

TLA⁺ Notation

for readers of

A Science of Computer Programs

Leslie Lamport

24 October 2024

This document describes how to write formulas that appear in the book *A Science of Computer Programs* in the TLA⁺ language. It explains all the notation used in the examples from the book that are on the web. How to write all TLA⁺ formulas can be found elsewhere [1, 2].

Symbols

TLA⁺ is written in ASCII. (Unicode input may be supported in the future.) Symbols that have obvious ASCII versions, such as +, \ (set difference), and .. (integer interval), are represented in the obvious way. The ASCII versions of other symbols that are not part of larger constructs are given in Figure 1. Symbols written in the book in small capitals like TRUE, THEN, and ELSE are written in TLA⁺ with ordinary capitals like TRUE, THEN, and ELSE.

Functions

The biggest difference between the book's notation and TLA⁺ involves functions. In TLA⁺, most notation for functions uses the square brackets []. If f is a function, the application of f to an expression exp , which is written in the book as $f(exp)$, is written in TLA⁺ as $f[exp]$. Thus, if f is a defined symbol, we can write $f(exp)$ iff f is defined by $f(p) \triangleq \dots$. We can write $f[exp]$ if f is any expression, but it is a meaningful expression iff f is a function. (See Section 2.6 of the book.)

Here are the other constructs involving functions:

<u>Book</u>	<u>TLA⁺</u>
\neg (negation)	\sim (tilde)
\leadsto (leads to)	$\sim>$
\times (Cartesian product)	$\backslash X$
$\#$ (cardinality)	Cardinality ¹
\Rightarrow (implication)	\Rightarrow
\equiv (equivalence)	\Leftrightarrow or $\backslash equiv$
\neq (not equal to)	$\#$ or \neq
\leq (less than or equal)	\leq or $\backslash leq$
\geq (greater than or equal)	\geq or $\backslash geq$
\triangleq (equals by definition)	$==$
\circ (sequence concatenation)	$\backslash o$
\oplus	$(+)$
\square (always)	$[]$
\diamond (eventually)	$\langle \rangle$
\subseteq (is a subset of)	$\backslash subseteq$
$'$ (prime)	$'$
$\langle \rangle$ (tuple notation)	$\ll \gg$
\wedge (conjunction)	\wedge or $\backslash land$
\cap (set intersection)	$\backslash cap$
\vee (disjunction)	\vee or $\backslash lor$
\cup (set union)	$\backslash cup$
\forall (universal quantification)	$\backslash A$
\exists (existential quantification)	$\backslash E$
\in (is an element of)	$\backslash in$
\notin (is not an element of)	$\backslash notin$
\div (integer division)	$\backslash div$
\mathbb{I}	Int ²
\mathbb{N}	Nat ²
\mathbb{E}	ENABLED

¹Defined in the *FiniteSets* module.

²Defined in the *Integers* module.

Figure 1: ASCII representation of symbols.

<u>Book</u>	<u>TLA⁺</u>
$v \in S \mapsto \text{exp}$	$[v \ \backslash \text{in } S \mid \rightarrow \text{exp}]$
$f \text{ EXCEPT } \text{exp1} \mapsto \text{exp2}$	$[f \text{ EXCEPT } ![\text{exp1}] = \text{exp2}]$
$D \rightarrow S$	$[D \rightarrow S]$
$\text{DOMAIN}(f)$	$\text{DOMAIN } f$

The TLA⁺ EXCEPT construct has generalizations that at least partly explain its ugliness, but which were not needed in the book.

Subscripts and Superscripts

An expression v appearing as a subscript in a formula of the book is written in TLA⁺ as $_v$. For example, $\text{WF}_v(A)$ is written as $\text{WF}_v(A)$. We can also write $[A]_{\langle x, y \rangle}$ as $[A]_ \langle\langle x, y \rangle\rangle$. However, if you want to write something like $\langle A \rangle_{\text{set}}$, you'd best use parentheses and write $\langle\langle A \rangle\rangle_ (\text{s} \backslash \text{o } \text{t})$ to make sure it's parsed correctly.

The only mathematical notation involving superscripts that is used by TLA⁺ is for exponentiation. A formula written x^2 in the book is written in TLA⁺ as x^2 , where the exponentiation operator \wedge is defined in the *Integers* module.

The book also uses sub- and superscripts for pedagogical purposes. For example, when adding a stuttering variable s to program *ICen2* in Section 7.3.2, it calls the resulting program *ICen2^s*. In the TLA⁺ representation of that example, the formula *ICen2^s* is named *ICen2_s*. Underscores “_” are used in this way for names that in the book are written with sub- or superscripts.

Modules and WITH

The smallest complete unit of a TLA⁺ description of a program is a module. The scope of any declaration or definition that appears in a module lies within that module. A module is contained in a file, and it ends with four or more consecutive “=” characters. Anything following the module in the file is ignored.

A module can import other modules in two ways. The first is with an **EXTENDS** statement that must be the first non-comment in the module. Extending modules in this way is essentially equivalent to copying the contents of the extended modules, including their declarations and definitions, into the current module. While **EXTENDS** statements can be used to decompose the definition of a program into multiple modules, in the examples it is used

mainly just to import standard modules. The *Integers* standard module, which defines the usual operators of arithmetic, is imported in this way by almost all the modules in the example.

The second way one module can import another is by *instantiation* with an **INSTANCE** statement. The book expresses substitution with the **WITH** construct, where

$$exp \text{ WITH } v \leftarrow exp_1, w \leftarrow exp_2$$

is the formula obtained by substituting the expression exp_1 for variable v and the expression exp_2 for variable w in expression exp . TLA^+ does not have such a way to express substitution in an individual formula. Instead, substitution is expressed with module instantiation.

Suppose *Mod* is the name of a module that declares only two identifiers, which may be variables or constants, that are named v and w . The statement

$$IM == \text{INSTANCE } Mod \text{ WITH } v \leftarrow exp_1, w \leftarrow exp_2$$

imports all the definitions from module *Mod* into the current module as follows. If the definition of the mapping **F** defined in module *Mod* were written in the book as $F(p, q, r) \triangleq exp$, then the mapping **IM!F** would be imported into the current module with a definition that would be written in the book as

$$IM!F(p, q, r) \triangleq (exp \text{ WITH } v \leftarrow exp_1, w \leftarrow exp_2)$$

In general, the **INSTANCE** statement must declare a substitution for every variable and constant defined in module *Mod*. However, if module *Mod* has a declared variable or constant named **id**, and the identifier **id** is declared or defined in the current module, then the substitution $id \leftarrow id$ can be omitted in the **INSTANCE** statement.

Proofs

Some of the examples contain proofs that are at least partially checked with TLAPS, the TLA^+ proof checker. The TLA^+ syntax for formal proofs is close to that used in the book, except for two things:

- Step numbers are written like $\langle 3 \rangle 4$, as mentioned in the book's Math VIII section and Section A.2.

- Paragraph proofs are replaced by proofs of the form **BY ... DEF** The **BY** section lists the facts (previous proved steps theorems) that the prover needs to use to prove the current goal. It may also contain **PTL**, which instructs TLAPS to use temporal-logic reasoning. (As explained in the book, most of a TLA proof consists of ordinary math, not temporal logic.) The **DEF** section lists the defined symbols whose definitions the prover needs to expand.

References

- [1] Leslie Lamport. TLA—temporal logic of actions. A web page at <https://lamport.azurewebsites.net/tla/tla.html>.
- [2] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003.