Chapter 17

The Meaning of a Module

Chapter 16 defines the meaning of the built-in TLA$^+$ operators. In doing so, it defines the meaning of a basic expression—that is, of an expression containing only built-in operators, declared constants, and declared variables. We now define the meaning of a module in terms of basic expressions. Since a TLA$^+$ specification consists of a collection of modules, this defines the semantics of TLA$^+$.

We also complete the definition of the syntax of TLA$^+$ by giving the remaining context-dependent syntactic conditions not described in Chapter 15. Here's a list of some illegal expressions that satisfy the grammar of Chapter 15, and where in this chapter you can find the conditions that make them illegal.

- $F(x)$, if F is defined by $F(x, y) \triangleq x + y$ (Section 17.1)
- $(x' + 1)'$ (Section 17.2)
- $x + 1$, if x is not defined or declared (Section 17.3)
- $F \triangleq 0$, if F is already defined (Section 17.5)

This chapter is meant to be read in its entirety. To try to make it as readable as possible, I have made the exposition somewhat informal. Wherever I could, I have used examples in place of formal definitions. The examples assume that you understand the approximate meanings of the TLA$^+$ constructs, as explained in Part I. I hope that mathematically sophisticated readers will see how to fill in the missing formalism.

17.1 Operators and Expressions

Because it uses conventional mathematical notation, TLA$^+$ has a rather rich syntax, with several different ways of expressing the same basic type of mathematical operation. For example, the following expressions are all formed by applying an operator to a single argument e:

$$Len(e) \qquad -e \qquad \{e\} \qquad e'$$

This section develops a uniform way of writing all these expressions, as well as more general kinds of expressions.

17.1.1 The Arity and Order of an Operator

An operator has an *arity* and an *order*. An operator's arity describes the number and order of its arguments. It's the arity of the Len operator that tells us that $Len(s)$ is a legal expression, while $Len(s, t)$ and $Len(+)$ are not. All the operators of TLA$^+$, whether built-in or defined, fall into three classes: 0th-, 1st-, and 2nd-order operators.[1] Here is how these classes, and their arities, are defined:

Len is defined in the *Sequences* module on page 341.

0. $E \triangleq x' + y$ defines E to be the 0th-order operator $x' + y$. A 0th-order operator takes no arguments, so it is an ordinary expression. We represent the arity of such an operator by the symbol $\_$ (underscore).

1. $F(x, y) \triangleq x \cup \{z, y\}$ defines F to be a 1st-order operator. For any expressions e_1 and e_2, it defines $F(e_1, e_2)$ to be an expression. We represent the arity of F by $\langle \_, \_ \rangle$.

   In general, a 1st-order operator takes expressions as arguments. Its arity is the tuple $\langle \_, \ldots, \_ \rangle$, with one $\_$ for each argument.

2. $G(f(\_, \_), x, y) \triangleq f(x, \{x, y\})$ defines G to be a 2nd-order operator. The operator G takes three arguments: its first argument is a 1st-order operator that takes two arguments; its last two arguments are expressions. For any operator Op of arity $\langle \_, \_ \rangle$, and any expressions e_1 and e_2, this defines $G(Op, e_1, e_2)$ to be an expression. We say that G has arity $\langle \langle \_, \_ \rangle, \_, \_ \rangle$.

   In general, the arguments of a 2nd-order operator may be expressions or 1st-order operators. A 2nd-order operator has an arity of the form $\langle a_1, \ldots, a_n \rangle$, where each a_i is either $\_$ or $\langle \_, \ldots, \_ \rangle$. (We can consider a 1st-order operator to be a degenerate case of a 2nd-order operator.)

It would be easy enough to define 3rd- and higher-order operators. TLA$^+$ does not permit them because they are of little use and would make it harder to check level-correctness, which is discussed in Section 17.2 below.

[1] Even though it allows 2nd-order operators, TLA$^+$ is still what logicians call a first-order logic because it permits quantification only over 0th-order operators. A higher-order logic would allow us to write the formula $\exists\, x(\_) : exp$.

17.1.2 λ Expressions

When we define a 0th-order operator E by $E \triangleq exp$, we can write what the operator E equals—it equals the expression *exp*. We can explain the meaning of this definition by saying that it assigns the value *exp* to the symbol E. To explain the meaning of an arbitrary TLA$^+$ definition, we need to be able to write what a 1st- or 2nd-order operator equals—for example, the operator F defined by

$$F(x, y) \;\triangleq\; x \cup \{z, y\}$$

TLA$^+$ provides no way to write an expression that equals this operator F. (A TLA$^+$ expression can equal only a 0th-order operator.) We therefore generalize expressions to λ *expressions*, and we write the operator that F equals as the λ expression

$$\lambda\, x, y \,:\, x \cup \{z, y\}$$

The symbols x and y in this λ expression are called λ parameters. We use λ expressions only to explain the meaning of TLA$^+$ specifications; we can't write a λ expression in TLA$^+$.

We also allow 2nd-order λ expressions, where the operator G defined by

$$G(f(\_, \_), x, y) \;\triangleq\; f(y, \{x, z\})$$

is equal to the λ expression

(17.1) $\lambda\, f(\_, \_), x, y \,:\, f(y, \{x, z\})$

The general form of a λ expression is $\lambda\, p_1, \ldots, p_n : exp$, where *exp* is a λ expression, each parameter p_i is either an identifier id_i or has the form $id_i(\_, \ldots, \_)$, and the id_i are all distinct. We call id_i the *identifier* of the λ parameter p_i. We consider the $n = 0$ case, the λ expression $\lambda : exp$ with no parameters, to be the expression *exp*. This makes a λ expression a generalization of an ordinary expression.

A λ parameter identifier is a bound identifier, just like the identifier x in $\forall\, x : F$. As with any bound identifiers, renaming the λ parameter identifiers in a λ expression doesn't change the meaning of the expression. For example, (17.1) is equivalent to

$$\lambda\, abc(\_, \_), qq, m \,:\, abc(m, \{qq, z\})$$

For obscure historical reasons, this kind of renaming is called α *conversion*.

If *Op* is the λ expression $\lambda\, p_1, \ldots, p_n : exp$, then $Op(e_1, \ldots, e_n)$ equals the result of replacing the identifier of the λ parameter p_i in *exp* with e_i, for all i in $1 \,..\, n$. For example,

$$(\lambda\, x, y \,:\, x \cup \{z, y\})\,(TT, w + z) \;=\; TT \cup \{z, (w + z)\}$$

This procedure for evaluating the application of a λ expression is called β *reduction.*

17.1.3 Simplifying Operator Application

To simplify the exposition, I assume that every operator application is written in the form $Op(e_1, \ldots, e_n)$. TLA$^+$ provides a number of different syntactic forms for operator application, so I have to explain how they are translated into this simple form. Here are all the different forms of operator application and their translations.

- Simple constructs with a fixed number of arguments, including infix operators like $+$, prefix operators like ENABLED , and constructs like WF, function application, and IF/THEN/ELSE. These operators and constructs pose no problem. We can write $+(a, b)$ instead of $a + b$, $IfThenElse(p, e_1, e_2)$ instead of

  $$\text{IF } p \text{ THEN } e_1 \text{ ELSE } e_2$$

  and $Apply(f, e)$ instead of $f[e]$. An expression like $a + b + c$ is an abbreviation for $(a + b) + c$, so it can be written $+(+(a, b), c)$.

- Simple constructs with a variable number of arguments—for example, $\{e_1, \ldots, e_n\}$ and $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$. We can consider each of these constructs to be repeated application of simpler operators with a fixed number of arguments. For example,

  $$\begin{array}{rcl} \{e_1, \ldots, e_n\} & = & \{e_1\} \cup \ldots \cup \{e_n\} \\ [h_1 \mapsto e_1, \ldots, h_n \mapsto e_n] & = & [h_1 \mapsto e_1] \,@@\, \ldots \,@@[h_n \mapsto e_n] \end{array}$$

  where @@ is defined in the *TLC* module, on page 248. Of course, $\{e\}$ can be written $Singleton(e)$ and $[h \mapsto e]$ can be written $Record(\text{“}h\text{”}, e)$. Note that an arbitrary CASE expression can be written in terms of CASE expressions of the form

  $$\text{CASE } p \rightarrow e \,\square\, q \rightarrow f$$

  using the relation

  $$\begin{array}{l} \text{CASE } p_1 \rightarrow e_1 \,\square\, \ldots \,\square\, p_n \rightarrow e_n \quad = \\ \qquad \text{CASE } p_1 \rightarrow e_1 \,\square\, (p_2 \lor \ldots \lor p_n) \rightarrow (\text{CASE } p_2 \rightarrow e_2 \,\square\, \ldots \,\square\, p_n \rightarrow e_n) \end{array}$$

- Constructs that introduce bound variables—for example,

  $$\exists\, x \in S \,:\, x + z > y$$

  We can rewrite this expression as

  $$ExistsIn(S,\ \lambda\, x : x + z > y)$$

  where *ExistsIn* is a 2nd-order operator of arity $\langle\, \_\,,\ \langle\, \_\,\rangle\,\rangle$. All the variants of the $\exists$ construct can be represented as expressions using either $\exists\, x \in S : e$ or $\exists\, x : e$. (Section 16.1.1 shows how these variants can be translated into expressions using only $\exists\, x : e$, but those translations don't maintain the scoping rules—for example, rewriting $\exists\, x \in S : e$ as $\exists\, x : (x \in S) \wedge e$ moves S inside the scope of the bound variable x.)

All other constructs that introduce bound variables, such as $\{x \in S : exp\}$, can similarly be expressed in the form $Op(e_1, \ldots, e_n)$ using λ expressions and 2nd-order operators Op. (Chapter 16 explains how to express constructs like $\{\langle x, y \rangle \in S : exp\}$, which have a tuple of bound identifiers, in terms of constructs with ordinary bound identifiers.)

- Operator applications such as $M(x)!\,Op(y, z)$ that arise from instantiation. We write this as $M!\,Op(x, y, z)$.
- LET expressions. The meaning of a LET expression is explained in Section 17.4 below. For now, we consider only LET-free λ expressions—ones that contain no LET expressions.

For uniformity, I will call an operator symbol an *identifier*, even if it is a symbol like + that isn't an identifier according to the syntax of Chapter 15.

17.1.4 Expressions

We can now inductively define an expression to be either a 0th-order operator, or to have the form $Op(e_1, \ldots, e_n)$ where Op is an operator and each e_i is either an expression or a 1st-order operator. The expression must be *arity-correct*, meaning that Op must have arity $\langle a_1, \ldots, a_n \rangle$, where each a_i is the arity of e_i. In other words, e_i must be an expression if a_i equals $\_$; otherwise it must be a 1st-order operator with arity a_i. We require that Op not be a λ expression. (If it is, we can use β reduction to evaluate $Op(e_1, \ldots, e_n)$ and eliminate the λ expression Op.) Hence, a λ expression can appear in an expression only as an argument of a 2nd-order operator. This implies that only 1st-order λ expressions can appear in an expression.

We have eliminated all bound identifiers except the ones in λ expressions. We maintain the TLA$^+$ requirement that an identifier that already has a meaning cannot be used as a bound identifier. Thus, in any λ expression $\lambda\, p_1, \ldots, p_n : exp$, the identifiers of the parameters p_i cannot appear as parameter identifiers in any λ expression that occurs in exp.

Remember that λ expressions are used only to explain the semantics of TLA$^+$. They are not part of the language, and they can't be used in a TLA$^+$ specification.

17.2 Levels

TLA$^+$ has a class of syntactic restrictions that come from the underlying logic TLA and have no counterpart in ordinary mathematics. The simplest of these is that "double-priming" is prohibited. For example, $(x' + y)'$ is not syntactically well-formed, and is therefore meaningless, because the operator $'$ (priming) can be applied only to a state function, not to a transition function like $x' + y$. This class of restriction is expressed in terms of *levels*.

In TLA, an expression has one of four basic levels, which are numbered 0, 1, 2, and 3. These levels are described below, using examples that assume x, y, and c are declared by

$$\text{VARIABLES } x, y \qquad \text{CONSTANT } c$$

and symbols like $+$ have their usual meanings.

0. A *constant*-level expression is a constant; it contains only constants and constant operators. Example: $c + 3$.

1. A *state*-level expression is a state function; it may contain constants, constant operators, and unprimed variables. Example: $x + 2 * c$.

2. A *transition*-level expression is a transition function; it may contain anything except temporal operators. Example: $x' + y > c$.

3. A *temporal*-level expression is a temporal formula; it may contain any TLA operator. Example: $\Box[x' > y + c]_{\langle x, y \rangle}$.

Chapter 16 assigns meanings to all basic expressions—ones containing only the built-in operators of TLA$^+$ and declared constants and variables. The meaning assigned to an expression depends as follows on its level.

0. The meaning of a constant-level basic expression is a constant-level basic expression containing only primitive operators.

1. The meaning of a state-level basic expression e is an assignment of a constant expression $s[\![e]\!]$ to any state s.

2. The meaning of a transition-level basic expression e is an assignment of a constant expression $\langle s, t \rangle[\![e]\!]$ to any transition $s \to t$.

3. The meaning of a temporal-level basic expression F is an assignment of a constant expression $\sigma \models F$ to any behavior σ.

An expression of any level can be considered to be an expression of a higher level, except that a transition-level expression is not a temporal-level expression.[2] For example, if x is a declared variable, then the state-level expression $x > 2$ is the

[2]More precisely, a transition-level expression that is not a state-level expression is not a temporal-level expression.

temporal-level formula such that $\sigma \models x$ is the value of $x > 2$ in the first state of σ, for any behavior σ.[3]

A set of simple rules inductively defines whether a basic expression is *level-correct* and, if so, what its level is. Here are some of the rules:

- A declared constant is a level-correct expression of level 0.
- A declared variable is a level-correct expression of level 1.
- If *Op* is declared to be a 1st-order constant operator, then the expression $Op(e_1, \ldots, e_n)$ is level-correct iff each e_i is level-correct, in which case its level is the maximum of the levels of the e_i.
- $e_1 \in e_2$ is level-correct iff e_1 and e_2 are, in which case its level is the maximum of the levels of e_1 and e_2.
- e' is level-correct and has level 2 iff e is level-correct and has level at most 1.[4]
- ENABLED e is level-correct and has level 1 iff e is level-correct and has level at most 2.
- $\exists\, x : e$ is level-correct and has level l iff e is level-correct and has level l, when x is considered to be a declared constant.
- $\boldsymbol{\exists}\, x : e$ is level-correct and has level 3 iff e is level-correct and has any level other than 2, when x is considered to be a declared variable.

There are similar rules for the other TLA$^+$ operators.

A useful consequence of these rules is that level-correctness of a basic expression does not depend on the levels of the declared identifiers. In other words, an expression e is level-correct when c is declared to be a constant iff it is level-correct when c is declared to be a variable. Of course, the level of e may depend on the level of c.

We can abstract these rules by generalizing the concept of a level. So far, we have defined the level only of an expression. We can define the level of a 1st- or 2nd-order operator *Op* to be a rule for determining the level-correctness and level of an expression $Op(e_1, \ldots, e_n)$ as a function of the levels of the arguments e_i. The level of a 1st-order operator is a rule, so the level of a 2nd-order operator *Op* is a rule that depends in part on rules—namely, on the levels of the arguments that are operators. This makes a rigorous general definition of levels for 2nd-order operators rather complicated. Fortunately, there's a simpler, less general

[3] The expression $x + 2$ can be considered to be a temporal-level expression that, like the temporal-level expression $\Box(x + 2)$, is silly. (See the discussion of silliness in Section 6.2 on page 67.)

[4] If e is a constant expression, then e' equals e, so we could consider e' to have level 0. For simplicity, we consider e' to have level 2 even if e is a constant.

definition that handles all the operators of TLA$^+$. Even more fortunately, you don't have to know it, so I won't bother writing it down. All you need to know is that there exists a way of assigning a level to every built-in operator of TLA$^+$. The level-correctness and level of any basic expression is then determined by those levels and the levels of the declared identifiers that occur in the expression.

One important class of operator levels are the *constant* levels. Any expression built from constant-level operators and declared constants has constant level. The built-in constant operators of TLA$^+$, listed in Tables 1 and 2 (pages 268 and 269) all have constant level. Any operator defined solely in terms of constant-level operators and declared constants has constant level.

We now extend the definition of level-correctness from expressions to λ expressions. We define the λ expression $\lambda p_1, \ldots, p_n : exp$ to be level-correct iff *exp* is level-correct when the λ parameter identifiers are declared to be constants of the appropriate arity. For example, $\lambda p,\ q(\_) : exp$ is level-correct iff *exp* is level-correct with the additional declaration

$$\text{CONSTANTS } p,\ q(\_)$$

This inductively defines level-correctness for λ expressions. The definition is reasonable because, as observed a few paragraphs ago, the level-correctness of *exp* doesn't depend on whether we assign level 0 or 1 to the λ parameters. One can also define the level of an arbitrary λ expression, but that would require the general definition of the level of an operator, which we want to avoid.

17.3 Contexts

Syntactic correctness of a basic expression depends on the arities of the declared identifiers. The expression $Foo = \{\}$ is syntactically correct if *Foo* is declared to be a variable, and hence of arity $\_$, but not if it's declared to be a (1st-order) constant of arity $\langle \_ \rangle$. The meaning of a basic expression also depends on the levels of the declared identifiers. We can't determine those arities and levels just by looking at the expression itself; they are implied by the context in which the expression appears. A nonbasic expression contains defined as well as declared operators. Its syntactic correctness and meaning depend on the definitions of those operators, which also depend on the context. This section defines a precise notion of a context.

For uniformity, built-in operators are treated the same as defined and declared operators. Just as the context might tell us that the identifier x is a declared variable, it tells us that $\in$ is declared to be a constant-level operator of arity $\langle \_, \_ \rangle$ and that $\notin$ is defined to equal $\lambda a,\ b : \neg(\in (a,\ b))$. We assume a standard context that specifies all the built-in operators of TLA$^+$.

To define contexts, let's first define declarations and definitions. A *declaration* assigns an arity and level to an operator name. A *definition* assigns a LET-free λ expression to an operator name. A *module definition* assigns the meaning of a module to a module name, where the meaning of a module is defined in Section 17.5 below.[5] A *context* consists of a set of declarations, definitions, and module definitions such that

C1. An operator name is declared or defined at most once by the context. (This means that it can't be both declared and defined.)

C2. No operator defined or declared by the context appears as the identifier of a λ parameter in any definition's expression.

C3. Every operator name that appears in a definition's expression is either a λ parameter's identifier or is declared (not defined) by the context.

C4. No module name is assigned meanings by two different module definitions.

Module and operator names are handled separately. The same string may be both a module name that is defined by a module definition and an operator name that is either declared or defined by an ordinary definition.

Here is an example of a context that declares the symbols $\cup$, a, b, and $\in$, defines the symbols c and *foo*, and defines the module *Naturals*:

$$(17.2)\quad \{\ \cup : \langle\_,\_\rangle, \quad a : \_, \quad b : \_, \quad \in\ : \langle\_,\_\rangle, \quad c \triangleq \cup(a, b), \\ \mathit{foo} \triangleq \lambda p, q(\_) : \in(p, \cup(q(b), a)), \quad \mathit{Naturals} \overset{m}{=} \ldots\ \}$$

Not shown are the levels assigned to the operators $\cup$, a, b, and $\in$ and the meaning assigned to *Naturals*.

If $\mathcal{C}$ is a context, a *$\mathcal{C}$-basic λ expression* is defined to be a λ expression that contains only symbols declared in $\mathcal{C}$ (in addition to λ parameters). For example, $\lambda x : \in(x, \cup(a, b))$ is a $\mathcal{C}$-basic λ expression if $\mathcal{C}$ is the context (17.2). However, neither $\cap(a, b)$ nor $\lambda x : c(x, b)$ is a $\mathcal{C}$-basic λ expression because neither $\cap$ nor c is declared in $\mathcal{C}$. (The symbol c is defined, not declared, in $\mathcal{C}$.) A $\mathcal{C}$-basic λ expression is *syntactically correct* if it is arity- and level-correct with the arities and levels assigned by $\mathcal{C}$ to the expression's operators. Condition C3 states that if $Op \triangleq exp$ is a definition in context $\mathcal{C}$, then exp is a $\mathcal{C}$-basic λ expression. We add to C3 the requirement that it be syntactically correct.

We also allow a context to contain a special definition of the form $Op \triangleq ?$ that assigns to the name Op an "illegal" value ? that is not a λ expression. This definition indicates that, in the context, it is illegal to use the operator name Op.

17.4 The Meaning of a λ Expression

We now define the meaning $\mathcal{C}[\![e]\!]$ of a λ expression e in a context $\mathcal{C}$ to be a $\mathcal{C}$-basic λ expression. If e is an ordinary (nonbasic) expression, and $\mathcal{C}$ is the

[5]The meaning of a module is defined in terms of contexts, so these definitions appear to be circular. In fact, the definitions of context and of the meaning of a module together form a single inductive definition.

context that specifies the built-in TLA$^+$ operators and declares the constants and variables that occur in e, then this will define $\mathcal{C}[\![e]\!]$ to be a basic expression. Since Chapter 16 defines the meaning of basic expressions, this defines the meaning of an arbitrary expression. The expression e may contain LET constructs, so this defines the meaning of LET, the one operator whose meaning is not defined in Chapter 16.

Basically, $\mathcal{C}[\![e]\!]$ is obtained from e by replacing all defined operator names with their definitions and then applying β reduction whenever possible. Recall that β reduction replaces

$$(\lambda\ p_1, \ldots, p_n\ :\ exp)\,(e_1, \ldots, e_n)$$

with the expression obtained from *exp* by replacing the identifier of p_i with e_i, for each i. The definition of $\mathcal{C}[\![e]\!]$ does not depend on the levels assigned by the declarations of $\mathcal{C}$. So, we ignore levels in the definition. The inductive definition of $\mathcal{C}[\![e]\!]$ consists of the following rules:

- If e is an operator symbol, then $\mathcal{C}[\![e]\!]$ equals (i) e if e is declared in $\mathcal{C}$, or (ii) the λ expression of e's definition in $\mathcal{C}$ if e is defined in $\mathcal{C}$.

- If e is $Op(e_1, \ldots, e_n)$, where Op is declared in $\mathcal{C}$, then $\mathcal{C}[\![e]\!]$ equals the expression $Op(\mathcal{C}[\![e_1]\!], \ldots, \mathcal{C}[\![e_n]\!])$.

- If e is $Op(e_1, \ldots, e_n)$, where Op is defined in $\mathcal{C}$ to equal the λ expression d, then $\mathcal{C}[\![e]\!]$ equals the β reduction of $\overline{d}(\mathcal{C}[\![e_1]\!], \ldots, \mathcal{C}[\![e_n]\!])$, where $\overline{d}$ is obtained from d by α conversion (replacement of λ parameters) so that no λ parameter's identifier appears in both $\overline{d}$ and some $\mathcal{C}[\![e_i]\!]$.

- If e is $\lambda p_1, \ldots, p_n : exp$, then $\mathcal{C}[\![e]\!]$ equals $\lambda p_1, \ldots, p_n : \mathcal{D}[\![exp]\!]$, where $\mathcal{D}$ is the context obtained by adding to $\mathcal{C}$ the declarations that, for each i in $1 \,..\, n$, assign to the i^{th} λ parameter's identifier the arity determined by p_i.

- If e is where d is a λ expression and *exp* an expression, then $\mathcal{C}[\![e]\!]$ equals $\mathcal{D}[\![exp]\!]$, where $\mathcal{D}$ is the context obtained by adding to $\mathcal{C}$ the definition that assigns $\mathcal{C}[\![d]\!]$ to Op.

- If e is

  $$\text{LET}\ \ Op(p_1, \ldots, p_n)\ \triangleq\ \text{INSTANCE}\ \ldots\ \ \text{IN}\ exp$$

  then $\mathcal{C}[\![e]\!]$ equals $\mathcal{D}[\![exp]\!]$, where $\mathcal{D}$ is the new current context obtained by "evaluating" the statement

  $$Op(p_1, \ldots, p_n)\ \triangleq\ \text{INSTANCE}\ \ldots$$

  in the current context $\mathcal{C}$, as described in Section 17.5.5 below.

The last two conditions define the meaning of any LET construct, because

- The operator definition $Op(p_1, \ldots, p_n) \triangleq d$ in a LET means

  $$Op \;\triangleq\; \lambda p_1, \ldots, p_n : d$$

- A function definition $Op[x \in S] \triangleq d$ in a LET means

  $$Op \;\triangleq\; \text{CHOOSE } Op : Op = [x \in S \mapsto d]$$

- The expression LET $Op_1 \triangleq d_1 \;\ldots\; Op_n \triangleq d_n$ IN exp is defined to equal

  $$\text{LET } Op_1 \triangleq d_1 \text{ IN } (\text{LET } \ldots \text{ IN } (\text{LET } Op_n \triangleq d_n \text{ IN } exp) \ldots)$$

The λ expression e is defined to be legal (syntactically well-formed) in the context $\mathcal{C}$ iff these rules define $\mathcal{C}[\![e]\!]$ to be a legal $\mathcal{C}$-basic expression.

17.5 The Meaning of a Module

The meaning of a module depends on a context. For an external module, which is not a submodule of another module, the context consists of declarations and definitions of all the built-in operators of TLA$^+$, together with definitions of some other modules. Section 17.7 below discusses where the definitions of those other modules come from.

The meaning of a module in a context $\mathcal{C}$ consists of six sets:

Dcl	A set of declarations. They come from CONSTANT and VARIABLE declarations and declarations in extended modules (modules appearing in an EXTENDS statement).
GDef	A set of global definitions. They come from ordinary (non-LOCAL) definitions and global definitions in extended and instantiated modules.
LDef	A set of local definitions. They come from LOCAL definitions and LOCAL instantiations of modules. (Local definitions are not obtained by other modules that extend or instantiate the module.)
MDef	A set of module definitions. They come from submodules of the module and of extended modules.
Ass	A set of assumptions. They come from ASSUME statements and from extended modules.
Thm	A set of theorems. They come from THEOREM statements, from theorems in extended modules, and from the assumptions and theorems of instantiated modules, as explained in Section 17.5.5 below.

The λ expressions of definitions in *GDef* and *LDef*, as well as the expressions in *Ass* and *Thm*, are $(\mathcal{C} \cup Dcl)$-basic λ expressions. In other words, the only operator symbols they contain (other than λ parameter identifiers) are ones declared in $\mathcal{C}$ or in *Dcl*.

The meaning of a module in a context $\mathcal{C}$ is defined by an algorithm for computing these six sets. The algorithm processes each statement in the module in turn, from beginning to end. The meaning of the module is the value of those sets when the end of the module is reached.

Initially, all six sets are empty. The rules for handling each possible type of statement are given below. In these rules, the *current context* $\mathcal{CC}$ is defined to be the union of $\mathcal{C}$, *Dcl*, *GDef*, *LDef*, and *MDef*.

When the algorithm adds elements to the context $\mathcal{CC}$, it uses α conversion to ensure that no defined or declared operator name appears as a λ parameter's identifier in any λ expression in $\mathcal{CC}$. For example, if the definition $foo \triangleq \lambda x : x + 1$ is in *LDef*, then adding a declaration of x to *Dcl* requires α conversion of this definition to rename the λ parameter identifier x. This α conversion is not explicitly mentioned.

17.5.1 Extends

An EXTENDS statement has the form

EXTENDS $M_1, \ldots, M_n$

where each M_i is a module name. This statement must be the first one in the module. The statement sets the values of *Dcl*, *GDef*, *MDef*, *Ass*, and *Thm* equal to the union of the corresponding values for the module meanings assigned by $\mathcal{C}$ to the module names M_i.

This statement is legal iff the module names M_i are all defined in $\mathcal{C}$, and the resulting current context $\mathcal{CC}$ does not assign more than one meaning to any symbol. More precisely, if the same symbol is defined or declared by two or more of the M_i, then those duplicate definitions or declarations must all have been obtained through a (possibly empty) chains of EXTENDS statements from the same definition or declaration. For example, suppose M_1 extends the *Naturals* module, and M_2 extends M_1. Then the three modules *Naturals*, M_1, and M_2 all define the operator $+$. The statement

EXTENDS *Naturals*, M_1, M_2

can still be legal, because each of the three definitions is obtained by a chain of EXTENDS statements (of length 0, 1, and 2, respectively) from the definition of $+$ in the *Naturals* module.

When decomposing a large specification into modules, we often want a module M to extend modules $M_1, \ldots, M_n$, where the M_i have declared constants and/or variables in common. In this case, we put the common declarations in a module P that is extended by all the M_i.

17.5.2 Declarations

A declaration statement has one of the forms

$$\text{CONSTANT } c_1, \ldots, c_n \qquad \text{VARIABLE } v_1, \ldots, v_n$$

where each v_i is an identifier and each c_i is either an identifier or has the form $Op(\_, \ldots, \_)$, for some identifier Op. This statement adds to the set Dcl the obvious declarations. It is legal iff none of the declared identifiers is defined or declared in $\mathcal{CC}$.

17.5.3 Operator Definitions

A global operator definition[6] has one of the two forms

$$Op \triangleq exp \qquad Op(p_1, \ldots, p_n) \triangleq exp$$

where Op is an identifier, exp is an expression, and each p_i is either an identifier or has the form $P(\_, \ldots, \_)$, where P is an identifier. We consider the first form an instance of the second with $n = 0$.

This statement is legal iff Op is not declared or defined in $\mathcal{CC}$ and the λ expression $\lambda p_1, \ldots, p_n : exp$ is legal in context $\mathcal{CC}$. In particular, no λ parameter in this λ expression can be defined or declared in $\mathcal{CC}$. The statement adds to $GDef$ the definition that assigns to Op the λ expression $\mathcal{CC}[\![\lambda p_1, \ldots, p_n : exp]\!]$.

A local operator definition has one of the two forms

$$\text{LOCAL } Op \triangleq exp \qquad \text{LOCAL } Op(p_1, \ldots, p_n) \triangleq exp$$

It is the same as a global definition, except that it adds the definition to $LDef$ instead of $GDef$.

17.5.4 Function Definitions

A global function definition has the form

$$Op[fcnargs] \triangleq exp$$

[6]An operator definition statement should not be confused with a definition clause in a LET expression. The meaning of a LET expression is described in Section 17.4.

where *fcnargs* is a comma-separated list of elements, each having the form $Id_1, \ldots, Id_n \in S$ or $\langle Id_1, \ldots, Id_n \rangle \in S$. It is equivalent to the global operator definition

$$Op \;\triangleq\; \text{CHOOSE } Op \;:\; Op = [fcnargs \mapsto exp]$$

A local function definition, which has the form

$$\text{LOCAL } Op[fcnargs] \;\triangleq\; exp$$

is equivalent to the analogous local operator definition.

17.5.5 Instantiation

We consider first a global instantiation of the form

(17.3) $I(p_1, \ldots, p_m) \;\triangleq\; \text{INSTANCE } N \text{ WITH } q_1 \leftarrow e_1, \ldots, q_n \leftarrow e_n$

For this to be legal, N must be a module name defined in $\mathcal{CC}$. Let *NDcl*, *NDef*, *NAss*, and *NThm* be the sets *Dcl*, *GDef*, *Ass*, and *Thm* in the meaning assigned to N by $\mathcal{CC}$. The q_i must be distinct identifiers declared by *NDcl*. We add a WITH clause of the form $Op \leftarrow Op$ for any identifier *Op* that is declared in *NDcl* but is not one of the q_i, so the q_i constitute all the identifiers declared in *NDcl*.

Neither I nor any of the identifiers of the definition parameters p_i may be defined or declared in $\mathcal{CC}$. Let $\mathcal{D}$ be the context obtained by adding to $\mathcal{CC}$ the obvious constant-level declaration for each p_i. Then e_i must be syntactically well-formed in the context $\mathcal{D}$, and $\mathcal{D}[\![e_i]\!]$ must have the same arity as q_i, for each $i \in 1 \mathrel{..} n$.

The instantiation must also satisfy the following level-correctness condition. Define module N to be a *constant* module iff every declaration in *NDcl* has constant level, and every operator appearing in every definition in *NDef* has constant level. If N is *not* a constant module, then for each i in $1 \mathrel{..} n$:

- If q_i is declared in *NDcl* to be a constant operator, then $\mathcal{D}[\![e_i]\!]$ has constant level.
- If q_i is declared in *NDcl* to be a variable (a 0th-order operator of level 1), then $\mathcal{D}[\![e_i]\!]$ has level 0 or 1.

The reason for this condition is explained in Section 17.8 below.

For each definition $Op \triangleq \lambda r_1, \ldots, r_p : e$ in *NDef*, the definition

(17.4) $I!Op \triangleq \lambda\; p_1, \ldots, p_m, r_1, \ldots, r_p \;:\; \overline{e}$

is added to *GDef*, where $\overline{e}$ is the expression obtained from e by substituting e_i for q_i, for all $i \in 1 \mathrel{..} n$. Before doing this substitution, α conversion must be applied to ensure that $\mathcal{CC}$ is a correct context after the definition of $I!Op$ is added to *GDef*. The precise definition of $\overline{e}$ is a bit subtle; it is given in Section 17.8 below. We require that the λ expression in (17.4) be level-correct. (If N is a nonconstant module, then level-correctness of this λ expression is implied by the level condition on parameter instantiation described in the preceding paragraph.) Legality of the definition of *Op* in module N and of the WITH substitutions implies that the λ expression is arity-correct in the current context. Remember that $I!Op(c_1, \ldots, c_m, d_1, \ldots, d_n)$ is actually written in TLA$^+$ as $I(c_1, \ldots, c_m)!Op(d_1, \ldots, d_n)$.

Also added to *GDef* is the special definition $I \triangleq ?$. This prevents I from later being defined or declared as an operator name.

If *NAss* equals the set $\{A_1, \ldots, A_k\}$ of assumptions, then for each theorem T in *NThm*, we add to *Thm* the theorem

$$\overline{A_1} \wedge \ldots \wedge \overline{A_k} \;\Rightarrow\; \overline{T}$$

(As above, $\overline{T}$ and the $\overline{A_j}$ are obtained from T and the A_j by substituting e_i for q_i, for each i in $1 \,..\, k$.)

A global INSTANCE statement can also have the two forms

$$I \;\triangleq\; \text{INSTANCE } N \text{ WITH } q_1 \leftarrow e_1, \ldots, q_n \leftarrow e_n$$
$$\text{INSTANCE } N \text{ WITH } q_1 \leftarrow e_1, \ldots, q_n \leftarrow e_n$$

The first is just the $m = 0$ case of (17.3); the second is similar to the first, except the definitions added to *GDef* do not have $I!$ prepended to the operator names. The second form also has the legality condition that none of the defined symbols in N may be defined or declared in the current context, except in the following case. An operator definition may be included multiple times through chains of INSTANCE and EXTENDS statements if it is defined in a module[7] having no declarations. For example, suppose the current context contains a definition of $+$ obtained through extending the *Naturals* module. Then an INSTANCE N statement is legal even though N also extends *Naturals* and therefore defines $+$. Because the *Naturals* module declares no parameters, instantiation cannot change the definition of $+$.

In all forms of the INSTANCE statement, omitting the WITH clause is equivalent to the case $n = 0$ of these statements. (Remember that all the declared identifiers of module N are either explicitly or implicitly instantiated.)

A local INSTANCE statement consists of the keyword LOCAL followed by an INSTANCE statement of the form described above. It is handled in a similar fashion to a global INSTANCE statement, except that all definitions are added to *LDef* instead of *GDef*.

[7] An operator J!Op is defined in the module that contains the $J \triangleq$ INSTANCE ... statement.

17.5.6 Theorems and Assumptions

A theorem has one of the forms

$\textsc{theorem}\ \mathit{exp} \qquad \textsc{theorem}\ \mathit{Op}\ \triangleq\ \mathit{exp}$

where *exp* is an expression, which must be legal in the current context $\mathcal{CC}$. The first form adds the theorem $\mathcal{CC}[\![\mathit{exp}]\!]$ to the set *Thm*. The second form is equivalent to the two statements

$\mathit{Op}\ \triangleq\ \mathit{exp}$
$\textsc{theorem}\ \mathit{Op}$

An assumption has one of the forms

$\textsc{assume}\ \mathit{exp} \qquad \textsc{assume}\ \mathit{Op}\ \triangleq\ \mathit{exp}$

The expression *exp* must have constant level. An assumption is similar to a theorem except that $\mathcal{CC}[\![\mathit{exp}]\!]$ is added to the set *Ass*.

17.5.7 Submodules

A module can contain a submodule, which is a complete module that begins with

┌──────────── MODULE *N* ────────────┐

for some module name *N*, and ends with

└────────────────────────────────────┘

This is legal iff the module name *N* is not defined in $\mathcal{CC}$ and the module is legal in the context $\mathcal{CC}$. In this case, the module definition that assigns to *N* the meaning of the submodule in context $\mathcal{CC}$ is added to *MDef*.

A submodule can be used in an INSTANCE statement that appears either later in the current module or in a module that extends the current module. Submodules of a module *M* are *not* added to the set *MDef* of a module that instantiates *M*.

17.6 Correctness of a Module

Section 17.5 above defines the meaning of a module to consist of the six sets *Dcl*, *GDef*, *LDef*, *MDef*, *Ass*, and *Thm*. Mathematically, we can view the meaning of a module to be the assertion that all the theorems in *Thm* are consequences of the assumptions in Ass. More precisely, let A be the conjunction of all the assumptions in Ass. The module asserts that, for every theorem T in Thm, the formula $A \Rightarrow T$ is valid.[8]

An assumption or theorem of the module is a $(\mathcal{C} \cup Dcl)$-basic expression. For an outermost module (not a submodule), $\mathcal{C}$ declares only the built-in operators of TLA$^+$, and Dcl declares the declared constants and variables of the module. Therefore, each formula $A \Rightarrow T$ asserted by the module is a basic expression. We say that the module is *semantically correct* if each of these expressions $A \Rightarrow T$ is a valid formula in the context Dcl. Chapter 16 defines what it means for a basic expression to be a valid formula.

By defining the meaning of a theorem, we have defined the meaning of a TLA$^+$ specification. Any mathematically meaningful question we can ask about a specification can be framed as the question of whether a certain formula is a valid theorem.

17.7 Finding Modules

For a module M to have a meaning in a context $\mathcal{C}$, every module N extended or instantiated by M must have its meaning defined in $\mathcal{C}$—unless N is a submodule of M or of a module extended by M. In principle, module M is interpreted in a context containing declarations and definitions of the built-in TLA$^+$ operator names and module definitions of all modules needed to interpret M. In practice, a tool (or a person) begins interpreting M in a context $\mathcal{C}_0$ initially containing only declarations and definitions of the built-in TLA$^+$ operator names. When the tool encounters an EXTENDS or INSTANCE statement that mentions a module named N not defined in the current context $\mathcal{CC}$ of M, the tool finds the module named N, interprets it in the context $\mathcal{C}_0$, and then adds the module definition for N to $\mathcal{C}_0$ and to $\mathcal{CC}$.

The definition of the TLA$^+$ language does not specify how a tool finds a module named N. A tool will most likely look for the module in a file named N`.tla`.

The meaning of a module depends on the meanings of the modules that it extends or instantiates. The meaning of each of those modules in turn may depend on the meanings of other modules, and so on. Thus, the meaning of a module depends on the meanings of some set of modules. A module M is syntactically incorrect if this set of modules includes M itself.

[8]In a temporal logic like TLA, the formula $F \Rightarrow G$ is not in general equivalent to the assertion that G is a consequence of assumption F. However, the two are equivalent if F is a constant formula, and TLA$^+$ allows only constant assumptions.

17.8 The Semantics of Instantiation

Section 17.5.5 above defines the meaning of an INSTANCE statement in terms of substitution. I now define precisely how that substitution is performed and explain the level-correctness rule for instantiating nonconstant modules.

Suppose that module M contains the statement

$$I \triangleq \text{INSTANCE } N \text{ WITH } q_1 \leftarrow e_1, \ldots, q_n \leftarrow e_n$$

where the q_i are all the declared identifiers of module N, and that N contains the definition

$$F \triangleq e$$

where no λ parameter identifier in e is defined or declared in the current context of M. The INSTANCE statement then adds to the current context of M the definition

(17.5) $I!F \triangleq \overline{e}$

where $\overline{e}$ is obtained from e by substituting e_i for q_i, for all i in $1 \,..\, n$.

A fundamental principle of mathematics is that substitution preserves validity; substituting in a valid formula yields a valid formula. So, we want to define $\overline{e}$ so that, if F is a valid formula in N, then $I!F$ is a valid formula in M.

A simple example shows that the level rule for instantiating nonconstant modules is necessary to preserve the validity of F. Suppose F is defined to equal $\Box[c' = c]_c$, where c is declared in N to be a constant. Then F is a temporal formula asserting that no step changes c. It is valid because a constant has the same value in every state of a behavior. If we allowed an instantiation that substitutes a variable x for the constant c, then $I!F$ would be the formula $\Box[x' = x]_x$. This is not a valid formula because it is false for any behavior in which the value of x changes. Since x is a variable, such a behavior obviously exists. Preserving validity requires that we not allow substitution of a nonconstant for a declared constant when instantiating a nonconstant module. (Since $\Box$ and $'$ are nonconstant operators, this definition of F can appear only in a nonconstant module.)

In ordinary mathematics, there is one tricky problem in making substitution preserve validity. Consider the formula

(17.6) $(n \in Nat) \Rightarrow (\exists\, m \in Nat : m \geq n)$

This formula is valid because it is true for any value of n. Now, suppose we substitute $m + 1$ for n. A naive substitution that simply replaces n by $m + 1$ would yield the formula

(17.7) $(m + 1 \in Nat) \Rightarrow (\exists\, m \in Nat : m \geq m + 1)$

Since the formula $\exists\, m \in Nat : m \geq m + 1$ is equivalent to FALSE, (17.7) is obviously not valid. Mathematicians call this problem *variable capture*; m is "captured" by the quantifier $\exists\, m$. Mathematicians avoid it by the rule that, when substituting for an identifier in a formula, one does not substitute for bound occurrences of the identifier. This rule requires that m be removed from (17.6) by α conversion before $m + 1$ is substituted for n.

Section 17.5.5 defines the meaning of the INSTANCE statement in a way that avoids variable capture. Indeed, formula (17.7) is illegal in TLA$^+$ because the subexpression $m + 1 \in Nat$ is allowed only in a context in which m is defined or declared, in which case m cannot be used as a bound identifier, so the subexpression $\exists\, m \ldots$ is illegal. The α conversion necessary to produce a syntactically well-formed expression makes this kind of variable capture impossible.

The problem of variable capture occurs in a more subtle form in certain nonconstant operators of TLA$^+$, where it is not prevented by the syntactic rules. Most notable of these operators is ENABLED. Suppose x and y are declared variables of module N, and F is defined by

$$F \;\triangleq\; \text{ENABLED}\,(x' = 0 \wedge y' = 1)$$

Then F is equivalent to TRUE, so it is valid in module N. (For any state s, there exists a state t in which $x = 0$ and $y = 1$.) Now suppose z is a declared variable of module M, and let the instantiation be

$$I \;\triangleq\; \text{INSTANCE } N \text{ WITH } x \leftarrow z,\, y \leftarrow z$$

With naive substitution, $I!F$ would equal

$$\text{ENABLED}\,(z' = 0 \wedge z' = 1)$$

which is equivalent to FALSE. (For any state s, there is no state t in which $z = 0$ and $z = 1$ are both true.) Hence, $I!F$ would not be a theorem, so instantiation would not preserve validity.

Naive substitution in a formula of the form ENABLED A does not preserve validity because the primed variables in A are really bound identifiers. The formula ENABLED A asserts that *there exist* values of the primed variables such that A is true. Substituting z' for x' and y' in the ENABLED formula is really substitution for a bound identifier. It isn't ruled out by the syntactic rules of TLA$^+$ because the quantification is implicit.

To preserve validity, we must define $\overline{e}$ in (17.5) so it avoids capture of identifiers implicitly bound in ENABLED expressions. Before performing the substitution, we first replace the primed occurrences of variables in ENABLED expressions with new variable symbols. That is, for each subexpression of e of the form ENABLED A and each declared variable q of module N, we replace every primed occurrence of q in A with a new symbol, which we write $\$q$, that does not appear in A. This new symbol is considered to be bound by the ENABLED operator. For example, the module

```
──────────────────── MODULE N ────────────────────
```
$\text{VARIABLE } u$

$G(v, A) \triangleq \text{ENABLED } (A \lor (\{u, v\}' = \{u, v\}))$

$H \triangleq (u' = u) \land G(u, u' \neq u)$

```
──────────────────────────────────────────────────
```

has as its global definitions the set

$$\begin{aligned} \{\, G &\triangleq \lambda v, A : \text{ENABLED } (A \lor (\{u, v\}' = \{u, v\})), \\ H &\triangleq (u' = u) \land \text{ENABLED } ((u' \neq u) \lor (\{u, u\}' = \{u, u\}))\ \} \end{aligned}$$

The statement

$$I \triangleq \text{INSTANCE } N \text{ WITH } u \leftarrow x$$

adds the following definitions to the current module:

$$\begin{aligned} I!G &\triangleq \lambda v, A : \text{ENABLED } (A \lor (\{\$u, v\}' = \{x, v\})) \\ I!H &\triangleq (x' = x) \land \text{ENABLED } ((\$u' \neq x) \lor (\{\$u, \$u\}' = \{x, x\})) \end{aligned}$$

Observe that $I!H$ does not equal $(x' = x) \land I!G(x, x' \neq x)$, even though H equals $(u' = u) \land G(u, u' \neq u)$ in module N and the instantiation substitutes x for u.

As another example, consider the module

```
──────────────────── MODULE N ────────────────────
```
$\text{VARIABLES } u, v$

$A \triangleq (u' = u) \land (v' \neq v)$

$B(d) \triangleq \text{ENABLED } d$

$C \triangleq B(A)$

```
──────────────────────────────────────────────────
```

The instantiation

$$I \triangleq \text{INSTANCE } N \text{ WITH } u \leftarrow x,\ v \leftarrow x$$

adds the following definitions to the current module:

$$\begin{aligned} I!A &\triangleq (x' = x) \land (x' \neq x) \\ I!B &\triangleq \lambda d : \text{ENABLED } d \\ I!C &\triangleq \text{ENABLED } ((\$u' = x) \land (\$v' \neq x)) \end{aligned}$$

Observe that $I!C$ is not equivalent to $I!B(I!A)$. In fact, $I!C \equiv \text{TRUE}$ and $I!B(I!A) \equiv \text{FALSE}$.

We say that instantiation *distributes* over an operator Op if

$$\overline{Op(e_1, \ldots, e_n)} = Op(\overline{e_1}, \ldots, \overline{e_n})$$

for any expressions e_i, where the overlining operator ($\overline{}$) denotes some arbitrary instantiation. Instantiation distributes over all constant operators—for example, $+$, $\subseteq$, and $\exists$.[9] Instantiation also distributes over most of the nonconstant operators of TLA$^+$, like priming ($'$) and $\Box$.

If an operator *Op* implicitly binds some identifiers in its arguments, then instantiation would not preserve validity if it distributed over *Op*. Our rules for instantiating in an ENABLED expression imply that instantiation does not distribute over ENABLED. It also does not distribute over any operator defined in terms of ENABLED—in particular, the built-in operators WF and SF.

There are two other TLA$^+$ operators that implicitly bind identifiers: the action composition operator "$\cdot$", defined in Section 16.2.3, and the temporal operator $\stackrel{+}{-\!\!\triangleright}$, introduced in Section 10.7. The rule for instantiating an expression $A \cdot B$ is similar to that for ENABLED A—namely, bound occurrences of variables are replaced by a new symbol. In the expression $A \cdot B$, primed occurrences of variables in A and unprimed occurrences in B are bound. We handle a formula of the form $F \stackrel{+}{-\!\!\triangleright} G$ by replacing it with an equivalent formula in which the quantification is made explicit.[10] Most readers won't care, but here's how that equivalent formula is constructed. Let $\mathbf{x}$ be the tuple $\langle x_1, \ldots, x_n \rangle$ of all declared variables; let b, $\widehat{x_1}$, ..., $\widehat{x_n}$ be symbols distinct from the x_i and from any bound identifiers in F or G; and let $\widehat{e}$ be the expression obtained from an expression e by substituting the variables $\widehat{x_i}$ for the corresponding variables x_i. Then $F \stackrel{+}{-\!\!\triangleright} G$ is equivalent to

$$
\text{(17.8)}\quad \boldsymbol{\forall}\, b : \begin{array}[t]{l} (\ \wedge\ (b \in \textsc{boolean}) \wedge \Box[b' = \textsc{false}]_b \\ \quad \wedge\ \boldsymbol{\exists}\, \widehat{x_1}, \ldots, \widehat{x_n} : \widehat{F} \wedge \Box(b \Rightarrow (\mathbf{x} = \widehat{\mathbf{x}}))\ ) \\ \Rightarrow\ \boldsymbol{\exists}\, \widehat{x_1}, \ldots, \widehat{x_n} : \widehat{G} \wedge (\mathbf{x} = \widehat{\mathbf{x}}) \wedge \Box[b \Rightarrow (\mathbf{x}' = \widehat{\mathbf{x}}')]_{\langle b, \mathbf{x}, \widehat{\mathbf{x}} \rangle} \end{array}
$$

Here's a complete statement of the rules for computing $\overline{e}$, for an arbitrary expression e.

1. Remove all $\stackrel{+}{-\!\!\triangleright}$ operators by replacing each subformula of the form $F \stackrel{+}{-\!\!\triangleright} G$ with the equivalent formula (17.8).

2. Recursively perform the following replacements, starting from the innermost subexpressions of e, for each declared variable x of N:

   - For each subexpression of the form ENABLED A, replace each primed occurrence of x in A by a new symbol $\$x$ that is different from any identifier and from any other symbol that occurs in A.

[9] Recall the explanation on pages 320–321 of how we consider $\exists$ to be a second-order operator. Instantiation distributes over $\exists$ because TLA$^+$ does not permit variable capture when substituting in λ expressions.

[10] Replacing ENABLED and "$\cdot$" expressions by equivalent formulas with explicit quantifiers before substituting would result in some surprising instantiations. For example, if N contains the definition $E(A) \triangleq$ ENABLED A, then $I \triangleq$ INSTANCE N would effectively obtain the definition $I!E(A) \triangleq A$.

- For each subexpression of the form $B \cdot C$, replace each primed occurrence of x in B and each unprimed occurrence of x in C by a new symbol $\$x$ that is different from any identifier and from any other symbol that occurs in B or C.

For example, applying these rules to the inner ENABLED expression and to the "$\cdot$" expression converts

$$\text{ENABLED}\,((\text{ENABLED}\,(x' = x))' \land ((y' = x) \cdot (x' = y)))$$

to

$$\text{ENABLED}\,((\text{ENABLED}\,(\$x' = x))' \land ((\$y' = x) \cdot (x' = \$y)))$$

and applying them again to the outer ENABLED expression yields

$$\text{ENABLED}\,((\text{ENABLED}\,(\$x' = \$xx))' \land ((\$y' = x) \cdot (\$xx' = \$y)))$$

where $\$xx$ is some new symbol different from x, $\$x$, and $\$y$.

3. Replace each occurrence of q_i with e_i, for all i in $1 \mathrel{..} n$.

Chapter 18

The Standard Modules

Several standard modules are provided for use in TLA$^+$ specifications. Some of the definitions they contain are subtle—for example, the definitions of the set of real numbers and its operators. Others, such as the definition of $1 \mathrel{..} n$, are obvious. There are two reasons to use standard modules. First, specifications are easier to read when they use basic operators that we're already familiar with. Second, tools can have built-in knowledge of standard operators. For example, the TLC model checker (Chapter 14) has efficient implementations of some standard modules; and a theorem-prover might implement special decision procedures for some standard operators. The standard modules of TLA$^+$ are described here, except for the *RealTime* module, which appears in Chapter 9.

18.1 Module *Sequences*

The *Sequences* module was introduced in Section 4.1 on page 35. Most of the operators it defines have already been explained. The exceptions are

$SubSeq(s,\ m,\ n)$	The subsequence $\langle s[m],\ s[m+1],\ \ldots,\ s[n]\rangle$ consisting of the m^{th} through n^{th} elements of s. It is undefined if $m < 1$ or $n > Len(s)$, except that it equals the empty sequence if $m > n$.
$SelectSeq(s,\ Test)$	The subsequence of s consisting of the elements $s[i]$ such that $Test(s[i])$ equals TRUE. For example, $$PosSubSeq(s) \triangleq \text{LET } IsPos(n) \triangleq n > 0 \text{ IN } SelectSeq(s, IsPos)$$ defines $PosSubSeq(\langle 0,\ 3,\ -2,\ 5\rangle)$ to equal $\langle 3,\ 5\rangle$.