Computational Proof Theory

A Computer Implementation of Fitch Calculus

Joohee Jeong

September 19, 2023

github.com/jhjeong314/proofmood

Contents

1 Formal Language

1.1 Symbols, strings and languages

Language is constructed using *symbols*, or equivalently *characters*, 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 any collection of strings composed of symbols.

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

Example 1.1 Let $\Sigma = \{a, \dots, z, A, \dots, Z\}$ 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) $\{a^n bc^n \mid n \in \mathbb{N}\} = \{b, abc, aabcc, aaabccc, \dots\}$ $\dashv$

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 $x = a_1a_2\cdots a_n$, then $\mathrm{set}(x) := \{a_1, a_2, \ldots, a_n\}$ and $|\mathrm{set}(x)| \le |x| = n$.

Please take note that in [Example 1.1], the cardinality of the set Σ is given. Specifically, $|\Sigma| = |\{a, \ldots, z, A, \ldots, Z\}| = 52$. However, it's important to recognize that in general, the cardinality of a set $\{a_1, \ldots, a_n\}$ is not necessarily equal to n. To illustrate this, consider the case where $a_1 = a$, $a_2 = b$, and $a_3 = a$. In this scenario, $|\{a_1, a_2, a_3\}| = 2$, which contradicts the assumption that the cardinality should be 3. It's possible to regard the symbols a_i as meta-symbols, and their scope or domain of applicability is defined by an alphabet that will be determined based on the context.

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.

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

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 $L_{\text{ar}} = L$. 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.

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

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.

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

(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

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

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:

$$\begin{aligned} &S \Rightarrow \lambda, && (1.6) \\ &S \Rightarrow aSb \Rightarrow a\lambda b \equiv ab && (1.7) \\ &S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb && (1.8) \\ &\vdots \quad \vdots \end{aligned}$$

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.

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

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

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

with production rules

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

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.

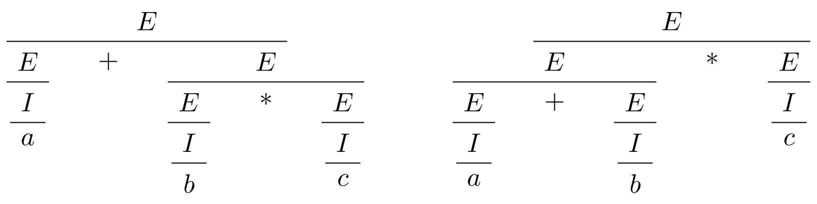
$$\begin{aligned} E &\Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + E * E \\ &\Rightarrow a + I * E \Rightarrow a + b * E \Rightarrow a + b * I \Rightarrow a + b * c \end{aligned} \tag{1.9}$$

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

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

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.

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

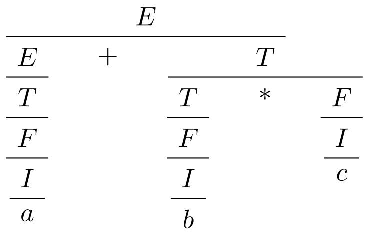
Here T stands for *term* and F stands for *factor.* We call the old grammar G_{ar1} and new grammar G_{ar2}. Be aware that T is also used to denote the set of terminal symbols. We hope that this will not cause any confusion.

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

$$\begin{aligned} E &\Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow I + T \Rightarrow a + T \\ &\Rightarrow a + T * F \Rightarrow a + F * F \Rightarrow a + I * F \Rightarrow a + b * F \\ &\Rightarrow a + b * I \Rightarrow a + b * c \end{aligned} \tag{1.10}$$

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}.

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow I + T \Rightarrow a + T \Rightarrow a + F \Rightarrow a + I \Rightarrow a + b \tag{1.11}$$
$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + I \Rightarrow E + b \Rightarrow T + b \Rightarrow F + b \Rightarrow I + b \Rightarrow a + b \tag{1.12}$$

(1.11) is an example of a *leftmost derivation*, which is defined to be a derivation such that for each intermediate expression in the derivation, when there are more than one non-terminal symbols, the production rule is applied to the leftmost one. (1.10) is a leftmost derivation too. But (1.12) is not a leftmost derivation—it is a *rightmost derivation*.

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}.

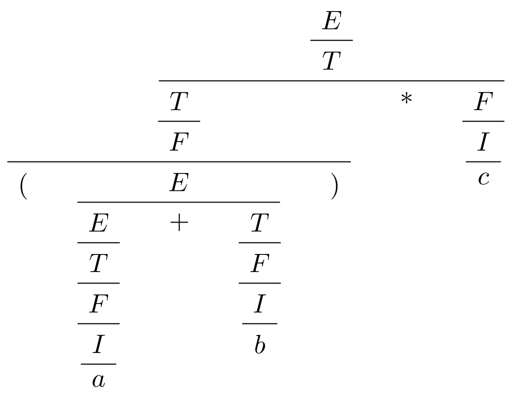
Exercise 1.4 Draw parse trees for $(a + b) + c$ and $a + (b + c)$ in G_{ar2}. ⊣

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 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*. ⊣

[2]This was defined at (1.5).

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



Most, but not all, 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 $|x| \le |y|$, 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.*

$$\mathcal{L}_{REG} \subsetneqq \mathcal{L}_{DCF} \subsetneqq \mathcal{L}_{CF} \subsetneqq \mathcal{L}_{CS} \subsetneqq \mathcal{L}_{REC} \subsetneqq \mathcal{L}_{RE} \tag{1.13}$$

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.

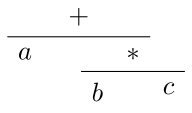
Some examples of abstract machines are finite state accepter(deterministic and non-deterministic), pushdown automaton(deterministic and non-deterministic), and Turing machine. We will not go into details about these machines here.

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.

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

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 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?

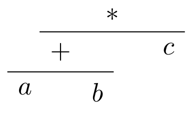
Come to think of it, syntax trees were never defined rigorously, while parse trees were. Syntax trees are called *abstract* because they do not represent every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure (i.e., partial order structure).

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

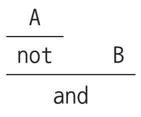
In this book, parsing will mean the process of obtaining a syntax tree(i.e., AST) from input text. Text may consist of a single line or multiple lines. We will consider the former case only for the time being.

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*.[1]

*Glyph*s are the visual representation of symbols or tokens. For instance, the glyph for the token '`not`' and '`and`' are '$\neg$' and '$\wedge$' respectively. The glyph for the token '`A`' is 'A', and the glyph for the token '`B`' is 'B' and so on. As you might have guessed, we use LaTeX to display these glyphs.

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]`.

[1]Usually we call them *lexemes* instead of tokens. Token may have many attributes and lexeme is just a string. But we will not make this distinction in this book.

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.

```python
# "value" argument is input through tokenizer().
# So a certain degree of validity check is done already.

class Token:
  def __init__(self, value):
  self.value = value
    if value in ("+", "-"):
      self.token_type = 'op_type1' # precedence 1
    elif value in ("*", "/"):
      self.token_type = 'op_type2' # precedence 2
    elif value == "(":
      self.token_type = 'lparen'
    elif value == ")":
      self.token_type = 'rparen'
    elif value.isdecimal():
      self.token_type = 'numeral'
    elif value.isalnum() and value[0].isalpha():
      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.

```python
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.*[2]

[2]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.

We define our Parser class as follows.

```
class Parser:
  def __init__(self, tokens):
    self.tokens = tokens
    self.current_token = None
    self.index = -1
    self.advance()  # set self.current_token to
                    # the first(i.e. self.index=0) element of tokens

  def advance(self): # increment self.index and set self.current_token
    self.index += 1
    if self.index < len(self.tokens):
      self.current_token = self.tokens[self.index]
    else:
      self.current_token = None

  def parse(self):
    return self.expr() # expr() corresponds to the starting symbol <expr>

  def expr(self):
    node = self.term()

    while(self.current_token is not None and
          self.current_token.token_type in ('op_type1')):
      # If we are at '+' in "a + b * c - ..." then the next token is '-'
      # because we will consume tokens by self.advance() and self.term().
      token = self.current_token
      self.advance()
      right_term = self.term()
      node = Node(token, [node, right_term]) # left associative

    return node

  def term(self):
    node = self.factor()

    while(self.current_token is not None and
          self.current_token.token_type in ('op_type2')):
      token = self.current_token
      self.advance()
      right = self.factor()
      node = Node(token, [node, right])

    return node

  def factor(self):
    if(self.current_token is not None and
       self.current_token.token_type == 'lparen'):
      self.advance()
      node = self.expr()
      if(self.current_token is not None and
         self.current_token.token_type == 'rparen'):
```

```
        self.advance()
      else:
        raise SyntaxError("Expected ')' after expression, in factor()")
    else:
      node = self.atom()

    return node

  def atom(self):
    if self.current_token is not None:
      token = self.current_token
      if token.token_type in ('numeral', 'identifier'):
        self.advance()
        return Node(token)
      else:
        raise SyntaxError(f"Expected numeral or identifier, in atom(): token")
    else:
      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.

```python
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.

```python
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.

```
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(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.

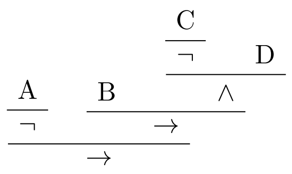
I attempted to find Python graphic packages that could fulfill this task, but unfortunately, I couldn't find any suitable options. Although the graphviz library is very nice, it doesn't offer a convenient way to draw the Abstract Syntax Tree (AST) in the style of the bussproofs package.

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.

———— ∘ ———— ∘ ————

```
not A imp B imp not C and D
```

$$\neg A \to B \to \neg C \land D$$



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 $\neg$, $\land$, $\lor$, $\to$ 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")
```

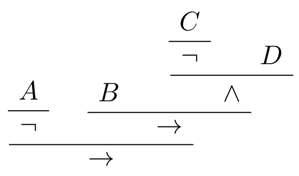
Following is a sample run of the codes in this section.

```
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}
```



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 $A_1 \rightarrow \mathrm{Bob}_{12} \wedge \mathrm{Binary\_Node}_{ij}$ 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. $\not\leftrightarrow$ 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[3] 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] }
```

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

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.

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. $\dashv$

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 $\rightarrow$, $\leftrightarrow$, $\not\leftrightarrow$, $\wedge$, $\vee$, then $\alpha \circ \beta$ is a formula.

- The set of formulas is the smallest set satisfying the above conditions. $\dashv$

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

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

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 $\forall x \,.\, \alpha \rightarrow \beta$ because the dot seems to strongly separate the determiner and the rest of the formula, and the formula might be interpreted as $\forall x\, (\alpha \rightarrow \beta)$ instead of $(\forall x\, \alpha) \rightarrow \beta$.

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.[4] 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.

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

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` 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.

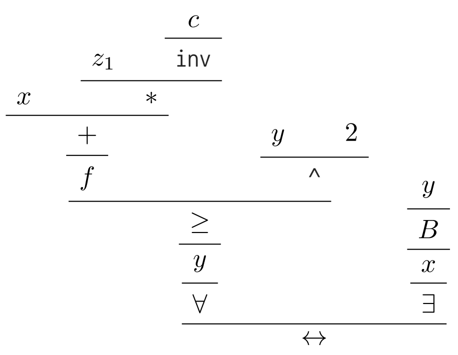
For example, the following formula

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

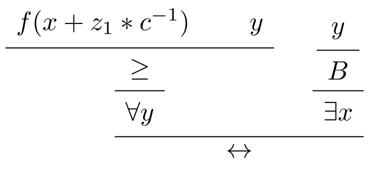
which is infix-LaTeX-rendered as

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

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 obtain the truth-value of α under $\bar{x}$ with respect to $\vec{A}$ using the truth table given in [Figure 3.1].

Truth table shows the truth-values of a formula under all possible truth-value assignments. So, the number of rows in a truth table is 2^n where n is the length of $\vec{A}$. Note that

Figure 3.1: Truth table for propositional connectives

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \to B$	$A \leftrightarrow B$	$A \not\leftrightarrow B$	$\perp$
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

n is less than or equal to the number of propositional variables appearing in α.

Obtaining the truth table for arbitrary formula α can be done by induction on the complexity of α. For example, if $\alpha \equiv (C \to \neg B) \to C$ and $\vec{A} = \langle A, B, C \rangle$, then the truth table for α is given as shown in [Example 3.1]. First we fill in the columns with level 1. Then we fill in the columns with level 2, and so on. The column with the highest level shows the truth-values of α.

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 5.2] 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 $\bar{x} = \langle 0,0,0 \rangle$. $\dashv$

Henceforth, a truth-value assignment is simply denoted by a string of 0's and 1's. For instance $\bar{x} = 101$ is short for $\bar{x} = \langle 1,0,1 \rangle$ and $\bar{x} = 0110$ is short for $\bar{x} = \langle 0,1,1,0 \rangle$.

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 *tautologically 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.

The expression $\Gamma \vDash_0 \varphi$ can be read as "Γ *tautologically entails* φ".

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 tautological contradiction iff $\vDash_0 \neg\varphi$. $\dashv$

In the definition above, the term 'tautologically' is used to distinguish those concepts with their first-order counterpart shown below.

- tautology: logically valid formula
- tautological contradiction: (logical) contradiction
- tautologically satisfied/unsatisfied: (logically) satisfied/unsatisfied
- tautologically satisfiable/unsatisfiable: (logically) satisfiable/unsatisfiable
- tautological consequence: logical consequence
- $\vDash_0$: $\vDash$

Semantics of 1st-order logic is described in [Section 5.1].

Using 'tautologically' repeatedly can become bothersome, and in cases where there is no ambiguity, we can skip its usage. For instance, if it's evident from the context that α represents a propositional formula, we can replace the phrase 'tautologically satisfiable' with 'satisfiable' when referring to α.

Likewise we may use the symbol $\vDash$ instead of $\vDash_0$.

Example 3.3 $A \wedge B$ is satisfiable because it is satisfied at $\bar{x} = 11$. But $A \wedge B$ is not a tautology because it is not satisfied at $\bar{x} = 10$. In fact it is not satisfied at any $\bar{x} \neq 11$.

$\{A \wedge B, \neg A\}$ 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 $\bar{x} := 111$ and $\bar{x} := 110$, and in both cases B is true.

Similarly, it is easily seen in [Example 3.1] 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. $\dashv$

Definition 3.4 For $\alpha \in$ `Formulas0` and $\Gamma \subseteq$ `Formulas0`, we give the symbol $\vDash_0$ the second meaning.

$$\bar{x} \vDash_0 \alpha \text{ means } \tilde{\alpha}(\bar{x}) = 1 \text{ and } \bar{x} \vDash_0 \Gamma \text{ means } \{\tilde{\alpha}(\bar{x}) \mid \alpha \in \Gamma\} = \{1\}.$$

For instance, in [Example 3.1], we see that

$$001 \vDash_0 \{A \to B,\ B \vee C,\ (C \to \neg B) \to C\} \quad \text{and} \quad 101 \nvDash_0 A \to B$$

Then we define Mod($*$), which means the set of *models* of $*$, as follows:

$$\mathrm{Mod}(\Gamma) \overset{\text{def}}{=} \{\bar{x} \mid \bar{x} \vDash_0 \Gamma\} \quad \text{and} \quad \mathrm{Mod}(\alpha) \overset{\text{def}}{=} \{\bar{x} \mid \bar{x} \vDash_0 \alpha\}$$

When $\vDash$ is used in this context, it is read "models", not "entails". So, for instance, the expression $101 \vDash_0 \Gamma$ is read "101 models gamma", not "101 entails gamma". $\dashv$

Caution. In *sequent calculus*, $\{\alpha_1, \dots, \alpha_n\} \vDash \{\beta_1, \dots, \beta_m\}$ means $\alpha_1 \wedge \cdots \wedge \alpha_n \vDash \beta_1 \vee \cdots \vee \beta_m$, but we will not use expression such as things in this book.

Fact 3.5 For $\alpha, \beta \in$ `Formulas0` and $\Gamma \subseteq$ `Formulas0`,

- $\Gamma \vDash \varphi$ iff $\mathrm{Mod}(\Gamma) \subseteq \mathrm{Mod}(\varphi)$,
- Γ is satisfiable iff $\mathrm{Mod}(\Gamma) \neq \varnothing$,
- α is a contradiction iff $\mathrm{Mod}(\alpha) = \varnothing$,
- $\mathrm{Mod}(\alpha \wedge \beta) = \mathrm{Mod}(\alpha) \cap \mathrm{Mod}(\beta)$,
- $\mathrm{Mod}(\alpha \vee \beta) = \mathrm{Mod}(\alpha) \cup \mathrm{Mod}(\beta)$,
- and so on, ... $\dashv$

Consider the following questions.

(1) Is the following formula a tautology?

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

(2) Is the following tautologically 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 3.6 A formula $\alpha \in$ `Formulas` is called *prime* iff the root of its AST is not a logical connective of positive arity. $\dashv$

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.

- $\perp$ is a prime formula but not a propositional variable.
- $(x < y) \land (x > y)$ is not a tautological contradiction although it is not satisfied in most first-order *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, $\vDash_1$, 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 $\vDash$ instead of $\vDash_1$, 'satisfiable' instead of 'first-order satisfiable' etc.

Note that $\alpha \vDash_0 \beta$ implies $\alpha \vDash \beta$ 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 *axiom*s 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 3.7 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. $\dashv$

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

$$\Gamma \vDash \varphi \iff \varphi \in f(\Gamma).$$

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.

[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.

Definition 3.8 Assuming that the syntactic definition of f such that $\Gamma \vDash \varphi \iff \varphi \in f(\Gamma)$ has been obtained, we will write $\Gamma \vdash \varphi$ instead of $\varphi \in f(\Gamma)$, or equivalently, instead of $\Gamma \vDash \varphi$. For propositional logic, from a function $f_0 : \mathcal{P}(\texttt{Formulas}) \to \mathcal{P}(\texttt{Formulas})$ such that $\Gamma \vDash_0 \varphi \Leftrightarrow \varphi \in f_0(\Gamma)$, we can define $\vdash_0$.

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

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

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

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. $\dashv$

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 of Hilbert system 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 3.9 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

[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.

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 `Formulas`. In a syntactic assignment statement, the l.h.s. must 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 $\rightarrow$ and $\neg$. $\dashv$

Definition 3.10 (Additional connectives) We define the following 5 connectives. The last one is 0-ary and the others are binary.

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

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

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

$$\text{A1} : \alpha \rightarrow (\beta \rightarrow \alpha)$$

$$\text{A2} : (\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$$

$$\text{A3} : (\neg\beta \rightarrow \neg\alpha) \rightarrow (\alpha \rightarrow \beta)$$

Note that all axioms are tautologies.

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

Note that axioms alone cannot prove anything other than axioms themselves. For instance how would you $A \vdash B \rightarrow A$ (or $\{A\} \vdash B \rightarrow A$ 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 3.12 (Modus Ponens) From the two formulas $\alpha \rightarrow \beta$ 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.

[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.

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

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

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

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

$$\{\alpha \to \beta,\ \alpha\} \vdash \beta$$ ⊣

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

If $\Gamma \in \mathcal{P}(\mathtt{Formulas})$ and $\varphi \in \mathtt{Formulas}$, 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.*

It should be clear from the definition that if $\Gamma_1 \subseteq \Gamma_2$, then a proof from Γ_1 is also a proof from Γ_2 but not vice versa. ⊣

Example 3.14 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 3.13], 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. ⊣

Example 3.15 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-symbols (or equivalently, a syntactic abbreviation), 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 3.21].

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

Theorem 3.16 (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.

$\vdots$	$\vdots$	$\vdots$
$n.$	$\alpha \to \beta$	..

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.

$\vdots$	$\vdots$	$\vdots$
$n.$	$\alpha \to \beta$	..
$n+1.$	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 $\langle \varphi_1, \dots, \varphi_n \equiv \beta \rangle$ be a Hilbert proof of β from $\Gamma \cup \{\alpha\}$.

[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.

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

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 3.14].

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 should 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 3.17 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 same is true for $\vDash$ in place of $\vdash$. ⊣

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 3.18 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 3.19 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 real 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 3.14)
- $\vdash \alpha \to ((\alpha \to \beta) \to \beta)$ (Example 3.18)
- $\vdash (\alpha \to \beta) \to ((\beta \to \gamma) \to (\alpha \to \gamma))$ (Example 3.19)

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 3.20 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 3.18
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 3.21 *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) □

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.)

3.4 Completeness and Soundness of proof systems

Definition 3.22 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*.

Each member of the theory of Γ is called a *theorem* of Γ and denoted by $\mathrm{Thm}(\Gamma)$.

A theory is said to be *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. ⊣

It can be shown that all tautologies can be proved in Hilbert system. This fact is called the *completeness* of Hilbert system (for propositional logic). So the completeness of a theory and the completeness of a proof system are two distinct concepts.

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.

Definition 3.23 If $\vdash$ denotes the proof relation of a proof system, then the proof system is said to be *complete* if for all formula α and all sets of formulas Γ

$$\Gamma \vDash \alpha \;\Rightarrow\; \Gamma \vdash \alpha \tag{3.1}$$

holds. The proof system is said to be *sound* if

$$\Gamma \vdash \alpha \;\Rightarrow\; \Gamma \vDash \alpha \tag{3.2}$$

holds.

When we impose the condition ‘Γ is finite’ in the above definitions, we obtain the *weak* version, or equivalently, the *finitary* version. ⊣

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. For $\vDash$, this is immediate from the definition[7], and for $\vdash$, most proof systems are set up so that this holds. Therefore, in the definition of the weak version of completeness and soundness, we could let $\Gamma = \varnothing$ without loss of generality.

The proof of the completeness theorem for Hilbert system is not easy. We will not present the proof here. For other proof systems, the direct proof of completeness is even harder. So it is usually done indirectly by showing that Thm(Γ) in the proof system is the same as Thm(Γ) in Hilbert system.

For any infinite or finite set Γ of formulas such that $\Gamma \vdash \varphi$, there exists a finite subset $\Gamma_{\text{fin}} \subseteq \Gamma$ such that $\Gamma_{\text{fin}} \vdash \varphi$. For Hilbert system (for propositional case) this is easily seen from the definition. Practically all proof systems are set up so that such a Γ_{fin} exists.

If we replace $\vdash$ by $\vDash$, then it is highly non-trivial to prove the existence of Γ_{fin}. The statement that this is the case is called the *compactness theorem*, whose proof is also highly non-trivial. The proof of the compactness theorem for the propositional case is simpler compared to the first-order case but still not that easy. We do not present the proof here.

The proof of the weak version of the completeness theorem for propositional logic is rather lengthy but doable. I want to mention that the proof is constructive. So, in principle, we can write a neat 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 will be very long.

However, I 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 for a wide class of logically valid first-order formulas.

The soundness theorem given in (3.2), which is the converse of the completeness theorem, can be proved easily using induction on the length of the Hilbert proof.

Proof. We assume $\Gamma = \varnothing$ without loss of generality. Suppose $\vdash \alpha$. We will show $\vDash \alpha$.

If the length of a proof of α is 1, then it must be a one of the 3 types of logical axioms given in [Definition 3.11]. In each case, the truth table for α shows that $\vDash \alpha$.

If α follows from modus ponens and earlier formulas α_i and $\alpha_j :\equiv \alpha_i \to \alpha$, then by the induction hypothesis, we have $\vDash \alpha_i$ and $\vDash \alpha_i \to \alpha$. Suppose that $\nvDash \alpha$ and $\bar{x}$ is a truth-value assignment such that $\tilde{\alpha}(\bar{x}) = 0$. We will get a contradiction. Since the truth value of $\alpha_i \to \alpha$ at $\bar{x}$ is 1, $\tilde{\alpha}_i(\bar{x})$ must be 0, which contradicts our induction hypothesis $\vDash \alpha_i$. □

3.5 Extending the Hilbert system

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 3.10], we introduced connectives $\vee, \wedge, \leftrightarrow, \not\leftrightarrow$ and $\perp$ 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 $A \vee \neg A \equiv \neg A \to \neg A \in L_{taut}$.

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

[7]We will define $\vDash$ for first-order logic later so that this is the case.

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.

- $\alpha \lor \beta \leftrightarrow (\neg\alpha \to \beta)$
- $\alpha \land \beta \leftrightarrow \neg(\alpha \to \neg\beta)$
- $(\alpha \leftrightarrow \beta) \to (\alpha \to \beta)$
- $(\alpha \leftrightarrow \beta) \to (\beta \to \alpha)$
- $(\alpha \to \beta) \to ((\beta \to \alpha) \to (\alpha \leftrightarrow \beta))$
- $(\alpha \not\leftrightarrow \beta) \leftrightarrow ((\alpha \land \neg\beta) \lor (\neg\alpha \land \beta))$
- $\bot \leftrightarrow (\alpha \land \neg\alpha)$

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 $\land$.

Figure 3.2: Inference rules for $\land$

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

We could introduce additional inference rules for the remaining connectives, but we will defer the discussion of these new rules until the section on Fitch calculus.

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.

We can extend the Hilbert system for more efficiency using meta-theorems. We have seen a meta-theorem called the Deduction theorem and found it very helpful.

The first meta-theorem we will introduce here is *substitution of propositional variables with formulas.* The idea is that if we have a Hilbert proof of α from Γ, then we can substitute all occurrences of any fixed propositional variable A in α and Γ with any fixed formula φ and obtain a Hilbert proof of α' from Γ', where α' and Γ' are obtained from α and Γ by substituting A with φ.

For example, from a Hilbert proof of $\vdash A \lor \neg A$ we can construct a Hilbert proof of $\vdash \varphi \lor \neg\varphi$ for any formula φ. We can prove this meta-theorem by induction on the length of the Hilbert proof. The proof is left as an exercise. In the proof, devising a suitable notational convention is the key.

The second meta-theorem is *replacement of subformula with equivalent formula.* The idea is that if we have a Hilbert proof of α from Γ, then we can replace any fixed occurrences of a fixed subformula φ of α and Γ with a formula φ' such that $\vdash \varphi \leftrightarrow \varphi'$ and obtain a Hilbert proof of α' from Γ', where α' and Γ' are obtained from α and Γ by replacing φ with φ'. If α has 3 occurrences of φ, we could replace any one of them, or any two of them, or all of them by φ'. We can prove this meta-theorem by induction on the length of the Hilbert proof. The proof is left as an exercise.

With these newly introduced tools, our extended Hilbert system undoubtedly becomes much more powerful and efficient. But still, it is far from being efficient enough for practical use. We will introduce more powerful proof systems in the following sections.

3.6 Natural Deduction

We will describe ND, the Natural Deduction system for propositional calculus only. The first-order version can be straightforwardly deduced from the propositional version of ND and the first-order version of Fitch calculus.

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

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

Figure 3.3: Inference rules for $\to$

$$\frac{\begin{matrix}[\alpha]\\ \vdots\\ \beta\end{matrix}}{\alpha \to \beta}\ \to \text{intro} \qquad\qquad \frac{\alpha \to \beta \qquad \alpha}{\beta}\ \to \text{elim}$$

although there is a difference which we will discuss shortly. '$\to$ 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 $\alpha \to \beta$. 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 $\bot$

$$\frac{\bot}{\alpha}\ \text{falsum} \qquad\qquad \frac{\begin{matrix}[\alpha \to \bot]\\ \vdots\\ \bot\end{matrix}}{\alpha}\ \text{RAA}$$

in a sense, eliminations, because the connective $\bot$ does not appear in the conclusions. So we gave them different names, other than '$\bot$ intro' and '$\bot$ elim'. 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 $\{\beta,\ \neg\beta :\equiv \beta \to \bot\} \vdash \alpha$, because this translates to $\{\beta,\ \beta \to \bot\} \vdash \bot \vdash \alpha$.

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 $\alpha \to \bot$, then we can derive α. The square bracket around $\alpha \to \bot$ means that $\alpha \to \bot$ 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 a set of vertices and a set of edges) having a designated vertex(called the root) with no cycles. But we will use a different definition as follows.

Definition 3.24 (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 leaf is sometime called a *terminal node.* 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.[8] 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. $\dashv$

A proof object should be a poset labeled with formulas.[9] This is because a formula, which is not a hypothesis[10], in the proof is obtained using a rule of inference applied to

[8]Precisely speaking, subtree T' determined by a node $x \in T$ of a tree $T = \langle T, \leq \rangle$ is a poset $\langle T', \leq' \rangle$ where $T' = \{y \in T \mid y \geq x\}$ and $\leq'$ is the restriction of $\leq$ on T'.

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

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

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.[11]

First, in Hilbert system, we cannot discharge a hypothesis or a premise. 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.

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 premises 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 3.25 (hypothesis and premise) The application of a rule of inference has exactly one *conclusion* and 0 or more *premises*. This application of a rule of inference 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 object of $\Gamma \vdash \varphi$, φ is also called the conclusion. $\dashv$

Definition 3.26 (ND proof) An *ND proof* is a labeled tree $(T, \leq, L)$ such that

(1) $T = \langle T, \leq \rangle$ 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) A premise of a conclusion must be a child of the conclusion or it is a square bracketed leaf, i.e., it is being discharged or in a position that can be discharged.

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

(7) (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.

[11] 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.

(8) 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 id 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 (7). 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 3.27

(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.

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

The discharged premise β doesn't present as explained in item (7) of [Definition 3.26]. If β was present, then that would make α an internal node which should then must 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.

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

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

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

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

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

(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, because ND doesn't need the Deduction theorem. 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. So I think ND is not good enough for practical use.

In the upcoming chapter, we will present a more efficient proof system called the Fitch calculus. Following that, we will create a computer-based implementation of this system, which will cover both propositional and first-order logic. This implementation will serve as a valuable practical tool for working with proofs in these logical frameworks.

3.7 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[12].

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

[12] `https://coq.inria.fr`

4 Fitch Proof, propositional logic

4.1 Introduction

Fitch proofs are syntactic objects constructed in the Fitch calculus proof system. In this chapter, we describe the propositional fragment of the Fitch calculus.

This proof system has no axioms and 14 rules of inference. Each of the six connectives, $\bot, \neg, \land, \lor, \to, \leftrightarrow$, has two corresponding rules of inference, an *introduction* and an *elimination*.

So far we have 12 rules associated with connectives. These are enough to make this proof system complete. But for efficiency, we will use 2 more rule, namely `repeat` and `LEM`. The rule `repeat` allows us to repeat a formula that has already been proved or assumed. The rule `LEM` is the law of excluded middle, which means we can use $\alpha \lor \neg\alpha$ for any 1st-order formula α.

We may describe the language of Fitch proofs with the following grammar.[1] In this grammar, `<formula>` means the first-order formula defined in p30.

This proof system deals with first-order formulas but only the propositional reasoning is allowed. For example, we cannot obtain $P(c)$ from the hypothesis $\forall x\, P(x)$. But we can obtain $P(c)$, for instance, from the two hypotheses $\exists x\, Q(x, c) \to P(c)$ and $\exists x\, Q(x, c)$.

Our context-free grammar for Fitch proofs is as follows.

```
<proof> ::= <hypothesis> "proves\n" <conclusion>
<hypothesis> ::= { <line> ".hyp" }+
<line> ::= <line_num> "." (<formula> "\n" ¦ <comment> ¦ <blank>)
<blank> ::= "\n"
<comment> ::= { <white_space> } "#" <utf8string> "\n"
<utf8string> ::= { [.] } # no "\n" allowed
<conclusion> ::= { (<line_annotated> ¦ <subproof>) }+
<subproof> ::= "{{" <hypothesis0> "proves\n" <conclusion> "}}"
<hypothesis0> ::= <line>
<line_annotated> ::= <line_num> "." (<formula> <ann> "\n" ¦ <comment> ¦ <blank>)
<ann> ::= <rule_of_inference> <premise>
<premise> ::= { <node_identifier> "," }
```

[1] In the actual implementation of the Fitch proof system, we do not exactly follow this grammar, which is given here for the sake of conveying the general idea.

```
<node_identifier> ::= <numeral> ¦ <numeral> "-" <numeral>
<rule_of_inference> ::= "." <conn> ("intro" ¦ "elim") ¦ "repeat" ¦ "LEM"
<conn> ::= "bot" ¦ "not" ¦ "and" ¦ "or" ¦ "imp" ¦ "iff"
<numeral> ::= [1-9] { [0-9] }
```

In the grammar presented above, subproofs are enclosed in double curly braces '{{' and '}}' to maintain context-free grammar. However, in our actual implementation, we diverge from this grammar practice. Instead, we employ an indentation system similar to Python to signify the subproof structure, which offers greater convenience for human users.

4.2 Fitch proof and its verification

As an example of a Fitch proof of $\{A \to B,\ B \to C\} \vdash A \to C$, we present the following.

Figure 4.1: A Fitch proof, #1

```
1. A imp B .hyp
2. B imp C .hyp
proves
  3. A .hyp
  proves
  4. B .imp elim 1,3
  5. C .imp elim 2,4
6. A imp C .imp intro 3-5
```

A Fitch proof consists of two parts: the hypothesis and the conclusion. These two parts are separated by the keyword '`proves`'.

Roughly, hypothesis is a set of formulas(called hypotheses) that are assumed to be true, and a conclusion is a list of formulas and subproofs that are logical consequences of the hypotheses. We set up our Fitch proof system so that this holds.

Every formula in conclusion, of the proof or of a subproof, need verification. Annotation does the job of this verification. A formula in a hypothesis does not need verification.

In the proof above, (line 1) and (line 2) form the hypothesis and (line 3)–(line 6) form the conclusion. The conclusion consists of a subproof (lines 3–5) and a formula (line 6). The subproof's hypothesis is (line 3) and its conclusion consists of (line 4) and (line 5). The formula (line 6) is the conclusion of the whole proof.

The meaning of the term 'conclusion' is twofold: a Fitch proof $\mathbb{P}$ has just one conclusion, which is a list of formulas and subproofs, and if the last member of the conclusion is a formula, then the formula is also called the conclusion of $\mathbb{P}$.

Let $\mathbb{P}$ be a Fitch proof, Γ be the hypothesis of $\mathbb{P}$, φ be a formula in the conclusion of $\mathbb{P}$, and $\mathbb{P}'$ be a subproof in the conclusion of $\mathbb{P}$.

In the example above, Γ is $\{A \to B, B \to C\}$, φ is $A \to C$, and $\mathbb{P}'$ is the subproof (lines 3–5).

Then we should have $\Gamma \vdash \varphi$ because φ is a logical consequence of Γ. We can verify this by the *annotation* of (line 6), namely '`imp intro 3-5`'. This annotation means that (line 6) is obtained by applying the inference rule '`imp intro`' to the subproof `3-5`. The subproof (lines 3–5) says that we can obtain C from the hypothesis A, which is exactly what we need to prove $A \to C$.

What do we mean by saying that $\mathbb{P}'$ is a logical consequence of Γ? The answer is as follows: Let γ be the hypothesis of $\mathbb{P}'$ and φ' be any formula in the conclusion of $\mathbb{P}'$. Then we should have $\Gamma, \gamma \vdash \varphi'$. Please note that, in this case, $\Gamma \nvdash \varphi'$ in general because φ' is not in the conclusion of $\mathbb{P}$. Recall that the conclusion of $\mathbb{P}$ consists of just two members, namely $\mathbb{P}'$ and φ.

When φ' is (line 4), $\varphi' :\equiv B$ is verified by the inference rule '$\rightarrow$ elim' which is also known as modus ponens. In this verification we use two premises (line 1), which is $A \rightarrow B$ and (line 3), which is A. When φ' is (line 5), $\varphi' :\equiv C$ is verified by the inference rule `imp elim` again, and this time we use (line 2) and (line 4) as premises.

Note that a proof may have several formulas in its hypothesis but a subproof may have only one formula in there.[2] This is just a convention. In case we want to use several formulas in the hypothesis of a subproof, we may just use the conjunction of those formulas instead.

The separator '`proves`' that exists between hypothesis and conclusion is usually replaced by a short horizontal line, and the indentation for a subproof is usually replaced by vertical border on the left. For example, the proof in [Figure 4.1] may be written as follows.

Figure 4.2: A Fitch proof, #2

```
|1. A imp B .hyp
|2. B imp C .hyp
|——
| |3. A .hyp
| |——
| |4. B .imp elim 1,3
| |5. C .imp elim 2,4
|6. A imp C .imp intro 3-5
```

Using LaTeX, we can obtain a prettier output like this:

Figure 4.3: A Fitch proof, #3

```
| 1. A → B       hyp
| 2. B → C       hyp
|———
| | 3. A         hyp
| |———
| | 4. B         → elim  3,1
| | 5. C         → elim  2,4
|
| 6. A → C       → intro  3-5
```

If we replace (line 6) with formula $B \wedge C$ and annotation $\wedge$ `intro` $4, 5$, then we obtain an invalid proof, which is shown below.

[2]An empty hypothesis is allowed.

```
| 1. A → B            hyp
| 2. B → C            hyp
|___
| | 3. A              hyp
| |___
| | 4. B              → elim  3,1
| | 5. C              → elim  2,4
|
| 6. B ∧ C          × ∧ intro  4,5
```

The reason why we cannot verify (line 6) with the annotation is that the premise nodes 4, 5 are not earlier than node 6 in the Fitch proof tree, although the line numbers 4, 5 are. We'll return to this issue shortly.

4.3 Fitch proof tree

The Fitch proof shown in [Figure 4.3] can be considered as a tree, consisting of 8 nodes. 6 of them are formulas and 2 of them are (sub)proofs. This Fitch proof tree is shown in [Figure 4.4]. In a Fitch proof tree, formulas are terminal nodes and (sub)proofs are internal nodes. Parents are earlier than children. Among two children of the same parent, the left child is earlier than the right child—this defines the partial order relation on the nodes of the Fitch proof tree.

Figure 4.4: Fitch proof tree #1

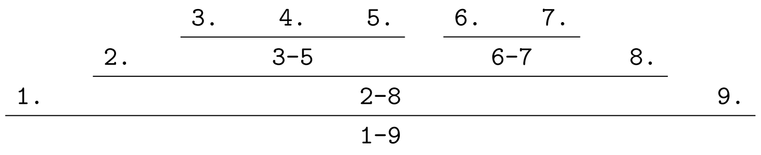
$$\cfrac{\texttt{1.}\ A \to B \qquad \texttt{2.}\ B \to C \qquad \cfrac{\texttt{3.}\ A \qquad \texttt{4.}\ B \qquad \texttt{5.}\ C}{\texttt{3-5}} \qquad \texttt{6.}\ A \to C}{\texttt{1-6}}$$

Here is another example of a Fitch proof, which proves $A \to B \to (A \lor C \to B \lor C)$. In this proof, we use '$\lor$ elim' and ''$\lor$ intro' rules which we did not introduce yet.

```
| 1. A → B                 hyp
|___
| | 2. A ∨ C               hyp
| |___
| | | 3. A                 hyp
| | |___
| | | 4. B                 → elim  1,3
| | | 5. B ∨ C             ∨ intro  4
| |
| | | 6. C                 hyp
| | |___
| | | 7. B ∨ C             ∨ intro  6
| |
| | 8. B ∨ C               ∨ elim  2,3-5,6-7
|
| 9. A ∨ C → B ∨ C         → intro  2-8
```

The corresponding tree is shown in [Figure 4.5] below. This time, for the leaf nodes, only the line numbers are shown and the formulas are not shown to save space.

Figure 4.5: Fitch proof tree #2



Remark 4.1 In [Figure 4.5], we can find many node pairs that are *incomparable*, e.g., (3, 8), (4, 7). Incomparable pair means that no element of the pair is earlier than the other. Note that the following node pairs are comparable: $3 < 4 < 5$, $6 < 7$, $2 < 5$, $1 < 7$, $3\text{-}5 < 8$.

It is important to note that we define the order relation between Fitch tree nodes so that 3-5 is not earlier than 5. This is because 3-5 cannot be used as a premise of 5. ⊣

Exercise 4.2 In the Fitch proof corresponding to [Figure 4.5], if we remove (line 9), the result is still a valid proof, say $\mathbb{P}_1$. Can we conclude $A \to B \vdash B \vee C$ from $\mathbb{P}_1$? If not, why? What about $A \to B,\ A \vee C \vdash B \vee C$? ⊣

4.4 Parsing Fitch Proofs

We aim to develop a capability to parse input text resembling [Figure 4.1] and generate a corresponding Fitch proof tree object. This object should have data structure so that it can be easily presented in a format similar to [Figure 4.4]. It's essential that the text in [Figure 4.2] can be parsed into the same object as the text in [Figure 4.1]. This enables us to easily copy, paste, make minor modifications, and then reparse the content.

[Figure 4.1] is shown below again for convenience.

```
1. A imp B .hyp
2. B imp C .hyp
proves
  3. A .hyp
  proves
  4. B .imp elim 1,3
  5. C .imp elim 2,4
6. A imp C .imp intro 3-5
```

We'll implement the parser so that the line number in each line may be omitted. This should make entering the text easier.

Input text consists of lines. Each line is either a hypothesis line, a conclusion line, and the '`proves` line' which separates the hypothesis part and the conclusion part. Blank lines and comment lines are allowed too. But we will not pay attention to these rather trivial lines for the time being.

A hypothesis line consists of a line number, a formula, and a keyword '`hyp`'. A conclusion line consists of a line number, a formula, and an annotation.

4.4.1 Annotation

Naturally, line parsing is the first step in parsing a Fitch proof. Line numbers are trivial, and we already know how to parse formulas. So it remains to work on annotations only, for line parsing of course.

Before we begin delving into annotations, it is useful to define the following two classes: `Connective` and `RuleInfer`, and some global constants too.

```
from enum import Enum

class Connective(Enum):
  BOT = 'bot'
  NOT = 'not'
  AND = 'and'
  OR = 'or'
  IMP = 'imp'
  IFF = 'iff'

class RuleInfer(Enum):
  BOT_INTRO = "bot intro"
  BOT_ELIM = "bot elim"
  NOT_INTRO = "not intro"
  NOT_ELIM = "not elim"
  AND_INTRO = "and intro"
  AND_ELIM = "and elim"
  OR_INTRO = "or intro"
  OR_ELIM = "or elim"
  IMP_INTRO = "imp intro"
  IMP_ELIM = "imp elim"
  IFF_INTRO = "iff intro"
  IFF_ELIM = "iff elim"
  REPEAT = "repeat"
  LEM = "LEM"
  HYP = "hyp"

CONN_LIST = [conn.value for conn in Connective]
INTRO_OR_ELIM = ['intro', 'elim']
RULES_AUX = ['repeat', 'LEM', 'hyp']
VERT = ' │ '
PROVES = ' ├──'
TAB = ' '
```

In the following code for annotations, we omit the tedious details of parsing and focus on the essential part. It should be enough to convey the general idea. The full code is available in `/proofs/propositional.py`

```
class Ann:
  """Annotation of a line of a proof tree"""
  def __init__(self, input_str: str):
    self.input_str = input_str
    self.rule = None # RuleInfer type
    self.premise = None # [node_code,.. , ]
                        # node_code ::= ln: digit ¦ ln_s-ln_e: str
                        # ln = line number
    self.parse()

  def __str__(self) -> str:
    if self.rule is None:
      return ''
    else:
      premise_str = ','.join(self.premise) if self.premise else ''
      return f"{self.rule.value} {premise_str}"

  def build_str(self) -> str:
    if self.rule is None:
      return ''
    else:
      return (f"rule: {self.rule.value}\n"
              f"premise: {self.premise}")

  def parse(self) -> None:
    """From self.input_str, set self.rule and self.premise."""
    .. snip ..
```

Following is a sample run.

```
# Construct an Ann object from a string.
s = " imp  intro   4-8 "
try:
  ann = Ann(s)
  print(f"input_str = '{ann.input_str}'")
  print(ann.build_str())
  print("\noutput from __str__():")
  print(" ", ann)
except Exception as e:
    print(e)

input_str = ' imp  intro   4-8 '
rule: imp intro
premise: ['4-8']

output from __str__():
  imp intro 4-8
```

4.4.2 Structure of the Fitch proof tree

The data structure of Fitch proof trees is shown in the following two classes. The first one is for the nodes of a proof tree, and the second one is for the proof tree itself.

```
class NodeLabel:
  """label of a node of a proof tree (like a token)"""
  def __init__(self, type: str = 'subproof', line: str = '',
               formula: Formula | None = None, ann: Ann | None = None):
    """ type ::= 'subproof' | 'formula' | 'comment' | 'blank'
        For subproofs, there's not much we need to do.
        The same is true for comments and blank lines.
        For formulas, initialization is done in two ways:
          (1) from a string, and
          (2) from a Formula object and an Ann object.
    """
    self.type = type
    self.line = line
    self.formula = formula
    self.ann = ann
    self.is_hyp = None # type: bool | None
    if self.type != 'formula':
      pass
    elif self.line != '':
      self.parse_line()
    elif isinstance(formula, Formula) and isinstance(ann, Ann):
      self.is_hyp = ann.rule == RuleInfer.HYP
    else:
      raise ValueError("NodeLabel.init(): Wrong arguments")

  def parse_line(self) -> None:
    """ Parse self.line and set self.formula, self.ann and self.is_hyp.
        self.line does not have the line number part."""
    line = self.line
    # Isolate formula part and annotation part from line.
    pos = word_in_str(CONN_LIST + RULES_AUX, line)
    #^ line[pos] == '.' if pos != -1
    if pos == -1:
      pos = len(line)
    fmla_part = line[:pos].lstrip()
    ann_part = line[pos:].rstrip()
    if ann_part.startswith('.'):
      ann_part = ann_part[1:]
    else:
      ann_part = '' # actually, this is redundant
    # If fmla_part is illegal, then show the ugly messages and exit.
    self.formula = parse_ast(fmla_part)
    # If ann_part is illegal, then set self.ann to ann_part.
    # So self.ann is either an Ann object or a string or None.
    try:
      self.ann = Ann(ann_part)
      self.is_hyp = self.ann.rule == RuleInfer.HYP
      # isinstance(self.ann, Ann)
    except Exception as e:
      # error, but keep it anyway
      # not isinstance(self.ann, Ann)
      self.ann = ann_part

  def __str__(self) -> str:
    if self.type == 'formula':
      return f"{self.formula}\t .{self.ann}"
    else: # self.type == 'subproof' | 'comment' | 'blank'
      return self.line

  def build_str(self) -> str:
    .. snip ..
```

Following is a sample run.

```
# Construct a NodeLabel object from a string.
```

```
line = "A imp B .hyp"
label = NodeLabel('formula', line)
print(label.build_str())
print()
print(label)
```

```
type: formula
line: A imp B .hyp
formula: A imp B
ann: hyp
is_hyp: True

A imp B         .hyp
```

Here is the second class. Among the 5 properties, only the 1st, 2nd and the 4th are used in this section §4.4. The 3rd and 4th are set in the `build_index()` method, and the 5th one is set in the `build_index_dict()` method. The `build_index()` method is called after the proof tree is constructed. The `build_index_dict()` method is called the `build_index()` method is called.

The goal of this section is to parse the Fitch proof text and print it. For this purpose, `self.index_dict` are not needed. We will not present the code of these methods here. Instead it is shown in the next section §4.5. If you are curious about the significance of `self.index`, please refer to the text surrounding [Figure 4.7], and then come back here.

```
class ProofNode:
  # In order to get node: Node from p_node: ProofNode, use
  #   node = p_node.label.formula.ast
  def __init__(self, label: NodeLabel, children=None):
    self.label = label
    self.children = children if children else []
    #^ list of ProofNode objects, not the list of labels
    # The following 3 attributes are set by the build_index() method.
    self.index = None # type: List[int] ¦ None
    self.line_num = None # type: str ¦ None # e.g., '4', '6-10'
    self.index_dict = None # type: Dict[str, List[int]] ¦ None

  def __str__(self) -> str:
    return self.build_fitch_text()

  def build_fitch_text(self) -> str:
  """ Recursively build Fitch-style proof text which looks like:
     │1. A imp B  .hyp
     │2. B imp C  .hyp
     ├──
     │ │3. A   .hyp
     │ ├──
     │ │4. B   .imp elim 1,3
     │ │5. C   .imp elim 2,4
     │6. A imp C  .imp intro 3-5
  """
    assert self.label.type == 'subproof', \
      "build_fitch_text(): this method must be called for a subproof type node"

    ret_str = ''
    b_hyp = True
    level = len(self.index) if self.index else 0 # always >= 1
    for kid in self.children:
      if b_hyp and not kid.label.is_hyp:
        b_hyp = False
        ret_str += VERT * (level - 1) + ' ├──\n'
      if kid.label.type != 'subproof':
        line_str = f"{kid.label}"
        ret_str += VERT * level + f'{kid.line_num}. ' + line_str + '\n'
      else:
        ret_str += kid.build_fitch_text()
    return ret_str

  def build_fitch_latex(self) -> str:
    """ Recursively build a Fitch-style proof latex source. """
    raise NotImplementedError("build_fitch_latex() not implemented")
```

Here is an example run where we create a proof tree comprising 5 nodes and then display it. It's important to note that in practice, we do not manually construct proof trees in this manner; rather, we parse input text to generate the proof tree.

```
# Construct a ProofNode manually. (not by using the parser)
line1 = "A imp B .hyp"
label1 = NodeLabel('formula', line1)
node1 = ProofNode(label1)
line2 = "A .hyp"
label2 = NodeLabel('formula', line2)
node2 = ProofNode(label2)
line3 = "B .imp elim 1,2"
label3 = NodeLabel('formula', line3)
node3 = ProofNode(label3)
line4 = "A .repeat 2"
label4 = NodeLabel('formula', line4)
node4 = ProofNode(label4)
label0 = NodeLabel()
node0 = ProofNode(label0, [node1, node2, node3, node4])
node0.build_index() # necessary for printing node0
print(node0)
```

```
│1. A imp B         .hyp
│2. A         .hyp
├──
│3. B         .imp elim 1,2
│4. A         .repeat 2
```

4.4.3 The Fitch proof parser

In a Fitch-style proof, indentation plays a pivotal role in its parsing. Indentation is not inherently a part of the context-free grammar, so to handle it effectively, we need to preprocess the input text by substituting indentation with corresponding double brace pairs. The parsing process can be broadly divided into three main steps:

(1) Extract a list of lines from the input text.

(2) Replace indentation with suitable double brace pairs.

(3) Utilize the standard recursive descent parsing algorithm designed for context-free grammars.

Furthermore, step (2) above can be broken down into the following sub-steps:

(*i*) Translate leading spaces, tabs, and VERTs into indentation levels represented as 1, 2, and so on, respectively. Assign each line its appropriate indentation level.

(*ii*) Remove any leading line numbers from each line.

(*iii*) By observing the changes in indentation levels between lines, convert the indentation into double brace pairs to denote subproofs.

(*iv*) When there is a change of line type from conclusion to hypothesis, add `["}}", "{{"]` to the list.

Step (1) and (2) are handled by the following function. It is not a method of any class but a standalone utility function.

```
  def get_str_li(proof_str: str, tabsize: int) -> List[str]:
  """
  This is a preprocessing function for parse_fitch().

  Convert proof_str to a list of strings, where the subproofs are
  indicated by double brace pairs.
  """
  # split proof_str into lines
  str_li = proof_str.split('\n') # this will be converted to str_li_ret
   .. snip snip ..

def print_lines(lines: List[str]) -> None:
  """ This is a test driver function for get_str_li().
      Print lines with indentation. """
  level = 1
  for str in lines:
    if str == "}}":
      print(f"{('  ' * (level - 2))}{str}")
      level -= 1
    else:
      print(f"{('  ' * (level - 1))}{str}")
    if str == "{{":
      level += 1
```

Here is a sample run. This is a Fitch proof for $A \to B \vdash A \lor C \to B \lor C$.

Example 4.3 We will refer to this proof string multiple times in the future.

```
prf_str = '''
1. A imp B .hyp
proves
```

```
    2. A or C .hyp
    proves
      3. A .hyp
      proves
      4. B .imp elim 1,3
      5. B or C .or intro 4
      6. C .hyp
      proves
      7. B or C .or intro 6
    8. B or C .or elim 2,3-5,6-7
  9. A or C imp B or C .imp intro 2-8'''
  lines = get_str_li(prf_str, tabsize=2)
  print_lines(lines)
```

```
A imp B .hyp
{{
  A or C .hyp
  {{
    A .hyp
    B .imp elim 1,3
    B or C .or intro 4
  }}
  {{
    C hyp
    B or C .or intro 6
  }}
  B or C .or elim 2,3-5,6-7
}}
A or C imp B or C .imp intro 2-8
```

Now that the indentations have been successfully converted into double brace pairs, it is evident that we can straightforwardly apply the standard recursive descent parsing algorithm. Following is our main class for parsing Fitch proofs. It is relatively simple as was expected.

```python
class ProofParser:
  def __init__(self, lines: List[str], tabsize: int):
    self.lines = lines
    self.current_line = None
    self.level = 1 # indentation level (ground level = 1)
    self.tabsize = tabsize
    self.index = -1
    self.advance()

  def advance(self): # increment self.index and set self.current_line
    self.index += 1
    if self.index < len(self.lines):
      self.current_line = self.lines[self.index]
    else:
      self.current_line = None

  def proof(self) -> ProofNode:
    children = []
    while (line_str := self.current_line) is not None:
      if line_str == '{{':
        self.level += 1
        self.advance()
        children.append(self.proof())
      else:
        if line_str == '}}':
          self.level -= 1
          if self.level <= 0:
            raise ValueError("ProofParser.proof(): below ground level")
          self.advance()
          break
        elif line_str == '':
          label_type = 'blank'
        elif line_str.lstrip().startswith('#'):
          label_type = 'comment'
        else:
          label_type = 'formula'
        label = NodeLabel(label_type, line_str)
        children.append(ProofNode(label)) # leaf node
        self.advance()

    return ProofNode(NodeLabel(), children)
```

For actual parsing, we use the following function.

```
def parse_fitch(proof_str: str, tabsize: int = 2) -> ProofNode:
  lines = get_str_li(proof_str, tabsize)
  #^ list of strings, where the subproofs are indicated by double brace pairs
  #^ each element of the list corresponds to a line of the proof
  parser = ProofParser(lines, tabsize)
  proof_node = parser.proof()
  proof_node.build_index() # necessary for printing
  proof_node.index_dict = proof_node.build_index_dict()
  # ^necessary for verifying the proof
  if parser.level != 1:
    raise ValueError("parse_fitch(): parsing ended with non-ground level")
  return proof_node
```

We show a sample run below. We use the proof string from [Example 4.3]. The output is shown in [Figure 4.6] for future reference.

It is worth noting that this proof contains two adjacent subproofs at the same level. In both the input string and the output, it might be somewhat hard to discern that line 5 ends one subproof and line 6 starts a new subproof. However, this issue will be addressed when we generate the LaTeX output later on.

```
prf_str = '''
  .. snip ..
'''
print(proof := parse_fitch(prf_str))
```

Of course, we cannot appreciate the full power of the parser until we use it in verifying proofs. We'll see that shortly.

Figure 4.6: Fitch proof for $A \to B \vdash A \lor C \to B \lor C$

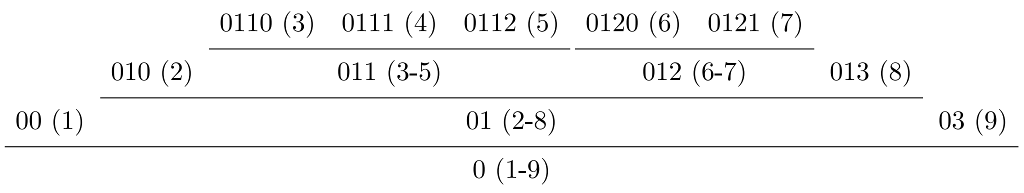
```
│1. A imp B         .hyp
├──
│ │2. A or C         .hyp
│ ├──
│ │ │3. A         .hyp
│ │ ├──
│ │ │4. B         .imp elim 1,3
│ │ │5. B or C    .or intro 4
│ │ │6. C         .hyp
│ │ ├──
│ │ │7. B or C       .or intro 6
│ │8. B or C         .or elim 2,3-5,6-7
│9. A or C imp B or C         .imp intro 2-8
```

4.5 Indexing tree nodes

When working with a list, we can efficiently retrieve its elements using their respective indices. In a similar fashion, we can access nodes within any tree using their node indices. A *node index* is represented as a list of nonnegative integers, which uniquely specifies the location of a node within the tree.

The Fitch proof shown in [Figure 4.6] is represented as a tree in [Figure 4.7]. Actually, this figure is almost the same as [Figure 4.5] we've seen earlier. In this figure, each node is labeled with the index and the line number in parentheses. The formula and annotation are not shown to save space. Index $[a_1, a_2, \ldots, a_n]$ is shown as a string $a_1a_2 \cdots a_n$. For example, the node index $[0, 1, 2]$ is shown as 012.

Figure 4.7: Node indices



The exact definition of index and line number can be found in the code of `build_index()` given below. This is a method of the `ProofNode` class.

```
def build_index(self, p_index: List[int] = [], i: int = 0,
  l_num: int = 1) -> int:
  """ Recursively set self.index and self.line_num.
    Automatically called by the parse_fitch() function, where
    self is the root of the whole proof.
    p_index means the parent's (tree)index(: List[int]).
    i is the (list)index(: int) of self in the parent's children list.
    l_num is the line number to be given to self for leaf nodes.
    Return value is the increment of line number for the next leaf.
    self.index is recursively set from the root to the leaves.
    self.line_num is recursively set from the leaves to the root. """
  self.index = p_index + [i]
  if self.children: # subproof case
    line_inc = 0 # tentative return value
    for i, kid in enumerate(self.children):
      line_inc += kid.build_index(self.index, i, l_num + line_inc)
      self.line_num = f"{l_num}-{l_num + line_inc - 1}"
  else: # leaf node case
    self.line_num = str(l_num)
    line_inc = 1

  return line_inc
```

The Fitch proof shown in [Figure 4.6] displays only the leaf nodes (i.e., formulas) as lines, concealing the internal nodes (i.e., subproofs). In Fitch proof, formulas are conveniently identified by their corresponding line numbers (type coerced to string), while subproofs are denoted by the string s-e, where s represents the line number of the subproof's initial line and e signifies the line number of the subproof's concluding line. We will call the string s-e the line number of the subproof.

To enhance the functionality of the ProofNode class, we introduce a property index_dict for the ProofNode class that enables us to convert between indices and line numbers efficiently. Additionally, we need to implement methods that facilitate the retrieval of a node based on its node index and/or line number. We update the ProofNode class as follows.

```
class ProofNode:
  ..
  def build_index_dict(self) -> Dict[str, List[int]]:
    """ Recursively build a dictionary with line numbers as keys and
        corresponding tree indices as values.
        Automatically called by the parse_fitch() function after build_index()
        has been called, where self is the root of the whole proof.
        The Return value is saved as proof_root.index_dict.
    """
    ret_dict = {}
    ret_dict[self.line_num] = self.index
    for kid in self.children:
      ret_dict.update(kid.build_index_dict())
    return ret_dict

  def get_p_node(self, node_code: List[int] | str): # ProofNode type
    """ Get and return the p_node specified by node_code, which
        is either a tree index(: List[int]) or a line number(:str).
        node_code of the form 's-e' is accepted too.
        self must be the root of the whole proof. """
    assert self.index_dict is not None, "get_p_node(): index_dict is None"
    p_node = self
    if isinstance(node_code, str):
      index = self.index_dict.get(node_code)
      if index is not None:
        p_node = self.get_p_node(index)
      else:
        raise ValueError(f"get_p_node(): line number {node_code} not found")
    else:
      if node_code == self.index:
        p_node = self
      else:
        for i in node_code[1:]:
          p_node = p_node.children[i]
    return p_node

  def toggle_node_code(self, node_code: List[int] | str) -> str | List[int]:
    """ Toggle node_code type between index and line number. """
    node = self.get_p_node(node_code)
    if node is None:
      raise ValueError("toggle_node_code(): node not found for " \
                       f"{node_code}")
    if isinstance(node_code, str):
      return node.index # type: ignore
    else:
      return node.line_num # type: ignore
```

Here is a sample run for the Fitch proof shown in [Figure 4.6] and [Figure 4.7].

```
prf_str = '''
  .. snip ..
'''
proof := parse_fitch(prf_str)
index_dict = proof.build_index_dict()
pprint(index_dict, sort_dicts=False) # from pprint import pprint

{'1-9': [0],
 '1': [0, 0],
 '2-8': [0, 1],
 '2': [0, 1, 0],
 '3-5': [0, 1, 1],
 '3': [0, 1, 1, 0],
 '4': [0, 1, 1, 1],
 '5': [0, 1, 1, 2],
 '6-7': [0, 1, 2],
 '6': [0, 1, 2, 0],
 '7': [0, 1, 2, 1],
 '8': [0, 1, 3],
 '9': [0, 2]}
```

4.6 Verifying Fitch Proofs

The order relation between nodes in a Fitch proof tree is very important. Basically, the formula or the subproof at node1 can be used as a premise of the formula at node2 if and only if node1 *is earlier* than node2. A general idea of this concept can be obtained from [Remark 4.1]. The following code shows the precise definition of this concept.

```
class ProofNode:
  ..
  def is_earlier(self, node_code1, node_code2) -> bool:
    """ Test if node_code1 is earlier than node_code2, which is equivalent
```

```
        to saying that node_code1 can be used as a premise of node_code2. """
    assert self.index_dict is not None, "get_node(): index_dict is None"
    # make sure that both node_code1 and node_code2 are of the type List[int]
    node_code1 = node_code1 if not isinstance(node_code1, str) \
      else self.index_dict[node_code1]
    node_code2 = node_code2 if not isinstance(node_code2, str) \
      else self.index_dict[node_code2]
    # This order is somewhat unusual but very important.
    # It is at the heart of the Fitch proof system.
    len1 = len(node_code1)
    return len1 <= len(node_code2) and \
      node_code1[:-1] == node_code2[:len1 - 1] and \
      node_code1[-1] < node_code2[len1 - 1]
```

Let's continue with the Fitch proof in [Figure 4.6] and [Figure 4.7].

```
proof := parse_fitch(prf_str)
index_dict = proof.build_index_dict()
for i in index_dict:
  for j in index_dict:
    label_j = proof.get_node(j).label
    if label_j.type == 'formula' and not label_j.is_hyp: # conclusion
      if proof.is_earlier(i, j):
        print(f"{i} <= {j}")
```

```
1 <= 4
1 <= 5
1 <= 7
1 <= 8
1 <= 9
2-8 <= 9
2 <= 4
2 <= 5
2 <= 7
2 <= 8
3-5 <= 7
3-5 <= 8
3 <= 4
3 <= 5
4 <= 5
6-7 <= 8
6 <= 7
```

To implement the verification of Fitch proofs, we need to define a new class called `FormulaProp`, which is a subclass of `Formula` with lots of methods added for proof verification of propositional logic.

```
class FormulaProp(Formula):
  def __init__(self, input: str | Node):
      super().__init__(input)

  def is_fmla_type(self, fmla_type: Connective) -> bool:
    return self.ast.token.value == fmla_type.value

  def get_kid_node1(self) -> Node:
    """ Get the kid node of self, which is a negation formula. """
    assert self.is_fmla_type(Connective.NOT), \
            "FormulaProp.get_kid_node1(): not negation formula"
    return self.ast.children[0]

  def get_kid_node2(self) -> Tuple[Node, Node]:
    """ Get the kids of self, whose root is a binary connective. """
    root_token = self.ast.token
    assert root_token.value in CONN_LIST and root_token.arity ==2, \
            "FormulaProp.get_kid_node2(): not binary connective formula"
    return (self.ast.children[0], self.ast.children[1])
```

Now, the core method of this class is given below. This method is straightforward but lengthy. So it is not shown in its entirety here. The full code can be found in the file `fitch_proof.py`.

```
  def verified_by(self, rule_inf: RuleInfer, premise: List[Node] = []) -> bool:
    """ Test if self is verified from (rule_inf, premise).
        If a subproof need be a member of premise, then use the formula
        A imp B where A is the hypothesis and B is the last formula
        of the subproof.
    """
    match rule_inf:
      case RuleInfer.BOT_INTRO:
         .. snip ..
      case RuleInfer.BOT_ELIM:
       .. snip snip ..
      case RuleInfer.IMP_ELIM: # modus ponens
        if len(premise) != 2:
          return  False
        longer_i = 0 if premise[0].longer_than(premise[1]) else 1
        longer_prem = premise[longer_i]
        shorter_prem = premise[1 - longer_i]
        longer_fmla = FormulaProp(longer_prem)
        if longer_fmla.is_fmla_type(Connective.IMP):
          left, right = longer_fmla.get_kid_node2()
          if left == shorter_prem and right == self.ast:
            return True
        return False
       .. snip snip ..
      case RuleInfer.HYP:
        # We can assume any formula as a hypothesis.
        # In other words, hypotheses do not need verification.
        return not premise
      case _:
        raise ValueError("FormulaProp.verified(): wrong rule_inf")
    return True # type checker needs this
```

A sample run is shown below.

```
fmla = FormulaProp('B1(x)')
node1 = parse_ast('(A iff not C_1) imp B1(x)')
```

```
node2 = parse_ast('A iff not C_1')
print(fmla.verified_by(RuleInfer.IMP_ELIM, [node1, node2]))
print(fmla.verified_by(RuleInfer.IMP_ELIM, [node2, node1]))
print(fmla.verified_by(RuleInfer.AND_INTRO, [node1, node2]))
```

```
True
True
False
```

To enable the verification of a line in a Fitch proof, the `verified_by()` method should accept the premise argument with the type `List[str]`. This allows us to provide the premises as a list of line numbers, which are of string type. This design choice aligns with the fact that each line's premises are determined by its annotation, which uses line numbers to identify both the lines and the formulas within them.

We will create a method called `ProofNode.verified_by()`, making use of the previously implemented `FormulaProp.verified_by()` method. While both methods serve the same purpose, it is essential to note that they have different arguments. The latter verifies a formula of type `FormulaProp`, while the former is designed to verify the formula specified by a line number. Consequently, their argument structures should differ accordingly.

We will create two additional methods within the `ProofNode` class, specifically named `verified()` and `verified_all()`.

```
ProofNode:
  def verified_by(self, conc: str, rule_inf: RuleInfer,
                  premise: List[str] = []) -> bool:
    """ Test if the conclusion with line number conc is verified by
        (rule_inf, premise). conc of the form 's-e' is not accepted.
        self must be the root of the whole proof. """
    assert bFmla(conc), f"verified_by(): conclusion='{conc}' is not accepted"
    assert self.index_dict is not None, "verified_by(): index_dict is None"
    p_node = self.get_p_node(conc) # ProofNode type
    if p_node.label.is_hyp:
      return True
    else:
      fmla = p_node.label.formula
      assert isinstance(fmla, Formula)
      conclusion = FormulaProp(fmla.ast)
      premise_nodes = []
      for p in premise:
        if not self.is_earlier(p, conc):
          return False
        if bFmla(p):
          fmla = self.get_p_node(p).label.formula
          premise_nodes.append(fmla.ast) # type: ignore
        else: # convert subproof to an implication formula
          str_li = [str.strip() for str in p.split('-')]
          s = str_li[0]
          e = str_li[1]
          node_s = self.get_p_node(s).label.formula.ast # type: ignore
          node_e = self.get_p_node(e).label.formula.ast # type: ignore
          conn = Token("imp")
          node = Node(conn, [node_s, node_e])
          premise_nodes.append(node)
      return conclusion.verified_by(rule_inf, premise_nodes)

  def verified(self, conc: str) -> bool:
    """ Test if the conclusion with line number conc is verified
        by its annotation.  conc of the form 's-e' is not accepted.
        self must be the root of the whole proof.
    """
    assert bFmla(conc), f"verified_by(): conclusion='{conc}' is not accepted"
    assert self.index_dict is not None, "verified_by(): index_dict is None"
    p_node = self.get_p_node(conc) # ProofNode type
    ann = p_node.label.ann
    if not isinstance(ann, Ann):
      return False
    else:
      return self.verified_by(conc, ann.rule, ann.premise) # type: ignore

  def verified_all(self) -> bool:
    """ Test if all conclusions are verified by their annotations.
        self must be the root of the whole proof. """
    assert self.index_dict is not None, "verified_by(): index_dict is None"
    for node_code in self.index_dict:
      if bSubproof(node_code):
        continue
      p_node = self.get_p_node(node_code) # ProofNode type
      if (label := p_node.label).type == 'formula':
        if label.is_hyp:
          continue
        if self.verified(node_code):
          continue
        else:
          return False
    return True
```

Here is an example run. This time, we have replaced the line `5. B or C .or intro 4` with `5. B or C .and intro 4` to intentionally make the proof invalid. Let's observe what happens in this scenario.

```
proof.verified_by("4", RuleInfer.IMP_ELIM, ["1", "3"])
```

```
True
```

```
for node_code in proof.index_dict: # type: ignore
  if bSubproof(node_code):
    continue
  p_node = proof.get_p_node(node_code)
  if (label := p_node.label).type == 'formula':
    if label.is_hyp:
      print(f"{node_code}: {label.formula} is hyp")
    else:
      if proof.verified(node_code):
        print(f"{node_code}: {label.formula} is verified")
      else:
        print(f"{node_code}: {label.formula} is *not* verified")
```

```
1: A imp B is hyp
2: A or C is hyp
3: A is hyp
4: B is verified
5: B or C is *not* verified
6: C is hyp
7: B or C is verified
8: B or C is verified
9: A or C imp B or C is verified
```

```
proof.verified_all()
```

```
False
```

4.7 Things to do

We have implemented a Fitch proof parser and a verifier for propositional logic. However, there are still many things to do. Here is a list of some of them.

- LaTeX output
- Graphical editor for Fitch proofs (VS Code extension maybe?)
- Save and retrieve Fitch proofs in database
- Web based Fitch proof editor and verifier
- Meta-theorems

Meta-theorems can be a valuable tool for simplifying the process of writing and verifying Fitch proofs. However, in my experience, these are not necessarily required for propositional logic. For most logical consequences in propositional logic, it's feasible to construct Fitch proofs without significant difficulty.

On the other hand, when it comes to predicate logic, meta-theorems become essential. Without them, our ability to write and verify Fitch proofs is severely limited. Here is a list of some meta-theorems that can be particularly useful in the context of predicate logic:

- Replacement of subformulas with equivalent formulas: Allows for the substitution of a subformula with an equivalent formula, preserving the validity of the proof.
- Replacement of subterms with equivalent terms: Permits the substitution of a subterm with an equivalent term, maintaining the integrity of the proof.
- Introducing meta-variables for formulas and substituting them with arbitrary formulas
- Introducing meta-variables for terms and substituting them with arbitrary terms
- Declaring new symbols with various attributes
- Introducing new symbols, such as abstract terms and predicates, once some necessary equivalence has been proved, and using them in proofs to make conservative extensions of the language

- Saving a proved logical consequence as a lemma in a database and utilizing it later

These meta-theorems play a pivotal role in extending the capabilities of Fitch proofs, particularly in the realm of predicate logic.

5 Fitch Proof, 1st-order logic

5.1 Semantics of 1st-order logic

5.2 Tautological Consequence in 1st-order logic

There are 14 rules of inference available for the propositional logic in our implementation of the Fitch proof system. In constructing a first-order Fitch proof, we combine all these rules into just one single rule, namely the *tautological consequence rule*.

In order to implement this rule of inference, we should be able to deal with truth tables. 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 each of α, β and φ, so that we can if

$$\mathrm{Mod}(\alpha) \cap \mathrm{Mod}(\beta) \subseteq \mathrm{Mod}(\varphi).$$

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 approach to examine the Abstract Syntax Tree (AST) of a given formula. The following code demonstrates this process in detail.

We prepared a Python module called `truth_table.py` which consists of 2 classes and 3 utility functions. The `Formula` class object has only one property, namely `ast` of `Node` type, and has lots of methods for building truth tables and other syntactic operations.

An important difference between the `Formula` class and the `Node` class is in their constructor methods. The constructor of the `Formula` class takes a string or a `Node` object as input and returns a `Formula` object. The constructor of the `Node` class takes a `Token` object and a list of `Node` objects as input and returns a `Node` object.

Please focus your attention on the dunder method `__eq__()` present in both the `Formula` and `Node` classes.

```python
from typing import List, Tuple

from first_order_logic_parse import *

class Formula:
  def __init__(self, input: str | Node):
    try:
      if isinstance(input, str):
        self.ast = parse_text(input)
      else:
        self.ast = input
      if self.ast.type == 'term':
        str_infix = (str if isinstance(input, str)
          else input.build_infix('text'))
        print(f"Input \"{str_infix}\" is a term, not a formula.")
    except ValueError as e:
      print(f"ValueError: e")
    except SyntaxError as e:
      print(f"SyntaxError: e")

  def __str__(self):
    return self.ast.build_infix('text')

  def __eq__(self, other) -> bool:
    return self.ast == other.ast

  ..

  def get_prime_subformulas(self) -> set:
    # This method is used in TruthTable.get_prime_subformulas().
    MAX_N = 8 # maximum number of prime subformulas
    prime_subs = set()
    tree = self.ast
    if tree.token.token_type in Token.NON_PRIME_ROOTS:
      # connectives of positive arity
      for kid in tree.children:
        prime_subs |= Formula(kid).get_prime_subformulas()
    elif tree.token.token_type != 'conn_0ary':
      prime_subs |= { self.ast.build_infix('text') }

    assert len(prime_subs) <= MAX_N, \
      f"Error: number of prime subformulas exceeds {MAX_N}."
    return prime_subs

  ..

FList = List[Formula]

class TruthTable:
  def  __init__(self, f_list: FList):
    self.f_list = f_list

  def get_prime_subformulas(self) -> set:
    MAX_N = 8 # maximum number of prime subformulas
    prime_subs = set()
    for f in self.f_list:
      prime_subs |= f.get_prime_subformulas()

    assert len(prime_subs) <= MAX_N, \
      f"Error: number of prime subformulas exceeds {MAX_N}."
    return prime_subs

  def show_truth_table(self, opt: str='text') -> None:
    # opt ::== 'text' | 'latex'

# Utility functions
  ..

```

The TruthTable class has only one property of type List[Formula] and lots of methods for building truth tables. The show_truth_table() method is the most important one, which is used to display the truth table of a given list of formulas. The get_prime_subformulas() method is used to get the set of prime subformulas of a given set of formulas. It utilizes a method in the class Formula with the same name.

Note that we set an upper bound 8 for the number of prime subformulas. We decided to use this rather small value of 8 because the number of propositional variables in tautological consequence reasoning for a human mind cannot be too big. Normally we use two or three, or at most four prop. variables in this circumstances, I guess.

Shown below are some codes in the module first_order_logic_parse.py relevant to the construction of truth tables.

```
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
  NON_PRIME_ROOTS = ("conn_1ary", "conn_2ary", "conn_arrow")
  PRIME_ROOTS = ("pred_pre", "pred_in", "equality", "prop_letter",
    "quantifier", "conn_0ary")

  def __init__(self, value):
    self.value = value # a string
    self.token_type = None
    self.arity = None
    self.precedence = None
    ..

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')
    self.index = -1 # 0,1,2,.. for truth tree, prime nodes
    self.bValue = -1 # 0,1 for truth tree, all nodes
    self.level = -1 # 0,1,2,.. for truth tree, all nodes
    self.alt_str = '' # 'P_1', 'P_2', .. for truth tree, prime nodes

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

  def __eq__(self, other):
    infix_self = self.build_infix('text')
    infix_other = other.build_infix('text')
    return infix_self == infix_other

  ..
```

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]

Truth tree is a subtree of the AST of a formula obtained by removing specific nodes. The root node belongs to the truth tree. If a node is labeled with a connective with positive arity, then each of its children belongs to the truth tree. Therefore, every node within the truth tree is a prime subformula of the formula or a subformula built from earlier nodes within the truth tree and connectives. In brief, a truth tree is a tree of subformulas where each node is to be assigned a truth value.

It's important to note that not all prime subformulas are included in the truth tree. For instance, consider the formula $\forall x\, P(x) \to Q$. This formula has four subformulas, among which $\forall x\, P(x)$ and Q are prime subformulas and present in the truth tree. The entire formula, representing the root node, is in the truth tree, but the subformula $P(x)$ is a prime subformula that does not belong to the truth tree.

Nodes labeled with `bot`, which represents the constant False, are prime subformulas but are not included in the return value of the `get_prime_subformulas()` method due to technical reasons. We refer to the elements in the return value of this method as *prime nodes*.

Prime nodes within the truth tree should be assigned *indices*. We will explain the role of indices by an example. If $[A, \forall x\, P(x), Q(x, y)]$ is the sorted list of all prime nodes within a set of formulas, then any node in the truth tree of a formula labeled A is assigned index 0. Similarly, nodes labeled with $\forall x\, P(x)$ are assigned index 1, and $Q(x, y)$ is assigned index 2. Non-prime nodes within the truth tree are not assigned indices.

When a specific truth-value assignment $\bar{x}$ is provided, each node in the proof tree is assigned a truth value in the following manner. Prime nodes are assigned values based on $\bar{x}$. Non-prime nodes are assigned values according to the truth functions of the corresponding connectives.

These concepts provide the foundation for implementing the inference rule of tautological consequence. However, for the purpose of creating a well-organized truth table, it becomes necessary to assign *labels* to the prime nodes. Without labels, the header of the truth table might become excessively wide and unwieldy. For propositional letters, their own tokens are used as labels. For other types of prime nodes, labels such as P_1, P_2, and so forth are used.

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

Additional implementation details for the inference rule of tautological consequence can be found in `truth_table.py`, and examples of truth tables are provided in `truth_table.ipynb`.

We can execute the following code in `truth_table.ipynb` and see the truth table of the 3 formulas.

```
from truth_table import *

# Initialize 3 1st-order formulas.
f1 = Formula('A imp forall x B1(x)')
f2 = Formula('forall x B1(x) or D2(f(a), b+1)')
f3 = Formula('(D2(f(a), b+1) imp not forall x B1(x)) imp D2(f(a), b+1)')
# Initialize a truth table object.
t_table = TruthTable([f1, f2, f3])
t_table.show_truth_table('latex') # latex mode
```

The `show_truth_table()` method requires some explanation. In essence, it utilizes the following methods to establish the scaffold of the truth tree.

1. `get_prime_subformulas()`
2. `label_prime_subs(prime_subs_li)`

Then, for each truth value assignment, it uses the following methods to fill in the truth tree, and then prepare and output a row of the truth table.

1. `get_truth_tree(tVal_assign)`
2. `get_bValues()`
3. `print_truth_table_row()`

The output is shown below.

```
Truth table for the following 3 formulas.
```

$$A \to \forall x\, B(x)$$
$$\forall x\, B(x) \vee D(f(a), b+1)$$
$$(D(f(a), b+1) \to \neg\forall x\, B(x)) \to D(f(a), b+1)$$

```
Prime subformulas and their alternate labels.
```

$$[A, \forall x\, B(x), D(f(a), b+1)]$$
$$[A, P_1, P_2]$$

A	P_1	P_2	A	$\to$	P_1	P_1	$\vee$	P_2	(	P_2	$\to$	$\neg$	P_1	)	$\to$	P_2
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

Actually, the output in Jupyter Notebook is the LaTeX source code of the truth table. When we compile the code, we get the nice looking truth table shown above.

If we choose the text mode, then the output is as follows.

```
t_table.show_truth_table()
```

```
prime subformulas = ['A', 'forall x B1(x)', 'D2(f(a), b + 1)']
alt prop. letters = ['A', 'P_1', 'P_2']

A P1 P2 : A imp P1, P1 or P2, (P2 imp not P1) imp P2
----------------------------------------------------
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
```

6
Extending the Fitch Proof System

Index