SRC Technical Note
1997 - 006a

April 12, 1997
Corrected June 8, 1997

The Operators of TLA$^+$

Leslie Lamport

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

The Operators of TLA$^+$

Leslie Lamport
lamport@src.dec.com

12 April 1997
Minor correction: 8 June 1997

This document is an introduction to the syntax and semantics of the operators of TLA$^+$. It assumes that you are familiar with ordinary mathematics (sets and functions) and are at least acquainted with TLA. It should enable you to understand the expressions that appear in TLA$^+$ specifications.

This is a preliminary document; suggestions are welcome.

Contents

The operators of TLA$^+$ can be classified as constant operators, action operators, and temporal operators. For convenience, all these operators are listed in Figures 1–4 on the next page through page 4. They are described in three separate sections.

The ASCII versions of typeset special characters are shown in Figure 5 on page 5. In the absence of a reasonable alternative, TLA$^+$ uses the TEX command name for a symbol. (Sometimes there is more than one ASCII version for the same symbol; the different versions are synonymous.)

1 The Constant Operators

All the constant operators of TLA$^+$ are listed in Figure 1 on page 3 and Figure 2 on page 4. In these figures, p and p_i are formulas, x is a bound variable, h and h_i are sequences of characters, the c_i are characters, the d_i are digits, and all other letters are terms. The operators are explained below, in more or less the order in which they appear in the figures. (The "miscellaneous operators" are defined earlier so they can be used in defining other operators.)

The figures show the simple forms of the operators. Some operators have more general forms. For example, you can write $\exists\, x, y : p$ instead of $\exists\, x : \exists\, y : p$. The more general forms are described with the individual operators.

1.1 Untypes

TLA$^+$ is based on ZFC (Zermelo-Fraenkel set theory with the axiom of choice). This is an untyped formalism. In an untyped formalism, every syntactically well-formed expression, no matter how silly, has a meaning—for example, the expression $3 \in \sqrt{\text{“abc”}}$. If the expression is silly, its meaning is probably unspecified. All we can tell about the expression $3 \in \sqrt{\text{“abc”}}$ is that it is a Boolean, so it equals either TRUE or FALSE.

Mathematicians write silly expressions all the time. For example the expression $1/0$ is a silly expression. But if you substitute 0 for x in the formula $(x \neq 0) \Rightarrow (x * (1/x) = 1)$, you get the valid formula $(0 \neq 0) \Rightarrow (0 * (1/0) = 1)$ that contains the silly expression $1/0$. A correct formula can contain silly expressions. However, the validity of a correct formula cannot depend on the meaning of a silly expression.

Logic

TRUE FALSE $\wedge$ $\vee$ $\neg$ $\Rightarrow$ $\equiv$

$\forall x : p$ $\exists x : p$ $\forall x \in S : p$ $\exists x \in S : p$

CHOOSE $x : p$ [Equals some x satisfying p]

Sets

$=$ $\neq$ $\in$ $\notin$ $\cup$ $\cap$ $\subseteq$ $\setminus$ [set difference]

$\{e_1, \ldots, e_n\}$	[Set consisting of elements e_i]
$\{x \in S : p\}$	[Set of elements x in S satisfying p]
$\{e : x \in S\}$	[Set of elements e such that x in S]
SUBSET S	[Set of subsets of S]
UNION S	[Union of all elements of S]

Functions

$f[e]$	[Function application]
DOMAIN f	[Domain of function f]
$[x \in S \mapsto e]$	[Function f such that $f[x] = e$ for $x \in S$]
$[S \rightarrow T]$	[Set of functions f with $f[x] \in T$ for $x \in S$]
$[f$ EXCEPT $![e_1] = e_2]$	[Function $\widehat{f}$ equal to f except $\widehat{f}[e_1] = e_2$]
$\{\![f$ EXCEPT $![e] \in S]\!\}$	[Set of functions $\widehat{f}$ equal to f except $\widehat{f}[e] \in S$]

Records

$e.h$	[The h-component of record e]
$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$	[The record whose h_i component is e_i]
$[h_1 : S_1, \ldots, h_n : S_n]$	[Set of all records with h_i component in S_i]
$[r$ EXCEPT $!.h = e]$	[Record $\widehat{r}$ equal to r except $\widehat{r}.h = e$]
$\{\![r$ EXCEPT $!.h \in S]\!\}$	[Set of records $\widehat{r}$ equal to r except $\widehat{r}.h \in S$]

Tuples

$e[i]$	[The i^{th} component of tuple e]
$\langle e_1, \ldots, e_n \rangle$	[The n-tuple whose i^{th} component is e_i]
$S_1 \times \ldots \times S_n$	[The set of all n-tuples with i^{th} component in S_i]

Strings and Numbers

"$\mathtt{c}_1 \ldots \mathtt{c}_n$"	[A literal string of n characters]
STRING	[The set of all strings]
$d_1 \ldots d_n$ $\quad d_1 \ldots d_n.d_{n+1} \ldots d_m$	[Numbers]

Figure 1: Simple forms of the constant operators of TLA$^+$.

Miscellaneous

if p **then** e_1 **else** e_2	[Equals e_1 if p true, else e_2]
case $p_1 \rightarrow e_1 \;\Box\; \ldots \;\Box\; p_n \rightarrow e_n$	[Equals e_i if p_i true]
let $x_1 \triangleq e_1 \ldots x_n \triangleq e_n$ **in** e	[Equals e in the context of the definitions]

$\land\ p_1$ [the conjunction $p_1 \land \ldots \land p_n$]
$\ \ \ldots$
$\land\ p_n$

$\lor\ p_1$ [the disjunction $p_1 \lor \ldots \lor p_n$]
$\ \ \ldots$
$\lor\ p_n$

Figure 2: Simple forms of the constant operators of TLA$^+$ (continued).

p'	[p true in final state of step]
$[A]_e$	[$A \lor (e' = e)$]
$\langle A \rangle_e$	[$A \land (e' \neq e)$]
ENABLED A	[An A step is possible]
UNCHANGED e	[$e' = e$]
$A \cdot B$	[Composition of actions]

Figure 3: The action operators of TLA$^+$.

$\Box F$	[F is always true]
$\Diamond F$	[F is eventually true]
$\mathrm{WF}_e(A)$	[Weak fairness for action A]
$\mathrm{SF}_e(A)$	[Strong fairness for action A]
$F \leadsto G$	[F leads to G]
$F \stackrel{+}{\rhd} G$	[F guarantees G]
$\boldsymbol{\exists}\, x : F$	[Temporal existential quantification (hiding).]
$\boldsymbol{\forall}\, x : F$	[Temporal universal quantification.]

Figure 4: The temporal operators of TLA$^+$.

<table>
<tr><td>$\land$</td><td><code>/\</code></td><td>$\exists$</td><td><code>\E</code></td><td>$\equiv$</td><td><code>\equiv</code></td></tr>
<tr><td>$\lor$</td><td><code>\/</code></td><td>$\forall$</td><td><code>\A</code></td><td>$\equiv$</td><td><code><=></code></td></tr>
<tr><td>$\Rightarrow$</td><td><code>=></code></td><td>$\in$</td><td><code>\in</code></td><td>CHOOSE</td><td><code>CHOOSE</code></td></tr>
<tr><td>$\neq$</td><td><code>#</code></td><td>$\notin$</td><td><code>\notin</code></td><td>SUBSET</td><td><code>SUBSET</code></td></tr>
<tr><td>$\neq$</td><td><code>/=</code></td><td>$\neg$</td><td><code>~</code></td><td>UNION</td><td><code>UNION</code></td></tr>
<tr><td>$\langle$</td><td><code><<</code></td><td>$<$</td><td><code><</code></td><td>DOMAIN</td><td><code>DOMAIN</code></td></tr>
<tr><td>$\rangle$</td><td><code>>></code></td><td>$>$</td><td><code>></code></td><td>EXCEPT</td><td><code>EXCEPT</code></td></tr>
<tr><td>$\rightarrow$</td><td><code>-></code></td><td>$\leq$</td><td><code>\leq</code></td><td>STRING</td><td><code>STRING</code></td></tr>
<tr><td>$\mapsto$</td><td><code>|-></code></td><td>$\geq$</td><td><code>\geq</code></td><td>case</td><td><code>CASE</code></td></tr>
<tr><td>“</td><td><code>"</code></td><td>$\triangleq$</td><td><code>==</code></td><td>other</td><td><code>OTHER</code></td></tr>
<tr><td>”</td><td><code>"</code></td><td>$\Box$</td><td><code>[]</code></td><td>if</td><td><code>IF</code></td></tr>
<tr><td>$\cdot$</td><td><code>\cdot</code></td><td>$\circ$</td><td><code>\o</code></td><td>then</td><td><code>THEN</code></td></tr>
<tr><td>let</td><td><code>LET</code></td><td>in</td><td><code>IN</code></td><td>else</td><td><code>ELSE</code></td></tr>
<tr><td>$\cap$</td><td><code>\cap</code></td><td>$\times$</td><td><code>\X</code></td><td>$\subseteq$</td><td><code>\subseteq</code></td></tr>
<tr><td>$\cup$</td><td><code>\cup</code></td><td>$\{\!\!\{$</td><td><code>{|</code></td><td>$\}\!\!\}$</td><td><code>|}</code></td></tr>
</table>

Figure 5: ASCII versions of typeset special characters.

1.2 Logic

1.2.1 Propositional Logic

In TLA$^+$, we use the ordinary operators of propositional logic. Negation is denoted by $\neg$ (`~` in ASCII), implication by $\Rightarrow$ (`=>`), and logical equivalence by $\equiv$ (`\equiv` or `<=>`), where $A \equiv B$ equals $(A \Rightarrow B) \land (B \Rightarrow A)$.

Among the propositional-logic operators, $\neg$ has highest precedence, $\land$ and $\lor$ have next lower precedence, then comes $\equiv$, and $\Rightarrow$ has lowest precedence. Because $\land$ and $\lor$ have equal precedence, an expression like $A \lor B \land C$ is illegal; parentheses must be used to disambiguate it.

1.2.2 Predicate Logic

We also use the ordinary universal quantifier $\forall$ (`\A`) and existential quantifier $\exists$ (`\E`) of predicate logic. Bounded quantifiers are defined in terms of unbounded ones in the usual way:

$$\begin{aligned} \forall x \in S : p &\triangleq \forall x : (x \in S) \Rightarrow p \\ \exists x \in S : p &\triangleq \exists x : (x \in S) \land p \end{aligned}$$

In these expressions, the bound variable x may not occur in the expression S.

TLA$^+$ uses the customary abbreviations for multiple quantification: $\forall\, x_1, \ldots, x_n : p$ and $\exists\, x_1, \ldots, x_n : p$. For example

$$\forall\, x, y, z : P \;\triangleq\; \forall\, x : \forall\, y : \forall\, z : P$$

The general form of bounded universal quantification is

$$\forall\, q_1 \in S_1, \ldots, q_n \in S_n : p$$

where each q_i is either a list $x_1, \ldots, x_m$ of variables or a tuple $\langle x_1, \ldots, x_m \rangle$ of variables. (See Section 1.6 for a discussion of tuples.) None of these bound variables may appear in any of the S_i. Existential quantification is similar. For example,

$$\begin{array}{l} \exists\, \langle x, y \rangle \in S, z, w \in T : p \;\triangleq \\ \quad \exists\, x, y, z, w : (\langle x, y \rangle \in S) \land (z \in T) \land (w \in T) \land p \end{array}$$

where neither x, y, z, nor w may appear in S or T.

An expression begun by $\forall$ or $\exists$, is terminated by the end of the statement or expression that contains it. The containing expression may be ended by some form of right parenthesis, by indentation rules (see Section 1.3.3), or by the end of a statement containing the expression.

1.2.3 The CHOOSE Operator

The CHOOSE operator is known to logicians as Hilbert's ε [4]. If there is an x such that p holds, then CHOOSE $x : p$ equals some such x. If there is more than one such x, it is not specified which one is chosen. If there is no such x, then the value of CHOOSE $x : p$ is unspecified.

The most common use for the CHOOSE operator is to "name" a uniquely specified value. For example, one possible definition of the operator / on the set $Real$ of real numbers is:

$$r/s \;\triangleq\; \text{CHOOSE } v : (v \in Real) \land (v * s = r)$$

If r is a nonzero real number, then there is no real v such that $v * 0 = r$. Therefore, $r/0$ has a completely unspecified value. We don't know what a real number times a string equals, so we cannot say whether or not there is a real number v such that $v *$ "abc" equals r. Hence, we don't know what the value of $r/$"abc" is.

The CHOOSE operator is often used in the following idiom, which defines $notAnS$ to be some arbitrary element not in S.

$$notAnS \;\triangleq\; \text{CHOOSE } x : x \notin S$$

The CHOOSE operator cannot be used to introduce nondeterminism. If q is equivalent to p, then CHOOSE $x : q$ equals CHOOSE $x : p$. Hence, CHOOSE "chooses the same value every time."

A CHOOSE expression, like a quantifier expression, is ended by the end of the expression or statement that contains it.

1.2.4 Typeless Logic

Because TLA$^+$ is completely untyped, an expression like $3 \wedge$ "abc" is legal. We can therefore ask whether it equals "abc" $\wedge\ 3$, and whether $(3 \wedge \text{"abc"}) \vee \neg(3 \wedge \text{"abc"})$ equals TRUE? There are two reasonable ways to define the semantics of the Boolean operators which give two sets of answers to such questions.

In the weak semantics, one knows nothing about the meaning of an expression like $a \wedge b$ if a and b are not both Booleans. With this semantics, $3 \wedge$ "abc" need not be a Boolean, and there is no reason to expect it to equal "abc" $\wedge\ 3$. We can reason about logical operators only when applied to Booleans.

In the strong semantics, $a \wedge b$ is a Boolean regardless of what a and b are, and logical operators obey most of the usual laws. In particular, $a \wedge b$ equals $b \wedge a$ for any a and b. The easiest way to define the strong semantics is to assume an operator Ψ such that $\Psi(e)$ is a Boolean for every e, and $\Psi(e)$ equals e if e is a Boolean. We then define the logical operators so that, for example, $a \wedge b$ equals $\Psi(a) \wedge \Psi(b)$, for all a and b.

The strong semantics is more convenient to use when writing proofs, because we can apply the laws of logic without having first to prove that the arguments of Boolean operators are Booleans. However, when writing specifications, we should assume only the weak semantics. The meaning of a specification should never depend on the value obtained by applying a Boolean operator to a nonBoolean.

1.3 Miscellaneous Operators

1.3.1 Constructs Stolen from Programming Languages

The expression

$$\textbf{if } p \textbf{ then } e_1 \textbf{ else } e_2$$

equals e_1 if p is true, otherwise it equals e_2. The **case** expression is defined by

$$\begin{array}{l}\textbf{case } p_1 \rightarrow e_1 \Box \ldots \Box p_n \rightarrow e_n \ \triangleq \\ \quad \text{CHOOSE } \$x : (p_1 \Rightarrow (\$x = e_1)) \wedge \ldots \wedge (p_n \Rightarrow (\$x = e_n))\end{array}$$

In the **case** expression, p_n can be **other**, which is an abbreviation for $\neg(p_1 \vee \ldots \vee p_{n-1})$.

Like a quantifier expression, an **else** clause or the final arm of a **case** expression is terminated by the end of the statement or expression that contains it. This rule means that if one **case** expression appears within another, then the inner **case** encompasses as much of the expression as possible. For example

$$\textbf{case}\ p_1 \rightarrow\ a + \textbf{case}\ p_2 \rightarrow\ e_2 \ \Box\ p_3 \rightarrow\ e_3 \ \Box\ p_4 \rightarrow\ e_4$$

is parsed as

$$\textbf{case}\ p_1 \rightarrow\ a + (\textbf{case}\ p_2 \rightarrow\ e_2 \ \Box\ p_3 \rightarrow\ e_3 \ \Box\ p_4 \rightarrow\ e_4)$$

if none of the p_i are **other** expressions. If p_3 is **other**, then the only possible parsing is

$$\textbf{case}\ p_1 \rightarrow\ a + (\textbf{case}\ p_2 \rightarrow\ e_2 \ \Box\ \textbf{other} \rightarrow\ e_3) \ \Box\ p_4 \rightarrow\ e_4$$

It is good practise to enclose an inner **case** in parentheses to avoid ambiguity. (Perhaps this practise should be required by the language syntax, but at the moment it isn't.)

1.3.2 The let Construct

The **let** construct allows one to make and use local definitions within an expression. For example

$$\begin{array}{ll} \textbf{let} & x \ \triangleq\ a * c \\ & F(n) \ \triangleq\ (a+b+c)\hat{\ } n \\ \textbf{in} & x * F(1) + F(2) + F(2+x) \end{array}$$

equals

$$(a * c) * (a+b+c)\hat{\ }1 + (a+b+c)\hat{\ }2 + (a+b+c)\hat{\ }(2 + (a * c))$$

The identifiers x and F must not have a meaning in the context where the **let** expression appears.

1.3.3 Junction Lists

TLA$^+$ uses the notation for conjunctions and disjunctions explained in [2]. A list of formulas prefixed by $\wedge$ or $\vee$ denotes the conjunction or disjunction of the formulas, and indentation is used to eliminate parentheses. For example:

$$\begin{array}{l} \wedge \vee A \vee B \\ \quad \vee C \\ \wedge \vee D \\ \quad \vee E \wedge F \\ \quad \vee G \end{array}$$

equals $((A \lor B) \lor C) \land (D \lor (E \land F) \lor G)$. The following are the precise rules for parsing a bulleted list of conjuncts or disjuncts, where general parentheses pairs are ( ), { }, [ ], and $\langle\ \rangle$.

- The $\land$ or $\lor$ tokens must line up.
- Each conjunct or disjunct is terminated by either the end of the entire expression or else the first token, not appearing inside general parentheses, that appears at the same vertical position or to the left of its $\land$ or $\lor$. (In the ASCII representation, the position of a symbol is that of its first character.)

1.4 Sets

1.4.1 Infix and Prefix Operators

TLA$^+$ uses the conventional infix operators for sets: $\in$ (`\in`), $\notin$ (`\notin`), $\cap$ (`\cap`), $\cup$ (`\cup`), and $\subseteq$ (`\subseteq`). The operator $\setminus$ denotes set difference; $A \setminus B$ is the set of elements in A that are not in B.

The operators $=$ and $\neq$ (`#` or `/=`) should be used for equality and inequality of nonBoolean values. Although $=$ can be used for equality of Boolean values, we prefer to use $\equiv$ in that case. (The precedence rules encourage this use.)

In TLA$^+$, SUBSET is the powerset operator, so SUBSET S is the set of all subsets of S. Thus, $S \in$ SUBSET T is equivalent to $S \subseteq T$. We write UNION S (instead of the more conventional $\bigcup S$) for the union of all the elements of S. For example, UNION $\{S, T\}$ equals $S \cup T$, and UNION (SUBSET S) equals S, for any sets S and T. The UNION and SUBSET operators can be confusing. Until you get used to them, you may want to write comments describing typical elements of each set in each subexpression in which they appear in your specifications.

The operators $\cap$, $\cup$, $\setminus$, SUBSET, and UNION all have equal precedence, which is higher than the precedence shared by $=$, $\neq$, $\in$, and $\subseteq$. That precedence is in turn lower than the precedence of $\neg$ (the highest-precedence Boolean operator). TLA$^+$ adopts the principle that eliminating ambiguity is more important than saving keystrokes. Therefore, operators are given equal precedence if there is any reasonable doubt about their relative precedence.

1.4.2 Set Constructors

Finite sets can be enumerated explicitly with an expression of the form $\{e_1, \ldots, e_n\}$, which denotes the set whose elements are the e_i. For example, $\{1, 2, 1, 3, 2, 3\}$ is the set consisting of the three elements 1, 2, and 3. The empty set is written $\{\}$.

TLA$^+$ provides two set-constructing operators that are often confused. The first is the subset constructor $\{x \in S : p\}$, which denotes the set of all elements of S satisfying p. For example, $\{x \in Nat : x \leq 3\}$ equals $\{0, 1, 2, 3\}$, where Nat is the set of natural numbers. The second is the set-of-all constructor $\{e : x \in S\}$, which denotes the set of expressions of the form e, for all x in S. For example, $\{2 * n : n \in Nat\}$ is the set of even natural numbers. In the expressions $\{x \in S : p\}$ and $\{e : x \in S\}$, the bound variable x may not occur in S.

The subset constructor also has the form

$$\{\langle x_1, \ldots, x_n \rangle \in S : p\}$$

which is defined to equal

$$\{x \in S : \exists x_1, \ldots, x_n : (x = \langle x_1, \ldots, x_n \rangle) \land p\}$$

where the x_i are variables that may not appear in S. The set-of-all constructor has the general form

$$\{e : q_1 \in S_1, \ldots, q_n \in S_n\}$$

where the q_i and S_i are the same as for bounded quantifiers—each q_i is either a list or a tuple of variables, and none of these bound variables may appear in any of the S_i. For example,

$$\{e : x \in S, y \in T\} \triangleq \text{UNION } \{\{e : x \in S\} : y \in T\}$$

In set theory, logical inconsistencies (such as Russell's paradox) are avoided by restricting the kinds of collections that are sets. These restrictions are built into the TLA$^+$ operators, which make it impossible to build sets that are "too big". For example, the collection of all finite sets is not a set. However, the set of all finite sets of natural numbers is a set, and can be written using the TLA$^+$ operators.

1.5 Functions

1.5.1 Functions and their Domains

In conventional mathematics, a function is a set of ordered pairs satisfying certain properties. In TLA$^+$, we do not specify any particular set-theoretic representation of functions. We simply assume that there is a certain kind of set called a function that has a domain. Function application is denoted by square brackets, so $f[e]$ is the value obtained by applying f to e. The expression $f[e]$ is meaningful even if e is not in the domain of f, or if f is not a function; but we have no way of specifying what the meaning of $f[e]$ is unless f is a function and e is in its domain.

The expression DOMAIN f equals the domain of f, if f is a function. The expression $[S \rightarrow T]$ denotes the set of all functions with domain S and range a subset of T. In other words, f is in $[S \rightarrow T]$ iff it is a function with domain S such that $f[x] \in T$ for all $x \in S$.

The TLA$^+$ notation for a "lambda expression" is the function constructor $[x \in S \mapsto e]$, which equals the function f whose domain is the set S, such that $f[x]$ equals e for each x in S. For example, $[i \in Nat \mapsto i+1]$ defines the successor function on the set Nat of natural numbers. An expression f denotes a function iff f equals $[x \in \text{DOMAIN}\, f \mapsto f[x]]$, where x is a variable that does not occur in f.

1.5.2 The EXCEPT Construct

Functions If f is a function, then $[f \text{ EXCEPT } ![e_1] = e_2]$ equals the function $\widehat{f}$ that is the same as f except with $\widehat{f}[e_1] = e_2$. Formally,

$$[f \text{ EXCEPT } ![e_1] = e_2] \;\stackrel{\Delta}{=}\; [x \in \text{DOMAIN}\, f \mapsto \textbf{if } x = e_1 \textbf{ then } e_2 \textbf{ else } f[x]]$$

where x is a variable that does not occur in the expressions f, e_1, or e_2. Think of $[f \text{ EXCEPT } ![e_1] = e_2]$ as the function obtained from f by the assignment $f[e_1] := e_2$.

The character @ appearing in the expression e_2 stands for $f[e_1]$. For example, $[Fcn \text{ EXCEPT } ![3] = @ * 5]$ is the function $\widehat{f}$ that is the same as Fcn except that $\widehat{f}[3]$ equals $Fcn[3] * 5$. Think of @ as an abbreviation for the "old value" of f applied to e_1.

The EXCEPT notation is generalized in two ways. First, we let $[f \text{ EXCEPT } ![e_1]...[e_n] = e]$ be the function obtained from f by the assignment $f[e_1]...[e_n] := e$, where @ in e denotes $f[e_1]...[e_n]$. Formally, $[f \text{ EXCEPT } ![e_1]...[e_n] = e]$ equals

$$[f \text{ EXCEPT } ![e_1] = [@ \text{ EXCEPT } ![e_2] = [@ \;\ldots\; \text{EXCEPT } ![e_n] = e]\ldots]$$

Next, we let $[f \text{ EXCEPT } ![\ldots] = e_1, \;\ldots\;, \; ![\ldots] = e_n]$ be an abbreviation for

$$[\ldots[f \text{ EXCEPT } ![\ldots] = e_1] \text{ EXCEPT } \ldots \; ![\ldots] = e_n]\ldots]$$

Think of $[f \text{ EXCEPT } ![2][7] = @+1, \; ![3][5] = @+2]$ as the array obtained from f by performing the two assignments

$$f[2][7] := f[2][7] + 1 \; ; \quad f[3][5] := f[3][5] + 2$$

Sets of Functions It is also convenient to have a notation for sets of functions that differ from f only in their value on some element in their domain. We let

$$\{\![f \text{ EXCEPT } ![e] \in S]\!\}$$

be the set of all functions

$$[f \text{ EXCEPT } ![e] = x]$$

with $x \in S$ (assuming x does not appear in the expression f or S). This set is written in ASCII as

```
{|f EXCEPT ![e] \in S|}
```

There is a similar @ convention to the one described above. For example, $\{\![Fcn \text{ EXCEPT } ![3] \in \{@, @+1\}]\!\}$ is the set consisting of the functions Fcn and $[Fcn \text{ EXCEPT } ![3] = @ + 1]$.

We generalize the $\{\![f \text{ EXCEPT } ![e_1] \in e_2]\!\}$ construct in the same ways we generalized the construct $[f \text{ EXCEPT } ![e_1] = e_2]$ above. For example,

$$\{\![f \text{ EXCEPT } ![a][b] \in S,\ ![c] \in T]\!\}$$

is the set of all functions

$$[f \text{ EXCEPT } ![a][b] = s,\ ![c] = t]$$

with $s \in S$ and $t \in T$. There is also one additional abbreviation: we can write "$= e$" instead of "$\in \{e\}$". For example,

$$\{\![f \text{ EXCEPT } ![a][7] \in S,\ ![b] = g]\!\}$$

is the set

$$\{\, [f \text{ EXCEPT } ![a][7] = s,\ ![b] = g] \;:\; s \in S\}$$

1.5.3 Recursive Function Definitions

TLA$^+$ does not allow recursive definitions of operators. However, using CHOOSE, it is easy to write recursive function definitions. For example, the classic recursive definition of the factorial function on the set of natural numbers can be written as

$$Fact \;\triangleq\; \text{CHOOSE}\, f \;:\; f = [n \in Nat \mapsto \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * f[n-1]\,]$$

TLA$^+$ provides the following abbreviation for this definition:

$$Fact[n : Nat] \;\triangleq\; \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * Fact[n-1]$$

In general, $f[x : S] \triangleq e$ is an abbreviation for

$$f \triangleq \text{CHOOSE } f : f = [x \in S \mapsto e]$$

The bound variable x may not appear in S.

Although TLA$^+$ does not allow recursive operator definitions, recursive function definitions can often be used to define operators. For example, the following defines the operator $\mathit{Cardinality}$ so that if S is a finite set, then $\mathit{Cardinality}(S)$ is the number of elements in S.

$$\begin{array}{lll}\mathit{Cardinality}(S) & \triangleq & \textbf{let } F[T : \text{SUBSET } S] \triangleq \\ & & \quad\quad \textbf{if } T = \{\} \\ & & \quad\quad\quad \textbf{then } 0 \\ & & \quad\quad\quad \textbf{else } 1 + F[T \setminus \{\text{CHOOSE } t : t \in T\}] \\ & & \textbf{in } F[S]\end{array}$$

1.5.4 Functions Versus Operators

It is important to understand the distinction between an operator like $\mathit{Cardinality}$ and a function like Fact. If f is a function, then f and $f[x]$ are both expressions. Hence, both $1 + f[x]$ and $1 + f$ are syntactically legal expressions. (What they mean will depend on the definition of $+$.) If O is an operator that takes a single argument, then $O(e)$ is an expression, for any expression e. However, O by itself is not an expression. One can write $1 + O(e)$, but $1 + O$ is nonsense—it is not a syntactically valid expression. Writing "$1 + O$" makes as little sense as writing "$1 + ($" or "$C + +$".

Since functions are expressions, they are often more convenient to use than operators. However, a function has a domain, which is a set, and it is possible to specify the value of $f[x]$ only for x in its domain. One cannot define a function f such that $f[S]$ equals $\mathit{Cardinality}(S)$ for every finite set S. The domain of such a function would have to be the collection of all finite sets, and that collection is not a set.

The most common use of functions is as the values of variables. In a program written in a conventional programming language, one often declares a variable x to be an array variable indexed by some set S. In a TLA$^+$ specification, one lets x be a variable whose value is always a function with domain S.

1.6 Tuples and Functions of Multiple Arguments

In TLA$^+$, $\langle e_1, \ldots, e_n \rangle$ denotes an ordered n-tuple. TLA$^+$ provides the usual Cartesian product notation, where $S_1 \times \ldots \times S_n$ is the set of all n-tuples

$\langle e_1, \ldots, e_n \rangle$ with each e_i in S_i. Unlike conventional mathematics, TLA$^+$ defines the n-tuple $\langle e_1, \ldots, e_n \rangle$ to be the function with domain $\{1, \ldots, n\}$ that maps i to e_i. Thus, if e is a tuple with at least i components, then $e[i]$ is its ith component, for any positive integer i. In particular, the zero-tuple $\langle\,\rangle$ is the unique function with empty domain.

A function of multiple arguments is a function whose domain is a Cartesian product. We can write $f[e_1, \ldots, e_n]$ as an abbreviation for $f[\langle e_1, \ldots, e_n \rangle]$. The general form of an explicit function constructor is

$$[q_1 \in S_1, \ldots, q_n \in S_n \mapsto e]$$

where the q_i and S_i are the same as for bounded quantifiers—each q_i is either a list or a tuple of variables, and none of these bound variables may appear in any of the S_i. For example,

$$[\langle x, y \rangle \in S, z \in T \mapsto e]$$

is equivalent to

$$\begin{array}{l} [w \in S \times T \mapsto \textbf{let } \; x \triangleq w[1][1] \quad y \triangleq w[1][2] \quad z \triangleq w[2] \\ \qquad\qquad\qquad\quad \textbf{in } \; e] \end{array}$$

if w does not occur in S, T, or e.

The EXCEPT construct and recursive function definitions extend in the obvious way to functions of multiple arguments. For example, the general form of a function definition is

$$f[q_1 : S_1, \ldots, q_n : S_n] \;\triangleq\; e$$

where the q_i and S_i are as before.

1.7 Strings and Numbers

A string is written "...", as in "abc". Formally, the string "abc" is a 3-tuple, so it equals $\langle$"abc"[1], "abc"[2], "abc"[3]$\rangle$. The elements of this 3-tuple (the three letters making up the string "abc") are unspecified. Of course, "" equals $\langle\,\rangle$. The set of all strings is written STRING.

A sequence of digits is a number. Formally, the sequence 376 of digits denotes *Numeral*("376"), where *Numeral* must be a defined operator. The *Naturals* module defines *Numeral* so that *Numeral*("376") equals the number 376. (More precisely, *Numeral*("376") equals the value that represents the number 376 in the particular representation of the natural numbers defined by the *Naturals* module.) Since 376 is equivalent to writing *Numeral*("376"), it is an error to write 376 in a module where $Numeral$ is not defined—which usually means in a module that does not extend or instance $Naturals$ or some other module that defines numbers.

The $Naturals$ module also defines the set Nat of natural numbers, and the following infix operators:

$$+ \ \ - \text{(substraction)} \ \ * \text{(times)} \ \ \hat{} \text{(exponentiation)} \ \ < \ \ \leq \ \ > \ \ \geq$$

Decimal numbers are defined in terms of the $Decimal$ operator. For example, 3.14159 is an abbreviation for $Decimal$(“3”, “14159”). The $Reals$ module defines $Decimal$ appropriately.

1.8 Records

TLA$^+$ provides notation for records, which are like the records of conventional programming languages. The xyz component of a record r is written $r.xyz$. The expression $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ is the record r with components $h_1, \ldots, h_n$ such that $r.h_i$ equals e_i, for $i = 1, \ldots, n$. The expression $[h_1 : S_1, \ldots, h_n : S_n]$ is the set of all records r with components $h_1, \ldots, h_n$ such that $r.h_i$ is an element of the set S_i, for $i = 1, \ldots, n$. In other words,

$$\begin{aligned} &[h_1 : S_1, \ldots, h_n : S_n] \ \triangleq \\ &\qquad \{[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n] : e_1 \in S_1, \ldots, e_n \in S_n\} \end{aligned}$$

In both of these notations, n has to be at least one; [ ] is not a legal expression.

In TLA$^+$, a record is a function whose domain is a finite set of strings, and $r.xyz$ is an abbreviation for r[“xyz”]. Taking $!.xyz$ to be an abbreviation for $!$[“xyz”], the EXCEPT notation applies to records as well as functions. For example $[r \text{ EXCEPT } !.h[e] = @ + 1]$ is the record $\widehat{r}$ that is the same as r, except $\widehat{r}.h$ is the function that is the same as $r.h$ except $\widehat{r}.h[e]$ equals $r.h[e] + 1$.

Because records are functions, one can always write r[“foo”] instead of $r.foo$ and can use function constructors for expressing records. This can be quite useful at times.

2 Action Operators

The action operators of TLA$^+$ come from TLA. They are listed in Figure 3 on page 4. Except for “$\cdot$”, these operators are all described in [3].

TLA has two classes of variables: rigid variables and flexible variables. Rigid variables are called *constants* in TLA$^+$. They are the variables of ordinary predicate logic. The bound variables introduced by the constant operators of Section 1 are rigid variables. Flexible variables are called simply *variables* in TLA$^+$.

An *action* is a Boolean formula that may contain primed and unprimed flexible variables. In an action, an unprimed instance of a variable denotes its value in "the current state", and a primed instance denotes its value in "the next state". For action reasoning, x and x' can be considered to be completely unrelated variables.

The action operators of TLA$^+$ can be defined in terms of primed variables as follows. (These definitions are applied from the inside out, replacing the arguments of an operator by their definitions and then replacing the operator by its definition.)

p' Equals p with every flexible variable x replaced by its primed version x'.

$[A]_e$ Equals $A \lor (e' = e)$.

$\langle A \rangle_e$ Equals $A \land (e' \neq e)$.

ENABLED A Let $x_1, \ldots, x_n$ be the flexible variables that occur primed in A, let $\widehat{x_1}, \ldots, \widehat{x_n}$ be new rigid variables that do not occur in A, and let $\widehat{A}$ be the formula obtained from A by replacing each occurrence of x'_i by $\widehat{x_i}$, for $i = 1, \ldots n$. Then ENABLED A equals $\exists\, \widehat{x_1}, \ldots, \widehat{x_n} : \widehat{A}$. For example

$$\text{ENABLED}\,(x' * x = y' * z) \;\triangleq\; \exists\, \widehat{x}, \widehat{y} : \widehat{x} * x = \widehat{y} * z$$

UNCHANGED e Equals $e' = e$.

$A \cdot B$ Equals the composition of the actions A and B. Intuitively, it is the action that first does A then B. More formally, let $x_1, \ldots, x_n$ be all the flexible variables that occur primed in A or unprimed in B, let $\widehat{x_1}, \ldots, \widehat{x_n}$ be new rigid variables that do not occur in A or B, let $\widehat{A}$ equal A with each x'_i replaced by $\widehat{x_i}$, and $\widetilde{B}$ equal B with each x_i replaced by $\widehat{x_i}$. Then $A \cdot B$ equals $\exists\, \widehat{x_1}, \ldots, \widehat{x_n} : \widehat{A} \land \widetilde{B}$. For example,

$$\begin{pmatrix} \land\, x' + y - x = z \\ \land\, z > y' + 7 \end{pmatrix} \cdot \begin{pmatrix} \land\, z' - x * y' \in w' \\ \land\, y \notin w \\ \land\, x' > y + z \end{pmatrix} \;\triangleq$$
$$\exists\, \widehat{w}, \widehat{x}, \widehat{y}, \widehat{z} : \begin{pmatrix} \land\, \widehat{x} + y - x = z \\ \land\, z > \widehat{y} + 7 \end{pmatrix} \land \begin{pmatrix} \land\, z' - \widehat{x} * y' \in w' \\ \land\, \widehat{y} \notin \widehat{w} \\ \land\, x' > \widehat{y} + \widehat{z} \end{pmatrix}$$

(The TLA$^+$ convention of using bulleted lists of conjuncts and disjuncts is explained in Section 1.3.)

3 Temporal Operators

A temporal formula is one that is true or false of a behavior, which is an infinite sequence of states. The TLA operators for writing temporal formulas are listed in Figure 4 on page 4. The temporal quantifiers have the same generalizations as the unbounded predicate-logic quantifiers—for example, $\exists\, x_1, \ldots, x_n : F$.

The temporal operators of TLA$^+$ are all described in [3] except for $\stackrel{+}{\rhd}$, which is introduced in [1]. (This operator is used only for assumption/guarantee specifications.) They are defined in terms of $\Box$, and $\exists$ as follows.

$\Diamond F$ Equals $\neg\Box\neg F$.

$\mathrm{WF}_e(A)$ Equals $(\Box\Diamond\neg\textsc{Enabled}\ \langle A\rangle_e) \lor (\Box\Diamond\langle A\rangle_e)$.

$\mathrm{SF}_e(A)$ Equals $(\Diamond\Box\neg\textsc{Enabled}\ \langle A\rangle_e) \lor (\Box\Diamond\langle A\rangle_e)$.

$F \leadsto G$ Equals $\Box(F \Rightarrow \Diamond G)$.

$\forall\, x : F$ Equals $\neg\exists\, x : \neg F$.

$F \stackrel{+}{\rhd} G$ This formula asserts that G remains true at least one step longer than F does. (And G remains true forever if F does.) A formula H is true for the first k steps of a behavior iff there is some way to continue those steps to a complete behavior for which H is true. This leads to the following formal definition of $F \stackrel{+}{\rhd} G$.

Let $\mathbf{x}$ be the tuple $\langle x_1, \ldots, x_n\rangle$ of all flexible variables that occur free in F or G; let $\widehat{\mathbf{x}}$ be an n-tuple of flexible variables $\langle \widehat{x_1}, \ldots, \widehat{x_n}\rangle$ distinct from any variables appearing (bound or free) in F or G; let $\widehat{F}$ and $\widehat{G}$ be the formulas obtained from F and G by substituting the variables $\widehat{x_i}$ for the corresponding variables x_i, and let b be a flexible variable distinct from the x_i and $\widehat{x_i}$. Then $F \stackrel{+}{\rhd} G$ is defined to equal

$$\begin{array}{l} \forall\, b : \land\ (b = \text{TRUE}) \land \Box[b' = \text{FALSE}]_b \\ \qquad \land\ \exists\, \widehat{x_1}, \ldots, \widehat{x_n} : \widehat{F} \land \Box(b \Rightarrow (\mathbf{x} = \widehat{\mathbf{x}})) \\ \qquad \Rightarrow\ \exists\, \widehat{x_1}, \ldots, \widehat{x_n} : \widehat{G} \land (\mathbf{x} = \widehat{\mathbf{x}}) \land \Box[b \Rightarrow (\mathbf{x}' = \widehat{\mathbf{x}}')]_{\langle b, \mathbf{x}, \widehat{\mathbf{x}}\rangle} \end{array}$$

References

[1] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[2] Leslie Lamport. How to write a long formula. *Formal Aspects of Computing*, 6:580–584, 1994. First appeared as Research Report 119, Digital Equipment Corporation, Systems Research Center.

[3] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[4] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach, New York, 1969.

Index