# PLMan (Propositional LogicMan) User's Manual — version 2.5.1

Takayuki Hoshi

*E-mail address*: `th279@cornell.edu`

# Contents

CHAPTER 1

# Introduction to PLMan

**PLMan**, or Propositional LogicMan, is a user-friendly and powerful propositional logic (sometimes called sentential logic or propositional calculus) sentence shell/interpreter written in Java. Many important propositional systems has been implemented on PLMan as will be listed shortly.

With PLMan, one can

- Evaluate formulas under different logical systems
- Compare each logic and decide the best logic that models a certain environment of your consideration.
- Tell if the given formula is grammatically correct (well-formed) or not
- Translate a given *well-formed formula* (or *wff*) into English
- Write formulas in a visually pleasant manner with standard logical connectives and Greek characters (both uppercase and lowercase) in Unicode
- Display truth tables for a given wff
- Determine the satisfiability, or validity, of a given wff
- Tell if a set of wffs entails a given wff or not
- Assign and refer to the description of each propositional atom
- Build his/her own knowledge system (or even axiomatic systems)

A list of propositional logics currently available so far is shown below:

- Classical two-valued propositional logic (CPL)
- Kleene three-valued logic ($K_3$)
- Łukasiewicz three-valued logic ($Ł_3$)
- Logic of paradox (LP)
- R-Mingle three valued logic ($RM_3$)
- A System of Fuzzy logic (Ł)
- Łukasiewicz' continuum-valued logic ($Ł_{\aleph}$)
- First degree entailment as a four-valued logic (FDE)

## 1. Installation

### 1.1. Linux or Unix.

(1) Unzip the downloaded PLMan distribution using some uncompression utility.

(2) There are two environment variables to be set in order to obtain PLMan's full functionality: "`PLMAN_PATH`" and "`PLMAN_SCRIPT_PATH`".

`PLMAN_PATH` is the directory under which PLMan can be found. If, for example, you unzipped PLMan archive under "`/home/user/app/`" directory, then "`/home/user/app/plman`" would be appropriate value for it.

PLMAN_SCRIPT_PATH is the directory under which PLMan script files reside. This path is used by ":inputFile" command in interactive mode (c.f. ":inputFile" subsection under chapter "Command"). If this environment variable is set, PLMan will look for the file specified as an argument to :inputFile command under and only under the directory.

(3) Once these two environment variables are set, go under "$PLMAN_PATH/bin" and copy "plman" directory to one of the directories where your OS looks for commands to execute (for example, "/home/user/bin", "/usr/local/bin", etc.).
(4) Finally, in order to execute PLMan, type:

```
plman
```

at your favorite command shell.

Another way to execute plman is simply to go under the plman path and type:

```
java -classpath lib/plman.jar:lib/ant.jar \
    PropositionalLogicParser
```

or

```
java -classpath lib/plman.jar:lib/ant.jar \
    PropositionalLogicParser <FILENAME>
```

**1.2. Windows.**

(1) Unzip the downloaded PLMan distribution using some uncompression utility.
(2) There are two environment variables to be set in order to obtain PLMan's full functionality: "PLMAN_PATH" and "PLMAN_SCRIPT_PATH".

PLMAN_PATH is the address of the folder under which plman can be found. If, for example, you unzipped PLMan archive under "C:\" folder, then "C:\plman" would be value that has to be set; similarly, if you unzipped PLMan archive under "C:\Documents and Settings\Administrator" but moved newly created "plman" folder to "C:\Program Files" folder, then "C:\Program Files\plman" is the correct value to be set to PLMAN_PATH.

PLMAN_SCRIPT_PATH is the address of the folder under which plman script files reside. This path is used by ":inputFile" command in interactive mode. If this environment variable is set, PLMan will look for the file specified by an argument under and only under the folder.

**Note**: If you who don't know how to set these values, see section "Setting Environment Variables."

(3) Go under %PLMAN_PATH and then under 'bin\' folder. Copy 'plman.bat' inside the folder to either 'C:\WINNT' or 'C:\Windows' folder (the existence of either of which depends on a type of Windows used by the user).
(4) Restart your computer.
(5) Open a command prompt and type:

```
plman
```

Another way to execute plman is simply to go under the plman path and type:

```
java -classpath lib/plman.jar:lib/ant.jar \
    PropositionalLogicParser
```

or

```
java -classpath lib/plman.jar:lib/ant.jar \
    PropositionalLogicParser <FILENAME>
```

**1.3. Setting Environment Variables.** Approaches to setting values to environment variables differ from one environment (and OS) to another. In this section, I will present several ways of doing so under different environments. Here, to be specific, let us take 'PLMAN_SCRIPT_PATH' as our particular environment variable to be set.

**Bash shell**. On the shell, type something like

```
export PLMAN_SCRIPT_PATH="/path/to/PLMan/script/files"
```

Example:

```
export PLMAN_SCRIPT_PATH="/home/user_dir/plman/script"
```

**Windows 9x**. Run "cmd.exe" (command prompt). At the prompt, type something like:

```
C:\>set PLMAN_SCRIPT_PATH=C:\path\to\PLMan\script\files
```

Example:

```
C:\>set PLMAN_SCRIPT_PATH=C:\plman\script
```

You could add your exact input line to `AUTOEXEC.BAT` if you want the variable to be set whenever you start your OS.

**Windows 2000 and XP**. You can follow either the same procedure for Windows 9x or the following instruction: Open the System icon in the the Control Panel. Under the Advanced tab, there is a button labeled "Environment Variables". Click on it and then add `PLMAN_SCRIPT_PATH` and its correct value to the system. Finally, reboot your computer. The procedure should be similar for XP users.

**Eshell (Emacs Shell), csh, and tcsh**. On the shell prompt, type:

```
setenv PLMAN_SCRIPT_PATH "/path/to/PLMan/script/files"
```

## 2. PLMan Syntax

Unlike many existing languages that add unnecessary complication to syntax, PLMan's syntax is both simple and intuitive, so that the users, I suppose, would have no hassle with grammatical expressions on PLMan.

PLMan's interpretation of a program is line-oriented, i.e., it evaluates and executes code line by line. Any input line to PLMan may be either a COMMAND expression, or one or more sequence of STATEMENT expressions, each of which can be separated by a semicolon ';'. Thus, a PLMan input would be either the form

```
<COMMAND_EXPR>
```

or of the form

```
<STATEMENT1> ; <STATEMENT2> ; ... ; <STATEMENTn>
```

With several exceptions, every PLMan command starts with a colon ':', which gets immediately followed by one or more English alphabetical characters or digits. Hence, each command has a line syntax of its own. The command ':exit' for example takes no argument, whereas ':setSystem' command takes one argument: the name of a propositional logic. (For detailed explanations and demonstrations of PLMan commands, see chapter "Commands".)

```
plman[i]> :setSystem "R3"
plman[i]> :exit        -- ':exit' exits PLMan
```

A STATEMENT can be either an ASSIGNMENT expression or a FORMULA expression. An ASSIGNMENT expression may take on one of the following forms:

(1) $P = V$
(2) $P_1 = V_1, P_2 = V_2, ...$
(3) $P =$ FORMULA
(4) $P_1 =$ FORMULA$_1$, $P_2 =$ FORMULA$_2$, ...
(5) $P$ : "Its description"

where (1) assigns a value $V$ to a propositional atom $P$; (2) assigns a value $V_i$ to each $P_i$ sequentially in order; (3) assigns a value which is obtained by interpreting FORMULA to a propositional atom $P$; (4) assigns a value which is obtained by interpreting FORMULA$_i$ to a propositional atom $P_i$; and (5) gives a description (or its English semantics) to the propositional atom $P$.

Example:

```
    plman[i]> x = 0
    plman[i]> y = 1, z = 0
    plman[i]> (x | y ) => z
    out[i]> False       [ 0 ; False ]
    plman[i]> G : "Godel was a great logician."
    plman[i]> GE : "Godel was a friend of Einstein."   ;   GA = 1
    plman[i]> :: G & GE         -- '::' is an abbreviation
of command ':translate'
    "Godel was a great logician." and "Godel was a friend of Einstein."
```

Precise specifications of (well-formed) formulas and FORMULA expressions will be covered in the next chapter, but PLMan employs the standard syntax for such basic constructs.

Example:

```
plman[5]> a = 0, b = 1, c = 0, d = 1, e = 0
plman[6]> (a & b) | c  -- is a formula
out[6]> False        [ 0 ; False ]
plman[7]> ((a) & b) => c | ~d <=> e  -- is a formula
```

```
    out[7]> True        [ 1 ; True ]
    plman[8]> -- whereas,

    plman[17]>  &~ c    -- is not.
    SYNTAX ERROR: The input formula is ill-formed (i.e., doesn't
follow the syntax).  Input ignored...
```

Comment lines start with either “--” or “//”.

Example:

```
    plman[i]> -- your comments here..
    plman[i]> a & b        -- your comments here..
    plman[i]> a & b        // your comments here..
```

**2.1. Modes.** There are two distinct modes in PLMan: the ‘interactive mode’ and the ‘file-input mode’.

***Interactive mode*** is the one in which users can directly communicate with PLMan line by line through the PLMan shell. Any user input will be evaluated on the fly, and the output will be shown immediately.

***File-input mode*** is the one enacted when a user specifies a plman script file as the first argument to “plman” command. In this mode, PLMan will read the file contents line by line until PLMan gets to the EOF (end of file) character. Switching from file-input mode to interactive mode can be conveniently done by placing the exact line

```
:interactiveMode
```

somewhere in the input file. As soon as PLMan parses a line that looks like the above, PLMan switches into the interactive mode, ignoring the rest of the file content.

CHAPTER 2

# The Language of Propositional Logics under PLMan

In this chapter, I will explain the language of propositional logics currently available on PLMan. Overall, there are eight propositional logics so far, but all systems use the same alphabet and the same definition of well-formed formulas (hence “language” in singular form). (In this and any other subsequent chapters, I will not necessarily give proofs to important facts that may be stated in the context (since this is just a user’s manual, not a text); but proofs may be found in any rigorous introductory textbook on mathematical logic.) After the formal specification of well-formed formulas, the informal, but standard, conventions for wffs will be listed.

## 1. Well-Formed Formulas

The ***Alphabet*** $\mathcal{A}$ of the language of propositional logics under PLMan is a set consisting of the following strings:

(1) <TRUE> : “1” , “T” , “TRUE”
(2) <FALSE> : “0”, “F”, “FALSE”
(3) <NOT> : “$\neg$”, “~”, “NOT”
(4) <AND> : “$\wedge$” , “&” , “AND”
(5) <OR> : “$\vee$” , “|” , “OR”
(6) <MATERIAL_CONDITIONAL> : “$\Rightarrow$” , “=>” , “IMPLIES”
(7) <IFF> : “$\Leftrightarrow$” , “<=>” , “IFF”
(8) <PARENTHESES> : “(” , “)”
(9) <SPACE> : “ ”
(10) <NON_NEGATIVE_REAL>
(11) <PROPOSITIONAL_ATOM>

where <PROPOSITIONAL_ATOM> is defined as a set of the strings consisting of a letter, including every Greek letter, followed by zero or more succession of either a letter, a digit, or a ‘-’, but excluding any string defined in (1)-(10). In regular expression, it would be expressed as [A-Za-z_[:greek:]][A-Za-z0-9_-[:greek:]]* (again, not including an element in $(1) \cup (2) \cup ... \cup (10)$ ). <NON_NEGATIVE_REAL> is a set of real numbers that are not negative (3, 1.302, etc.). In any logical system, <IFF> may be omitted (actually, if one would like to oversimplify the language, he could also cut off <AND> and <OR> as well), but we nevertheless include the symbol for our convenience. A complete list of Greek letters recognized by PLMan follows:

[:greek:] : “$\alpha$” , “$\beta$” , “$\gamma$” , “$\delta$” , “$\epsilon$” , “$\zeta$” , “$\eta$”, “$\theta$” , “$\iota$” , “$\kappa$” , “$\lambda$” , “$\mu$” , “$\nu$” , “$\xi$” , “$o$” , “$\pi$”, “$\rho$” , “$\sigma$” , “$\tau$” , “$\upsilon$” , “$\phi$” , “$\chi$” , “$\psi$” , “$\omega$” , “$\Gamma$”, “$\Delta$” , “$\Theta$”

, “Λ” , “Ξ” , “Π” , “Σ” , “Υ” , “Φ” , “Ψ”, “Ω”

(in Unicode: ”0x03B1” to ”0x03C1”, ”0x03C3” to ”0x03C9”, ”0x0393”, ”0x0394”, ”0x0398”, ”0x039B”, ”0x039E”, ”0x03A0”, ”0x03A3”, ”0x03A5”, ”0x03A6”, ”0x03A8”, and ”0x03A9”, respectively.)

Semantically speaking, for each of the sets in (1) to (7), there is only one “real” or correct string within the set; any other string in the set is merely an abbreviation for the real one. For example, in `<TRUE>` we see the strings “1”, “T” and “TRUE”; but the real representation within the set is actually “1”, and the rest — “T” , “TRUE” — is an abbreviated (or human-understandable) representation of “1”. The table of the “real” and its abbreviations is shown below.

| Set | Real | Abbreviations |
|---|---|---|
| `<TRUE>` | “1” | “T” , “TRUE” |
| `<FALSE>` | “0” | “F”, “FALSE” |
| `<NOT>` | “¬” | “~”, “NOT” |
| `<AND>` | “∧” | “&” , “AND” |
| `<OR>` | “∨” | “\|” , “OR” |
| `<MATERIAL_CONDITIONAL>` | “⇒” | “=>” , “IMPLIES” |
| `<IFF>` | “⇔” | “<=>” , “IFF” |

Putting them all together, we obtain an unambiguous definition of the alphabet for the language to be

$\mathcal{A}$ := `<TRUE>` ∪ `<FALSE>` ∪ `<NOT>` ∪ `<AND>` ∪ `<OR>` ∪ `<MATERIAL_CONDITIONAL>` ∪ `<IFF>` ∪ `<SPACE>` ∪ `<NON_NEGATIVE_REAL>` ∪ `<PROPOSITIONAL_ATOM>`

Now, define $\mathcal{A}^*$ to be the set of finite strings over $\mathcal{A}$, so that “`p => (q&r)`’ $\in \mathcal{A}^*$, “$\zeta\eta 1(T)) \Rightarrow)$” $\in \mathcal{A}^*$, “I love PLMan”$\in \mathcal{A}^*$, and so forth.

We now give a formal definition of ***well-formed formulas*** (wffs).

Definition. Let $\alpha, \beta \in \mathcal{A}^*$. (Well-formed) formulas $\mathcal{W} \subseteq \mathcal{A}^*$ of propositional logics under PLMan is the smallest subset of $\mathcal{A}^*$ recursively satisfying the following conditions[1]:

**1:** Any element in `<PROPOSITIONAL_ATOM>` ∪ `<NON_NEGATIVE_REAL>` ∪ `<TRUE>` ∪ `<FALSE>` is an element of $\mathcal{W}$.

**2:** If $\alpha \in \mathcal{W}$, then $(\neg\alpha) \in \mathcal{W}$

**3:** If $\alpha, \beta \in \mathcal{W}$, then $(\alpha \wedge \beta) \in \mathcal{W}$, $(\alpha \vee \beta) \in \mathcal{W}$, $(\alpha \Rightarrow \beta) \in \mathcal{W}$, and $(\alpha \Leftrightarrow \beta) \in \mathcal{W}$, respectively.

We call any formula obtained in (1) an *atomic formula* and any other formulas *composite formulas*. As an important fact, each element constructed by 1-3 are unique (Unique Readability).

[1]Another neat definition of wffs would be to first fix $\mathcal{W}_0 = \langle \text{PROPOSITIONAL\_ATOM} \rangle \cup \langle \text{NON\_NEGATIVE\_REAL} \rangle \cup \langle \text{TRUE} \rangle \cup \langle \text{FALSE} \rangle$ and define $\mathcal{W}_{n+1} := \mathcal{W}_n \cup \{(\neg\alpha) \mid \alpha \in \mathcal{W}_n\} \cup \{(\alpha \wedge \beta), (\alpha \vee \beta), (\alpha \Rightarrow \beta), (\alpha \Leftrightarrow \beta) \mid \alpha, \beta \in \mathcal{W}_n\}$. The resulting set obtained by the union of $\mathcal{W}_0$ to $\mathcal{W}_\infty$ — i.e., $\cup_{n=0}^{\infty} \mathcal{W}_n$ — is actually the same as $\mathcal{W}$, whose proof again can be seen in any rigorous introductory book on mathematical logic.

## 2. Informal Conventions

Formally in the language of propositional logic, an expression, say, $((\neg p) \vee (q \Rightarrow ((\neg r) \wedge s)))$ is a well-formed formula. However, since it is a bit cumbersome to keep the formal syntax and write every formula in this manner, it is common to introduce some informal conventions which is supposed to make it easier for humans to read and write a formula without destroying its semantics. In our case, conventions are as follows:

Let $p$, $q$, $r$, $s$ be propositional atoms; $\alpha$, $\beta$, and $\gamma$ be wffs. Then,

(1) We may drop the outermost parentheses in a formula. For example, $(p \vee q)$ can also be written as $p \vee q$.
(2) We may let the negation symbol $\neg$ take precedence over any other connectives when parentheses are missing, and the symbol binds as little as possible. We also let other propositional connective symbols — $\wedge, \vee, \Rightarrow$ and $\Leftrightarrow$ — follow the same scheme according to the order of precedence: 1. $\neg$, 2. $\wedge$ and $\vee$ (same), and 3. $\Rightarrow$ and $\Leftrightarrow$ (same). Thus, $p \Leftrightarrow \neg q$ is now a shorthand for $(p \Leftrightarrow (\neg q))$.
(3) We group repetitions of propositional connective symbols with the same precedence to the right when parentheses are missing. For example, $\alpha \vee \beta \vee \gamma$ is a shorthand for $\alpha \vee (\beta \vee \gamma)$. Likewise, $\alpha \Rightarrow \beta \Leftrightarrow \gamma$ will be interpreted as $\alpha \Rightarrow (\beta \Leftrightarrow \gamma)$. (Note that some authors let $\Rightarrow$ take precedence over $\Leftrightarrow$, yielding $(\alpha \Rightarrow \beta) \Leftrightarrow \gamma$. But since this can be confusing sometimes, I decided not to do so.) The same applies to $\wedge$ and $\vee$.

As an example, following our conventions, the formula $((\neg p) \vee (q \Rightarrow ((\neg r) \wedge s)))$ can simply be: $\neg p \vee (q \Rightarrow \neg r \wedge s)$.

A FORMULA expression in PLMan then is any well-formed formula, with or without the above informal conventions being applied.

CHAPTER 3

# Systems of Logic

In this chapter, we will first review the important concepts of propositional logic and then move on to the general overviews of each logical systems implemented in PLMan.

## 1. General Overview

In many of propositional logics whose interpretation is a function, the system can be defined as a structure $\langle \mathrm{TV}, D, \mathcal{T} \rangle$ where

- TV is a set of truth values
- $D$ is a set of truth values for which the system yields True (a designated set).
- $\mathcal{T} = \{\ \tau_c \mid c \in \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}\}$ is a set of truth functions for each connective available to the system.

Let PA be an abbreviation for <PROPOSITIONAL_ATOM>. A ***truth assignment*** $v$ is a function $v : \mathrm{PA} \to \mathrm{TV}$ which, given a propositional atom, assigns a truth value to it. (We will use PA to represent a set of propositional atoms henceforth.)

Given a truth assignment $v$, ***the interpretation function*** $\bar{v} : \mathcal{W} \to \mathrm{TV}$, which assigns a truth value to a well-formed formula, is defined recursively as follows:

- $\bar{v}(\mathrm{tv}) = 1$ if $\mathrm{tv} \in \mathrm{TV}$ and $\mathrm{tv} \in D$
- $\bar{v}(\mathrm{tv}) = 0$ if $\mathrm{tv} \in \mathrm{TV}$ and $\mathrm{tv} \notin D$
- $\bar{v}(\alpha) = v(\alpha)$ if $\alpha \in \mathrm{PA}$
- $\bar{v}(\neg\alpha) = \tau_{\neg}(\bar{v}(\alpha))$
- $\bar{v}(\alpha \Box \beta) = \tau_{\Box}(\bar{v}(\alpha), \bar{v}(\beta))$ where $\Box \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, $\alpha, \beta \in \mathcal{W}$

where $\tau_{\neg} : \mathrm{TV} \to \mathrm{TV}$ and $\tau_{\Box} : \mathrm{TV} \times \mathrm{TV} \to \mathrm{TV}$ are semantic functions for each connective defined independently in each system. Suppose we are given a well-formed formula $\alpha$ and a truth assignment $v$ on the basis of which $\bar{v}$ yields its return values. Then we call $\bar{v}(\alpha)$ ***the interpretation of*** $\alpha$ ***with respect to*** $v$.

In PLMan, the initial truth assignment $v$ assigns False to all (undeclared) propositional atoms. Thus, for example, the interpretation of an input formula "a | b" with respect to initial assignment $v$, will return false as illustrated below:

```
[ Note that the underlying system here is "CPL" ]

plman[i]> a | b
Propositional atom ‘a’ is undeclared. Returning False ( value: 0 ) instead.
Propositional atom ‘b’ is undeclared. Returning False ( value: 0 ) instead.
```

```
out[i]> False        [ 0 ; False ]
```

This initial assignment $v$ may be replaced with a new assignment if the user changes the behavior of $v$ (i.e., assigns some specific values to some propositional atoms).

This initial behavior of $v$ can be rejected by letting it return a different value for certain propositional atoms, thus creating a new truth assignment with which the interpretation function $\bar{v}$ evaluates input formulas:

```
[ Note that the underlying logic here is "CPL" ]

plman[i]> a = 0, b = 1
plman[i]> a | b
out[i]> True         [ 1 ; True ]
```

One can compute $\bar{v}$ for any possible truth assignment $v$. In particular, given distinct propositional atoms that appear in a wff $\alpha$, we can simply list all possible combinations of truth values for the propositions obtainable from truth assignments; the list of all combinations, if each of which is paired with an answer computed by $\bar{v}$ for the given truth values, is called the ***truth table*** for $\alpha$.

```
[ Note that the underlying system here is "CPL" ]

plman[i]> :table (a & ~b) => b

a   b   ( a & ~b ) => b
1   1   1
1   0   0
0   1   1
0   0   1
```

Say a truth assignment $v$ ***satisfies*** a formula $\alpha$ if and only if the the returned value of the interpretation of $\alpha$ with respect to $v$ is an element of $D$ — that is, $\bar{v}(\alpha) \in D$. Here, we say that a formula $\alpha$ is ***satisfiable*** if there exists some truth assignment $v$ that satisfies $\alpha$; similarly, $\alpha$ is called ***valid***, or a ***tautology***, if all possible truth assignments satisfy it. Finally, given a set $\Sigma$ of formulas (possibly empty) and a formula $\alpha$, fix a set $S$ of all truth assignments that satisfy every formula in $\Sigma$. If every satisfying truth assignment $s$ in $S$ also satisfies $\alpha$ as well, then we say that $\Sigma$ ***entails*** $\alpha$ (or $\Sigma$ ***tautologically implies*** $\alpha$); symbolically, we express this situation by '$\Sigma \models \alpha$'.

In PLMan, we have simple commands and expression for checking each of the above status:

```
[ Note that the underlying system here is "CPL" ]

plman[i]> :satisfiable x & y
out[i]> True         [ 1 ; True ]
```

```
plman[i]> :satisfiable ~(x => y) & ~( x & ~y )
out[i]> False        [ 0 ; False ]
plman[i]> :valid (a | b) | ~(a | b)
out[i]> True         [ 1 ; True ]
plman[i]> { a , ~ a } |= everything
out[i]> True         [ 1 ; True ]
plman[i]> { a | b , a <=> ~ b } |=  (a | b) & ~( a & b)
out[i]> True         [ 1 ; True ]
```

As stated earlier, PLMan currently provides 8 systems of propositional logic:

- Classical two-valued propositional logic (CPL)
- Kleene three-valued logic ($K_3$)
- Łukasiewicz three-valued logic ($Ł_3$)
- Logic of paradox (LP)
- R-Mingle three valued logic ($RM_3$)
- A System of Fuzzy logic (Ł)
- Łukasiewicz' continuum-valued logic ($Ł_{\aleph}$)
- First degree entailment as a four-valued logic (FDE)

The general descriptions of each system will occupy the rest of the chapter.

## 2. Classical two-valued propositional logic (CPL)

Classical two-valued propositional logic (CPL) is certainly one of the most important systems in propositional logic: it is indeed embedded in classical First Order Logic (FOL), the one employed as a base logic system in ZFC.

CPL is the structure $\langle \mathrm{TV}, D, \mathcal{T} \rangle = \langle \{0,1\}, \{1\}, \{\ \tau_c \mid c \in \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}\} \rangle$. The definitions of each function for connectives are given below.

$\tau_\neg$ is a unary function, i.e., a function whose arity is one, defined by a truth table as follows:

| $x$ | $\tau_\neg$ |
|---|---|
| 1 | 0 |
| 0 | 1 |

$\tau_\wedge$ is a binary function, i.e., a function with arity two, defined accordingly:

| $x$ | $y$ | $\tau_\wedge(x,y)$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\tau_\vee$:

| $x$ | $y$ | $\tau_\vee(x,y)$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

$\tau_\Rightarrow$:

| $x$ | $y$ | $\tau_\Rightarrow(x,y)$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

$\tau_\Leftrightarrow$:

| $x$ | $y$ | $\tau_\Leftrightarrow(x,y)$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

## 3. Philosophical Objections to CPL

Logical strictness and limitation sacrifices a huge partition of what actually *is* and leaves no room for what's plausible, uncertain, or unlikely, thus drawing a dogmatic boundary to our thought, namely, to "what can be said." CPL, for instance, works well in the domain of Physics because our Universe, when seen non-microscopically, looks deterministic (not all deterministic however[1]), which in some respect explains why many formulations in mathematics directly apply to Nature without modification. But is it really the case that every object has a truth value of either True or False? — Or even, that truth and falsity of every objective proposition is knowable within a totality of its own? It turned out that the latter, when posed to the domain of mathematics, is false: Gödel's first incompleteness theorem tells us that any consistent and decidable axiomatic system which extends the axioms of Peano Arithmetic has at least one unprovable sentence with the axioms in the system.

The former is also objectionable[2]; for, if we are to accept this to be the case, then it is up to each human mind to decide the truth and falsity of (possibly) unknowable propositions. For example, ZFC takes for granted without much justification the existence of a set that contains infinitely many sets, which is disguised under the name of "the Axiom of Infinity." But the problem is that the truth and falsity of such proposition, if the justification isn't satisfactory, pretty much depends on the observers (I for one do not believe this to be the case unlike ZFC). True, it is useful to postulate this axiom so as to not worry about the finiteness of the objects of their consideration. But this observation indicates that mathematics somehow takes a pragmatic stand toward the understanding of the world and is not completely objective, that there in fact is some relativity within it.[3]

CPL (and many other propositional logics as well) is also notorious for ignoring the "relevance" between propositions in a formula[4]. To see this, consider the following inference in CPL:

(1) If Wittgenstein died on April 29 of 1951 (call it $W$), then no one can see him in reality anymore ($I$).
(2) If no one can see Wittgenstein in reality anymore, Arthur Schopenhauer disliked Georg Hegel ($S$).
(3) Therefore, we may infer that if Wittgenstein died on April 29 of 1951, then Schopenhauer disliked Hegel.

---

[1]Modern quantum physics confirms physical interaction of objects at particle level is only statistically predicted and thus is non-deterministic.

[2]There actually is a logic which tries to reconcile this problem: the ***Intuitionist logic***. In its essence, it says that the meaning of a formula (or a sentence) is determined not by the conditions under which it is true, but by the conditions under which its proof is found (*proof condition*). Conceivably, this system yields more accurate verification of the truth values of each statement than CPL in that intuitionism won't allow, for example, $\Phi \vee \neg\Phi$ to be true unless one of proposition, either $\Phi$ or $\neg\Phi$, is proven. True, truth values obtained by this system is highly credible; as a matter of fact, many computational automatic theorem provers now use intuitionist logic as their basis. But this draws a even stricter boundary to the domain of our thought than an already strict and inperfect logic: CPL. (Yet I do feel this is a much better logic than CPL itself is.)

[3]Consider also the unsolvability of *Continuum Hypothesis*, or another funny example *Skolem Paradox*.

[4]There had been attempts to resolve this problem, however; one of them is called *Conditional logic*.

Notice that (1)-(3) is a perfectly sound inference under CPL: (1) is true because we can't see someone who is deceased already; (2) is the case because $I$ is true and, as a historical fact, $S$ is also the case; thus, by transitivity, we may conclude that (3) is true. In symbols, this may be expressed as $\{ W \Rightarrow I, I \Rightarrow S \} \models W \Rightarrow S$ or, because the completeness theorem holds, $\{ W \Rightarrow I, I \Rightarrow S \} \vdash W \Rightarrow S$.

But we all notice that either of (2) and (3) is absurd, and in a real debate we would immediately dismiss such an inference. It is clear therefore that when these irrelevant statements are consciously interpreted within the domain of humans and human communication — namely, dynamic systems involving time, memory, uncertainly and belief — they suddenly turns into illogical statements. (Mathematicians *implicitly* excludes (or conceals) this type of irrelevance between the antecedent and the consequent; thus it is fair to say that mathematics involves human psychology.)

## 4. Kleene three-valued logic ($K_3$)

$K_3$ is a classical 3-valued propositional logic with the structure $\langle \mathrm{TV}, D, \mathcal{T} \rangle = \langle \{0, 1, 2$
$\}, \{1\}, \{ \tau_c \mid c \in \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}\}\rangle$ This is a system which inherits the behavior of CPL's interpretation involving truth values 0 and 1. As is the case for CPL, 1 means true and 0 false; the truth value '2' (often denoted as 'u') in $K_3$ is used to represent the condition of being "unknown." For example, the proposition, as some contemporary astrophysicists believe, that "there are 1 or more universes distinct from the Universe in which the Earth resides" in this system will have the truth value of 2 because it have not been (or can't be) confirmed yet. The proposition "Quine died on Christmas Day of 2000," however, will have the value 1 because it is a historical fact. The definitions of each function for connectives follow.

$\tau_\neg$ is a unary function, i.e., a function whose arity is one, defined below:

| $x$ | $\tau_\neg(x)$ |
|---|---|
| 2 | 2 |
| 1 | 0 |
| 0 | 1 |

$\tau_\wedge$ is a binary function, i.e., a function with arity two, defined accordingly:

| $x$ | $y$ | $\tau_\wedge(x, y)$ |
|---|---|---|
| 2 | 2 | 2 |
| 2 | 1 | 2 |
| 2 | 0 | 0 |
| 1 | 2 | 2 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 2 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\tau_\vee$:

| $x$ | $y$ | $\tau_\vee(x,y)$ |
|---|---|---|
| 2 | 2 | 2 |
| 2 | 1 | 1 |
| 2 | 0 | 2 |
| 1 | 2 | 1 |
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 2 | 2 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

$\tau_\Rightarrow$:

| $x$ | $y$ | $\tau_\Rightarrow(x,y)$ |
|---|---|---|
| 2 | 2 | 2 |
| 2 | 1 | 1 |
| 2 | 0 | 2 |
| 1 | 2 | 2 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 2 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

$\tau_\Leftrightarrow$:

| $x$ | $y$ | $\tau_\Leftrightarrow(x,y)$ |
|---|---|---|
| 2 | 2 | 2 |
| 2 | 1 | 2 |
| 2 | 0 | 2 |
| 1 | 2 | 2 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 2 | 2 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

## 5. Łukasiewicz three-valued logic (Ł$_3$)

System Ł$_3$ is a very slight modification of $K_3$ which contains the fatal problem that the law of identity $a \Rightarrow a$, which seems intuitively correct to humans, is not even valid. $L_3$ modifies $K_3$ so that this problem be fixed. Thus, only functions modified are $\tau_\Rightarrow$ and $\tau_\Leftrightarrow$ (again, $\tau_\Leftrightarrow$ can be obtained by $\tau_\wedge(\tau_\Rightarrow(x,y), \tau_\Rightarrow(x,y)))$.

$\tau_\Rightarrow$:

| $x$ | $y$ | $\tau_\Rightarrow(x,y)$ |
|---|---|---|
| 2 | 2 | 1 |
| 2 | 1 | 1 |
| 2 | 0 | 2 |
| 1 | 2 | 2 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 2 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

$\tau_\Leftrightarrow$:

| $x$ | $y$ | $\tau_\Leftrightarrow(x,y)$ |
|---|---|---|
| 2 | 2 | 1 |
| 2 | 1 | 2 |
| 2 | 0 | 2 |
| 1 | 2 | 2 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 2 | 2 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

Example:

```
plman[i]> :setSystem "L3"
plman[i]> :valid a => a
out[i]> True        [ 1 ; True ]
```

## 6. Logic of paradox ($LP$)

System $LP$ is exactly the same as $K_3$ except for the elements of $D$: in $LP$, the designated set is defined as $D = \{1, 2\}$, as opposed to $K_3$ whose $D$ contains only 1. In this system, then, one can translate the truth value 2 as “both true and false”, 1 as “true and true only”, and 0 as “false and false only”, which are exactly the PLMan translations of those values.
Example:

```
plman[i]> :setSystem "K3"
plman[i]> zeta = 2
plman[i]> zeta
```

```
out[i]> False      [ 2 ; Unknown ]
plman[i]> :setSystem "LP"
plman[i]> zeta
out[i]> True       [ 2 ; Both true and false ]
```

## 7. R-Mingle three-valued logic ($\mathrm{RM}_3$)

System $RM_3$ modifies $LP$ so that ***modus ponens*** be valid; the rest is the same as $LP$ otherwise.

$\tau_{\Rightarrow}$:

| $x$ | $y$ | $\tau_{\Rightarrow}(x,y)$ |
|---|---|---|
| 2 | 2 | 2 |
| 2 | 1 | 1 |
| 2 | 0 | 0 |
| 1 | 2 | 0 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 2 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

$\tau_{\Leftrightarrow}$:

| $x$ | $y$ | $\tau_{\Leftrightarrow}(x,y)$ |
|---|---|---|
| 2 | 2 | 2 |
| 2 | 1 | 0 |
| 2 | 0 | 0 |
| 1 | 2 | 0 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 2 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

## 8. A Fuzzy Logic (Ł)

System Ł is a system of fuzzy logic which can take an infinite number of truth values between 0 and 1, inclusive. Thus, the system may be represented by the structure $\langle \mathrm{TV}, D, \mathcal{T}\rangle = \langle\{x \in \mathbb{R} \mid 0.0 \le x \le 1.0\}, \{\, x \in \mathbb{R} \mid \lambda \le x \le 1.0\}, \{\, \tau_c \mid c \in \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}\}\rangle$ where $\lambda$ is some real number between 0 and 1, inclusive. A proposition which takes the truth value of 0.0 is sometimes said to be "completely false"; the truth value of 1.0 "completely true." One may interpret $\lambda$ as a sort of the very borderline between truth and falsity — values above and equal to $\lambda$ are true, below false. In PLMan, the default value of $\lambda$ is 0.5; one can change the value by typing;

```
:setBorderline x
```

where $0 \le x \le 1$.

Semantic functions for connectives are defined as follows:

$\tau_{\neg}$:

$$\tau_{\neg}(x) = 1 - x$$

$\tau_{\wedge}$:

$$\tau_{\wedge}(x,y) = \min(x,y)$$

$\tau_{\vee}$:

$$\tau_{\vee}(x,y) = \max(x,y)$$

$\tau_{\Rightarrow}$:

$$\tau_{\Rightarrow}(x,y) = \begin{cases} 1 & \text{if } x \leq y \\ 1-(x-y) & \text{otherwise} \end{cases}$$

$\tau_{\Leftrightarrow}$:

$$\tau_{\Leftrightarrow}(x,y) = \tau_{\wedge}(\tau_{\Rightarrow}(x,y), \tau_{\Rightarrow}(y,x))$$

Just as a note, if we let TV be $\{0.0, 0.5, 1.0\}$ and $D$ be $\{1\}$ for Ł, then the resulting system behaves exactly the same as $Ł_3$.

## 9. Łukasiewicz' three-valued continuum-valued logic ($Ł_{\aleph}$)

$Ł_{\aleph}$ is the same as system Ł except that the designated set $D$ contains only value 1. Thus the structure becomes $\langle \mathrm{TV}, D, \mathcal{T} \rangle = \langle \{x \in \mathbb{R} \mid 0.0 \leq x \leq 1.0\}, \{1.0\}, \{\ \tau_c \mid c \in \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}\}\rangle$ where every $\tau_c$ is the same as the ones in Ł.

## 10. First Degree Entailment as a Four-valued Logic (FDE)

Logic FDE differs from other systems we've seen so far in one fundamental manner: its truth assignments are "relations", not functions. Thus, although there are actually only two truth values 1 (True) and 0 (False), i.e., TV = $\{0, 1\}$, any propositional atom $p$ of FDE may be in one of four states:

- $p$ relates to 1 only [ True and not False ]
- $p$ relates to 0 only [ False and not True ]
- $p$ relates to both 1 and 0 [ Both True and False ]
- $p$ relates to neither 1 nor 0 [ Neither True nor False ]

PLMan considers all of the above states as being distinct and labels these states as 0, 1, 2, and 3, respectively. The designated values in FDE, then, are 1 and 2 (i.e., $D = \{1, 2\}$ ). If being interpreted this way, FDE may be considered as a 4-valued logic.

A truth assignment in FDE is a relation R from PA to $\{0, 1\}$. As was the case for logics explained earlier, we can extend a given assignment R and create a new interpretation relation $\bar{\mathrm{R}} : \mathcal{W} \to \mathrm{TV}$, i.e. a relation from a set of wffs to a set of truth values, recursively:

- $\alpha\ \bar{\mathrm{R}}\ 1$ *iff* $\alpha \in \mathrm{PA}$ and $\alpha\ \mathrm{R}\ d$ where $d \in D$
- $\alpha\ \bar{\mathrm{R}}\ 0$ *iff* $\alpha \in \mathrm{PA}$ and $\alpha\ \mathrm{R}\ d$ where $d \in D$
- $\neg\alpha\ \bar{\mathrm{R}}\ 1$ *iff* $\alpha\ \bar{\mathrm{R}}\ 0$
- $\neg\alpha\ \bar{\mathrm{R}}\ 0$ *iff* $\alpha\ \bar{\mathrm{R}}\ 1$
- $\alpha \wedge \beta\ \bar{\mathrm{R}}\ 1$ *iff* $\alpha\ \bar{\mathrm{R}}\ 1$ and $\beta\ \bar{\mathrm{R}}\ 1$
- $\alpha \wedge \beta\ \bar{\mathrm{R}}\ 0$ *iff* $\alpha\ \bar{\mathrm{R}}\ 0$ or $\beta\ \bar{\mathrm{R}}\ 0$
- $\alpha \vee \beta\ \bar{\mathrm{R}}\ 1$ *iff* $\alpha\ \bar{\mathrm{R}}\ 1$ or $\beta\ \bar{\mathrm{R}}\ 1$
- $\alpha \vee \beta\ \bar{\mathrm{R}}\ 0$ *iff* $\alpha\ \bar{\mathrm{R}}\ 0$ and $\beta\ \bar{\mathrm{R}}\ 0$
- $\alpha \Rightarrow \beta\ \bar{\mathrm{R}}\ 1$ *iff* $\neg\alpha\ \bar{\mathrm{R}}\ 1$ or $\beta\ \bar{\mathrm{R}}\ 1$
- $\alpha \Rightarrow \beta\ \bar{\mathrm{R}}\ 0$ *iff* $\neg\alpha\ \bar{\mathrm{R}}\ 0$ and $\beta\ \bar{\mathrm{R}}\ 0$
- $\alpha \Leftrightarrow \beta\ \bar{\mathrm{R}}\ 1$ *iff* $\alpha \Rightarrow \beta\ \bar{\mathrm{R}}\ 1$ and $\beta \Rightarrow \alpha\ \bar{\mathrm{R}}\ 1$

- $\alpha \Leftrightarrow \beta \ \bar{\mathrm{R}}\ 0$ *iff* $\alpha \Rightarrow \beta \ \bar{\mathrm{R}}\ 0$ or $\beta \Rightarrow \alpha \ \bar{\mathrm{R}}\ 0$

The truth conditions above yields the following truth tables. Here, one must notice that the values that appear in the tables are "states" rather than "truth values" as explained earlier.

| $x$ | $\neg x$ |
|---|---|
| 3 | 3 |
| 2 | 2 |
| 1 | 0 |
| 0 | 1 |

| $x$ | $y$ | $x \wedge y$ |
|---|---|---|
| 3 | 3 | 3 |
| 3 | 2 | 0 |
| 3 | 1 | 3 |
| 3 | 0 | 0 |
| 2 | 3 | 0 |
| 2 | 2 | 2 |
| 2 | 1 | 2 |
| 2 | 0 | 0 |
| 1 | 3 | 3 |
| 1 | 2 | 2 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 3 | 0 |
| 0 | 2 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

| $x$ | $y$ | $x \vee y$ |
|---|---|---|
| 3 | 3 | 3 |
| 3 | 2 | 1 |
| 3 | 1 | 1 |
| 3 | 0 | 3 |
| 2 | 3 | 1 |
| 2 | 2 | 2 |
| 2 | 1 | 1 |
| 2 | 0 | 2 |
| 1 | 3 | 1 |
| 1 | 2 | 1 |
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 3 | 3 |
| 0 | 2 | 2 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

| $x$ | $y$ | $x \Rightarrow y$ |
|---|---|---|
| 3 | 3 | 3 |
| 3 | 2 | 1 |
| 3 | 1 | 1 |
| 3 | 0 | 3 |
| 2 | 3 | 1 |
| 2 | 2 | 2 |
| 2 | 1 | 1 |
| 2 | 0 | 2 |
| 1 | 3 | 3 |
| 1 | 2 | 2 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 3 | 1 |
| 0 | 2 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

| $x$ | $y$ | $x \Leftrightarrow y$ |
|---|---|---|
| 3 | 3 | 3 |
| 3 | 2 | 1 |
| 3 | 1 | 3 |
| 3 | 0 | 3 |
| 2 | 3 | 1 |
| 2 | 2 | 2 |
| 2 | 1 | 2 |
| 2 | 0 | 2 |
| 1 | 3 | 3 |
| 1 | 2 | 2 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 3 | 3 |
| 0 | 2 | 2 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

**FDE on PLMan.**

Setting FDE as a current logic can be done by:

```
:setSystem "FDE"
```

In order to interpret any formula under FDE, one must first provide a truth assignment (a relation); a truth assignment can be created by any input of the form

```
:createR <RELATION_SYMBOL> <RELATION>
```

where <RELATION> is any expression of the form:

$$< a_1, v_1 >, < a_2, v_2 >, ..., < a_n, v_n >$$

and where $a_i \in \mathrm{PA}$ and $v_i \in \{0, 1\}$. Thus,

```
:createR R1 { <a,0> }
```

creates a relation R1 with an initial element <a,0>.

Once a relation is created, one can set the relation to FDE by:

```
:setR ExistingRelation
```

ExistingRelation then will become the truth assignment of FDE with which PLMan can interpret formulas. The default truth assignment is the one that has no relational element (i.e., a relation for which every propositional atom is interpreted as neither true nor false).

In order to add or remove an element from an existing relation, type either of the following

```
:addR ExistingRelation <PAIR>
:addR ExistingRelation <PAIR>, <PAIR>, ...
:removeR ExistingRelation <PAIR>
:removeR ExistingRelation <PAIR>, <PAIR>, ...
```

where `<PAIR>` is of the form:

$$< a, v >$$

such that $a \in$ PA and $v \in$ TV $= \{0, 1\}$. There are neat abbreviations for these commands — please refer to ":addR" and ":removeR" sections in chapter "Commands".

To show the contents of an existing relation can be done by:

```
:showR ExistingRelation
```

Example:

```
plman[i]> :setSystem "FDE"
plman[i]> :createR R { <x,0>, <x,1>, <y,0> }
Relation 'R' is now created.
plman[i]> :setR R
Relation for the current system is now set to 'R'.
plman[i]> :showR R
<x,0>
<x,1>
<y,0>
plman[i]> :addR R <z,1>
Pair <z,1> added to the current relation.
plman[i]> :addR R <y,1>, <z,0>
Pair <y,1> added to the current relation.
Pair <z,0> added to the current relation.
plman[i]> :showR R
<x,0>
<x,1>
<y,0>
<y,1>
<z,0>
<z,1>
plman[i]> :removeR R <x,0>, <y,1>
Pair <x,0> removed from the current relation.
```

```
Pair <y,1> removed from the current relation.
plman[i]> :showR R
<x,1>
<y,0>
<z,0>
<z,1>
```

CHAPTER 4

# Commands

## 1. System Commands

### 1.1. :setSystem.

```
:setSystem "<SYSTEM_NAME>"
```

Command ':setSystem' sets the system of logic specified by `<SYSTEM_NAME>`, as the current underlying logical system in PLMan. The value of `<SYSTEM_NAME>` can be either "`CPL`", "`K3`", "`L3`", "`L`", "`LAleph`", "`LP`", "`RM3`", or "`FDE`".

**Example**:

```
plman[i]> :setSystem "K3"
plman[i]> x = 2, y = 2
plman[i]> x => y
out[i]> False        [ 2 ; Unknown ]
plman[i]> :setSystem "L3"
plman[i]> x => y
out[i]> True         [ 1 ; True ]
```

**Works on**:
- All systems of logic.

### 1.2. :currentSystem.

```
:currentSystem
```

Command ':currentSystem' outputs the name of the current logic under which we are evaluating expressions.

**Example**:

```
plman[i]> :currentSystem
CPL
plman[i]> :setSystem "FDE"
plman[i]> :currentSystem
```

```
FDE
```

**Works on**:

- All systems of logic.

### 1.3. :exit.

```
:exit
```

Command ':exit' exits PLMan.

**Example**:

```
plman[i]>  :exit
```

**Works on**:

- All systems of logic.

### 1.4. :interactiveMode.

```
:interactiveMode
```

Command ':interactiveMode' switches the ***FileInput mode*** to ***Interactive mode***. This is useful if one inputs a PLMan script file to PLMan, but wants to communicate with PLMan interactively based on the knowledge given by the script.

**Example**:

```
    $ ls
    test.plman
    $ cat test.plman
    -- Mode Switch Demo

    S : "PLMan reached :interactiveMode command input
line in a plman script file "
    E : "you executed the file"
    I : "you are in the PLMan interactive mode."

    :interactiveMode

    $ plman test.plman
    ------------------ PLMan: Reading input strings from file ‘test.plman’

    plman[i]> S : "PLMan reached :interactiveMode command input
line in a plman script file "
```

```
    plman[i]> E : "you executed the file"
    plman[i]> I : "you are in the PLMan interactive mode."
    plman[i]> :interactiveMode
    -- INTERACTIVE MODE SESSION STARTS HERE
    plman[i]> S = 1, E = 1, I = 0
    plman[i]> (S & E) => I      -- this should be because you are
in fact in interactive mode
    out[i]> False       [ 0 ; False ]
    plman[i]> I = 1
    plman[i]> (S & E) => I
    out[i]> True        [ 1 ; True ]
    plman[i]>
```

**Works on**:

- All systems of logic.

## 2. Commands involving Truth Values

### 2.1. :table.

```
:table <FORMULA>
```

Command ':table' outputs the truth table of the given FORMULA.

The maximum number of elements in the table excluding the elements in the answer (or formula) column, which can be calculated by the number of truth values in the current system raised to the power of the number of distinct propositional atoms in the given formula (e.g., the number of elements for "a | b & c" under CPL is $2^3 = 8$) , is set to the maximum value of the natural part of the long integer — namely, $2^{63} - 1$ (9,223, 372,036,854,775,807). Strictly speaking, this maximum value can be set to $2^{63}$, which is more succinct and more desirable, but there is a technical constraint that prevents me from doing so.

**Works on**:

- "CPL"
- "K3"
- "L3"
- "LP"
- "RM3"
- "FDE"

### 2.2. :satisfiable.

```
:satisfiable <FORMULA>
```

Command ':satisfiable' returns either True or False; if the formula is satisfiable then the system will return True; False otherwise.

As is the case for ':table' command, if we let the number of truth values in the current system be $b$ and the number of distinct propositional atoms in the given formula be $n$, then it must always be the case that $b^n < 2^{63}$, presupposing that $b^n \in \mathbb{N}$; otherwise, the PLMan will reject the request.

**Example**:

```
plman[i]> :setSystem "CPL"
plman[i]> :satisfiable a & ~a
out[i]> False      [ 0 ; False ]
plman[i]> :setSystem "FDE"
plman[i]> :satisfiable    a & ~a
out[i]> True      [ 1 ; True and not false ]
plman[i]> :setSystem "LP"
plman[i]> :satisfiable    a & ~a
out[i]> True      [ 1 ; True and true only ]
```

**Works on**:

- "CPL"
- "K3"
- "L3"
- "LP"
- "RM3"
- "FDE"

### 2.3. :valid.

```
:valid <FORMULA>
```

Command ':valid' returns either True or False; if the formula is valid it will return False; True otherwise.

As is the case for ':table' command, if we let the number of truth values in the current system be $b$, the number of distinct propositional atoms in the given formula be $n$, then it must always be the case that $b^n < 2^{63}$, presupposing that $b^n \in \mathbb{N}$; otherwise, the PLMan will reject the request.

**Example**:

```
plman[i]> :setSystem "CPL"
plman[i]> :valid   a | ~a
out[i]> True       [ 1 ; True ]

plman[i]> :setSystem "K3"
plman[i]> :valid   a | ~a
out[i]> False      [ 0 ; False ]
```

```
plman[i]> :setSystem "FDE"
plman[i]> :valid   a | ~a
out[i]> False      [ 0 ; False and not true ]
```

**Works on**:

- “CPL”
- “K3”
- “L3”
- “LP”
- “RM3”
- “FDE”

### 2.4. Entailment ‘| =’.

```
{ a, b, c,  ... }  |=  d
```

Greek alphabets above are formulas. Inside  must be contained one or more formulas (i.e., cannot be empty). It returns either True or False: True if `{a,b,c,...}` entails `d`, False otherwise.

**Example**:

```
plman[i]> :setSystem "CPL"
plman[i]> { a | b , a <=> ~ b } |=  (a | b) & ~( a & b)
out[i]> True        [ 1 ; True ]
```

**Works on**:

- “CPL”
- “K3”
- “L3”

## 3. Commands involving Relations

Currently, all commands explained in this section are used only on system “FDE”.

### 3.1. :createR.

```
:createR <RELATION_SYMBOL> <RELATION>
```

Command ‘:createR’ creates a relation of the name `<RELATION_SYMBOL>` from a set of propositional atoms to a set of truth values.

`<RELATION>` is any expression of the form:

$$< a_1, v_1 >, < a_2, v_2 >, ..., < a_n, v_n >$$

where $n$ is some natural number (thus includes 0, in which case the form will look like: `{ }` )

Here, $a_i$ is an element in <PROPOSITIONAL_ATOM>; $v_i$ in the set of truth values for they underlying logical system. (precisely speaking, however, PLMan allows $v_i$ to be in <PROPOSITIONAL_ATOM> for the future extensibility, but doing so will have no effect.) $a_i$ doesn't have to be an non-existent: it can be an already existing relation as well.

**Example**:

```
plman[i]> :setSystem "FDE"
plman[i]> :createR R1 { <a,0> }
Relation ‘R1’ is now created.
plman[i]> :showR R1
<a,0>
plman[i]> :createR R2 { }
Relation ‘R2’ is now created.
plman[i]> :showR R2     -- nothing will appear
plman[i]> :createR R1 { <a,1>, <b,1> , <b,0> , <d,0> }
Relation ‘R1’ is now created.
plman[i]> :showR R1
<a,1>
<b,0>
<b,1>
<d,0>
```

**Works on**:

- “FDE”

### 3.2. :setR.

```
:setR ExistingRelation
```

It sets a relation to a logical system whose interpretation depends on it. If the current logic do not require a relation to be set, then ‘:setR’ takes no effect.

**Example**:

```
plman[i]> :setSystem "CPL"
plman[i]> :createR R { <x,0> }
Relation ‘R’ is now created.
plman[i]> :setR R
WARNING: Symbol ‘R’ is not an relation object. The attempt failed...
plman[i]> :setSystem "FDE"
plman[i]> :setR R
Relation for the current system is now set to ‘R’.
```

**Works on**:

- "FDE"

### 3.3. :addR.

```
:addR ExistingRelation <PAIR>
:addR ExistingRelation <PAIR>, <PAIR>, ...
```

Command ':addR' adds an element expressed by `<PAIR>` to an already existing relation.

`<PAIR>` is of the form:

$$< a, v >$$

and where $a \in$ PA and $v \in$ TV.

Once a truth assignment relation R is set to the correct logic, then one can abbreviate

```
:addR R <PAIR>
:addR R <PAIR>, <PAIR>, ...
```

simply as

```
<PAIR>
<PAIR> , <PAIR>, ...
```

**Example**:

```
plman[i]> :setSystem "FDE"
plman[i]> :createR R { <x,0> }
Relation 'R' is now created.
plman[i]> :addR R <x,1>
Pair <x,1> added to the current relation.
plman[i]> :showR R
<x,0>
<x,1>
plman[i]> :setR R
plman[i]> <y,1> , <y,0>     -- adding those elements
Pair <y,1> added to the current relation.
Pair <y,0> added to the current relation.
plman[i]> :showR R
<x,0>
<x,1>
<y,0>
<y,1>
```

**Works on**:

- “FDE”

### 3.4. :removeR.

```
:removeR ExistingRelation <PAIR>
:removeR ExistingRelation <PAIR>, <PAIR>, ...
```

Command ‘:removeR’ removes an element expressed by `<PAIR>` to an already existing relation.

`<PAIR>` is of the form:

$$< a, v >$$

and where $a \in \mathrm{PA}$ and $v \in \mathrm{TV}$.

Once a truth assignment relation R is set to the correct logic, then one can abbreviate

```
:removeR R <PAIR>
:removeR R <PAIR>, <PAIR>, ...
```

as simply,

```
\ <PAIR>
\ <PAIR> , <PAIR>, ...
```

**Example**:

```
plman[i]> :setSystem "FDE"
plman[i]> :createR R { <x,0> , <x,1> , <y,0> , <y,1> }
Relation ‘R’ is now created.
plman[i]> :removeR <y,1> , <y,0>
plman[i]> :setR R
plman[i]> \ <y,1> , <y,0>     -- removing those elements
Pair <y,1> removed from the current relation.
Pair <y,0> removed from the current relation.
plman[i]> :removeR R <x,1>
Pair <x,1> removed from the current relation.
plman[i]> :showR R
<x,0>
```

**Works on**:

- “FDE”

### 3.5. :showR.

```
:showR ExistingRelation
```

Command ':showR' shows the contents of a given relation 'ExistingRelation'.

**Example**:

```
    plman[i]> :setSystem "FDE"
    plman[i]> :createR Philosophers-of-Mathematics-in-20th-century
{ <Plato,0>, <Godel,1> }
    Relation 'Philosophers-of-Mathematics-in-20th-century' is now created.
    plman[i]> :addR Philosophers-of-Mathematics-in-20th-century
<Brouwer,1> , <Kant,0> , <Quine,1>
    Pair <Brouwer,1> added to the current relation.
    Pair <Kant,0> added to the current relation.
    Pair <Quine,1> added to the current relation.
    plman[i]> :addR Philosophers-of-Mathematics-in-20th-century
<Curry,0>, <Curry,1>
    Pair <Curry,0> added to the current relation.
    Pair <Curry,1> added to the current relation.
    plman[i]> -- Because "Curry" in English is also a word for a
spiced dish with curry powder
    plman[i]> :showR Philosophers-of-Mathematics-in-20th-century
    <Brouwer,1>
    <Curry,0>
    <Curry,1>
    <Godel,1>
    <Kant,0>
    <Plato,0>
    <Quine,1>
```

**Works on**:

- "FDE"

## 4. Useful User-friendly Commands

### 4.1. :translate or ::

```
:translate <FORMULA>
:: <FORMULA>
```

Command ':translate' or its abbreviation '::' outputs the English translation of the given FORMULA.

**Example**:

```
    plman[i]> S : "Socrates is a man" -- ':' is the description
assignment operator
```

```
plman[i]> H : "Socrates is a human" , W : "Socrates is a woman"
plman[i]> :: S <=> ( H & ~W)
"Socrates is a man" if and only if [ "Socrates is a human" and
it is not the
case that "Socrates is a woman" ]
```

**Works on**:

- All systems of logic (but may not correctly translate the given connectives if their meaning differ from those given in CPL)

### 4.2. :inputFile.

```
:inputFile "<FILENAME>"
```

Command ':inputFile' takes a file of `<FILENAME>` from the path set to the environment variable 'PLMAN_SCRIPT_PATH'.

**Works on**:

- All systems of logic.