError as e:
    print(f"ValueError: {e}")
  except SyntaxError as e:
    print(f"SyntaxError: {e}")
  else:
    print(tree)
```

Below are some sample runs of the `testParser()` function. In a successful run, the output is the Polish notation representation of the AST. In case of an unsuccessful run, the output is the error message raised by the parser, providing information about the type and location of the error.

```
testParser("a + b * (c - d) + ab")
testParser("(a/b + 102)*(const - 2*var)")
# Some invalid inputs.
testParser("c - a + UpperCaseVar")
testParser("c1 - a + UpperCaseVar")
testParser("a + + b")
testParser("a + b *")
testParser("-a + b *")
testParser("a b")
```

```
+ + a * b - c d ab
* + / a b 102 - const * 2 var
```

```
ValueError: 'UpperCaseVar' is invalid (non-token)
ValueError: 'UpperCaseVar' is invalid (non-token)
SyntaxError: Expected numeral or identifier, in atom(): + (op_type1)
SyntaxError: Unexpected end of input, in atom()
SyntaxError: Expected numeral or identifier, in atom(): - (op_type1)
SyntaxError: Unexpected token b (identifier) at 1, in parse_input(). \
  Expected end of input.
```

2.3.2 Unary prefix/postfix and Exponentiation operators

The grammar G_{ar4} is lacking several common syntax elements found in arithmetic expressions. For example, it does not support unary prefix operators like the minus sign (-) or unary postfix operators like the exclamation mark (!). Additionally, it does not include the exponentiation operator (^). We will extend the grammar G_{ar4} to G_{ar5} to support these new operators.

Precedence among these new operators must also be considered. For example, the exponentiation operator (^) has higher precedence than the unary minus sign (-), which in turn has higher precedence than the operators (*) and (/). The unary postfix operator (!) has the highest precedence of all. We show some equivalent expressions below.

```
-a^b = -(a^b)
a^b! = a^(b!)
-a! = -(a!)
```

The exponentiation operator (^) should be right associative because $a^{b^c} = a^{(b^c)}$ and not $(a^b)^c$, which is interpreted as a^{bc} in mathematics. We will extend the grammar G_{ar4} to G_{ar5} to support these new operators.

The unary prefix operators (-) pauses a subtle problem. Consider $-a + b$ and $a + -b$. The first expression is good, but the second one looks a bit ugly and should be written as $a + (-b)$. So, a *negative term* can be the first term of an expression in itself but cannot be a term in the middle of an expression. It should be parenthesized if it appears as a non-first term of an expression.

In order to handle the unary (-) in G_{ar5}, we introduce a new variable symbol <nterm> for negative terms. The new production rules for <expr> are shown below.

```
<expr> ::= (<term> | <nterm>) { ("+" | "-") <term> }
<nterm> ::= "-" { "-" } <term>
<term> ::= <factor> { ("*" | "/") <factor> }
..
```

By the way, I realized that each method in `Parser` class, which is a recursive descent parser for the corresponding non-terminal symbol, has a token type checking routine at the beginning. It looks like

```
if self.current_token is not None and
   self.current_token.token_type == 'lparen':
  self.advance()
  ..
```

I have added a new method called `check_token_type()` to the `Parse` class. This method simplifies the code by eliminating repetitive code blocks (shown above) that were present throughout.

In the following code, `token_types` is either a tuple of strings or a string. If `token_types` is a tuple, then the method checks if the current token type is in the tuple. If `token_types` is a string, then the method checks if the current token type is equal to the string.

```
def check_token_type(self, token_types):
  # token_types can be a string or a tuple of strings
  token = self.current_token
  if token is None:
    return False
  elif(type(token_types) is not tuple): # must be a string in this case
      return token.token_type == token_types
  elif len(token_types) == 1:
    return token.token_type == token_types[0]
  elif token.token_type in token_types:
    return True
  else:
    return False
```

In order to handle ⟨nterm⟩, we added `nterm()` method and modified the `expr()` method as follows.

```
def expr(self):
  if(self.current_token is not None and
    self.current_token.value == '-'): # unary minus
    node = self.nterm()
  else: # not a negative term
    node = self.term()

  while self.check_token_type('op_bin_1'): # '+' or '-'
    token = self.current_token
    self.advance()
    right_term = self.term()
    node = Node(token, [node, right_term])

  return node

def nterm(self):
  token = self.current_token
  # For the first visit only, token.value == '-' is  guaranteed
  #   because we have checked it in self.expr().
  # But for subsequent recursive calls it can be otherwise.
  if(token is None or token.value != '-'):
    node = self.term()
  else:
    token.token_type = 'op_unary_prefix'
    self.advance()
    unary_node = self.nterm() # recursive call
    node = Node(token, [unary_node])

  return node
```

We modified the `build_polish_notation()` method of `Node` class to be able to show the token types. The token type is shown when we set the value of the newly introduced argument `opt` to `True`. The `testParser()` function was modified too.

```
def build_polish_notation(self, opt=False):
  ret_str = (f"self.token.value(self.token.token_type)" if opt
    else f"self.token.value")
  if self.children:
    ret_str += ' '
  ret_str += ' '.join(child.build_polish_notation(opt)
                      for child in self.children)
  return ret_str

def testParser(input_text, showOperType=False):
  try:
   ..
  else:
    print(ast.build_polish_notation(showOperType))
```

Some test results are shown below.

```
testParser("-a*b2 + c")
testParser("-a*(--b2) + c")
testParser("first + second* (hello + (-c1))+a23")
testParser("a - (-b)", showOperType=True)
# Some invalid expressions.
testParser("a + -b")
```

```
- - - a - * b c
+ - * a b2 c
+ - * a - - b2 c
+ + first * second + hello - c1 a23
-(op_bin_1) a(identifier) -(op_unary_prefix) b(identifier)
SyntaxError: Expected numeral or identifier at 2, in atom(): - (op_bin_1)
```

Before we dig into expressions having unary postfix operators, let us think what a ⟨factor⟩ really is. We can regard ⟨factor⟩ as an operand of a binary operator having precedence 2, namely (*) or (/), and a term is formed when the factors are combined together. This is exactly what the following production rule says.

```
⟨term⟩ ::= ⟨factor⟩ { ("*" ¦ "/") ⟨factor⟩ }
```

As we mentioned earlier, postfix operators have higher precedence than binary operators. So it is natural to introduce another variable symbol ⟨factor_post⟩ for a *postfix factor* and modify the production rules for ⟨factor⟩ as follows. Assume that we have two postfix unary operators (!) and (').

```
⟨factor⟩ ::= ⟨factor_post⟩ { ("!" ¦ "'") }
⟨factor_post⟩ ::= "(" ⟨expr⟩ ")" ¦ ⟨atom⟩
```

Another thing that we need to consider now is exponentiation. The exponentiation operator ^ has precedence higher than 2 and lower than that of a postfix operator, and it is right associative. So let us introduce another variable symbol factor_exp and modify the production rules for ⟨factor⟩ further accordingly.

Now the full list of the production rules of G_{ar5} is shown below.

```
<expr> ::= (<term> | <nterm>)  ("+" | "-") <term>
<nterm> ::= "-"  "-"  <term>
<term> ::= <factor>  ("*" | "/") <factor>
<factor> ::=  <factor_exp> "^"  <factor_exp>
<factor_exp> ::= <factor_post>  ("!" | "'")
<factor_post> ::= "(" <expr> ")" | <atom>
<atom> ::= <identifier> | <numeral>
..
```

Methods of `Parser` class related to various factors were modified as follows.

```
def factor(self):
  node = self.factor_exp()

  if self.check_token_type('op_bin_exp'): # '^'
    token = self.current_token
    self.advance()
    right_factor = self.factor() # recursive call for right associativity
    node = Node(token, [node, right_factor])

  return node

def factor_exp(self):
  node = self.factor_postfix()

  while self.check_token_type('op_postfix'):
    token = self.current_token
    self.advance()
    node = Node(token, [node])

  return node

def factor_postfix(self):
  if self.check_token_type('lparen'):
    self.advance()
    node = self.expr()
    if self.check_token_type('rparen'):
      self.advance()
    else:
      raise SyntaxError("Expected ')' after expression at " +
                        f"self.index, in factor_postfix()")
  else:
    node = self.atom()

  return node
```

The `Token` class and the `tokenizer()` function were modified accordingly, but we omit them here. Instead we show them in the next section.

Some test runs are shown below.

```
testParser("(-a!+b)*b*c!!'")
testParser("a^b^c")
testParser("(a)^(---b'!)^c")
testParser("(-a^b!)^c")
```

```
* * + - ! a b b ' ! ! c
^ a ^ b c
^ a ^ - - - ! ' b c
^ - ^ a ! b c
```

2.3.3 Function symbols

Functions are an integral part of mathematics. Therefore, we extend G_{ar5} to G_{ar6} in order to incorporate functions into the grammar. Relevant part of the new grammar is shown below.

```
..
<factor> ::=  <factor_exp> "^"  <factor_exp>
<factor_exp> ::= <factor_post>  ("!" | "'")
<factor_post> ::= "(" <expr> ")"  | <func_call> | <atom>
<func_call> ::= <func_symb> '(' <expr> {',' <expr>} ')'
  # 0-ary functions are not allowed
<atom> ::= <identifier> | <numeral>
..
```

These symbols should be implemented so that users can easily define their own symbols.

We define a list of `(string, arity)` tuples called `FUNCTION_SYMBOLS` to represent function symbols. The first element of each tuple is the name of the function symbol and the second element is the arity, i.e., the number of arguments.

The `Token` class was modified as follows. Only the relevant parts are shown.

```
class Token:
  def __init__(self, value):
    # You can put more function symbols here.
    # For example, ('tan', 1), ('log', 1), ('choose', 2), ('g', 2), ('f1', 1).
    FUNCTION_SYMBOLS = dict([('sin', 1), ('cos', 1), ('max', 2), ('min', 2), ('f', 3)])

    self.value = value
    self.arity = None
    self.precedence = None
    # input value is guaranteed to be a valid token
    if value == ",":
      self.token_type = 'comma'
    elif value in ("+", "-"):
    ..
    elif value in FUNCTION_SYMBOLS:
        self.token_type = 'func_symb'
        self.arity = FUNCTION_SYMBOLS[value]
        self.precedence = 5
    ..
```

The `Parser` class was modified as follows. Only the relevant parts are shown.

```
1 def factor_postfix(self):
2   if self.check_token_type('lparen'):
3     self.advance()
4     node = self.expr()
5     if self.check_token_type('rparen'):
```

```
      self.advance()
    else:
      raise SyntaxError(f"Expected ')' after expression at {self.index} " +
                  f"in factor_postfix(), but {self.current_token} is given.")
  elif self.check_token_type('func_symb'):
    node = self.func_call()
  else:
    node = self.atom()

  return node

def func_call(self):
  if self.current_token is not None:
    token = self.current_token
    if self.check_token_type('func_symb'):
      self.advance()
      if self.check_token_type('lparen'):
        self.advance()
        args = [] # list of arguments of the function

        while True:
          args.append(self.expr())
          if self.check_token_type('comma'):
            self.advance()
          elif self.check_token_type('rparen'):
            break
          else:
            raise SyntaxError("Expected ',' or ')' after function argument at " +
                      f"{self.index} in, but {self.current_token} is given.")

        # arity check
        if token.arity is None or token.arity != len(args):
          raise SyntaxError(f"Function {token.value} expects {token.arity} " +
                            f"arguments, but {len(args)} were given")

        self.advance()
        return Node(token, args)

      else:
        raise SyntaxError(f"Expected '(' after function symbol at {self.index}" +
                          f" in func_call(), but {self.current_token} is given.")
    else:
      raise SyntaxError(f"Expected function symbol at {self.index} in" +
                        f" func_call(), but {token} is given.")
  else:
    raise SyntaxError("Unexpected end of input, in func_call()")
```

So far, the parsed result is output only in polish notation. We added two methods `build_RPN()` and `build_latex_infix()` to the `Node` class to output the result in these format.

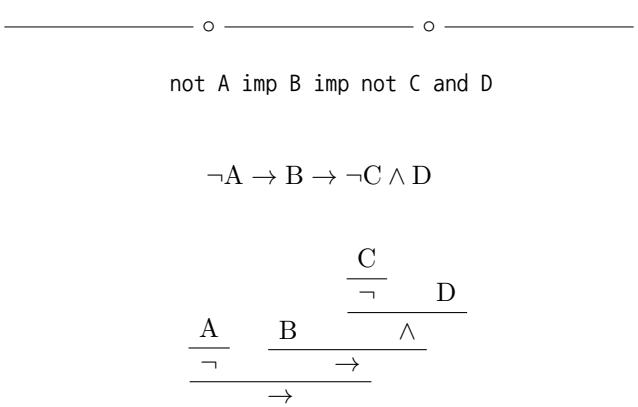
`build_RPN()` is easily written using the post-order traversal, while `build_latex_infix()` is non-trivial. Please refer to the source code for details.

2.3.4 Drawing the AST in bussproof style

For creating tree diagrams, the LaTeX `bussproofs` package is available. However, it poses a challenge when working with Jupyter Notebook or Google Colab since it is not directly usable in these environments.

I attempted to find Python graphic packages that could fulfill this task, but unfortunately, I couldn't find any suitable options. Although the `graphviz` library is very nice, it doesn't offer a convenient way to draw the Abstract Syntax Tree (AST) in the style of the `bussproofs` package.

As a result, I wrote my own solution. The code is available in `arith7_parse.py`, and it utilizes only the basic functions of `matplotlib`.



2.4 Language of Logic

2.4.1 Propositional Logic

The grammar of the language of propositional logic is relatively simple.

```
<expr> ::= { <term> "imp" } <term> | <term> { ( "iff" | "xor") <term> }
<term> ::= <factor> { ("and" | "or") <factor> }
<factor> ::= { "not" } '(' <expr> ')' | { "not" } <atom>
<atom> ::= [A-Z] { [A-Za-z0-9_] } | "bot"
```

One distinction between arithmetic and logic is that, in arithmetic, operator symbols such as $+$, $-$, $*$, and $/$ can be directly typed from the keyboard, whereas in logic, the symbols like $\neg$, $\wedge$, $\vee$, $\rightarrow$ are not readily available on standard keyboards.

So, for logical connective symbols, we use tokens like `not`, `and`, `or`, `imp`, `iff`, `xor` and `bot` and use the method `build_infix_latex()` to convert them to the corresponding symbols. Of course we have the method `draw_tree()` too to draw the AST of a logical formula.

Precedence among logical connectives is as follows:

- '`bot`' ($\bot$) is a logical connective but behaves like an atomic formula. It has the highest precedence.
- '`not`' ($\neg$) has the highest precedence among the connectives other than '`bot`'.
- '`and`' ($\wedge$) and '`or`' ($\vee$) have the same precedence, which is lower than '`not`'.

- '`imp`' ($\rightarrow$), '`iff`' ($\leftrightarrow$)and '`xor`' ($\nleftrightarrow$) have the same precedence, which is lower than '`and`' and '`or`'. (Note that $A \leftrightarrow B$ is logically equivalent to $\neg(A \texttt{ xor } B)$ and therefore the symbol $\nleftrightarrow$ is used for '`xor`'.)

Syntactic association of logical connectives is as follows:

- $\neg$ and $\rightarrow$ are right-associative.
- $\wedge$, $\vee$ and all other binary connectives are left-associative.

We have the `draw_tree()` method to draw AST in Jupyter Notebook environment. But for the purpose of writing articles in LaTeX, this is not good enough. We need a method that converts the AST to LaTeX source that can be compiled with `bussproofs.sty`. So I wrote a method called `build_bussproof()` to achieve this goal.

Now the `testParser()` function has total of 5 options: `polish`, `RPN`, `infix_latex`, `tree` and `bussproof`.

```
def testParser(input_text, showOption='polish', operOpt=False):
# showOption ::= 'polish' | 'RPN' | 'infix_latex' | 'tree' | 'bussproof'
# 'bussproof' output is LaTeX source text
# operOpt has effect only when showOption == 'polish' or 'RPN'
from IPython.display import display, Math

try:
  ast = parse_text(input_text)
except ValueError as e:
  print(f"ValueError: e")
except SyntaxError as e:
  print(f"SyntaxError: e")
else:
  if showOption=='polish':
    print(ast.build_polish_notation(operOpt))
  elif showOption=='RPN':
    print(ast.build_RPN(operOpt))
  elif showOption=='infix_latex':
    s = ast.build_infix_latex()
    print(s) # latex source text
    display(Math(f"$s$")) # render LaTeXed expression
  elif showOption=='tree':
    ast.draw_tree()
  elif showOption=='bussproof':
    s = ast.build_bussproof()
    print(s)
  else:
    raise ValueError(f"Invalid showOption: showOption")
```

Following is a sample run of the codes in this section.

```
input_formula = "not A imp B imp not C and D"
testParser(input_formula) # default: polish notation
testParser(input_formula, 'RPN')
testParser(input_formula, 'infix_latex')
testParser(input_formula, 'tree')
testParser(input_formula, 'bussproof')
```

```
imp not A imp B and not C D
A not B C not D and imp imp
\neg A \rightarrow B \rightarrow \neg C \wedge D
```

$$\neg A \rightarrow B \rightarrow \neg C \wedge D$$

```
\begin{prooftree}
\AxiomC{$A$}
\UnaryInfC{$\neg$}
\AxiomC{$B$}
\AxiomC{$C$}
\UnaryInfC{$\neg$}
\AxiomC{$D$}
\BinaryInfC{$\wedge$}
\BinaryInfC{$\rightarrow$}
\BinaryInfC{$\rightarrow$}
\end{prooftree}
```

$$\dfrac{\dfrac{A \quad \neg}{\neg A} \quad \dfrac{\dfrac{B \quad \dfrac{\dfrac{C}{\neg} \quad D}{\wedge}}{\rightarrow}}{\rightarrow}}{\rightarrow}$$

You can view/access all the codes mentioned above in `propositional_logic_parse.py` and `propositional_logic_parser.ipynb`.

———————— ○ ———————— ○ ————————

Underscore is allowed in the names of *propositional variables*, which can be understood of as another name for atoms. For example, '`A_1 and B_2`' is a valid formula and rendered as $A_1 \wedge B_2$ in `'infix_latex'` option and in `'tree'` option.

When the name of a propositional variable has length greater than 1, it is rendered in roman font. This is because the math italic font doesn't look good for strings with length greater than 1. You can clearly see the difference between $Henry$ and *Henry*. So I decided to use 'Henry' instead of $Henry$. Actually '*Henry*' looks better than 'Henry' but unfortunately `matplotlib` doesn't support `textsl` font.

When there are multiple underscores in the name of a propositional variable, only the last occurrence is rendered as subscript. For instance the formula `A_1 imp Bob_12 and Binary_Node_ij` is rendered as $A_1 \rightarrow \text{Bob}_{12} \wedge \text{Binary\_Node}_{ij}$ in `'infix_latex'` option and in `'tree'` option.

```
instr_li = ["Cats_and_Dogs_12", "Cats_and_Dogs_", "Cats_and_Dogs", "Cats"]
for instr in instr_li:
  print(identifier_to_latex(instr))
```

```
{\rm Cats\_and\_Dogs}_{12}
{\rm Cats\_and\_Dogs}
{\rm Cats\_and}_{Dogs}
{\rm Cats}
```

———————— ○ ———————— ○ ————————

The formula '`A and B and C`' is parsed as $(A \wedge B) \wedge C$ and rendered as $A \wedge B \wedge C$ in latex-infix form. We use $A \wedge B \wedge C$ because $(A \wedge B) \wedge C$ and $A \wedge (B \wedge C)$ are logically equivalent.

The formula '`A and B or C`' is parsed as $(A \wedge B) \vee C$ and rendered as $(A \wedge B) \vee C$ in latex-infix form. We use $(A \wedge B) \vee C$ because $(A \wedge B) \vee C$ and $A \wedge (B \vee C)$ are not logically equivalent.

The formula '`A imp B imp C`' is parsed as $A \to (B \to C)$ and rendered as $A \to (B \to C)$ in latex-infix form. We use $A \to (B \to C)$ because $A \to (B \to C)$ and $(A \to B) \to C$ are not logically equivalent.

Note that $\leftrightarrow$ is semantically associative which means $(A \leftrightarrow B) \leftrightarrow C$ and $A \leftrightarrow (B \leftrightarrow C)$ are logically equivalent. $\nleftrightarrow$ is semantically associative too.

People often use $A \leftrightarrow B \leftrightarrow C$ to mean A, B and C are all equivalent to each other. But this is not a good practice because $A \leftrightarrow B \leftrightarrow C$ is not equivalent to $(A \leftrightarrow B) \wedge (B \leftrightarrow C)$. Likewise $A \to B \to C$ is not equivalent to $(A \to B) \wedge (B \to C)$. Nevertheless, I will sometimes use $A \Leftrightarrow B \Leftrightarrow C$ to mean $\vDash A \leftrightarrow B$ and $\vDash B \leftrightarrow C$ informally. Likewise, I will use $A \Rightarrow B \Rightarrow C$ to mean $\vDash A \to B$ and $\vDash B \to C$ informally.

2.4.2 1st-order Logic

You can access all codes mentioned in this subsection in `first_order_logic_parse.py` and `first_order_logic_parser.ipynb`.

The language of 1st-order logic, denoted by `FirstOrder`, is traditionally defined as follows: The alphabet comprises 8 categories. The initial three categories, namely variables, constants, and functions, are utilized to construct *terms*. The subsequent four categories, predicates, equality, connectives, and quantifiers, are employed to combine terms in order to form *formulas*. Lastly, there exists a category of auxiliary symbols, such as parentheses and commas. These symbols do not appear in AST and thus can be regarded as non-terminal symbols[2] without any concerns.

`FirstOrder` is roughly defined as the union of terms and formulas, whose definitions are given later in [Defn. 2.1] and [Defn. 2.2], respectively.

Among those 8 categories of symbols, equality (`=`), connectives (`not`, `and`, `or`, `imp`, `iff`, `xor`, `bot`), quantifiers (`forall`, `exists`) and 3 auxiliary symbols (`(`, `)`, `,`) are fixed, or *reserved*.

The remaining 4 categories of symbols, namely variables (`VAR`), constants (`CONST`), functions (`FUNC`) and predicates (`PRED`), are user-defined. We will use the following conventions to distinguish them.

```
<var> ::= ( [u-z] ¦ [i-n] ) { [0-9] }
<const> ::= [a-e] { [A-Za-z0-9_] } ¦ <numeral>
<func> ::= [f-h] { [A-Za-z0-9_] }
<pred> ::= [A-Z] { [A-Za-z0-9_] }
<numeral> ::= [1-9] { [0-9] }
```

We can always declare and use a symbol against the naming convention above as we wish. For instance we can use `min`, `max`, `choose` etc. as binary function symbols. We can also use `prime`, `even`, `odd` etc. as unary predicate symbols. These user-defined symbols must be used as prefix symbols.

[2]Here, the 'non-terminal symbol' does not mean the 'variable symbol' used in the derivation of words in formal languages.

Predicate symbols are sometimes called the *relation symbols*. But we will use the term *predicate symbol* throughout this book.

Things would have been simple if we had only prefix notation. But we also have infix and postfix notation. To make matters worse, the same symbol is sometimes used as different tokens. For instance, (-) is sometimes used as a unary prefix function symbol (without parentheses) and sometimes used as a binary infix function symbol.

Infix function symbols may have different precedences and we have to take care of them. But infix predicate symbols do not have precedences because they take terms as operands (as function symbol does) and 'return' a formula, which belongs to a different category. An expression like $x < y \approx z$ is normally interpreted to mean $(x < y) \wedge (y \approx z)$, and we don't have to worry about the precedence between $<$ and $\approx$.

Now, let's proceed to define the languages of first-order, namely terms and formulas. We will provide an informal mathematical definition of these languages. The detailed EBNF (Extended Backus-Naur Form) definition will not be presented here, as it may complicate understanding. Instead, you can refer to the precise definition in the code located in `first_order_logic_parse.py`.

Definition 2.1 A *term* is defined as follows.

- A variable is a term.
- A constant is a term.
- Variables and constants are called the *primitive terms*.
- If $t_1, \ldots, t_n$ are terms and f is an n-ary prefix function symbol, where $n \geq 1$, then $f(t_1, \ldots, t_n)$ is a term.
- If t_1, t_2 are terms and $*$ is a binary infix function symbol, then $t_1 * t_2$ is a term.
- If t is a term and $'$ is a unary postfix function symbol, then t' is a term.
- If t is a term and if $-$ is considered a unary prefix function symbol, then $-t$ is a term.
- The set of terms is the smallest set satisfying the above conditions. ⊣

Definition 2.2 A *formula* is defined as follows.

- If $t_1, \ldots, t_n$ are terms and P is an n-ary prefix predicate symbol, where $n \geq 1$, then $P(t_1, \ldots, t_n)$ is a formula. If P is a 0-ary predicate symbol, then P is a formula.
- If t_1, t_2 are terms and $\approx$ is a binary infix predicate symbol, then $t_1 \approx t_2$ is a term. Equality (=) is a special case of binary infix predicate symbol.
- Formulas built from above two are called the *atomic formulas*. Non-atomic formulas, which are called the *compound formulas* are defined as follows.
- $\perp$ is a formula.
- If α is a formula and x is a variable, then $\forall x\, \alpha$ and $\exists x\, \alpha$ are formulas. The symbol $\forall$ and $\exists$ are called the *quantifiers*. $\forall$ is *universal* quantifier and $\exists$ is *existential* quantifier.

  An expression of the form $\forall x$ or $\exists x$, i.e., a quantifier followed by a variable is called a *determiner*.A determiner is neither a term nor a formula.

- If α is a formula, then $\neg\alpha$ is a formula.
- If α, β are formulas and $\circ$ is a binary infix connective symbol such as $\to$, $\leftrightarrow$, $\nleftrightarrow$, $\wedge$, $\vee$, then $\alpha \circ \beta$ is a formula.
- The set of formulas is the smallest set satisfying the above conditions. ⊣

Formulas of the form $\forall x\, \alpha$ and $\exists x\, \alpha$ are called a *universal formula* and an *existential formula* respectively. In a quantified formula, the formula α after the determiner is called the *scope* of the determiner(or of the quantifier)). α is called a *matrix* of the quantified formula.

A formula that has no occurrence of determiner is called a *quantifier free formula* or q.f. formula.

Formulas of the form $\neg\alpha$, $\neg\neg\alpha$, $\alpha \to \beta$ and $\alpha \leftrightarrow \beta$ are called the *negation formulas*, *double negation formulas*, *implication formula* and *biconditionals* respectively.

In an implication formula $\alpha \to \beta$, α is called an *antecedent* and β is called a *consequent*. In other formulas of the form $\alpha \circ \beta$, α and β are simply called the *left-hand side* and the *right-hand side* respectively.

Note that a formula of the form $\forall x\, \alpha \to \beta$ is not a universal formula. It is an implication formula because determiner has higher precedence than implication or any other connectives. $\forall x\, \alpha \to \beta$ is not equivalent to $\forall x (\alpha \to \beta)$. It should be interpreted as $(\forall x\, \alpha) \to \beta$.

Some people place a dot(.) between a determiner and its scope. For instance, $\forall x\, .\, \alpha$. This notation is somewhat confusing in a formula like $\forall x\, .\, \alpha \to \beta$ because the dot seems to strongly separate the determiner and the rest of the formula, and the formula might be interpreted as $\forall x (\alpha \to \beta)$ instead of $(\forall x\, \alpha) \to \beta$.

One more thing to note is that the scope of a determiner is always a formula. For instance, $\forall x\, x < y$ is to be interpreted as $\forall x (x < y)$, not $(\forall x\, x) < y$ which is not a well-formed formula. Maybe it is a good practice to put parentheses around the scope of a determiner in such a situation. To repeat, we normally do not write $\forall x (\alpha(x))$ because this expression has too many parentheses. But it is advised to write $\forall x (x < y)$ for readability.

We utilize the subsequent reserved words (and symbols) in association with types. For instance "emptyset" and "infty" are reserved words with type `CONST`, and `"!"` and `"^#"` are reserved words with type `OPER_POST`.

```
CONSTS = [ "emptyset", "infty" ]
OPER_PRE = [ "-" ]
OPER_POST = [ "!", "'", "^#", "^+", "^-", "^*", "^o", "^inv" ]
OPER_IN_1 = [ "+", "-", "cap", "cup", "oplus" ]
OPER_IN_2 = [ "*", "/", "%", "times", "div", "otimes", "cdot" ]
OPER_IN_3 = [ "^" ]
PRED_IN = [ "!=", "<", "<=", ">", ">=", "in", "nin", "subseteq",
            "nsubseteq", "subsetneqq", "supseteq", "nsupseteq",
            "supsetneqq", "divides", "ndivides", "sim", "simeq",
            "cong", "equiv", "approx" ]
```

Certain symbols can be found in multiple categories. In such instances, the symbol's type and arity are initially assigned during tokenization and later confirmed during parsing.

The appearance of a symbol can vary depending on its type. For example, the symbol `"*"` is typically displayed as an infix operator $*$. However, when it is used in `"^*"` as a

postfix operator, it is rendered as a superscript. For instance the string `"x^*"` (consisting of two tokens) is rendered as x^*. Special care is needed for tokens `"^o"` and `"^inv"`. They are rendered as x° and x^{-1} respectively.

The reserved function symbols which are given `OPER_*` types will be called 'operator symbols'. The user-defined function symbols, which are described in page 29, are given the type `FUNC_PRE`.

Prefix operator symbols have precedence 1 and do not use parentheses pair around its argument. So we write `"-x"` instead of `"-(x)"`. Postfix operator symbols have precedence 4. Infix operator symbols have precedences from 1 to 3. All user-defined function symbols are n-ary prefix with $n \geq 1$.

All infix predicate symbols have the same precedence. All other predicate symbols are n-ary prefix with $n \geq 0$.

The precedence of a token between different types are as follows:

$$\text{Parenthesis} > \text{Function} > \text{Predicate} > \text{Quantifiers} > \text{Connectives}$$

For example,

$$\forall x\, P(x) \to a + b > c \ \wedge R(x)$$

is to be interpreted as

$$\big(\forall x\, P(x)\big) \to \Big(\big((a + b) > c\big) \ \wedge R(x)\Big).$$

The numbered precedence of a token is applicable to operator types only. For example, `a + b * c ^ d !` is to be interpreted as

$$a + (b * (c^{(d!)}))$$

User-defined Symbols

For the time being, we have to follow the following rules for user-defined symbols.

```
⟨var⟩ ::= ( [u-z] ¦ [i-n] ) [ '_' { [0-9] } ]
⟨const⟩ ::= [a-e] { [A-Za-z0-9_] }
⟨func⟩ ::= [f-h] { [A-Za-z0-9_] }
⟨pred⟩ ::= [A-Z] { [A-Za-z0-9_] }
⟨numeral⟩ ::= [1-9] { [0-9] }
```

For example, for the time being, we cannot use a ternary function symbol like `substitute(source, target, position)` or a binary predicate symbol like `subformula(x,y)`. Moreover all user-defined functions and predicates are prefix (for the time being again).

Numerals exhibit a slight deviation from other user-defined symbols in that their interpretations are predefined.[3] To accommodate numerals, we modify the definition of terms (given in page 27) slightly. In this updated definition, primitive terms include variables, constants, and numerals.

Arities for functions and predicates are indicated by appending the number of arguments as a decimal digit to the end of the symbol name. The character right before the digit cannot be an underscore. For instance, `f2` represents a binary function symbol, while `P3`

[3] We are talking about semantics here. Syntactically, you can just ignore this comment.

denotes a ternary predicate symbol. Arities bigger than 9 cannot be indicated in this manner.

If the last character of the function or predicate symbol is not a digit or if the second last character is an underscore, then the arity of the symbol is assume to be 1 for functions and 0 for predicates. For example, `f` and `f_2` are unary function symbols and `P` and `P_3` are nullary predicate symbols, which are equivalent to propositional variables or *propositional letters*.

Arities of function/predicate symbols are not rendered in outputs because they can be inferred from the number of arguments. For example, `f2` is binary and rendered as f, `g12` is binary and rendered as $g1$, `g_12` is binary and rendered as g_1, and `g_2` is unary and rendered as g_2.

In the future, as we expand the language of logic to encompass the expression of proofs, we will provide the user with the ability to define the arities of function and predicate symbols in the *signature* section of the input text.

Grammar and Parser

The ⟨term⟩ structure bears a strong resemblance to ⟨expr⟩ in the language of arithmetic, allowing us to employ a similar parsing technique with minimal difficulty.

In the case of ⟨formula⟩, it shares similarities with ⟨expr⟩ in the language of propositional logic. However, ⟨atom⟩ in this context is more intricate, as it is constructed by combining terms with predicates in either prefix or infix form. Additionally, there exists a new set of tokens called quantifiers, distinct from connectives, used for combining atomic formulas.

```
⟨formula⟩ ::= { ⟨comp_fmla1⟩ "imp" } ⟨comp_fmla1⟩ ¦
                ⟨comp_fmla1⟩ { ( "iff" ¦ "xor") ⟨comp_fmla1⟩ }
⟨comp_fmla1⟩ ::= ⟨comp_fmla2⟩ { ("and" ¦ "or") ⟨comp_fmla2⟩ }
⟨comp_fmla2⟩ ::= { ("not" ¦ ⟨determiner⟩) }
                 ( '(' ⟨formula⟩ ')' ¦ ⟨atom⟩ ¦ "bot" )
⟨determiner⟩ ::= ⟨quantifier⟩ ⟨var⟩
⟨quantifier⟩ ::= "forall" ¦ "exists"
⟨atom⟩ ::= ⟨prop_letter⟩ ¦ ⟨pred_pre⟩ "(" ⟨term⟩ {',' ⟨term⟩} ")" ¦
           ⟨term⟩ ⟨pred_in⟩ ⟨term⟩
⟨term⟩ ::= (⟨term1⟩ ¦ ⟨nterm1⟩) { ⟨oper_in_1⟩ ⟨term1⟩ }
⟨nterm1⟩ ::= ⟨oper_pre⟩ { ⟨oper_pre⟩ } ⟨term1⟩
⟨term1⟩ ::= ⟨factor⟩ { ⟨oper_in_2⟩ ⟨factor⟩ }
⟨factor⟩ ::= { ⟨factor_exp⟩ ⟨oper_in_3⟩ } ⟨factor_exp⟩
⟨factor_exp⟩ ::= ⟨factor_postfix⟩ { ⟨oper_postfix⟩ }
⟨factor_postfix⟩ ::= "(" ⟨term⟩ ")"  ¦ ⟨func_call⟩ ¦ ⟨identifier⟩
⟨func_call⟩ ::= ⟨func_pre⟩ '(' ⟨term⟩ {',' ⟨term⟩} ')'
⟨identifier⟩ ::= ⟨const⟩ ¦ ⟨numeral⟩ ¦ ⟨var⟩
# oper_in_1, oper_pre, oper_in_2, oper_in_3, oper_post,
#   func_pre, const, numeral, var are defined in the Token class.
```

If the list of tokens fed to the parser has at least one predicate, then parse it as a formula. Otherwise parse it as a term. Also we need to give a new attribute `type ::= 'formula' ¦ 'term'` to the `Node` class. Please see the following code snippet for details.

```
class Token:
  ..
  FMLA_TOKENS = ("pred_pre", "pred_in", "equality", "prop_letter",
```

```
    'conn_0ary') # an expression is a formula iff it has a token in FMLA_TOKENS
  FMLA_ROOTS = FMLA_TOKENS + ("conn_1ary", "conn_2ary", "conn_arrow",
    "quantifier", "var_determiner")
    # a parsed node is a formula iff it has a token in FMLA_ROOTS
  ..

class Node:
  ..
  def __init__(self, token, children=None):
    self.token = token # the node is labeled with a Token object
    self.children = children if children else [] # list of Node objects
    self.type = 'formula' if self.token.token_type in Token.FMLA_ROOTS else 'term'
  ..
  def build_infix_latex(self):
    if(self.type == 'term'):
      return self.build_infix_latex_term()
    else: # self.type == 'formula'
      return self.build_infix_latex_formula()
  ..
class Parser:
  ..
  def parse(self) -> Node:
    # determine the type of self.tokens, whether it is a formula or a term
    is_formula = any([token.token_type in Token.FMLA_TOKENS
                      for token in self.tokens])
    if is_formula:
      return self.formula()
    else:
      return self.term()
  ..
```

The `Token.token_type` is a string that serves as an indicator for the type of the token. These token types play a significant role in the parsing of formulas and terms, as well as the rendering of formulas and terms in LaTeX.

Following is the list of token types. Please see the code of the `Token` class for details.

```
equality
conn_arrow, conn_2ary, conn_1ary, conn_0ary
quantifier, var_determiner
lparen, rparen, comma
oper_in_1, oper_in_2, oper_in_3
oper_pre, oper_post
pred_in
numeral
var
const # emptyset, infty, [a-e] { .. }
func_pre
pred_pre
prop_letter
```

When we render the AST, we do not want to display all the nodes for otherwise the tree could be unnecessarily big. All terms are rendered as a single node. Similarly, all quantifier determiner variable pairs are rendered as a single node.

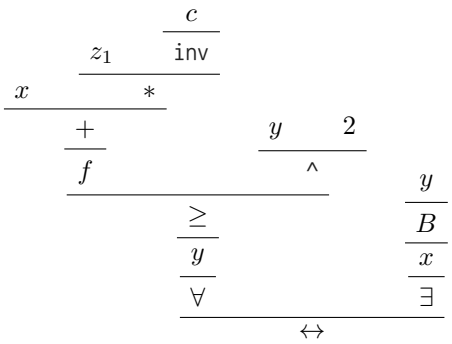
For example, the following formula

```
forall y f(x + z_1 * c^inv) >= y^2 iff exists x B1(y),
```

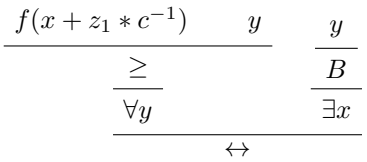
which is infix-LaTeX-rendered as

$$\forall y\,(f(x + z_1 * c^{-1}) \geq y^2) \;\leftrightarrow\; \exists x\, B(y),$$

is parsed as the following AST.



But the following simplified version is actually rendered to save space.



Caution. In Google Colab, certain frequently employed LaTeX symbols like `\le`, `\ge`, and others cannot be directly utilized. To facilitate the rendering of these symbols, the following command is typically employed:

```
plt.rcParams["text.usetex"] = True
```

However, it's worth noting that this command might not function as expected within the Google Colab environment. While there are potential workarounds available to address this issue, for the purpose of our current task, we will not delve into those solutions.

You have the option to continue working within the VS Code environment and/or utilize the `showOption='bussproof'` to access the LaTeX source. Subsequently, you can copy and paste this source into a LaTeX document, and then proceed to compile it in order to generate the intended output.

3
Formal Proof Systems

This chapter is intended for readers who are not acquainted with formal proof systems. If you are already familiar with such systems, feel free to skip this chapter.

Out of the various logics available, our focus will solely be on classical first-order logic. We will not delve into the discussion of other significant logics, including intuitionistic logic, modal logic, and linear logic.

Throughout, we denote the set of all first-order formulas, which corresponds to the variable ⟨formula⟩ defined in p30, by `Formulas`. A member of `Formulas` which is built from propositional letters(i.e., nullary predicate symbols) and connectives is called a propositional formula and the set of all such formulas is denoted by `Formulas0`.

Proof systems can be established independently of the semantics of first-order logic or propositional logic, focusing solely on symbol manipulation as the basis for proofs.

But in order to fully understand proofs, some semantic considerations should eventually come into play. While proof systems solely built on symbol manipulation can be effective for establishing formal validity, grasping the deeper meaning and implications of the proofs requires connecting them to the underlying semantics.

Semantics provide the interpretation and meaning of the logical symbols used in the proof. They help us understand why certain symbol manipulations are valid and what these manipulations represent in terms of logical relationships. Without delving into semantics, proofs might remain abstract symbol games lacking real-world significance.

3.1 Semantics for propositional logic

For a sequence $\vec{A} := \langle A_1, \ldots, A_n \rangle$ of propositional variables, any function

$$\bar{x} : \{A_1, \ldots, A_n\} \to \{0, 1\}$$

is called a *truth-value assignment*. Normally 1 signifies truth and 0 signifies falsity.

If α is a propositional formula having only the propositional variables among $A_1, \ldots, A_n$, then we can give the truth-value of α under $\bar{x}$ using the truth table given in [Figure 3.1] given below. Obtaining the truth table for arbitrary formula α can be done by induction on the complexity of α.

Figure 3.1: Truth table for propositional connectives

α	β	$\neg\alpha$	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\alpha \to \beta$	$\alpha \leftrightarrow \beta$	$\alpha \nleftrightarrow \beta$	$\bot$
1	1	0	1	1	1	1	0	0
1	0	0	0	1	0	0	1	0
0	1	1	0	1	1	0	1	0
0	0	1	0	0	1	1	0	0

So, $\langle \alpha, \vec{A} \rangle$ defines a map $\tilde{\alpha} : \{0,1\}^n \to \{0,1\}$ in a natural way. We call $\tilde{\alpha}$ the *truth function* of α (with respect to $\vec{A}$). We can also regard $\bar{x}$ as a member of $\{0,1\}^n$: i.e., a function $\{1,\ldots,n\}^n \to \{0,1\}$. So we write $\tilde{\alpha}(\bar{x})$ instead of $\tilde{\alpha}(\bar{x}(A_1),\ldots,\bar{x}(A_n))$.

When we deal with multiple formulas $\alpha_1,\ldots,\alpha_m \in$ `Formulas0`, normally we assume an underlying sequence $\vec{A}$ that contains all the propositional variables appearing in $\alpha_1,\ldots,\alpha_m$. The sequence $\vec{A}$ enumerates the propositional variables in some fixed order. In this case, we can regard $\tilde{\alpha}_i$ as a function $\{0,1\}^n \to \{0,1\}$ for each i, where n is the length of $\vec{A}$.

Example 3.1 Following are 3 truth tables combined into one for the propositional formulas $A \to B$, $B \vee C$ and $(C \to \neg B) \to C$. This table was generated using the `TruthTable.show_truth_table()` method in `truth_table.py`. See [Section 4.2] for more details.

A	B	C	A	$\to$	B	B	$\vee$	C	(	C	$\to$	$\neg$	B	)	$\to$	C
1	1	1	1	1	1	1	1	1		1	0	0	1		1	1
1	1	0	1	1	1	1	1	0		0	1	0	1		0	0
1	0	1	1	0	0	0	1	1		1	1	1	0		1	1
1	0	0	1	0	0	0	0	0		0	1	1	0		0	0
0	1	1	0	1	1	1	1	1		1	0	0	1		1	1
0	1	0	0	1	1	1	1	0		0	1	0	1		0	0
0	0	1	0	1	0	0	1	1		1	1	1	0		1	1
0	0	0	0	1	0	0	0	0		0	1	1	0		0	0
Level			1	2	1	1	2	1		1	3	2	1		4	1

For the 3 formulas, we took $\vec{A} := \langle A, B, C \rangle$ as the sequence of propositional variables. The first row corresponds to the truth-value assignment $\bar{x} = \langle 1,1,1 \rangle$, the second row to $\bar{x} = \langle 1,1,0 \rangle$, and so on. The last row corresponds to $\bar{x} = \langle 0,0,0 \rangle$. ⊣

Definition 3.2 A formula $\alpha \in$ `Formulas0` is called a *tautology* if $\tilde{\alpha}$ is the constant function 1. α is called a *tautological contradiction* if $\tilde{\alpha}$ is the constant function 0.

When $\tilde{\alpha}(\bar{x}) = 1$, we say that α is *tautologically satisfied* at $\bar{x}$. A set $\{\alpha_i \mid i \in I\}$ of formulas is said to be tautologically satisfied at $\bar{x}$ if $\tilde{\alpha}_i(\bar{x}) = 1$ for all $i \in I$.

A formula or a set of formulas is said to be *satisfiable* if it is satisfied at some truth-value assignment. Otherwise it is said to be *tautologically unsatisfiable*.

For $\Gamma \subseteq$ `Formulas` and $\varphi \in$ `Formulas`, we say that φ *tautologically follows* from Γ, or equivalently φ is a *tautological consequence* of Γ if $\tilde{\varphi}(\bar{x}) = 1$ for all truth-value assignment $\bar{x}$ such that $\tilde{\alpha}(\bar{x}) = 1$ for all $\alpha \in \Gamma$. We write $\Gamma \vDash_0 \varphi$ in this case.

In case $\Gamma = \varnothing$, we write $\vDash_0 \varphi$ instead of $\varnothing \vDash_0 \varphi$. So, φ is a tautology iff $\vDash_0 \varphi$, and φ is a contradiction iff $\vDash_0 \neg\varphi$. ⊣

Example 3.3 $A \wedge B$ is satisfiable because it is satisfied at $\bar{x} = \langle 1, 1\rangle$. But $A \wedge B$ is not a tautology because it is not satisfied at any $\bar{x} \neq \langle 0, 0\rangle$.

$\{A \wedge B, \neg A\}$ is not satisfiable because it is not satisfied at any $\bar{x}$.

$A \wedge \neg A$ is a contradiction, and $A \vee \neg A$ is a tautology.

In [Example 3.1], we see that $\{A, A \to B\} \vDash_0 B$ because the only truth-value assignments that make both A and $A \to B$ true are $\bar{x} := \langle 1, 1, 1\rangle$ and $\bar{x} := \langle 1, 1, 0\rangle$, and in both cases B is true.

Similarly, it is easily seen that $\{B,\ C,\ C \to \neg B\}$ is unsatisfiable.

In this [Example 3.3], we should have put 'tautologically' in many places but omitted it for brevity. ⊣

The symbol $\vDash$ serves a dual purpose. Firstly, it signifies the logical consequence relation between sets of formulas and formulas. An example of this type of usage is $\{A, A \to B\} \vDash B$. Secondly, it indicates the satisfaction relation between *interpretations* (i.e., truth-value assignment in the context of propositional logic) and sets of formulas. An example of this type of usage is $101 \vDash \{A, B \vee C, A \to C\}$.

The relation between the two meanings is established by the following fact.

Fact 3.4 Define $\mathrm{Mod}(\Gamma) \stackrel{\text{def}}{=} \{\bar{x} \mid \bar{x} \vDash \Gamma\}$. Then $\Gamma \vDash \varphi$ iff $\mathrm{Mod}(\Gamma) \subseteq \mathrm{Mod}(\varphi)$. ⊣

Consider the following questions.

(1) Is the following formula a tautology?

$$x \leq y \to x \leq y$$

(2) Is the following satisfiable?

$$\{\forall x\, P(x), \neg \forall x\, P(x)\}$$

According to our definition, the answer of the two questions above are both negative because the formulas appearing in (1) and (2) are not members of `Formulas0`. But we may want to answer the first question positively, and the second question negatively. In order to do so, we need the following definition.

Definition 3.5 A formula $\alpha \in$ `Formulas` is called *prime* iff the root of its AST is not a logical connective of positive arity. ⊣

Essentially, any truth-value can be assigned to a prime formula. Thus, we have the flexibility to treat a prime formula as a propositional variable, ensuring that distinct formulas are assigned distinct propositional variables. $\bot$ is an exception. It is a prime formula according to our definition, but it is not a propositional variable. It is something like a propositional constant whose truth-value is always 0.

As a result, we can redefine fundamental concepts like tautology, satisfiability, tautological consequence, and related notions, employing prime formulas instead of propositional variables. Going forward, we will adopt this revised definition for propositional semantics, unless specified otherwise.

Note the following.

- $\bot$ is a prime formula but not a propositional variable.

- $(x < y) \wedge (x > y)$ is not a tautological contradiction although it is not satisfied in most *interpretations*[1].
- $\forall x \neg P(x) \to \neg \exists x\, P(x)$ is not a tautology although it is *logically valid*.

We will soon define first-order logical consequence, $\models_1$, first-order satisfiability, etc. We will omit 'first-order' in most cases. So we will say 'logically follows' instead of 'first-order logically follows' and write $\models$ instead of $\models_1$, 'satisfiable' instead of 'first-order satisfiable' etc.

Note that $\alpha \models_0 \beta$ implies $\alpha \models \beta$ but not vice versa.

3.2 Axioms and Rules of Inference

There is no universally accepted definition of formal proof systems. But I believe most people will agree on the following.

We should have a well-defined set of logical formulas, for instance `Formulas`, for which we gave a definition on p33. Then we need a set of *axioms* and a set of *rules of inference*. The axioms are formulas that are assumed to be true. The rules of inference are rules that allow us to infer a formula from other formulas.[2] A proof is a sequence of formulas such that each formula is either an axiom or follows from previous formulas by a rule of inference, or a tree labeled with formulas such that a more sophisticated application of rules of inference is allowed.

We will use the following notation to facilitate the discussion of proof systems.

Notation 3.6 For a set X, we use $\mathcal{P}(X)$ to denote the set of all subsets of X. In other words, $\mathcal{P}(X)$ is the power set of X. ⊣

In a simplest setting, a proof system may be defined as a function

$$f : \mathcal{P}(\texttt{Formulas}) \to \mathcal{P}(\texttt{Formulas})$$

such that for all $\Gamma \in \mathcal{P}(\texttt{Formulas})$ and $\varphi \in \texttt{Formulas}$, we have

$$\Gamma \models \varphi \;\Leftrightarrow\; \varphi \in f(\Gamma).$$

So $f(\Gamma)$ is the set of all logical consequences of Γ. This definition does not look very useful, but it is a good starting point.

In order to implement the concept of proof in computers, it is necessary to establish a syntactic definition for the proof function f. By defining the function f in terms of the syntax of formulas, we will be able to effectively accomplish this objective.

Definition 3.7 Assuming that the syntactic definition of f such that $\Gamma \models \varphi \;\Leftrightarrow\; \varphi \in f(\Gamma)$ has been obtained, we will write $\Gamma \vdash \varphi$ instead of $\varphi \in f(\Gamma)$, or equivalently, instead of $\Gamma \models \varphi$. For propositional logic, from a function $f_0 : \mathcal{P}(\texttt{Formulas}) \to \mathcal{P}(\texttt{Formulas})$ such that $\Gamma \models_0 \varphi \Leftrightarrow \varphi \in f_0(\Gamma)$, we can define $\vdash_0$.

[1] 'Interpretation' is a key concept in first-order semantics. We will discuss this later in this book.

[2] This is not entirely true for some proof systems like Fitch calculus, which will be used throughout this book.

Both $\models$ and $\vdash$ can be thought of as a relation from $\mathcal{P}(\texttt{Formulas})$ into $\texttt{Formulas}$. In this respect, $\models$ is called the *logical consequence relation* and $\vdash$ is called the *proof relation*.

Similarly we have the propositional versions of logical consequence relation and proof relation. From this point forward, when discussing proofs, semantics, and related concepts in the context of the first-order version, we can apply similar approaches in the context of propositional logic, while excluding corresponding details for the sake of brevity.

When $\Gamma = \varnothing$ we may just omit it. So $\varnothing \vdash \varphi$ can be written as $\vdash \varphi$.

We want the notion of *proof* or *derivation*, which *witnesses* the proof relation $\Gamma \vdash \varphi$. In general there can be many derivations of φ from Γ. So we want to have something like

$$\exists p \in \texttt{Proofs}(\Gamma, \varphi) \;\Leftrightarrow\; \Gamma \vdash \varphi$$

where $\texttt{Proofs}(\Gamma, \varphi)$ is the set of all proofs of φ from Γ. The set $\texttt{Proofs}(\Gamma, \varphi)$ for a specific Γ and φ, as well as the union of all such sets where Γ and φ range over all possible values, is a formal language too.

When $p \in \texttt{Proofs}(\Gamma, \varphi)$ holds, we call φ the *conclusion* and Γ the *hypotheses* of the proof p. ⊣

The ternary relation $p \in \texttt{Proofs}(\Gamma, \varphi)$ should be recursive so that it can be implemented in computers. On the other hand, the relation $\Gamma \vdash \varphi$ is not recursive in general, although it is recursively enumerable.

The proof system we will be using is called *Fitch Calculus*[3]. It is a formal proof system for formal logics. We will focus on the Fitch system on classical first-order logic only. It is named after Frederic Fitch, who introduced it in 1952.

The Fitch calculus is closely connected to the widely recognized proof system known as *Natural Deduction*.[4]

To begin, we will introduce the Hilbert System, the oldest formal proof system in the modern sense. This introduction will serve as a foundation for comprehending the nuanced distinctions between the Fitch Calculus and the Natural Deduction systems.

3.3 Hilbert System

We will focus on explaining the Hilbert system for propositional calculus exclusively. Note that the first-order version can be deduced straightforwardly from the propositional version and the first-order version of Fitch calculus.

We assume that the only connectives are $\to$ and $\neg$. Other connectives can be considered as *abbreviations* (or *syntactic assignments*) of these two.

Notation 3.8 We will use $\equiv$ to denote *syntactic equality*. For example, it is not true in general that $\alpha \vee \beta \equiv \beta \vee \alpha$ although $\alpha \vee \beta$ is (generally interpreted as) semantically equivalent to $\beta \vee \alpha$. These two formulas, $\alpha \vee \beta$ and $\beta \vee \alpha$, are syntactically equal if and only if α and β are syntactically equal.

We use $:\equiv$ to denote *syntactic assignments*. In an expression $\alpha :\equiv \beta$, α and β are meta-symbols whose values are in $\texttt{Formulas}$. In a syntactic assignment statement, the l.h.s. must

[3] https://en.wikipedia.org/wiki/Fitch_notation

[4] The Fitch calculus is often regarded as a variation of Natural Deduction, with Natural Deduction itself being perceived as a category encompassing multiple proof systems, rather than a singular proof system.

be a single meta-symbol whereas the r.h.s. can be any (well-formed) expression consisting of several meta-symbols and/or object symbols such as $\to$ and $\neg$. ⊣

Definition 3.9 (Additional connectives) We define the following 5 connectives. The last one is 0-ary and the others are binary.

- $\alpha \wedge \beta :\equiv \neg(\alpha \to \neg\beta)$
- $\alpha \vee \beta :\equiv \neg\alpha \to \beta$
- $\alpha \leftrightarrow \beta :\equiv (\alpha \to \beta) \wedge (\beta \to \alpha)$
- $\alpha \nleftrightarrow \beta :\equiv (\alpha \wedge \neg\beta) \vee (\neg\alpha \wedge \beta)$
- $\bot :\equiv A \wedge \neg A$

In the expressions above, α, β etc. denote arbitrary members of `Formulas` and A denote an arbitrary but fixed propositional variable. ⊣

Recall that a proof system roughly consists of axioms and rules of inference. Our propositional Hilbert system has 3 axioms and 1 rule of inference.

Definition 3.10 Hilbert axioms for propositional logic are as follows.[5]

A1 : $\alpha \to (\beta \to \alpha)$

A2 : $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$

A3 : $(\neg\beta \to \neg\alpha) \to (\alpha \to \beta)$

Note that all axioms are tautologies.

To be precise, we have three *axiom schemes* A1, A2 and A3. For example, $B \wedge A \to (C \to B \wedge A)$ and $A \to ((A \vee \neg B) \to A)$ are instances of A1, and $(\neg A \to \neg\neg B) \to (\neg B \to A)$ is an instance of A3. Note that $(A \to B) \to (\neg B \to \neg A)$ is not an instance of A3. ⊣

Note that axioms alone cannot prove anything other than axioms themselves. For instance how would you $A \vdash B \to A$ (or $\{A\} \vdash B \to A$ to be rigorous)? It is not possible because axioms are just formulas and nothing more.

In order to prove any formula, you need some *rules of inference*. Hilbert system for propositional calculus has only one rule of inference, namely *modus ponens* which means the following rule:

Definition 3.11 (Modus Ponens) From the two formulas $\alpha \to \beta$ and α, infer β. This rule of inference is called *modus ponens* and is denoted by *mp*. We can represent this rule by the following tree diagram.

$$\frac{\alpha \to \beta \qquad \alpha}{\beta}\ (\text{mp})$$

In this tree, siblings are not ordered. So we can also write the same tree as follows.

[5] There are other formulations of the axioms for Hilbert system. This one is due to the Polish logician Jan Łukasiewicz, renowned for his contribution to Polish notation.

$$\frac{\alpha \qquad \alpha \to \beta}{\beta}\ (\text{mp})$$

Modus ponens can be represented more succinctly by the following proof relation:

$$\{\alpha \to \beta,\ \alpha\} \vdash \beta \qquad \dashv$$

Personally, I believe that the first step of understanding the concept of formal proofs is the distinction between axioms and rules of inference. I remember the day when I was wondering whether the modus ponens can be replaced by the axiom $\alpha \to ((\alpha \to \beta) \to \beta)$ or $\alpha \wedge (\alpha \to \beta) \to \beta$.

Definition 3.12 Recall that the Hilbert proofs for propositional logic form a formal language. We define this language mathematically here.

If $\Gamma \in \mathcal{P}(\texttt{Formulas})$ and $\varphi \in \texttt{Formulas}$, then a *Hilbert proof* (or *Hilbert derivation*) of φ from Γ is a finite sequence of formulas $\varphi_1, \varphi_2, \ldots, \varphi_n \equiv \varphi$ such that for each i, one of the following holds:

- $\varphi_i \in \Gamma$,
- φ_i is an instance of A1, A2 or A3,
- φ_i follows from previous formulas by modus ponens: i.e., there are $j, k < i$ such that $\varphi_k \equiv \varphi_j \to \varphi_i$. Neither $j < k$ nor $k < j$ is required.

We may call the members of Γ the *extralogical axioms* or hypotheses, while the members of A1, A2, A3 are called the *logical axioms*. $\dashv$

Example 3.13 We will prove $\alpha \to \alpha$ from empty hypotheses in Hilbert system, or equivalently construct a Hilbert proof for $\vdash \alpha \to \alpha$. (We should really use $\vdash_0 \alpha \to \alpha$ here to be precise, but let's be a little sloppy for convenience.)

1.	$\alpha \to ((\alpha \to \alpha) \to \alpha)$	A1
2.	$(\alpha \to ((\alpha \to \alpha) \to \alpha)) \to ((\alpha \to (\alpha \to \alpha)) \to (\alpha \to \alpha))$	A2
3.	$(\alpha \to (\alpha \to \alpha)) \to (\alpha \to \alpha)$	mp(1,2)
4.	$\alpha \to (\alpha \to \alpha)$	A1
5.	$\alpha \to \alpha$	mp(4,3)

According to [Definition 3.12], the Hilbert proof consists of the sequence of 5 formulas provided in the central column of the table above. But we usually consider the whole table as the Hilbert proof. We may call such a table an *annotated* Hilbert proof.

The first column is the line number, the second column is the formula, and the third column is the justification, or *annotation* of the formula. The justification is either a logical axiom, hypothesis(i.e., extralogical axiom) or modus ponens applied to two previous lines. The conclusion is the formula in the last line of the table. $\dashv$

Example 3.14 A Hilbert proof for $\vdash \neg\alpha \to (\alpha \to \beta)$ is as follows.

1.	$\neg\alpha \to (\neg\beta \to \neg\alpha)$	A1
2.	$(\neg\beta \to \neg\alpha) \to (\alpha \to \beta)$	A3
3.	$((\neg\beta \to \neg\alpha) \to (\alpha \to \beta)) \to (\neg\alpha \to ((\neg\beta \to \neg\alpha) \to (\alpha \to \beta)))$	A1
4.	$\varphi :\equiv \neg\alpha \to ((\neg\beta \to \neg\alpha) \to (\alpha \to \beta))$	mp(2,3)
5.	$\varphi \to ((\neg\alpha \to (\neg\beta \to \neg\alpha)) \to (\neg\alpha \to (\alpha \to \beta)))$	A2
6.	$(\neg\alpha \to (\neg\beta \to \neg\alpha)) \to (\neg\alpha \to (\alpha \to \beta))$	mp(4,5)
7.	$\neg\alpha \to (\alpha \to \beta)$	mp(1,6)

In line 4 and 5, we used syntactic assignment for readability. ⊣

Based on the two examples of Hilbert proofs presented above, it becomes evident that the Hilbert proof system is highly inefficient. Writing a Hilbert proof for any substantial mathematical statement is practically unattainable.

To enhance the efficiency of a formal proof system like the Hilbert system, several approaches can be employed. These may include the utilization of meta-theorems, lemmas, and extended rules of inference. These techniques serve to streamline the process and make formal proofs more manageable.

Utilizing lemma means that we can put formulas such as $\alpha \to \alpha$ and $\neg\alpha \to (\alpha \to \beta)$, which were proven earlier, on any line in any Hilbert proof.

Some examples of extended rules of inference can be found in [Lemma 3.20].

For meta-theorems, we introduce the *Deduction Theorem* which is perhaps the most widely used one.

Theorem 3.15 (The Deduction Theorem) *If* $\Gamma \in \mathcal{P}(\texttt{Formulas})$ *and* $\alpha, \beta \in \texttt{Formulas}$*, then*

$$\Gamma \cup \{\alpha\} \vdash \beta \;\Rightarrow\; \Gamma \vdash \alpha \to \beta$$

Proof. We first prove the converse of the theorem although it is not required. Suppose $\Gamma \vdash \alpha \to \beta$ and let the following be a Hilbert proof of $\alpha \to \beta$ from Γ.

⋮	⋮	⋮
$n.$	$\alpha \to \beta$	..

Then by adding just two lines to the proof given above, we can obtain a Hilbert proof of β from $\Gamma \cup \{\alpha\}$[6], which is shown below.

⋮	⋮	⋮
$n.$	$\alpha \to \beta$	..
$n+1.$	α	hyp
$n+2.$	β	mp(n, $n+1$)

In line $n+1$, the annotation *hyp* signifies that α is a member of the new hypothesis set $\Gamma \cup \{\alpha\}$. ✓

Now let us prove the theorem. Suppose $\Gamma \cup \{\alpha\} \vdash \beta$ and let $\langle \varphi_1, \ldots, \varphi_n \equiv \beta \rangle$ be a Hilbert proof of β from $\Gamma \cup \{\alpha\}$.

We will show that a Hilbert proof of $\Gamma \vdash \alpha \to \beta$ exists by induction on n.

[6]Here, we are using the fact that if $\Gamma_1 \subseteq \Gamma_2$, then a proof from Γ_1 is also a proof from Γ_2.

If $n = 1$, then we have 2 cases: (1) β is a logical axiom or $\beta \in \Gamma$, and (2) $\beta \equiv \alpha$.

In case (1), the sequence $\langle \beta, \beta \to (\alpha \to \beta), \alpha \to \beta \rangle$ is a Hilbert proof of $\Gamma \vdash \alpha \to \beta$. In case (2), then we know $\Gamma \vdash \alpha \to \alpha$ by the result of [Example 3.13].

If $n > 1$, we have 3 cases: (1) β is a logical axiom or $\beta \in \Gamma$, (2) $\beta \equiv \alpha$, and (3) there exist $i, j < n$ such that $\varphi_i \equiv \varphi_j \to \beta$.

For the cases (1) and (2), the proof is similar to the case $n = 1$. (In these cases, we don't need the Hilbert proof of $\Gamma \cup \{\alpha\} \vdash \beta$ in constructing the Hilbert proof of $\Gamma \vdash \alpha \to \beta$.)

For case (3), we have a Hilbert proof of $\Gamma \cup \{\alpha\} \vdash \varphi_j$ and a Hilbert proof of $\Gamma \cup \{\alpha\} \vdash \varphi_i :\equiv \varphi_j \to \beta$, and the lengths of both proofs are less than n. By the induction hypothesis, we have a Hilbert proof of $\Gamma \vdash \alpha \to \varphi_j$ and a Hilbert proof of $\Gamma \vdash \alpha \to (\varphi_j \to \beta)$. Let n_1 and n_2 be the lengths of these proofs, respectively. Then the following is a Hilbert proof of $\Gamma \vdash \alpha \to \beta$.

$\vdots$	$\vdots$	$\vdots$
$n_1.$	$\alpha \to \varphi_j$	..
$\vdots$	$\vdots$	$\vdots$
$n_3 := n_1 + n_2.$	$\alpha \to (\varphi_j \to \beta)$	..
$n_3 + 1.$	$(\alpha \to (\varphi_j \to \beta)) \to ((\alpha \to \varphi_j) \to (\alpha \to \beta))$	A2
$n_3 + 2.$	$(\alpha \to \varphi_j) \to (\alpha \to \beta)$	mp(n_3,$n_3 + 1$)
$n_3 + 3.$	$\alpha \to \beta$	mp(n_1,$n_3 + 2$)

Note that above proof is constructive—from a Hilbert proof of $\Gamma \cup \{\alpha\} \vdash \beta$, we can actually construct a Hilbert proof of $\Gamma \vdash \alpha \to \beta$ by following the argument used in the proof. □

Notation 3.16 We may write $\Gamma, \alpha \vdash \varphi$ instead of $\Gamma \cup \{\alpha\} \vdash \varphi$. We may write $\alpha \vdash \varphi$ and $\alpha, \beta \vdash \varphi$ instead of $\{\alpha\} \vdash \varphi$ and $\{\alpha, \beta\} \vdash \varphi$ respectively, and so on. ⊣

The following exercise is a good test of your understanding of the Deduction theorem. The solution might be a little bit trickier than you might expect.

Exercise 3.17 Using the argument of the proof of Deduction theorem, construct a Hilbert proof of $P \vdash (P \to Q) \to Q$ from a Hilbert proof of $P, P \to Q \vdash Q$. ⊣

Example 3.18 Using the Deduction theorem, we can prove the transitivity of implication, which means $\vdash (\alpha \to \beta) \to ((\beta \to \gamma) \to (\alpha \to \gamma))$ easily. We only need to show $\alpha \to \beta, \beta \to \gamma, \alpha \vdash \gamma$, which is trivial.

Please be aware that constructing a Hilbert proof of $\vdash (\alpha \to \beta) \to ((\beta \to \gamma) \to (\alpha \to \gamma))$ does not require any ingenuity but will be very tedious. ⊣

We may use the following facts as lemmas in the future.

- $\vdash \alpha \to \alpha$ (Example 3.13)
- $\vdash \alpha \to ((\alpha \to \beta) \to \beta)$ (Example 3.17)
- $\vdash (\alpha \to \beta) \to ((\beta \to \gamma) \to (\alpha \to \gamma))$ (Example 3.18)

Or equivalently, we may use the following extended rules of inference.

$$\frac{}{\alpha \to \alpha}\ \text{(iden)} \qquad \frac{\alpha \to \beta \qquad \beta \to \gamma}{\alpha \to \gamma}\ \text{(trans)}$$

Example 3.19 Using the tools we developed so far, we can prove the *double-negation elimination* $\vdash \neg\neg\alpha \to \alpha$ and *double-negation introduction* $\vdash \alpha \to \neg\neg\alpha$.

A Hilbert proof of $\vdash \neg\neg\alpha \to \alpha$ is as follows.

1.	$\varphi :\equiv \alpha \to \alpha$	iden
2.	$(\neg\neg\varphi \to \neg\neg\alpha) \to (\neg\alpha \to \neg\varphi)$	A3
3.	$(\neg\alpha \to \neg\varphi) \to (\varphi \to \alpha)$	A3
4.	$(\neg\neg\varphi \to \neg\neg\alpha) \to (\varphi \to \alpha)$	trans(2,3)
5.	$\neg\neg\alpha \to (\neg\neg\varphi \to \neg\neg\alpha)$	A1
6.	$\neg\neg\alpha \to (\varphi \to \alpha)$	trans(4,5)
7.	$\varphi \to ((\varphi \to \alpha) \to \alpha)$	Example 3.17
8.	$(\varphi \to \alpha) \to \alpha$	mp(1,7)
9.	$\neg\neg\alpha \to \alpha$	trans(6,8)

□

Imagine how long the proof would be if we did not use the extended rules of inference, lemmas and meta-symbols such as φ.

Now we may use the double-negation elimination, denoted by '$\neg\neg$ elim', as an extended rule of inference. Using the double-negation elimination, we can prove $\vdash \alpha \to \neg\neg\alpha$ easily.

1.	$\neg\neg\neg\alpha \to \neg\alpha$	$\neg\neg$ elim
2.	$(\neg\neg\neg\alpha \to \neg\alpha) \to (\alpha \to \neg\neg\alpha)$	A3
3.	$\alpha \to \neg\neg\alpha$	mp(1,2)

Lemma 3.20 *We may use the following extended rules of inference.*

$$\frac{\alpha \qquad \neg\alpha}{\beta}\ \textit{(falsum)} \qquad \frac{\neg\alpha \to \beta \qquad \neg\alpha \to \neg\beta}{\alpha}\ (\neg\ \textit{elim})$$

Proof. For the *falsum* rule, we present a Hilbert proof of $\alpha, \neg\alpha \vdash \beta$ below.

1.	$(\neg\beta \to \neg\alpha) \to (\alpha \to \beta)$	A3
2.	$\neg\alpha \to (\neg\beta \to \neg\alpha)$	A1
3.	$\neg\alpha$	hyp
4.	$\neg\beta \to \neg\alpha$	mp(2,3)
5.	$\alpha \to \beta$	mp(1,4)
6.	α	hyp
7.	β	mp(5,6)

For the $\neg$ *elim* rule, let $\Gamma = \{\neg\alpha \to \beta, \neg\alpha \to \neg\beta\}$ and we first prove

$$\Gamma, \neg\alpha \vdash \neg(\alpha \to (\beta \to \alpha))$$

by constructing a Hilbert proof.

1.	$\neg\alpha$	hyp
2.	$\neg\alpha \to \beta$	hyp
3.	β	mp(1,2)
4.	$\neg\alpha \to \neg\beta$	hyp
5.	$\neg\beta$	mp(1,4)
6.	$\neg(\alpha \to (\beta \to \alpha))$	falsum(3,5)

Now we have $\Gamma \vdash \neg\alpha \to \neg(\alpha \to (\beta \to \alpha))$ by the Deduction theorem. We complete the proof of $\neg$ elim rule by the following Hilbert proof in Γ.

1.	$\neg\alpha \to \neg(\alpha \to (\beta \to \alpha))$	Deduction theorem
2.	$(\neg\alpha \to \neg(\alpha \to (\beta \to \alpha))) \to ((\alpha \to (\beta \to \alpha)) \to \alpha)$	A3
3.	$(\alpha \to (\beta \to \alpha)) \to \alpha$	mp(1,2)
4.	$\alpha \to (\beta \to \alpha)$	A1
5.	α	mp(3,4) $\square$

It is easy to see that we can use the $\neg$ intro as an extended rule of inference by constructing a Hilbert proof of $\alpha \to \beta, \alpha \to \neg\beta \vdash \neg\alpha$. (We may need the $\neg\neg$ elim rule and the trans rule in this construction.)

Definition 3.21 For a set Γ of formulas the set $\{\varphi \mid \Gamma \vdash \varphi\}$ is called the *theory* of Γ. The theory of $\varnothing$ is called the *theory of propositional logic*.

A theory is called *consistent* if it does not have a contradiction as a member. Note that an there exists exactly one inconsistent theory, namely the set of all formulas. This is because we have the falsum rule of inference.

A theory is called *complete* if, for each formula φ either φ or $\neg\varphi$ is a member of the theory. $\dashv$

It can be shown that all tautologies can be proved in Hilbert system. This fact is called the *completeness* of Hilbert system. So the completeness of a theory and the completeness of a proof system are two distinct concepts. The latter is usually defined as $\Gamma \vDash \varphi \;\Rightarrow\; \Gamma \vdash \varphi$ for all $\Gamma \subseteq$ `Formulas` and all $\varphi \in$ `Formulas`.

If $\Gamma := \{\alpha_1, \ldots, \alpha_n\}$ were finite, then $\Gamma \vDash \varphi$ and $\Gamma \vdash \varphi$ would be equivalent to $\vDash \alpha_1 \wedge \cdots \wedge \alpha_n \to \varphi$ and $\vdash \alpha_1 \wedge \cdots \wedge \alpha_n \to \varphi$ respectively. Moreover, for an infinite Γ such that $\Gamma \vDash \varphi$ or $\Gamma \vdash \varphi$, we have a finite subset $\Gamma_0 \subseteq \Gamma$ such that $\Gamma_0 \vDash \varphi$ or $\Gamma_0 \vdash \varphi$ respectively. So our initial definition of completeness is equivalent to the second definition.

This equivalence is relatively easy to prove for propositional logic. But for the case of first-order logic, the equivalence is not that easy to prove, and in fact it is one of the greatest theorems in the foundation of mathematical logic.

Note that the theory of propositional logic is not complete. For instance there exists no Hilbert proof for neither $\vdash A$ nor $\vdash \neg A$ when A is a propositional variable.

We will not present the proof of the completeness theorem here. For the case of first-order logic it is far from trivial. For the case of propositional logic, it is doable but rather lengthy. I want to mention that the proof for propositional case is constructive. So, in principle, we can write a single computer program that takes any tautology φ and outputs a Hilbert proof of φ. But this process is extremely inefficient and the resulting Hilbert proof is very long.

However, we believe that an efficient automatic proof generation for tautologies is possible if we use Fitch calculus proof system. For first-order Fitch calculus, this is known to be impossible. Yet we believe that a cleverly written computer program can help human mathematician obtain a Fitch proof of first-order formulas when a proof of that formula actually exists.

The converse of the completeness theorem,

$$\vdash \varphi \text{ implies } \vDash \varphi$$

is called the *soundness* of Hilbert system and can be proved too. This proof can be done using induction on the length of Hilbert proof and relatively easy.

The set of all tautologies is a mathematically definable formal language, which will be denoted by L_{taut}. The alphabet of this language L_{taut} is the set of all propositional variables union the set of connectives $\{\neg, \to\}$.

In [Definition 3.9], we introduced connectives $\vee, \wedge, \leftrightarrow, \nleftrightarrow$ and $\bot$ as abbreviations. So, for instance, the formula $A \vee \neg A$ is an abbreviation of the formula $\neg A \to \neg A$. This means that $A \vee \neg A \notin L_{taut}$ while $\neg A \to \neg A \in L_{taut}$.

We can assume the alternative perspective that $\wedge, \vee$ etc. are not just abbreviations but are primitive connectives. Let L'_{taut} be the set of all tautologies over our new alphabet. An exact definition of L'_{taut} can be done either semantically using truth tables or syntactically using a new version of Hilbert system.

This new Hilbert system may be called the *Hilbert system with additional connectives*. The rules of inference are the same as the original Hilbert system, but for the axioms we need to add some new ones. The new axioms are not uniquely determined. We may use the following ones for instance.

- $\alpha \vee \beta \leftrightarrow (\neg\alpha \to \beta)$
- $\alpha \wedge \beta \leftrightarrow \neg(\alpha \to \neg\beta)$
- $(\alpha \leftrightarrow \beta) \to (\alpha \to \beta)$
- $(\alpha \leftrightarrow \beta) \to (\beta \to \alpha)$
- $(\alpha \to \beta) \to ((\beta \to \alpha) \to (\alpha \leftrightarrow \beta))$
- $(\alpha \nleftrightarrow \beta) \leftrightarrow ((\alpha \wedge \neg\beta) \vee (\neg\alpha \wedge \beta))$
- $\bot \leftrightarrow (\alpha \wedge \neg\alpha)$

If we let L be the set of all formulas in the original alphabet (with only two connectives), then $L \cap L'_{\text{taut}} = L_{\text{taut}}$. This means that the new Hilbert system is a *conservative extension* of the original Hilbert system. We will not prove this fact here.

The ability to define new symbols is an essential feature of efficient formal proof systems. The notion of conservative extension is very closely related to this feature. We will discuss this in more detail in later chapters.

In the new Hilbert system, we can use additional rules of inference for the newly added connectives. For instance, we can add the following rules of inference for $\wedge$.

We could add more inference rules for each of the remaining connectives. But we'll see these rules in Fitch calculus later.

In general, rules of inference is more convenient than axioms. The Natural deduction has no axioms at all and only uses rules of inference. The same is true for Fitch calculus.

Figure 3.2: Inference rules for $\wedge$

$$\frac{\alpha \qquad \beta}{\alpha \wedge \beta} \wedge \text{intro} \qquad \frac{\alpha \wedge \beta}{\alpha} \wedge \text{elim1} \qquad \frac{\alpha \wedge \beta}{\beta} \wedge \text{elim2}$$

3.4 Natural Deduction

We will describe the Natural Deduction for propositional calculus only. The first-order version can be easily deduced from the propositional version and the first-order Fitch calculus.

For simplicity, we will use 3 connectives only: $\bot, \rightarrow$ and $\wedge$, and other connectives are defined as abbreviations. For instance, $\neg\alpha$ is an abbreviation of $\alpha \rightarrow \bot$.

For the connective $\wedge$, we use the rules shown in [Figure 3.2]. For the connective $\rightarrow$, we use the following rules of inference. $\rightarrow$ elim rule looks the same as modus ponens although

Figure 3.3: Inference rules for $\rightarrow$

$$\frac{\begin{array}{c}[\alpha] \\ \vdots \\ \beta\end{array}}{\alpha \rightarrow \beta} \rightarrow \text{intro} \qquad \frac{\alpha \rightarrow \beta \qquad \alpha}{\beta} \rightarrow \text{elim}$$

there is a difference which we will discuss shortly. $\rightarrow$ intro rule is a bit more complicated. It is a rule for introducing an implication. Intuitively, the rule says that if we can derive β from the assumption α, then we can derive $\alpha \rightarrow \beta$. The vertical 3 dots in the diagram mean that there exists a derivation of β from α using suitable rules of inference. The square bracket around α means that α has been *discharged* and no longer a hypothesis. We will discuss this in more detail shortly.

For the connective $\bot$, we use the following rules of inference. The rules for $\bot$ are both,

Figure 3.4: Inference rules for $\bot$

$$\frac{\bot}{\alpha} \text{ falsum} \qquad \frac{\begin{array}{c}[\alpha \rightarrow \bot] \\ \vdots \\ \bot\end{array}}{\alpha} \text{ RAA}$$

in a sense, eliminations, because the connective $\bot$ does not appear in the conclusions. So we gave them different names. The falsum rule says that if we can derive $\bot$, then we can derive anything. This is essentially an ND(Natural Deduction) version of Hilbert system's falsum rule, which is $\beta, \neg\beta :\equiv \beta \rightarrow \bot \vdash \alpha$, because this translates to $\beta, \beta \rightarrow \bot \vdash \bot \vdash \alpha$.

The rule RAA(*reductio ad absurdum*) says that if we can derive $\bot$ from the assumption $\neg\alpha$, which is defined to be the abbreviation of $\alpha \rightarrow \bot$, then we can derive α. The square bracket around $\alpha \rightarrow \bot$ means that $\alpha \rightarrow \bot$ has been discharged and no longer a hypothesis.

The six rules of inference shown in [Figure 3.2], [Figure 3.3] and [Figure 3.4] are sufficient

for the completeness of the Natural Deduction proof system.

Now let us define ND proof rigorously. An ND proof is a tree of formulas—in other words, a tree labeled with formulas. The root of the tree is the conclusion, and the leaves of the tree are the hypotheses. Discharged leaves are removed from the hypotheses.

We first need a definition of a tree. Trees are typically defined as a undirected graph(ordered pair of vertices and edges) having a designated vertex(called the root) with no cycles. But we will use a different definition as follows.

Definition 3.22 (tree) A nonempty set with a reflexive, antisymmetric and transitive relation $\leq$ is called a *partially ordered set* or *poset*. A linearly ordered subset of a poset is called a *chain*. A *tree* is a poset $(T, \leq)$ such that (1) T has a least element called the *root*, and (2) for each $x \in T$ the subset $\{y \in T \mid y \leq x\}$ is a chain.

Members of T are called the *nodes*.

If $x, y \in T$ satisfy $x < y$ and $\neg(\exists z \in T)(x < z < y)$, then x is called the *parent* of y and y is called a *child* of x. Each non-root node has a unique parent. A node may have arbitrary finite number(including zero) of children. Note that x is a root iff it has no parent. A *leaf* is a node with no children. A non-leaf node is called an *internal node*.

If x is a node, then the *subtree* at x is the set of all *descendants* of x including x itself. A descendant of x is any $y \in T$ satisfying $y > x$.

A *branch* is a chain containing a leaf.

For $x \in T$, the nodes of T which share the same parent with x are called the *siblings* of x. Siblings are incomparable, i.e., if x_1 and x_2 are siblings of each other, then neither $x_1 < x_2$ nor $x_2 < x_1$.

An *ordered tree* is obtained from a tree by giving orders to the children of each node. So a parent node in an ordered tree may have the eldest(i.e., the least) child, the second eldest child, and so on. A tree which is not an ordered tree is also called an *unordered tree*. Some people call an ordered tree a *tree* and an unordered tree a *free tree*. We will use the term *tree* to mean an unordered tree.

A *labeled tree* is a function from a tree to a set (of labels). This function is called a *labeling* of the labeled tree. ⊣

A proof object should be a poset labeled with formulas.[7] This is because a formula, which is not a hypothesis[8], in the proof is obtained using a rule of inference applied to some *earlier* formulas. We say that a node x is earlier than a node y if $x > y$ in the tree order.

A Hilbert proof has a linearly ordered set as the underlying poset, while an ND proof has an unordered tree as the underlying poset. Since a linearly ordered set is a special case of a tree, one might think that a Hilbert proof is a special case of an ND proof. But this is not true even when we replace all axioms of Hilbert system with corresponding rules of inference.[9]

First, in Hilbert system, we cannot discharge a hypothesis. We need to keep all hypotheses until the end of the proof. When we want to discharge a hypothesis, we need to use the deduction theorem.

[7] This is not true in Fitch calculus, where a proof is an ordered tree labeled with formulas and subproofs.

[8] Note that the ND proof system do not have any axioms.

[9] A very simple way of converting an axiom to a rule of inference is the following: let the axiom be the conclusion of the rule of inference and let the number of the premises of the rule be zero.

Second, in ND, when applying a rule of inference, the premises of the rule must be the children of the conclusion unless the premise is being discharged. But in Hilbert system this is not necessary. For example if line 3 is $\alpha \to \beta$ and line 5 is α, we may obtain β in line 12. The distance between the hypotheses and the conclusion in an application of a rule of inference can be arbitrary, which is not the case in ND.

Perhaps this is a good place to distinguish 'hypothesis' from 'premise' used in formal proofs. See the definition below.

Definition 3.23 (Terminology) The application of a rule of inference has exactly one *conclusion* and 0 or more *premises*. This application is called the *verification* of the conclusion. The term *hypothesis* is only used for the members of Γ where our formal proof is $\Gamma \vdash \varphi$. In this proof φ is also called the conclusion. ⊣

Definition 3.24 (ND proof) An *ND proof* is a labeled tree $(T, \leq, L)$ such that

(1) T is a tree.

(2) L is a labeling of T with formulas.

(3) For all $x \in T$, if x is a leaf, then $L(x)$ is a hypothesis and do not need verification. For all internal node $x \in T$, $L(x)$ need be verified.

(4) A verification of an internal node x is done by applying a rule of inference, one among those given in [Figure 3.2, 3.3, 3.4], where $L(x)$ is the conclusion of the rule of inference.

(5) The label of a discharged node is no longer a member of the set of hypotheses.

(6) (This part is tricky.) In applying the two rules of inference, '$\to$ intro' and 'RAA', the formula in the bracket may or may not be discharged. If it is not discharged, then the formula remains in the set of hypotheses. In RAA, the premise must be present regardless of whether the formula in the bracket is discharged or not. But in $\to$ intro, the premise do not need be present.

(7) A discharge can be done for multiple leaves at once as long as the labels of the leaves are all the same. It is a good practice to superscript the nodes with the same number to indicate they were treated together.

    Indeed, to avoid any potential confusion about which conclusion corresponds to which premise, it is recommended to superscript all discharged nodes during each verification of the two types mentioned earlier in (6). This notation provides clarity by indicating the discharged premise for the specific application of the rule of inference. ⊣

The definition of ND proof can be fully understood only through examples.

Example 3.25

(1) The tree with a single node labeled with any formula φ is a valid ND proof. Its only node is a leaf and at the same time a root. So its label φ is a hypothesis and does not need verification. Also φ is a conclusion. So the tree is a valid ND proof of the proof relation $\{\varphi\} \vdash \varphi$.

(2) Let's construct an ND proof of $\alpha \vdash \beta \to \alpha$. The conclusion is an implication formula. So we should use the $\to$ intro rule for the verification.

$$\dfrac{\alpha}{\beta \to \alpha} \to \text{intro}$$

No discharge is necessary in this case. In fact no discharge is possible because that would make α an internal node which should then be verified, which is not possible.

(3) The ND proof of $\vdash \alpha \to (\beta \to \alpha)$ needs two verifications with $\to$ intro rule. Only one of them needs discharge.

$$\dfrac{\dfrac{[\alpha]^1}{\beta \to \alpha} \to \text{intro}}{\alpha \to (\beta \to \alpha)} \to \text{intro}^1$$

(4) The ND proof of $\vdash \alpha \to (\neg\alpha \to \beta)$ is as follows.

$$\dfrac{\dfrac{\dfrac{[\alpha]^1 \qquad [\neg\alpha]^2}{\bot} \to \text{elim}}{\beta} \text{falsum}}{\dfrac{\neg\alpha \to \beta}{\alpha \to (\neg\alpha \to \beta)} \to \text{intro}^1} \to \text{intro}^2$$

(5) The ND proof of $\vdash (\neg\beta \to \neg\alpha) \to (\alpha \to \beta)$ is as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{[\neg\beta]^1 \qquad [\neg\beta \to \neg\alpha]^2}{\neg\alpha} \to \text{elim} \qquad [\alpha]^3}{\bot} \to \text{elim}}{\beta} \text{RAA}^1}{\dfrac{\alpha \to \beta}{(\neg\beta \to \neg\alpha) \to (\alpha \to \beta)} \to \text{intro}^3} \to \text{intro}^2$$

(6) The ND proof of $\vdash (\alpha \wedge \beta) \to (\beta \wedge \alpha)$ is as follows.

$$\dfrac{\dfrac{\dfrac{[\alpha \wedge \beta]^1}{\beta} \wedge \text{elim2} \qquad \dfrac{[\alpha \wedge \beta]^1}{\alpha} \wedge \text{elim1}}{\beta \wedge \alpha} \wedge \text{intro}}{(\alpha \wedge \beta) \to (\beta \wedge \alpha)} \to \text{intro}^1$$

It is evident that Natural Deduction (ND) is a more efficient proof system in comparison to the Hilbert system. However, constructing an ND proof for a relatively simple proof relation such as $\vdash \neg(\alpha \leftrightarrow \neg\alpha)$ is still not straightforward, and the resulting proof tree can occupy a significant amount of space.

In the next chapter, we will introduce a more efficient proof system known as the Fitch calculus. Subsequently, we will develop a computer implementation of this system, encompassing both propositional and first-order logic. This implementation will provide a practical approach to working with proofs in these logics.

3.5 Other Systems

We studied two proof systems: Hilbert system and Natural Deduction. As I said before we will study Fitch calculus soon.

Although there are numerous other proof systems, we will not delve into them in this book. However, we will provide a brief mention of some prominent ones: logic tableaux, resolution, sequent calculus, and Coq[10].

(Will add some stuff here. Skip for now.)

[10] https://coq.inria.fr

4 Fitch Proof, propositional logic

4.1 Introduction

Fitch proofs are syntactic objects constructed in the Fitch calculus proof system. We describe the language of Fitch proofs with the following grammar. In this grammar, ⟨formula⟩ means the first-order formula defined in p30.

```
⟨proof⟩ ::= ⟨hypothesis⟩ "entails" ⟨conclusion⟩
⟨hypothesis⟩ ::= ⟨line⟩ { ⟨line⟩ }
⟨line⟩ ::= ⟨node_identifier⟩ (⟨formula⟩ "\n" ¦ ⟨comment⟩ ¦ ⟨blank⟩)
⟨blank⟩ ::= "\n"
⟨comment⟩ ::= { ⟨white_space⟩ } "#" ⟨utf8string⟩ "\n"
⟨utf8string⟩ ::= { [.] } # no "\n" allowed
⟨conclusion⟩ ::= (⟨line_annotated⟩ ¦ ⟨subproof⟩)  (⟨line_annotated⟩ ¦ ⟨subproof⟩)
⟨subproof⟩ ::= ⟨indent⟩ ⟨hypothesis0⟩ "entails" ⟨conclusion⟩
⟨indent⟩ ::= "\t" # tab
⟨hypothesis0⟩ ::= ⟨line⟩
⟨line_annotated⟩ ::= ⟨node_ident⟩ (⟨formula⟩ ⟨ann⟩ "\n" ¦ ⟨comment⟩ ¦ ⟨blank⟩)
⟨ann⟩ ::= "[ann]" ⟨rule_of_inference⟩ ⟨premise⟩
⟨premise⟩ ::= { ⟨node_ident⟩ "," }
⟨node_ident⟩ ::= ⟨numeral⟩ ¦ ⟨numeral⟩ "-" ⟨numeral⟩ # line number
⟨rule_of_inference⟩ ::= ("intro" ¦ "elim") (⟨conn⟩ ¦ ⟨quant⟩) ¦ "repeat" ¦ "LEM"
⟨conn⟩ ::= "bot" ¦ "not" ¦ "and" ¦ "or" ¦ "imp" ¦ "iff"
⟨quant⟩ ::= "forall" ¦ "exists"
```

In this chapter, we describe the propositional fragment of the Fitch calculus. This proof system has no axioms and 12 rules of inference. Each of the six connectives has two corresponding rules of inference, an *introduction* and an *elimination*. We do not need rules for the connective `xor`. This connective `xor` is considered as an abbreviation built from other connectives—for instance we may use the following definition: `A xor B` :≡ `not(A iff B)`.

4.2 Truth Table

We aim to extend the construction of truth tables to accommodate two key aspects:

1. We want to handle not only propositional formulas formed from propositional letters and connectives but also first-order formulas.

2. We aim to analyze not only individual formulas but also sets of formulas.

To illustrate, when determining whether $\{\alpha, \beta\} \vdash \varphi$ is justified through tautological consequence, we must construct a truth table for $\{\alpha, \beta, \neg\varphi\}$. Thus, the second aspect is necessary.

Regarding the first aspect, it is vital to identify prime subformulas, which are the fundamental elements of propositional reasoning. We can achieve this by using a bottom-up method to examine the Abstract Syntax Tree (AST) of a given formula. The following code demonstrates this process in detail.

Please note that we represent the prime formulas as infix-notation strings of the formulas. The reason behind this choice is to facilitate the usage of set operations.[1]

The truth tree corresponds to a subtree of the AST of a formula. In the truth tree, each node represents a subformula of the original formula, where the subformula can either be a prime subformula or a subformula constructed from earlier nodes in the truth tree and connectives. In short, the truth tree is a tree structure consisting of subformulas, and each node in the tree is assigned a truth value.

For instance, consider the formula $\forall x\, P(x) \to Q$. It has 4 subformulas, out of which $\forall x\, P(x)$ and Q are prime subformulas (thus in the truth tree), and the $P(x)$ is the only subformula not in the truth tree.

To uniquely identify prime subformulas in the truth tree, we assign indices to them. Suppose $[A, \forall x\, P(x), Q(x, y)]$ is a list of all prime subformulas derived from a given set of formulas. In the truth tree of a formula, any node labeled A is assigned index 0, $\forall x\, P(x)$ is assigned index 1, and $Q(x, y)$ is assigned index 2. Only the prime nodes in the truth tree receive indices. This indexing aids in efficiently representing and handling the prime subformulas within the truth tree.

[1] The Nodes of the AST are not hashable, which is a requirement for efficient set operations.

5

Fitch Proof, 1st-order logic

5.1 Tautological Consequence in 1st-order logic

Index