Chapter 11

Advanced Examples

It would be nice to provide an assortment of typical examples that cover most of the specification problems that arise in practice. However, there is no such thing as a typical specification. Every real specification seems to pose its own problems. But we can partition all specifications into two classes, depending on whether or not they contain VARIABLE declarations.

A specification with no variables defines data structures and operations on those structures. For example, the *Sequences* module defines various operations on sequences. When specifying a system, you may need some kind of data structure other than the ones provided by the standard modules like *Sequences* and *Bags*, described in Chapter 18. Section 11.1 gives some examples of data structure specifications.

A system specification contains variables that represent the system's state. We can further divide system specifications into two classes—high-level specifications that describe what it means for a system to be correct, and lower-level specifications that describe what the system actually does. In the memory example of Chapter 5, the linearizable memory specification of Section 5.3 is a high-level specification of correctness, while the write-through cache specification of Section 5.6 describes how a particular algorithm works. This distinction is not precise; whether a specification is high- or low-level is a matter of perspective. But it can be a useful way of categorizing system specifications.

Lower-level system specifications tend to be relatively straightforward. Once the level of abstraction has been chosen, writing the specification is usually just a matter of getting the details right when describing what the system does. Specifying high-level correctness can be much more subtle. Section 11.2 considers a high-level specification problem—formally specifying a multiprocessor memory.

11.1 Specifying Data Structures

Most of the data structures required for writing specifications are mathematically simple and are easy to define in terms of sets, functions, and records. Section 11.1.2 describes the specification of one such structure—a graph. On rare occasions, a specification will require sophisticated mathematical concepts. The only examples I know of are hybrid system specifications, discussed in Section 9.5. There, we used a module for describing the solutions to differential equations. That module is specified in Section 11.1.3 below. Section 11.1.4 considers the tricky problem of defining operators for specifying BNF grammars. Although not the kind of data structure you're likely to need for a system specification, specifying BNF grammars provides a nice little exercise in "mathematization". The module developed in that section is used in Chapter 15 for specifying the grammar of TLA$^+$. But, before specifying data structures, you should know how to make local definitions.

11.1.1 Local Definitions

In the course of specifying a system, we write lots of auxiliary definitions. A system specification may consist of a single formula *Spec*, but we define dozens of other identifiers in terms of which we define *Spec*. These other identifiers often have fairly common names—for example, the identifier *Next* is defined in many specifications. The different definitions of *Next* don't conflict with one another because, if a module that defines *Next* is used as part of another specification, it is usually instantiated with renaming. For example, the *Channel* module is used in module *InnerFIFO* on page 38 with the statement

$$InChan \triangleq \text{INSTANCE } Channel \text{ WITH } \ldots$$

The action *Next* of the *Channel* module is then instantiated as $InChan!Next$, so its definition doesn't conflict with the definition of *Next* in the *InnerFIFO* module.

A module that defines operations on a data structure is likely to be used in an EXTENDS statement, which does no renaming. The module might define some auxiliary operators that are used only to define the operators in which we're interested. For example, we need the *DifferentialEquations* module only to define the single operator *Integrate*. However, *Integrate* is defined in terms of other defined operators with names like *Nbhd* and *IsDeriv*. We don't want these definitions to conflict with other uses of those identifiers in a module that extends *DifferentialEquations*. So, we want the definitions of *Nbhd* and *IsDeriv* to be local to the *DifferentialEquations* module.[1]

[1] We could use the LET construct to put these auxiliary definitions inside the definition of *Integrate*, but that trick wouldn't work if the *DifferentialEquations* module exported other operators besides *Integrate* that were defined in terms of *Nbhd* and *IsDeriv*.

TLA$^+$ provides a LOCAL modifier for making definitions local to a module. If a module M contains the definition

LOCAL $Foo(x) \triangleq \ldots$

then *Foo* can be used inside module M just like any ordinary defined identifier. However, a module that extends or instantiates M does not obtain the definition of *Foo*. That is, the statement EXTENDS M in another module does not define *Foo* in that module. Similarly, the statement

$N \triangleq$ INSTANCE M

does not define $N!Foo$. The LOCAL modifier can also be applied to an instantiation. The statement

LOCAL INSTANCE *Sequences*

in module M incorporates into M the definitions from the *Sequences* module. However, another module that extends or instantiates M does not obtain those definitions. Similarly, a statement like

LOCAL $P(x) \triangleq$ INSTANCE N

makes all the instantiated definitions local to the current module.

The LOCAL modifier can be applied only to definitions and INSTANCE statements. It cannot be applied to a declaration or to an EXTENDS statement, so you *cannot* write either of the following:

LOCAL CONSTANT N

LOCAL EXTENDS *Sequences*

These are not legal statements.

If a module has no CONSTANT or VARIABLE declarations and no submodules, then extending it and instantiating it are equivalent. Thus, the two statements

EXTENDS *Sequences* INSTANCE *Sequences*

are equivalent.

In a module that defines general mathematical operators, I like to make all definitions local except for the ones that users of the module would expect. For example, users expect the *Sequences* module to define operators on sequences, such as *Append*. They don't expect it to define operators on numbers, such as +. The *Sequences* module uses + and other operators defined in the *Naturals* module. But instead of extending *Naturals*, it defines those operators with the statement

LOCAL INSTANCE *Naturals*

The definitions of the operators from *Naturals* are therefore local to *Sequences*. A module that extends the *Sequences* module could then define + to mean something other than addition of numbers.

11.1.2 Graphs

A graph is an example of the kind of simple data structure often used in specifications. Let's now write a *Graphs* module for use in writing system specifications.

We must first decide how to represent a graph in terms of data structures that are already defined—either built-in TLA$^+$ data structures like functions, or ones defined in existing modules. Our decision depends on what kind of graphs we want to represent. Are we interested in directed graphs or undirected graphs? Finite or infinite graphs? Graphs with or without self-loops (edges from a node to itself)? If we are specifying graphs for a particular specification, the specification will tell us how to answer these questions. In the absence of such guidance, let's handle arbitrary graphs. My favorite way of representing both directed and undirected graphs is to specify arbitrary directed graphs, and to define an undirected graph as a directed graph that contains an edge iff it contains the opposite-pointing edge. Directed graphs have a pretty obvious representation: a directed graph consists of a set of nodes and a set of edges, where an edge from node m to node n is represented by the ordered pair $\langle m, n \rangle$.

In addition to deciding how to represent graphs, we must decide how to structure the *Graphs* module. The decision depends on how we expect the module to be used. For a specification that uses a single graph, it is most convenient to define operations on that specific graph. So, we want the *Graphs* module to have (constant) parameters *Node* and *Edge* that represent the sets of nodes and edges of a particular graph. A specification could use such a module with a statement

INSTANCE *Graphs* WITH *Node* $\leftarrow \ldots$, *Edge* $\leftarrow \ldots$

where the "..."s are the sets of nodes and edges of the particular graph appearing in the specification. On the other hand, a specification might use many different graphs. For example, it might include a formula that asserts the existence of a subgraph, satisfying certain properties, of some given graph G. Such a specification needs operators that take a graph as an argument—for example, a *Subgraph* operator defined so *Subgraph*(G) is the set of all subgraphs of a graph G. In this case, the *Graphs* module would have no parameters, and specifications would incorporate it with an EXTENDS statement. Let's write this kind of module.

An operator like *Subgraph* takes a graph as an argument, so we have to decide how to represent a graph as a single value. A graph G consists of a set N of nodes and a set E of edges. A mathematician would represent G as the ordered pair $\langle N, E \rangle$. However, $G.node$ is more perspicuous than $G[1]$, so we represent G as a record with *node* field N and *edge* field E.

Having made these decisions, it's easy to define any standard operator on graphs. We just have to decide what we should define. Here are some generally useful operators:

IsDirectedGraph(G)
: True iff G is an arbitrary directed graph—that is, a record with *node* field N and *edge* field E such that E is a subset of $N \times N$. This operator is useful because a specification might want to assert that something is a directed graph. (To understand how to assert that G is a record with *node* and *edge* fields, see the definition of *IsChannel* in Section 10.3 on page 140.)

DirectedSubgraph(G)
: The set of all subgraphs of a directed graph G. Alternatively, we could define *IsDirectedSubgraph*(H, G) to be true iff H is a subgraph of G. However, it's easy to express *IsDirectedSubgraph* in terms of *DirectedSubgraph*:

$$\mathit{IsDirectedSubgraph}(H,\ G) \ \equiv\ H \in \mathit{DirectedSubgraph}(G)$$

On the other hand, it's awkward to express *DirectedSubgraph* in terms of *IsDirectedSubgraph*:

$$\mathit{DirectedSubgraph}(G) \ =\ \text{CHOOSE } S \ :\ \forall\, H \ :\ (H \in S) \equiv \mathit{IsDirectedSubgraph}(H,\ G)$$

Section 6.1 explains why we can't define a set of all directed graphs, so we had to define the *IsDirectedGraph* operator.

IsUndirectedGraph(G)
UndirectedSubgraph(G)
: These are analogous to the operators for directed graphs. As mentioned above, an undirected graph is a directed graph G such that for every edge $\langle m,\ n\rangle$ in $G.\mathit{edge}$, the inverse edge $\langle n,\ m\rangle$ is also in $G.\mathit{edge}$. Note that *DirectedSubgraph*(G) contains directed graphs that are not undirected graphs—except for certain "degenerate" graphs G, such as graphs with no edges.

Path(G)
: The set of all paths in G, where a path is any sequence of nodes that can be obtained by following edges in the direction they point. This definition is useful because many properties of a graph can be expressed in terms of its set of paths. It is convenient to consider the one-element sequence $\langle n\rangle$ to be a path, for any node n.

AreConnectedIn(m, n, G)
: True iff there is a path from node m to node n in G. The utility of this operator becomes evident when you try defining various common graph properties, like connectivity.

There are any number of other graph properties and classes of graphs that we might define. Let's define these two:

IsStronglyConnected(G)
: True iff G is strongly connected, meaning that there is a path from any node to any other node. For an undirected graph, strongly connected is equivalent to the ordinary definition of connected.

IsTreeWithRoot$(G,\ r)$
: True iff G is a tree with root r, where we represent a tree as a graph with an edge from each nonroot node to its parent. Thus, the parent of a nonroot node n equals

$$\text{CHOOSE } m \in G.node : \langle n,\ m\rangle \in G.edge$$

The *Graphs* module appears on the next page. By now, you should be able to work out for yourself the meanings of all the definitions.

11.1.3 Solving Differential Equations

Section 9.5 on page 132 describes how to specify a hybrid system whose state includes a physical variable satisfying an ordinary differential equation. The specification uses an operator *Integrate* such that $Integrate(D,\ t_0,\ t_1,\ \langle x_0, \ldots, x_{n-1}\rangle)$ is the value at time t_1 of the n-tuple

$$\langle x,\ dx/dt,\ \ldots,\ d^{n-1}x/dt^{n-1}\rangle$$

where x is a solution to the differential equation

$$D[t,\ x,\ dx/dt,\ \ldots,\ d^n x/dt^n] = 0$$

whose 0^{th} through $(n-1)^{\text{st}}$ derivatives at time t_0 are $x_0,\ \ldots\ ,\ x_{n-1}$. We assume that there is such a solution and that it is unique. Defining *Integrate* illustrates how to express sophisticated mathematics in TLA$^+$.

We start by defining some mathematical notation that we will use to define the derivative. As usual, we obtain from the *Reals* module the definitions of the set *Real* of real numbers and of the ordinary arithmetic operators. Let *PosReal* be the set of all positive reals:

$$PosReal \triangleq \{r \in Real : r > 0\}$$

and let $OpenInterval(a, b)$ be the open interval from a to b (the set of numbers greater than a and less than b):

$$OpenInterval(a, b) \triangleq \{s \in Real : (a < s) \land (s < b)\}$$

(Mathematicians usually write this set as $(a,\ b)$.) Let's also define $Nbhd(r, e)$ to be the open interval of width $2e$ centered at r:

$$Nbhd(r, e) \triangleq OpenInterval(r - e, r + e)$$

MODULE *Graphs*

A module that defines operators on graphs. A directed graph is represented as a record whose *node* field is the set of nodes and whose *edge* field is the set of edges, where an edge is an ordered pair of nodes.

LOCAL INSTANCE *Naturals*

LOCAL INSTANCE *Sequences*

$IsDirectedGraph(G) \triangleq$ True iff G is a directed graph.

$\land\ G = [node \mapsto G.node,\ edge \mapsto G.edge]$
$\land\ G.edge \subseteq (G.node \times G.node)$

$DirectedSubgraph(G) \triangleq$ The set of all (directed) subgraphs of a directed graph.

$\{H \in [node : \text{SUBSET } G.node,\ edge : \text{SUBSET } (G.node \times G.node)] :$
$\quad IsDirectedGraph(H) \land H.edge \subseteq G.edge\}$

$IsUndirectedGraph(G) \triangleq$ An undirected graph is a directed graph in which every edge has an oppositely directed one.

$\land\ IsDirectedGraph(G)$
$\land\ \forall\, e \in G.edge : \langle e[2],\ e[1]\rangle \in G.edge$

$UndirectedSubgraph(G) \triangleq$ The set of (undirected) subgraphs of an undirected graph.

$\{H \in DirectedSubgraph(G) : IsUndirectedGraph(H)\}$

$Path(G) \triangleq$ The set of paths in G, where a path is represented as a sequence of nodes.

$\{p \in Seq(G.node) : \land\ p \neq \langle\,\rangle$
$\qquad\qquad \land\ \forall\, i \in 1\ ..\ (Len(p) - 1) : \langle p[i],\ p[i+1]\rangle \in G.edge\}$

$AreConnectedIn(m,\ n,\ G) \triangleq$ True iff there is a path from m to n in graph G.

$\exists\, p \in Path(G) : (p[1] = m) \land (p[Len(p)] = n)$

$IsStronglyConnected(G) \triangleq$ True iff graph G is strongly connected.

$\forall\, m,\ n \in G.node : AreConnectedIn(m,\ n,\ G)$

$IsTreeWithRoot(G,\ r) \triangleq$ True if G is a tree with root r, where edges point from child to parent.

$\land\ IsDirectedGraph(G)$
$\land\ \forall\, e \in G.edge : \land\ e[1] \neq r$
$\qquad\qquad\qquad\qquad \land\ \forall f \in G.edge : (e[1] = f[1]) \Rightarrow (e = f)$
$\land\ \forall\, n \in G.node : AreConnectedIn(n,\ r,\ G)$

Figure 11.1: A module for specifying operators on graphs.

To explain the definitions, we need some notation for the derivative of a function. It's rather difficult to make mathematical sense of the usual notation df/dt for the derivative of f. (What exactly is t?) So, let's use a mathematically simpler notation and write the n^{th} derivative of the function f as $f^{(n)}$. (We don't have to use TLA$^+$ notation because differentiation will not appear explicitly in our definitions.) Recall that $f^{(0)}$, the 0^{th} derivative of f, equals f.

We can now start to define *Integrate*. If a and b are numbers, *InitVals* is an n-tuple of numbers, and D is a function from $(n+2)$-tuples of numbers to numbers, then

$$Integrate(D,\ a,\ b,\ InitVals) \ =\ \langle f^{(0)}[b],\ \ldots,\ f^{(n-1)}[b]\rangle$$

where f is the function satisfying the following two conditions:

- $D[r,\ f^{(0)}[r],\ f^{(1)}[r],\ \ldots, f^{(n)}[r]] \ =\ 0$, for all r in some open interval containing a and b.
- $\langle f^{(0)}[a],\ \ldots,\ f^{(n-1)}[a]\rangle \ =\ InitVals$

We want to define $Integrate(D,\ a,\ b,\ InitVals)$ in terms of this function f, which we can specify using the CHOOSE operator. It's easiest to choose not just f, but its first n derivatives as well. So, we choose a function g such that $g[i] = f^{(i)}$ for $i \in 0\ldots n$. The function g maps numbers in $0\ldots n$ into functions. More precisely, g is an element of

$$[0\ldots n \to [OpenInterval(a-e,\ b+e) \to Real]\,]$$

for some positive e. It is the function in this set that satisfies the following conditions:

1. $g[i]$ is the i^{th} derivative of $g[0]$, for all $i \in 0\ldots n$.
2. $D[r,\ g[0][r],\ \ldots,\ g[n][r]\,] \ =\ 0$, for all r in $OpenInterval(a-e,\ b+e)$.
3. $\langle\, g[0][a],\ \ldots,\ g[n-1][a]\rangle = InitVals$

We now have to express these conditions formally.

To express the first condition, we will define *IsDeriv* so that $IsDeriv(i,\ df,\ f)$ is true iff df is the i^{th} derivative of f. More precisely, this will be the case if f is a real-valued function on an open interval; we don't care what $IsDeriv(i,\ df,\ f)$ equals for other values of f. Condition 1 is then

$$\forall\, i \in 1\ldots n\ :\ IsDeriv(i,\ g[i],\ g[0])$$

To express the second condition formally, without the "...", we reason as follows:

$$\begin{array}{ll}
D[r,\ g[0][r],\ \ldots,\ g[n][r]\,] & \\
\ =\ D[\,\langle r,\ g[0][r],\ \ldots,\ g[n][r]\rangle\,] & \text{See page 50.} \\
\ =\ D[\,\langle r\rangle \circ \langle g[0][r],\ \ldots,\ g[n][r]\rangle\,] & \text{Tuples are sequences} \\
\ =\ D[\,\langle r\rangle \circ [i \in 1\ldots(n+1) \mapsto g[i-1][r]]\,] & \text{An } (n{+}1)\text{-tuple is a function with domain } 1\ldots n{+}1.
\end{array}$$

The third condition is simply

$$\forall\, i \in 1\,..\,n\,:\, g[i-1][a] = \mathit{InitVals}[i]$$

We can therefore write the formula specifying g as

$$\begin{array}{l}\exists\, e \in \mathit{PosReal}\,:\, \wedge\, g \in [0\,..\,n \to [\mathit{OpenInterval}(a-e,\, b+e) \to \mathit{Real}]\,] \\ \qquad\qquad \wedge\, \forall\, i \in 1\,..\,n\,:\, \wedge\, \mathit{IsDeriv}(i,\, g[i],\, g[0]) \\ \qquad\qquad\qquad\qquad\quad \wedge\, g[i-1][a] = \mathit{InitVals}[i] \\ \qquad\qquad \wedge\, \forall\, r \in \mathit{OpenInterval}(a-e,\, b+e)\,: \\ \qquad\qquad\quad D[\,\langle r\rangle \circ [i \in 1\,..\,(n+1) \mapsto g[i-1][r]]\,] \;=\; 0\end{array}$$

where n is the length of *InitVals*. The value of $\mathit{Integrate}(D,\, a,\, b,\, \mathit{InitVals})$ is the tuple $\langle g[0][b], \ldots, g[n-1][b]\rangle$, which can be written formally as

$$[i \in 1\,..\,n \mapsto g[i-1][b]]$$

To complete the definition of *Integrate*, we now define the operator *IsDeriv*. It's easy to define the i^{th} derivative inductively in terms of the first derivative. So, we define $\mathit{IsFirstDeriv}(df,\, f)$ to be true iff df is the first derivative of f, assuming that f is a real-valued function whose domain is an open interval. Our definition actually works if the domain of f is any open set.[2] Elementary calculus tells us that $df[r]$ is the derivative of f at r iff

$$df[r] \;=\; \lim_{s \to r} \frac{f[s] - f[r]}{s - r}$$

The classical "δ-ϵ" definition of the limit states that this is true iff, for every $\epsilon > 0$, there is a $\delta > 0$ such that $0 < |s - r| < \delta$ implies

$$\left| df[r] - \frac{f[s] - f[r]}{s - r} \right| < \epsilon$$

Stated formally, this condition is

$$\begin{array}{l}\forall\, \epsilon \in \mathit{PosReal}\,: \\ \quad \exists\, \delta \in \mathit{PosReal}\,: \\ \qquad \forall\, s \in \mathit{Nbhd}(r,\, \delta) \setminus \{r\}\,:\, \dfrac{f[s] - f[r]}{s - r} \in \mathit{Nbhd}(df[r],\, \epsilon)\end{array}$$

We define $\mathit{IsFirstDeriv}(df,\, f)$ to be true iff the domains of df and f are equal, and this condition holds for all r in their domain.

The definitions of *Integrate* and all the other operators introduced above appear in the *DifferentialEquations* module of Figure 11.2 on the next page. The LOCAL construct described in Section 11.1.1 above is used to make all these definitions local to the module, except for the definition of *Integrate*.

[2] A set S is open iff for every $r \in S$ there exists an $\epsilon > 0$ such that the interval from $r - \epsilon$ to $r + \epsilon$ is contained in S.

MODULE *DifferentialEquations*

This module defines the operator *Integrate* for specifying the solution to a differential equation. If a and b are reals with $a \leq b$; *InitVals* is an n-tuple of reals; and D is a function from $(n+1)$-tuples of reals to reals; then this is the n-tuple of values

$$\langle f[b],\ \frac{df}{dt}[b],\ \ldots,\ \frac{d^{n-1}f}{dt^{n-1}}[b]\rangle$$

where f is the solution to the differential equation

$$D[t,\ f,\ \frac{df}{dt},\ \ldots,\ \frac{d^{n}f}{dt^{n}}]\ =\ 0$$

such that

$$\langle f[a],\ \frac{df}{dt}[a],\ \ldots,\ \frac{d^{n-1}f}{dt^{n-1}}[a]\rangle\ =\ \mathit{InitVals}$$

LOCAL INSTANCE *Reals*
LOCAL INSTANCE *Sequences*
LOCAL $PosReal \triangleq \{r \in Real : r > 0\}$
LOCAL $OpenInterval(a, b) \triangleq \{s \in Real : (a < s) \land (s < b)\}$
LOCAL $Nbhd(r, e) \triangleq OpenInterval(r - e, r + e)$

The INSTANCE statement and these definitions are local, so a module that extends this one obtains only the definition of *Integrate*.

LOCAL $IsFirstDeriv(df, f) \triangleq$
$\land\ df \in [\text{DOMAIN } f \to Real]$
$\land\ \forall\, r \in \text{DOMAIN } f :$
$\quad \forall\, e \in PosReal :$
$\quad\quad \exists\, d \in PosReal :$
$\quad\quad\quad \forall\, s \in Nbhd(r,\ d) \setminus \{r\} : (f[s] - f[r])/(s - r) \in Nbhd(df[r],\ e)$

Assuming DOMAIN f is an open subset of *Real*, this is true iff f is differentiable and df is its first derivative. Recall that the derivative of f at r is the number $df[r]$ satisfying the following condition: for every ϵ there exists a δ such that $0 < |s - r| < \delta$ implies $|df[r] - (f[s] - f[r])/(s - r)| < \epsilon$.

LOCAL $IsDeriv(n,\ df,\ f) \triangleq$

True iff f is n times differentiable and df is its n^{th} derivative.

LET $IsD[k \in 0\ ..\ n,\ g \in [\text{DOMAIN } f \to Real]\,] \triangleq$ $\quad IsD[k, g] = IsDeriv(k, g, f)$
IF $k = 0$ THEN $g = f$
ELSE $\exists\, gg \in [\text{DOMAIN } f \to Real] : \land\ IsFirstDeriv(g, gg)$
$\land\ IsD[k - 1, gg]$
IN $IsD[n,\ df]$

$Integrate(D,\ a,\ b,\ InitVals) \triangleq$
LET $n \triangleq Len(InitVals)$
$gg \triangleq$ CHOOSE $g : \exists\, e \in PosReal : \land\ g \in [0\ ..\ n \to [OpenInterval(a - e,\ b + e) \to Real]\,]$
$\land\ \forall\, i \in 1\ ..\ n : \land\ IsDeriv(i, g[i], g[0])$
$\land\ g[i-1][a] = InitVals[i]$
$\land\ \forall\, r \in OpenInterval(a - e, b + e) :$
$D[\,\langle r\rangle \circ [i \in 1\ ..\ (n+1) \mapsto g[i-1][r]]\,]\ =\ 0$
IN $[i \in 1\ ..\ n \mapsto gg[i-1][b]]$

Figure 11.2: A module for specifying the solution to a differential equation.

11.1.4 BNF Grammars

BNF, which stands for Backus-Naur Form, is a standard way of describing the syntax of computer languages. This section develops the *BNFGrammars* module, which defines operators for writing BNF grammars. A BNF grammar isn't the kind of data structure that arises in system specification, and TLA$^+$ is not particularly well suited to specifying one. Its syntax doesn't allow us to write BNF grammars exactly the way we'd like, but we can come reasonably close. Moreover, I think it's fun to use TLA$^+$ to specify its own syntax. So, module *BNFGrammars* is used in Chapter 15 to specify part of the syntax of TLA$^+$, as well as in Chapter 14 to specify the syntax of the TLC model checker's configuration file.

Let's start by reviewing BNF grammars. Consider the little language SE of simple expressions described by the BNF grammar

$$\begin{array}{lll} \textit{expr} & ::= & \textbf{ident} \mid \textit{expr}\ \textbf{op}\ \textit{expr} \mid (\ \textit{expr}\ ) \mid \text{LET}\ \textit{def}\ \text{IN}\ \textit{expr} \\ \textit{def} & ::= & \textbf{ident} == \textit{expr} \end{array}$$

where **op** is some class of infix operators like $+$, and **ident** is some class of identifiers such as *abc* and x. The language SE contains expressions like

$$abc\ +\ (\text{LET}\ x == y + abc\ \text{IN}\ x * x)$$

Let's represent this expression as the sequence

$$\langle\ \text{“abc”},\ \text{“+”},\ \text{“(”},\ \text{“LET”},\ \text{“x”},\ \text{“==”},\ \text{“y”},\ \text{“+”},\ \text{“abc”},\ \text{“IN”},\ \text{“x”},\ \text{“*”},\ \text{“x”},\ \text{“)”}\ \rangle$$

of strings. The strings such as “abc” and “+” appearing in this sequence are usually called *lexemes*. In general, a sequence of lexemes is called a *sentence*; and a set of sentences is called a *language*. So, we want to define the language SE to consist of the set of all such sentences described by the BNF grammar.[3]

To represent a BNF grammar in TLA$^+$, we must assign a mathematical meaning to nonterminal symbols like *def*, to terminal symbols like **op**, and to the grammar's two productions. The method that I find simplest is to let the meaning of a nonterminal symbol be the language that it generates. Thus, the meaning of *expr* is the language SE itself. I define a *grammar* to be a function G such that, for any string “str”, the value of $G[\text{“str”}]$ is the language generated by the nonterminal *str*. Thus, if G is the BNF grammar above, then $G[\text{“expr”}]$ is the complete language SE, and $G[\text{“def”}]$ is the language defined by the production for *def*, which contains sentences like

$$\langle\ \text{“y”},\ \text{“==”},\ \text{“qq”},\ \text{“*”},\ \text{“wxyz”}\ \rangle$$

[3]BNF grammars are also used to specify how an expression is parsed—for example, that $a + b * c$ is parsed as $a + (b * c)$ rather than $(a + b) * c$. By considering the grammar to specify only a set of sentences, we are deliberately not capturing that use in our TLA$^+$ representation of BNF grammars.

Instead of letting the domain of G consist of just the two strings "expr" and "def", it turns out to be more convenient to let its domain be the entire set STRING of strings, and to let $G[s]$ be the empty language (the empty set) for all strings s other than "expr" and "def". So, a grammar is a function from the set of all strings to the set of sequences of strings. We can therefore define the set *Grammar* of all grammars by

$$Grammar \triangleq [\text{STRING} \to \text{SUBSET } Seq(\text{STRING})]$$

In describing the mathematical meaning of records, Section 5.2 explained that $r.ack$ is an abbreviation for $r[\text{"ack"}]$. This is the case even if r isn't a record. So, we can write $G.op$ instead of $G[\text{"op"}]$. (A grammar isn't a record because its domain is the set of all strings rather than a finite set of strings.)

A terminal like **ident** can appear anywhere to the right of a "::=" that a nonterminal like *expr* can, so a terminal should also be a set of sentences. Let's represent a terminal as a set of sentences, each of which is a sequence consisting of a single lexeme. Let a *token* be a sentence consisting of a single lexeme, so a terminal is a set of tokens. For example, the terminal **ident** is a set containing tokens such as $\langle \text{"abc"} \rangle$, $\langle \text{"x"} \rangle$, and $\langle \text{"qq"} \rangle$. Any terminal appearing in the BNF grammar must be represented by a set of tokens, so the == in the grammar for SE is the set $\{\langle \text{"=="} \rangle\}$. Let's define the operator *tok* by

tok is short for *token*.

$$tok(s) \triangleq \{\langle s \rangle\}$$

so we can write this set of tokens as $tok(\text{"=="})$.

A production expresses a relation between the values of $G.str$ for some grammar G and some strings "str". For example, the production

$$def \ ::= \ \textbf{ident} \ \texttt{==} \ expr$$

asserts that a sentence s is in $G.def$ iff it has the form $i \circ \langle \text{"=="} \rangle \circ e$ for some token i in **ident** and some sentence e in $G.expr$. In mathematics, a formula about G must mention G (perhaps indirectly by using a symbol defined in terms of G). So, we can try writing this production in TLA$^+$ as

$$G.def \ ::= \ ident \ \ tok(\text{"=="}) \ \ G.expr$$

In the expression to the right of the ::=, adjacency is expressing some operation. Just as we have to make multiplication explicit by writing $2 * x$ instead of $2x$, we must express this operation by an explicit operator. Let's use &, so we can write the production as

$$G.def \ ::= \ ident \ \& \ tok(\text{"=="}) \ \& \ G.expr \tag{11.1}$$

This expresses the desired relation between the sets $G.def$ and $G.expr$ of sentences if ::= is defined to be equality and & is defined so that $L \ \& \ M$ is the

set of all sentences obtained by concatenating a sentence in L with a sentence in M:

$$L \ \& \ M \ \triangleq \ \{s \circ t : s \in L, t \in M\}$$

The production

$$expr \ ::= \ \textbf{ident} \mid expr \ \textbf{op} \ expr \mid (\ expr\ ) \mid \text{LET } def \text{ IN } expr$$

can similarly be expressed as

The precedence rules of TLA$^+$ imply that $a \mid b \ \& \ c$ is interpreted as $a \mid (b \ \& \ c)$.

$$\begin{array}{lll} (11.2) & G.expr & ::= \quad ident \\ & & \quad \mid \ G.expr \ \& \ op \ \& \ G.expr \\ & & \quad \mid \ tok(\text{“(”}) \ \& \ G.exp \ \& \ tok(\text{“)”}) \\ & & \quad \mid \ tok(\text{“LET”}) \ \& \ G.def \ \& \ tok(\text{“IN”}) \ \& \ G.expr \end{array}$$

This expresses the desired relation if $\mid$ (which means *or* in the BNF grammar) is defined to be set union ($\cup$).

We can also define the following operators that are sometimes used in BNF grammars:

- *Nil* is defined so that $Nil \ \& \ S$ equals S for any set S of sentences:
$$Nil \ \triangleq \ \{\langle\rangle\}$$
- L^+ equals $L \mid L \ \& \ L \mid L \ \& \ L \ \& \ L \mid \ldots$:
$$\begin{array}{lll} L^+ & \triangleq & \text{LET } LL[n \in Nat] \ \triangleq \quad LL[n] = L \mid \ldots \mid \overbrace{L \ \& \ \ldots \ L}^{n+1 \text{ copies}}. \\ & & \quad \text{IF } n = 0 \text{ THEN } L \\ & & \qquad \text{ELSE } LL[n-1] \mid LL[n-1] \ \& \ L \\ & & \text{IN} \quad \text{UNION } \{LL[n] : n \in Nat\} \end{array}$$

L^+ is typed `L^+` and L^* is typed `L^*`.

- L^* equals $Nil \mid L \mid L \ \& \ L \mid L \ \& \ L \ \& \ L \mid \ldots$:
$$L^* \ \triangleq \ Nil \mid L^+$$

The BNF grammar for SE consists of two productions, expressed by the TLA$^+$ formulas (11.1) and (11.2). The entire grammar is the single formula that is the conjunction of these two formulas. We must turn this formula into a mathematical definition of a grammar *GSE*, which is a function from strings to languages. The formula is an assertion about a grammar G. We define *GSE* to be the smallest grammar G satisfying the conjunction of (11.1) and (11.2), where grammar G_1 smaller than G_2 means that $G_1[s] \subseteq G_2[s]$ for every string s. To express this in TLA$^+$, we define an operator *LeastGrammar* so that $LeastGrammar(P)$ is the smallest grammar G satisfying $P(G)$:

$$\begin{array}{l} LeastGrammar(P(\_)) \ \triangleq \\ \quad \text{CHOOSE } G \in Grammar : \\ \qquad \land \ P(G) \\ \qquad \land \ \forall H \in Grammar : P(H) \Rightarrow (\forall s \in \text{STRING} : G[s] \subseteq H[s]) \end{array}$$

Letting $P(G)$ be the conjunction of (11.1) and (11.2), we can define the grammar GSE to be $LeastGrammar(P)$. We can then define the language SE to equal $GSE.expr$. The smallest grammar G satisfying a formula P must have $G[s]$ equal to the empty language for any string s that doesn't appear in P. Thus, $GSE[s]$ equals the empty language $\{\}$ for any string s other than "expr" and "def".

To complete our specification of GSE, we must define the sets *ident* and *op* of tokens. We can define the set *op* of operators by enumerating them—for example:

$$op \triangleq tok(\text{“+”}) \mid tok(\text{“}-\text{”}) \mid tok(\text{“*”}) \mid tok(\text{“/”})$$

To express this a little more compactly, let's define $Tok(S)$ to be the set of all tokens formed from elements in the set S of lexemes:

$$Tok(S) \triangleq \{\langle s \rangle : s \in S\}$$

We can then write

$$op \triangleq Tok(\{\text{“+”}, \text{“}-\text{”}, \text{“*”}, \text{“/”}\})$$

Let's define *ident* to be the set of tokens whose lexemes are words made entirely of lower-case letters, such as "abc", "qq", and "x". To learn how to do that, we must first understand what strings in TLA$^+$ really are. In TLA$^+$, a string is a sequence of characters. (We don't care, and the semantics of TLA$^+$ doesn't specify, what a character is.) We can therefore apply the usual sequence operators on them. For example, $Tail(\text{“abc”})$ equals "bc", and "abc" $\circ$ "de" equals "abcde".

See Section 16.1.10 on page 307 for more about strings. Remember that we take *sequence* and *tuple* to be synonymous.

The operators like & that we just defined for expressing BNF were applied to sets of sentences, where a sentence is a sequence of lexemes. These operators can be applied just as well to sets of sequences of any kind—including sets of strings. For example, {"one", "two"} & {"s"} equals {"ones", "twos"}, and {"ab"}$^+$ is the set consisting of all the strings "ab", "abab", "ababab", etc. So, we can define *ident* to equal $Tok(Letter^+)$, where *Letter* is the set of all lexemes consisting of a single lower-case letter:

$$Letter \triangleq \{\text{“a”}, \text{“b”}, \ldots, \text{“z”}\}$$

Writing this definition out in full (without the "...") is tedious. We can make this a little easier as follows. We first define $OneOf(s)$ to be the set of all one-character strings made from the characters of the string s:

$$OneOf(s) \triangleq \{\langle s[i] \rangle : i \in \text{DOMAIN } s\}$$

We can then define

$$Letter \triangleq OneOf(\text{“abcdefghijklmnopqrstuvwxyz”})$$

```
GSE ≜ LET op ≜ Tok( {"+", "−", "∗", "/"} )
        ident ≜ Tok( OneOf("abcdefghijklmnopqrstuvwxyz")^+ )
        P(G) ≜ ∧ G.expr ::=      ident
                               |  G.expr & op & G.expr
                               |  tok("(") & G.expr & tok(")")
                               |  tok("LET") & G.def & tok("IN") & G.expr
               ∧ G.def  ::=  ident & tok("==") & G.expr
    IN  LeastGrammar(P)
```

Figure 11.3: The definition of the grammar *GSE* for the language SE.

The complete definition of the grammar *GSE* appears in Figure 11.3 on this page.

All the operators we've defined here for specifying grammars are grouped into module *BNFGrammars*, which appears in Figure 11.4 on the next page.

Using TLA$^+$ to write ordinary BNF grammars is a bit silly. However, ordinary BNF grammars are not very convenient for describing the syntax of a complicated language like TLA$^+$. In fact, they can't describe the alignment rules for its bulleted lists of conjuncts and disjuncts. Using TLA$^+$ to specify such a language is not so silly. In fact, a TLA$^+$ specification of the complete syntax of TLA$^+$ was written as part of the development of the Syntactic Analyzer, described in Chapter 12. Although valuable when writing a TLA$^+$ parser, this specification isn't very helpful to an ordinary user of TLA$^+$, so it does not appear in this book.

11.2 Other Memory Specifications

Section 5.3 specifies a multiprocessor memory. The specification is unrealistically simple for three reasons: a processor can have only one outstanding request at a time, the basic correctness condition is too restrictive, and only simple read and write operations are provided. (Real memories provide many other operations, such as partial-word writes and cache prefetches.) We now specify a memory that allows multiple outstanding requests and has a realistic, weaker correctness condition. To keep the specification short, we still consider only the simple operations of reading and writing one word of memory.

11.2.1 The Interface

The first thing we must do to specify a memory is determine the interface. The interface we choose depends on the purpose of the specification. There are many different reasons why we might be specifying a multiprocessor memory. We could