ematical 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

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

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,

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

  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

  $$\text{CASE } p \to e \,\Box\, q \to f$$

  using the relation

  $$\begin{aligned} &\text{CASE } p_1 \to e_1 \,\Box\, \ldots \,\Box\, p_n \to e_n \;= \\ &\qquad \text{CASE } p_1 \to e_1 \,\Box\, (p_2 \vee \ldots \vee p_n) \to (\text{CASE } p_2 \to e_2 \,\Box\, \ldots \,\Box\, p_n \to e_n) \end{aligned}$$

- 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 $\exists\, x : (x \in S) \land e$ 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

VARIABLES x, y 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.
- $\mathbf{\exists}\, x : e$ 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

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*:

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

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

  LET $Op_1 \triangleq d_1$ IN (LET $\ldots$ IN (LET $Op_n \triangleq d_n$ 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