2. a finite set of comparable TLC values (*comparable* is defined below), or
3. a function f whose domain is a TLC value such that $f[x]$ is a TLC value, for all x in DOMAIN f.

For example, the first two rules imply that

(14.3) $\{\{\text{“a”}, \text{“b”}\}, \{\text{“b”}, \text{“c”}\}, \{\text{“c”}, \text{“d”}\}\}$

is a TLC value because rules 1 and 2 imply that $\{\text{“a”}, \text{“b”}\}$, $\{\text{“b”}, \text{“c”}\}$, and $\{\text{“c”}, \text{“d”}\}$ are TLC values, and the second rule then implies that (14.3) is a TLC value. Since tuples and records are functions, rule 3 implies that a record or tuple whose components are TLC values is a TLC value. For example, $\langle 1, \text{“a”}, 2, \text{“b”} \rangle$ is a TLC value.

To complete the definition of what a TLC value is, I must explain what *comparable* means in rule 2. The basic idea is that two values should be comparable iff the semantics of TLA$^+$ determines whether or not they are equal. For example, strings and numbers are not comparable because the semantics of TLA$^+$ doesn't tell us whether or not “abc” equals 42. The set $\{\text{“abc”}, 42\}$ is therefore not a TLC value; rule 2 doesn't apply because “abc” and 42 are not comparable. On the other hand, $\{\text{“abc”}\}$ and $\{4, 2\}$ are comparable because sets having different numbers of elements must be unequal. Hence, the two-element set $\{\{\text{“abc”}\}, \{4, 2\}\}$ is a TLC value. TLC considers a model value to be comparable to, and unequal to, any other value. The precise rules for comparability are given in Section 14.7.2.

14.2.2 How TLC Evaluates Expressions

Checking a specification requires evaluating expressions. For example, TLC does invariance checking by evaluating the invariant in each reachable state—that is, computing its TLC value, which should be TRUE. To understand what TLC can and cannot do, you have to know how it evaluates expressions.

TLC evaluates expressions in a straightforward way, generally evaluating subexpressions “from left to right”. In particular:

- It evaluates $p \land q$ by first evaluating p and, if it equals TRUE, then evaluating q.
- It evaluates $p \lor q$ by first evaluating p and, if it equals FALSE, then evaluating q. It evaluates $p \Rightarrow q$ as $\neg p \lor q$.
- It evaluates IF p THEN e_1 ELSE e_2 by first evaluating p, then evaluating either e_1 or e_2.

To understand the significance of these rules, let's consider a simple example. TLC cannot evaluate the expression $x[1]$ if x equals $\langle \rangle$, since $\langle \rangle[1]$ is silly. (The empty sequence $\langle\,\rangle$ is a function whose domain is the empty set and hence does not contain 1.) The first rule implies that, if x equals $\langle\,\rangle$, then TLC can evaluate the formula

$$(x \neq \langle\,\rangle) \land (x[1] = 0)$$

but not the (logically equivalent) formula

$$(x[1] = 0) \land (x \neq \langle\,\rangle)$$

(When evaluating the latter formula, TLC first tries to compute $\langle\,\rangle[1] = 0$, reporting an error because it can't.) Fortunately, we naturally write the first formula rather than the second because it's easier to understand. People understand a formula by "mentally evaluating" it from left to right, much the way TLC does.

TLC evaluates $\exists\, x \in S : p$ by enumerating the elements $s_1, \ldots, s_n$ of S in some order and then evaluating p with s_i substituted for x, successively for $i = 1, \ldots, n$. It enumerates the elements of a set S in a very straightforward way, and it gives up and declares an error if the set is not obviously finite. For example, it can obviously enumerate the elements of $\{0, 1, 2, 3\}$ and $0 \,..\, 3$. It enumerates a set of the form $\{x \in S : p\}$ by first enumerating S, so it can enumerate $\{i \in 0 \,..\, 5 : i < 4\}$ but not $\{i \in \mathit{Nat} : i < 4\}$.

TLC evaluates the expressions $\forall\, x \in S : p$ and CHOOSE $x \in S : p$ by first enumerating the elements of S, much the same way as it evaluates $\exists\, x \in S : p$. The semantics of TLA$^+$ states that CHOOSE $x \in S : p$ is an arbitrary value if there is no x in S for which p is true. However, this case almost always arises because of a mistake, so TLC treats it as an error. Note that evaluating the expression

$$\text{IF } n > 5 \text{ THEN CHOOSE } i \in 1 \,..\, n : i > 5 \text{ ELSE } 42$$

will not produce an error because TLC will not evaluate the CHOOSE expression if $n \leq 5$. (TLC would report an error if it tried to evaluate the CHOOSE expression when $n \leq 5$.)

TLC cannot evaluate "unbounded" quantifiers or CHOOSE expressions—that is, expressions having one of the forms

$$\exists\, x : p \qquad \forall\, x : p \qquad \text{CHOOSE } x : p$$

TLC cannot evaluate any expression whose value is not a TLC value, as defined in Section 14.2.1 above. In particular, TLC can evaluate a set-valued expression only if that expression equals a finite set, and it can evaluate a function-valued expression only if that expression equals a function whose domain is a finite set. TLC will evaluate expressions of the following forms only if it can enumerate the set S:

$$\begin{array}{lll}
\exists\, x \in S : p & \forall\, x \in S : p & \text{CHOOSE } x \in S : p \\
\{x \in S : p\} & \{e : x \in S\} & [x \in S \mapsto e] \\
\text{SUBSET } S & \text{UNION } S &
\end{array}$$

TLC can often evaluate an expression even when it can't evaluate all subexpressions. For example, it can evaluate

$$[n \in Nat \mapsto n * (n+1)][3]$$

which equals the TLC value 12, even though it can't evaluate

$$[n \in Nat \mapsto n * (n+1)]$$

which equals a function whose domain is the set *Nat*. (A function can be a TLC value only if its domain is a finite set.)

TLC evaluates recursively defined functions with a simple recursive procedure. If f is defined by $f[x \in S] \triangleq e$, then TLC evaluates $f[c]$ by evaluating e with c substituted for x. This means that it can't handle some legal function definitions. For example, consider this definition from page 68:

$$\begin{array}{l} mr[n \in Nat] \;\triangleq \\ \quad [f \mapsto \textsc{if}\; n = 0 \;\textsc{then}\; 17 \;\textsc{else}\; mr[n-1].f * mr[n].g\,, \\ \quad\;\; g \mapsto \textsc{if}\; n = 0 \;\textsc{then}\; 42 \;\textsc{else}\; mr[n-1].f + mr[n-1].g\,] \end{array}$$

To evaluate $mr[3]$, TLC substitutes 3 for n and starts evaluating the right-hand side. But because $mr[n]$ appears in the right-hand side, TLC must evaluate the subexpression $mr[3]$, which it does by substituting 3 for n and starting to evaluate the right-hand side. And so on. TLC eventually detects that it's in an infinite loop and reports an error.

Legal recursive definitions that cause TLC to loop like this are rare, and they can be rewritten so TLC can handle them. Recall that we defined *mr* to express the mutual recursion:

$$\begin{array}{l} f[n] \;=\; \textsc{if}\; n = 0 \;\textsc{then}\; 17 \;\textsc{else}\; f[n-1] * g[n] \\ g[n] \;=\; \textsc{if}\; n = 0 \;\textsc{then}\; 42 \;\textsc{else}\; f[n-1] + g[n-1] \end{array}$$

The subexpression $mr[n]$ appeared in the expression defining $mr[n]$ because $f[n]$ depends on $g[n]$. To eliminate it, we have to rewrite the mutual recursion so that $f[n]$ depends only on $f[n-1]$ and $g[n-1]$. We do this by expanding the definition of $g[n]$ in the expression for $f[n]$. Since the ELSE clause applies only to the case $n \neq 0$, we can rewrite the expression for $f[n]$ as

$$f[n] \;=\; \textsc{if}\; n = 0 \;\textsc{then}\; 17 \;\textsc{else}\; f[n-1] * (f[n-1] + g[n-1])$$

This leads to the following equivalent definition of *mr*:

$$\begin{array}{l} mr[n \in Nat] \;\triangleq \\ \quad [f \mapsto \textsc{if}\; n = 0 \;\textsc{then}\; 17 \\ \qquad\qquad\qquad\quad \textsc{else}\; mr[n-1].f * (mr[n-1].f + mr[n-1].g)\,, \\ \quad\;\; g \mapsto \textsc{if}\; n = 0 \;\textsc{then}\; 42 \;\textsc{else}\; mr[n-1].f + mr[n-1].g\,] \end{array}$$

With this definition, TLC has no trouble evaluating $mr[3]$.

The evaluation of ENABLED predicates and the action-composition operator "·" are described on page 240 in Section 14.2.6. Section 14.3 explains how TLC evaluates temporal-logic formulas for temporal checking.

If you're not sure whether TLC can evaluate an expression, try it and see. But don't wait until TLC gets to the expression in the middle of checking the entire specification. Instead, make a small example in which TLC evaluates just that expression. See the explanation on page 14.5.3 of how to use TLC as a TLA$^+$ calculator.

14.2.3 Assignment and Replacement

As we saw in the alternating bit example, the configuration file must determine the value of each constant parameter. To assign a TLC value v to a constant parameter c of the specification, we write c `=` v in the configuration file's `CONSTANT` statement. The value v may be a primitive TLC value or a finite set of primitive TLC values written in the form $\{v_1, \ldots, v_n\}$—for example, `{1, -3, 2}`. In v, any sequence of characters like `a1` or `foo` that is not a number, a quoted string, or `TRUE` or `FALSE` is taken to be a model value.

In the assignment c `=` v, the symbol c need not be a constant parameter; it can also be a defined symbol. This assignment causes TLC to ignore the actual definition of c and to take v to be its value. Such an assignment is often used when TLC cannot compute the value of c from its definition. In particular, TLC cannot compute the value of *NotAnS* from the definition

$$NotAnS \triangleq \text{CHOOSE } n : n \notin S$$

because it cannot evaluate the unbounded CHOOSE expression. You can override this definition by assigning *NotAnS* a value in the `CONSTANT` statement of the configuration file. For example, the assignment

```
NotAnS = NS
```

causes TLC to assign to *NotAnS* the model value `NS`. TLC ignores the actual definition of *NotAnS*. If you used the name *NotAnS* in the specification, you'd probably want TLC's error messages to call it `NotAnS` rather than `NS`. So, you'd probably use the assignment

```
NotAnS = NotAnS
```

which assigns to the symbol *NotAnS* the model value `NotAnS`. Remember that, in the assignment c `=` v, the symbol c must be defined or declared in the TLA$^+$ module, and v must be a primitive TLC value or a finite set of such values.

Note that d is a defined symbol in the replacement c `<-` d, while `v` is a TLC value in the substitution c `=` v.

The `CONSTANT` statement of the configuration file can also contain *replacements* of the form c `<-` d, where c and d are symbols defined in the TLA$^+$

module. This causes TLC to replace c by d when performing its calculations. One use of replacement is to give a value to an operator parameter. For example, suppose we wanted to use TLC to check the write-through cache specification of Section 5.6 (page 54). The *WriteThroughCache* module extends the *MemoryInterface* module, which contains the declaration

$$\textsc{constants}\ Send(\_,\_,\_,\_),\ Reply(\_,\_,\_,\_),\ \ldots$$

We have to tell TLC how to evaluate the operators *Send* and *Reply*. We do this by first writing a module *MCWriteThroughCache* that extends the *WriteThroughCache* module and defines two operators

$$\begin{aligned} MCSend(p, d, old, new) &\triangleq \ldots \\ MCReply(p, d, old, new) &\triangleq \ldots \end{aligned}$$

We then add to the configuration file's `CONSTANT` statement the replacements

```
Send  <- MCSend
Reply <- MCReply
```

A replacement can also replace one defined symbol by another. In a specification, we usually write the simplest possible definitions. A simple definition is not always the easiest one for TLC to use. For example, suppose our specification requires an operator *Sort* such that *Sort*(S) is a sequence containing the elements of S in increasing order, if S is a finite set of numbers. Our specification in module *SpecMod* might use the simple definition

$$\begin{aligned} Sort(S) \;\triangleq\; &\textsc{choose}\ s \in [1 \,..\, Cardinality(S) \to S] : \\ &\quad \forall\, i, j \in \textsc{domain}\ s : (i < j) \Rightarrow (s[i] < s[j]) \end{aligned}$$

To evaluate *Sort*(S) for a set S containing n elements, TLC has to enumerate the n^n elements in the set $[1 \,..\, n \to S]$ of functions. This may be unacceptably slow. We can write a module *MCSpecMod* that extends *SpecMod* and defines *FastSort* so it equals *Sort* when applied to finite sets of numbers, but can be evaluated more efficiently by TLC. We can then run TLC with a configuration file containing the replacement

```
Sort <- FastSort
```

One possible definition of *FastSort* is given in Section 14.4, on page 250.

14.2.4 Evaluating Temporal Formulas

Section 14.2.2 (page 231) explains what kind of ordinary expressions TLC can evaluate. The specification and properties that TLC checks are temporal formulas; this section describes the class of temporal formulas it can handle.

TLC can evaluate a TLA temporal formula iff (i) the formula is *nice*—a term defined in the next paragraph—and (ii) TLC can evaluate all the ordinary expressions of which the formula is composed. For example, a formula of the form $P \leadsto Q$ is nice, so TLC can evaluate it iff it can evaluate P and Q. (Section 14.3 below explains on what states and pairs of states TLC evaluates the component expressions of a temporal formula.)

A temporal formula is nice iff it is the conjunction of formulas that belong to one of the following four classes:

State Predicate

Invariance Formula A formula of the form $\Box P$, where P is a state predicate.

Box-Action Formula A formula of the form $\Box[A]_v$, where A is an action and v is a state function.

Simple Temporal Formula To define this class, we first make the following definitions:

The terminology used here is not standard.

- The *simple Boolean operators* consist of the operators

  $$\land \quad \lor \quad \neg \quad \Rightarrow \quad \equiv \quad \text{TRUE} \quad \text{FALSE}$$

  of propositional logic together with quantification over finite, constant sets.
- A *temporal state formula* is one obtained from state predicates by applying simple Boolean operators and the temporal operators $\Box$, $\Diamond$, and $\leadsto$. For example, if N is a constant, then

  $$\forall\, i \in 1 \,..\, N : \Box((x = i) \Rightarrow \exists\, j \in 1 \,..\, i : \Diamond(y = j))$$

  is a temporal state formula.
- A *simple action formula* is one of the following, where A is an action and v a state function:

  $$\text{WF}_v(A) \qquad \text{SF}_v(A) \qquad \Box\Diamond\langle A\rangle_v \qquad \Diamond\Box[A]_v$$

  The component expressions of $\text{WF}_v(A)$ and $\text{SF}_v(A)$ are $\langle A\rangle_v$ and $\text{ENABLED}\,\langle A\rangle_v$. (The evaluation of ENABLED formulas is described on page 240.)

A simple temporal formula is then defined to be one constructed from temporal state formulas and simple action formulas by applying simple Boolean operators.

For convenience, we exclude invariance formulas from the class of simple temporal formulas, so these four classes of nice temporal formulas are disjoint.

TLC can therefore evaluate the temporal formula

$$\forall\, i \in 1 \,..\, N : \Diamond(y = i) \Rightarrow \text{WF}_y((y' = y + 1) \land (y \geq i))$$

if N is a constant, because this is a simple temporal formula (and hence nice) and TLC can evaluate all of its component expressions. TLC cannot evaluate $\Diamond\langle x' = 1\rangle_x$, since this is not a nice formula. It cannot evaluate the formula $\mathrm{WF}_x(x'[1] = 0)$ if it must evaluate the action $\langle x'[1] = 0\rangle_x$ on a step $s \rightarrow t$ in which $x = \langle\,\rangle$ in state t.

A `PROPERTY` statement can specify any formulas that TLC can evaluate. The formula of a `SPECIFICATION` statement must contain exactly one conjunct that is a box-action formula. That conjunct specifies the next-state action.

14.2.5 Overriding Modules

TLC cannot compute $2 + 2$ from the definition of $+$ contained in the standard *Naturals* module. Even if we did use a definition of $+$ from which TLC could compute sums, it would not do so very quickly. Arithmetic operators like $+$ are implemented directly in Java, the language in which TLC is written. This is achieved by a general mechanism of TLC that allows a module to be overridden by a Java class that implements the operators defined in the module. When TLC encounters an EXTENDS *Naturals* statement, it loads the Java class that overrides the *Naturals* module rather than reading the module itself. There are Java classes to override the following standard modules: *Naturals*, *Integers*, *Sequences*, *FiniteSets*, and *Bags*. (The *TLC* module described below in Section 14.4 is also overridden by a Java class.) Intrepid Java programmers will find that writing a Java class to override a module is not too hard.

14.2.6 How TLC Computes States

When TLC evaluates an invariant, it is calculating the invariant's value, which is either TRUE or FALSE. When TLC evaluates the initial predicate or the next-state action, it is computing a set of states—for the initial predicate, the set of all initial states, and for the next-state action, the set of possible *successor states* (primed states) reached from a given starting (unprimed) state. I will describe how TLC does this for the next-state action; the evaluation of the initial predicate is analogous.

Recall that a state is an assignment of values to variables. TLC computes the successors of a given state s by assigning to all unprimed variables their values in state s, assigning no values to the primed variables, and then evaluating the next-state action. TLC evaluates the next-state action as described in Section 14.2.2 (page 231), except for two differences, which I now describe. This description assumes that TLC has already performed all the assignments and replacements specified by the `CONSTANT` statement of the configuration file and has expanded all definitions. Thus, the next-state action is a formula containing only variables, primed variables, model values, and built-in TLA$^+$ operators and constants.