PlusCal / TLA$^+$: An Annotated Cheat Sheet

Stephan Merz
stephan.merz@loria.fr

Abstract

This document is intended to summarize the main constructs that a user of PlusCal and TLA$^+$ who is using the Toolbox and TLC is likely to encounter. It does not replace the available introductory material about TLA$^+$ and is also not meant as a reference manual. Feedback to the author is welcome.

1 PlusCal

PlusCal is an algorithmic language that has the "look and feel" of imperative pseudo-code for describing concurrent algorithms. It has a formal semantics, through translation to TLA$^+$, and algorithms can be verified using the TLA$^+$ tools. We describe here the "C syntax" of PlusCal, but there is also a "P syntax" closer to the Pascal programming language.

A PlusCal algorithm appears inside a comment within a TLA$^+$ module. Its top-level syntax is as follows.

```
--algorithm name {
  (* declaration of global variables *)
  (* operator definitions *)
  (* macro definitions *)
  (* procedures *)
  (* algorithm body or process declarations *)
}
```

Variable declarations, operator and macro definitions, as well as procedures are optional. There must either be an algorithm body (for a sequential algorithm) or process declarations (for a concurrent algorithm).

The PlusCal translator embeds the TLA$^+$ specification corresponding to the PlusCal algorithm in the module within which the algorithm appears, immediately following the algorithm. The translation is delimited by the lines

```
\* BEGIN TRANSLATION
...
\* END TRANSLATION
```

Users should not touch this area, as it will be overwritten whenever the PlusCal translator is invoked. However, TLA$^+$ definitions may appear either before the PlusCal algorithm or after the generated TLA$^+$ specification. In particular, operators defined before the PlusCal algorithm may be used in the algorithm. Correctness properties are defined below the generated specification because they refer to the variables declared by the translator.

Variable declarations. Variables are declared as follows, they may (but need not) be initialized:

```
variables x, y=0, z \in {1,2,3};
```

Note that there may be at most one "variables" clause, but that it may declare any number of variables. This example declares three variables x, y, and z. The variable x is not initialized, y is initialized to 0, and z may take either 1, 2 or 3 as initial values. During model checking, TLC will assign x a default value that is different from all ordinary values and will explore all three initial values for z.

Operator definitions. As in TLA$^+$ (see below), operators represent utility functions for describing the algorithm. The syntax is the same as for TLA$^+$ definitions, explained below. For example,

```
define {
  Cons(x,s) == << x >> \o s
  Front(s)  == [i \in 1 .. Len(s)-1 |-> s[i]]
}
```

declares an operator *Cons* that prepends an element x to a sequence s and an operator *Front* that returns the subsequence of a non-empty sequence without the last element. Again, there can be a single "define" clause, but it may contain any number of operator definitions, without any separator symbol.

Macros. A macro represents several PlusCal statements that can be inserted into an algorithm, and it may have parameters. In contrast to a defined operator, a macro need not be purely functional but may have side effects. In contrast to a procedure, it cannot be recursive and may not contain labels or complex statements such as loops or procedure calls or return statements. Here are two examples:

```
macro send(to, msg) {
  network[to] := Append(@, msg)
}
macro rcv(p, var) {
  await Len(network[p]) > 0;
  var := Head(network[p]);
  network[p] := Tail(@)
}
```

These macros represent send and receive operations in a network represented by FIFO queues indexed by processes. The first macro appends a message to the queue of the destination process. The second macro blocks until the queue of process p contains at least a message, then assigns the first message in the queue to variable *var* and removes it from that queue. When using a macro, simply insert an invocation as a statement in the code, such as `send(p, x+1)` for sending the value $x+1$ to process p.

Procedures. A procedure declaration is similar to that of a macro, but it may also contain the declaration of local variables.[1] The procedure body may contain arbitrary statements, including procedure calls (even recursive calls).

```
procedure p(x,y)
    variable z=x+1  \* procedure-local variable
{
    call q(z,y);
l:  y := y+1;
    return
}
```

Procedure parameters are treated as variables and may be assigned to. Any control flow in a procedure should reach a return statement, any result computed by the procedure should be stored in a variable—possibly a parameter. Procedure calls are preceded by the keyword call and must be followed by a label.

Since the PlusCal translator introduces a stack for handling procedures, a PlusCal algorithm using procedures must appear in a module EXTENDing the standard module `Sequences`.

[1] These variables may be initialized using the form $x = v$, but not $x \in S$.

Processes. A PlusCal algorithm may declare one or more process templates, each of which can have several instances. A process template looks as follows:

```
process (name [= | \in] e)
   variables ...   \* process-local variables
{
   (* process body *)
}
```

The form "*name* = *e*" will give rise to one instance of the process whose process identity is (the value of expression) *e*, whereas "*name* ∈ *e*" will create one process instance per element of the set *e*. When creating several process instances from different templates, it is important to ensure that all process identities are comparable and different: for example, use only integers, only strings or only model values. Within the process body, the process identity is referred to as `self` (and not as `name` as the syntax might suggest). Processes conceptually execute (asynchronously) in parallel, from the start of execution of the algorithms: PlusCal does not support dynamically spawning processes during execution.

PlusCal statements. The statements of PlusCal are those expected of a simple imperative language, with the addition of primitives for synchronization and for non-deterministic choice that are useful for specifications.

Assignments are written `x := e`, but PlusCal also supports assignments to components of an array `x[i,j] := e` whose translation involves EXCEPT forms in TLA$^+$. PlusCal also has a statement `print e` for printing the value of an expression.[2] However, due to the breadth-first state enumeration strategy used by default in TLC, it may not always be obvious to understand the relationship between outputs to the console and actual execution flow.

The conditional statement has the usual form

```
if (exp) ... else ...
```

where the "else" branch is optional; compound statements are enclosed in braces. Similarly, while loops are written

```
while (exp) ...
```

Other statements for transferring the flow of control are procedure call and return (see above), as well as `goto l` for jumping to label *l*—see below for a more detailed explanation of labels in PlusCal.

[2] Use of this statement requires EXTENDing the standard module TLC.

Assertions can be embedded in PlusCal in the form `assert e`. This statement has no effect if the predicate *e* is true and otherwise aborts (producing a counter-example trace in TLC). There is also `skip`, which does nothing and is equivalent to `assert TRUE`.

The statement `await e` (which can alternatively be written `when e`) blocks execution until the condition is true. This is useful for synchronizing parallel processes, for example for blocking message reception until a message has actually been received.

Finally, PlusCal offers two statements for modeling non-determinism. The first form,

```
either \* first choice
or     \* second choice
or     \* ...
```

is useful for expressing the choice between a fixed number of alternatives. A useful idiom for restricting this choice is to add `await` statements to branches: only those choices whose condition evaluate to `TRUE` can indeed be executed. (This is PlusCal's version of Dijkstra's guarded commands.)

Second, the form

```
with (x \in S) ...
```

allows for choices based on the elements of set *S*: the variable *x* acts as a local variable whose scope is restricted to the body of the `with` statement. For example, in a model where communication is not necessarily FIFO, *S* could be the set of messages that a process has received, and a `with` statement could be used to handle any one of these messages.

Semicolons. Unlike in C and similar languages, PlusCal uses semicolons to separate (and not end) statements. In particular, this means that no semicolon is required before the closing brace of a compound statement (although it is not considered as a syntax error). However, a semicolon is required *after* the closing brace if it is followed by another statement.

Labels. PlusCal statements can be labeled, and the primary purpose of labels is to indicate the "grain of atomicity" in the execution of parallel processes. The idea is that a group of statements that appears between two labels is executed without interruption by any other process. However, the PlusCal translator imposes certain labeling rules. The most important ones are:

- The first statement in the body of a procedure, a process or of the algorithm body must be labeled.
- A `while` statement must be labeled.
- Any statement following a `call`, a `return` or an `if` or `either` statement that contains a labeled statement must be labeled.
- A macro body and the body of a `with` statement may not contain labels.
- Any two assignments to the same variable (including to two components of the same array) must be separated by a label.

The full set of labeling rules is given in the PlusCal manual. The translator prints information about missing labels, it may also be run with the `-label` option to automatically add missing labels.

Specifying Fairness. Fairness conditions require that statements that are "often enough" enabled must eventually be executed; they are necessary in order to ensure that an algorithm satisfies any liveness property. In PlusCal, fairness conditions can be attached to the algorithm, process templates or labels. By writing `fair algorithm`, one requires that some statement (for some process) must eventually be executed if some statement is enabled. This is a weak condition: in particular, some process may never execute although it is always enabled, if other processes are also always enabled. Writing `fair process` ensures that the process (more precisely, each instance of the process template) will eventually execute if it remains enabled. The even stronger condition `fair+ process` requires strong fairness for the process, i.e. it will eventually execute if it is infinitely often enabled—even if it is also infinitely often disabled. For example, a process that requires a semaphore that is infinitely often free will eventually acquire it under strong fairness, even in the presence of competing processes.

The overall fairness requirement attached to a process can be modulated for individual actions (corresponding to a group of statements introduced by a label). Writing `l:-` instead of `l:` suppresses the fairness assumption for the group of statements introduced by label `l`. For example, in a mutual-exclusion algorithm, one may not want to assume any fairness condition for the non-critical section. On the contrary, writing `l:+` assumes strong fairness for that group of statements. (This has no effect for a strongly fair process.) For example, one may in this way require strong fairness only for acquiring a semaphore within a weakly fair process.

Even finer-grained fairness conditions can be expressed in TLA$^+$, details can be found in the literature on TLA$^+$.

2 TLA$^+$

PlusCal algorithms are translated into the TLA$^+$ specification language, and the TLA$^+$ tools (in particular the TLA$^+$ Toolbox and the model checker TLC) are used for verifying properties of algorithms. TLA$^+$ is based on ordinary mathematical set theory, and it is untyped. In order to effectively use PlusCal, you need to know at least some TLA$^+$ syntax:

- all expressions that appear in PlusCal algorithms are written in TLA$^+$, and
- properties to be verified (beyond assertions inserted into PlusCal code) are also written in TLA$^+$.

TLA$^+$ has ASCII syntax but can be pretty-printed from the Toolbox, using standard mathematical notation. We show the pretty-printed versions as well as the ASCII source below.

2.1 Overall structure

Modules and declarations. TLA$^+$ specifications are structured in *modules*. A TLA$^+$ module begins with a header line

```
---------- MODULE Name ----------
```

(at least four '-' signs) and ends with a bottom line

```
==================================
```

(at least four '=' signs). A module may import the contents of other modules by EXTENDing them, which is basically equivalent to copying the content of the extended module into the extending module. It may also create INSTANCEs of other modules whose (constant or variable) parameters are instantiated by expressions of the instantiating module. This is useful for example when showing that a lower-level specification refines (or implements) a higher-level one.

Modules usually declare constant or variable parameters,[3] for example

CONSTANTS *Node*, *Edge*
VARIABLES *leader network*

for declaring constant sets representing the nodes and edges of a graph and two variables used in a leader-election algorithm. (When the algorithm is written in PlusCal, the variable declaration is generated by the PlusCal translator.)

[3]The values of constant parameters remain fixed throughout any execution of the algorithm. TLA$^+$ variables are analogous to program variables and assume different values at different states of the execution.

Since TLA$^+$ is untyped, these declarations do not indicate the types of constants or variables. (Semantically, all TLA$^+$ values are sets.) However, a specification may contain assumptions on the constant parameters such as

ASSUME $Edge \subseteq Node \times Node$

and these assumptions are checked by TLC. Properties of variables are expressed using formulas, typically invariants or temporal logic properties.

Operator definitions. The bulk of a TLA$^+$ module consists in operator definitions. Operators represent subformulas of the overall specification, including useful operations on data as well as initial conditions and possible transitions of algorithms. Again, when using PlusCal, most definitions are generated by the PlusCal translator. However, a PlusCal algorithm may use operators defined in the TLA$^+$ module (or in some extended module), and correctness properties of a PlusCal algorithm are usually specified in the part of the TLA$^+$ module below the generated TLA$^+$ translation.

An operator definition has the form[4]

$$op(p_1, \ldots, p_n) \triangleq e$$

where $p_1, \ldots, p_n$ are parameters of the definition and e is a TLA$^+$ expression that represents the body of the definition and does not contain op. Each parameter p_i is either an identifier or of the form $f(\_, \ldots, \_)$. The latter case represents an operator parameter, and the number of underscores indicates the arity of the operator, i.e. the number of arguments that it expects. For example,

$$Apply(f(\_), x) \triangleq f(x)$$

takes a unary operator parameter f and an ordinary parameter x.

TLA$^+$ enforces the general rule that a name that already exists in the current context may not be reused. The above definition of *Apply* would therefore be illegal in a context where f or x have already been introduced as constant or variable parameters or as operator names. However, the scope of a parameter symbol is limited to the definition in which it appears, and it is therefore legal to reuse x as a parameter name in a subsequent definition.

Recursive operators. For the special case of a function definition, one may write

$$f[x \in S] \triangleq e$$

[4]The symbol $\triangleq$ is written == in ASCII.

in order to define a function with domain S that maps every $x \in S$ to the expression e (which may contain x). In this case, e may contain the symbol f. For example, one may write

$$fact[n \in Nat] \triangleq \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1]$$

for defining the factorial function over natural numbers. TLA$^+$ also allows for recursive operator definitions; in this case, the names and arities of recursive operators have to be declared before the definitions, as in

$$\text{RECURSIVE } Fact(\_)$$
$$Fact(n) \triangleq \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * Fact(n-1)$$

which defines a recursive *operator* (rather than a function) *Fact*; note in particular that *Fact* does not have a domain. Mutually recursive operators may be introduced by declaring all operator names and arities before their definitions.

Theorems. Finally, TLA$^+$ modules may assert theorems and contain proofs of these theorems. Proofs are checked by the TLA$^+$ Proof System (TLAPS), an interactive proof assistant for TLA$^+$. Theorems are ignored by TLC: instead, properties to be verified by model checking need to be entered in the TLC pane of the TLA$^+$ Toolbox. We will therefore not describe the syntax of theorems and proofs here.

2.2 Logic

$\wedge$, $\vee$, $\Rightarrow$, $\equiv$, $\neg$ `/\`, `\/`, `=>`, `<=>`, `~`

are the standard operators of propositional logic (conjunction, disjunction, implication, equivalence, and negation).

$=$, $\neq$ `=`, `#`

denote equality and disequality (the latter may also be written `~=`).

TRUE, FALSE `TRUE`, `FALSE`

are the logical constants true and false.

$\forall x : P(x)$, $\exists x : P(x)$ `\A x : P(x)`, `\E x : P(x)`

represent the logical quantifiers "forall" and "exists". You may not reuse a variable that is already used in the current scope. In other words, if x has already been introduced (as a constant, variable, operator or parameter of the current operator definition) then it is illegal to write $\forall x : P(x)$, but you have to write $\forall y : P(y)$ for some fresh variable y. There exist "bounded versions" of the quantifiers restricted to a set S and written as $[\forall|\exists]x \in S : P(x)$, and these are the only forms that TLC can evaluate.

CHOOSE $x : P(x)$ `CHOOSE x : P(x)`

denotes some arbitrary but fixed value x such that $P(x)$ is true, and some unspecified fixed value otherwise. Again, there is a bounded version that restricts the choice of possible values to some set S, and this is the only form accepted by TLC. In mathematical terminology, this operator is known as "Hilbert's ε-operator". It is important to understand that this represents deterministic choice: multiple evaluations of the expression will yield the same result. CHOOSE-expressions are most useful if there is a unique value satisfying the predicate. For example, the length of a sequence s can be defined as

$$\text{CHOOSE } n \in Nat : \text{DOMAIN } s = 1..n$$

Another useful paradigm is extending some set S by a "null value":[5]

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

The set $S \cup \{notAnS\}$ is then similar to an option type in functional programming.

2.3 Sets

$\{e_1, \ldots, e_n\}$ `{e1, ..., en}`

denotes the set containing (the values corresponding to) the expressions e_1, $\ldots$, e_n. In particular, $\{\}$ is the empty set.

$x \in S$, $x \notin S$, $S \subseteq T$ `x \in S`, `x \notin S`, `S \subseteq T`

is true if x is an element (or is not an element) of set S, respectively if S is a subset of T.

$\cup$, $\cap$, $\setminus$ `\cup` (or `\union`), `\cap` (or `\inter`), `\`

set union, intersection, and difference.

UNION S `UNION S`

generalized set union: S is expected to be a set of sets, and the result is the union of these sets. For example, UNION $\{\{1,2\},\{3,4\}\} = \{1,2,3,4\}$, and more generally, UNION $\{A,B\} = A \cup B$.

[5] The astute reader will notice that this form cannot be evaluated by TLC. However, the TLA$^+$ Toolbox will substitute a special "model value" for *notAnS*.

SUBSET S `SUBSET S`

denotes the set of all subsets of S. For example,

$$\text{SUBSET } \{1,2\} = \{\{\},\{1\},\{2\},\{1,2\}\}$$

$\{x \in S : P(x)\}$ `{x \in S : P(x)}`

denotes the subset of elements of S for which the predicate $P(x)$ is true. For example, assuming that $isPrime(n)$ is true for a natural number n if n is prime then $\{n \in 1..100 : isPrime(n)\}$ contains the set of prime numbers in the interval between 1 and 100.

$\{e(x) : x \in S\}$ `{e(x) : x \in S}`

denotes the set of values obtained by applying the operator e to all elements of S. This construction is similar to the `map` operator from functional programming. For example, $\{2*n+1 : n \in 0..99\}$ denotes the set of the first 100 odd natural numbers.

$Cardinality(S)$ `Cardinality(S)`

the number of elements of the finite set S. This operator is defined in the module `FiniteSets`, which you must EXTEND in order to use it. That module also defines the predicate *IsFiniteSet* for checking if a set is finite.

2.4 Functions

TLA$^+$ functions are total over their domain. Although every function is a set (just like any TLA$^+$ value), we don't know what set the function denotes and we think of functions as a primitive class of values. (For nerds: in many presentations of set theory, functions are sets of pairs. This is not enforced in TLA$^+$.) In terms of programming terminology, functions are analogous to arrays—although their index set or domain need not be finite—and TLA$^+$ uses array syntax for functions.

$[x \in S \mapsto e(x)]$ `[x \in S |-> e(x)]`

denotes the function with domain S that maps every element x of S to $e(x)$. For example, $[n \in Nat \mapsto n+1]$ is the successor function on natural numbers. This expression is similar to a λ-expression in functional programming.

DOMAIN f `DOMAIN f`

denotes the domain of the function f. Although TLA$^+$ has no standard notation for the range of a function f, it can easily be defined as

$$\{f[x] : x \in \text{DOMAIN } f\}$$

$f[x]$ `f[x]`

denotes the result of applying the function f to the argument x. This expression only makes sense if $x \in \text{DOMAIN } f$.

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

denotes the function that has the same domain as f and acts similarly as f, except that it maps the argument x to e. For example, if *succ* is the successor function from above, then $[succ \text{ EXCEPT } ![0] = 42]$ is the function that maps every natural number to its successor, but that maps 0 to 42. Within the expression e, one may use the symbol @ in order to refer to $f[x]$. For example,

$$[count \text{ EXCEPT } ![p] = @ + 1]$$

updates function *count* by incrementing $count[p]$ by 1.

Updating a function at several arguments can be written as

$$[f \text{ EXCEPT } ![x] = a,\ ![y] = b,\ ![z] = c]$$

$[S \to T]$ `[S -> T]`

denotes the set of all functions whose domain is S and such that $f[x] \in T$ holds for all $x \in S$.

For functions that take several arguments, TLA$^+$ adopts the convention that $f[x,y]$ is shorthand for $f[\langle x,y \rangle]$ and hence $f \in [X \times Y \to Z]$. This convention extends to EXCEPT expressions, so you can write $[f \text{ EXCEPT } ![x,y] = e]$.

2.5 Records

A TLA$^+$ record is a function from a finite set of fields (represented as strings) to values. In principle, standard function operations therefore apply to records. Nevertheless, TLA$^+$ introduces some specific syntax for records.

$[fld_1 \mapsto e_1, \ldots, fld_n \mapsto e_n]$ `[fld1 |-> e1, ..., fldn |-> en]`

constructs a record with fields fld_i holding values e_i. For example,

$$[kind \mapsto \text{"request"},\ sndr \mapsto p,\ clk \mapsto 42]$$

could represent a request message sent by process p with logical clock value 42.

r.fld `r.fld`

selects the value of field *fld* from record *r*.

$[r$ EXCEPT $!.fld = e]$ `[r EXCEPT !.fld = e]`

denotes the record that is similar to *r*, except that field *fld* holds value *e*. (The indicated field should exist in record *r*.) An analogous generalization to the EXCEPT construct for functions allows several record fields to be updated.

2.6 Numbers

You must import the arithmetical operators by EXTENDing one of the standard modules `Naturals` or `Integers`. (TLA$^+$ also has a standard module `Reals` for real numbers, but we won't need it.)

Nat, *Int* `Nat, Int`

denote the set of natural numbers $(0, 1, \ldots)$ and of integers.

$i..j$ `i .. j`

an interval of integers, such as `-3 .. 5`.

$+$, $-$, $*$, $\div$, *mod* `+, -, *, \div, %`

are the standard arithmetical operations (addition, subtraction, multiplication, integer division and modulus).

2.7 Tuples and sequences

Tuples and sequences are the same mathematical objects, but they are used differently: whereas a tuple has a fixed number of components, a sequence can be of varying length. In TLA$^+$, tuples and sequences are represented as functions with domain $1..n$, for some natural number n. (In particular, the empty sequence has domain $1..0$.) Standard function operations therefore apply, and the i-th element of a sequence s is obtained as $s[i]$. The sequence operations must be imported by EXTENDing the standard module `Sequences`.

$\langle e_1, \ldots, e_n \rangle$ `<<e1, ..., en>>`

denotes the sequence with elements e_1, ..., e_n, in that order. In particular, `<< >>` is the empty sequence.

$s \circ t$ `s \o t`

denotes the concatenation of the sequences s and t. For example, $\langle 1,2 \rangle \circ \langle 3,4 \rangle = \langle 1,2,3,4 \rangle$.

$Seq(S)$ `Seq(S)`

the set of all finite sequences with elements in set S.

$S \times T$ `S \X T`

denotes the set product of S and T, i.e. the set of all pairs $\langle s,t \rangle$ with $s \in S$ and $t \in T$.

$Len(s)$ `Len(s)`

denotes the length of sequence s.

$Append(s,e)$ `Append(s,e)`

adds element e at the end of the sequence s.

$Head(s)$, $Tail(s)$ `Head(s), Tail(s)`

denote the first element of the sequence s, respectively the rest of the sequence with the first element removed. These operators are well-defined only if the sequence s is non-empty.

$SubSeq(s,m,n)$ `SubSeq(s,m,n)`

the sequence $\langle s[m], s[m+1], \ldots, s[n] \rangle$.

$SelectSeq(s,P(\_))$ `SelectSeq(s, P(_))`

denotes the subsequence of elements of sequence s that satisfy predicate P. For example, if $isPrime(n)$ is true if n is a prime number, then

$$SelectSeq(\langle 2,3,4,5,6,7,8 \rangle, isPrime) = \langle 2,3,5,7 \rangle$$

Many more operators on sequences can easily be defined in TLA$^+$. For example, the "map" operation on sequences is defined as

$$Map(s, f(\_)) \triangleq [i \in \text{DOMAIN } s \mapsto f(s[i])]$$

Other operations require recursive definitions, such as a "reduce" operation that applies an operation op to all elements of the sequence s starting from the "null value" e:

$$\begin{array}{l} \text{RECURSIVE } Reduce(\_,\_,\_) \\ Reduce(s, op(\_,\_), e) \triangleq \text{IF } s = \langle\rangle \text{ THEN } e \\ \qquad\qquad\qquad\qquad\quad \text{ELSE } op(Head(s), Reduce(Tail(s), op, e)) \end{array}$$

Character strings are represented as sequences of characters. How characters are represented is left unspecified; literal strings are written as `"string"`. However, TLC handles strings natively and does not support sequence operations applied to strings.

2.8 Miscellaneous constructs

IF P THEN t ELSE e `IF P THEN t ELSE e`

is a conditional expression. It can be used as a formula if t and e are both formulas, but also more generally as a term. See the definition of the factorial function earlier for an example.

CASE $p_1 \rightarrow e_1 \Box p_2 \rightarrow e_2 \ldots \Box$ OTHER $\rightarrow e$
`CASE p1 -> e1 [] p2 -> e2 ... [] OTHER -> e`

generalizes conditional expressions to more than two branches. It equals some e_i such that p_i is true, or e if no guard p_i is true (The OTHER branch is optional.) In case several p_i are true, the choice of which branch is evaluated is fixed but unspecified: make sure that in this case, all corresponding e_i evaluate to the same value.

LET $x \triangleq t$ IN e `LET x == t IN e`

allows users to introduce local definitions, similar to the standard "let" construct of functional programming languages.

3 Online Resources

- `learntla.com`: A Web site for helping newcomers learn PlusCal and TLA$^+$. There is also a published book ("Practical TLA$^+$") by the same author with a more complete introduction.
- `http://lamport.azurewebsites.net/tla/tla.html`: The official TLA$^+$ Web site with lots of reference material.
- `http://lamport.azurewebsites.net/video/videos.html`: A video course on TLA$^+$ by Leslie Lamport (does not cover PlusCal, though).
- `http://lamport.azurewebsites.net/tla/c-manual.pdf`: The PlusCal manual for the C syntax that is also used in the lectures.