11 Compiler II: Code Generation

When I am working on a problem, I never think about beauty. But when I have finished, if the solution is not beautiful, I know it is wrong.

—R. Buckminster Fuller (1895–1993)

Most programmers take compilers for granted. But if you stop to think about it, the ability to translate a high-level program into binary code is almost like magic. In Nand to Tetris we devote four chapters (7–11) for demystifying this magic. Our hands-on methodology is based on developing a compiler for Jack—a simple, modern object-based language. As with Java and C#, the overall Jack compiler is based on two tiers: a virtual machine (VM) *back end* that translates VM commands into machine language and a *front end* compiler that translates Jack programs into VM code. Building a compiler is a challenging undertaking, so we divide it further into two conceptual modules: a *syntax analyzer*, developed in chapter 10, and a *code generator*—the subject of this chapter.

The syntax analyzer was built in order to develop, and demonstrate, a capability for parsing high-level programs into their underlying syntactical elements. In this chapter we'll morph the analyzer into a full-scale compiler: a program that converts the parsed elements into VM commands designed to execute on the abstract virtual machine described in chapters 7–8. This approach follows the modular analysis-synthesis paradigm that informs the construction of well-designed compilers. It also captures the very essence of translating text from one language to another: first, one uses the source language's *syntax* for analyzing the source text and figuring out its underlying *semantics*, that is, what the text seeks to say; next, one reexpresses the parsed semantics using the syntax of the target language. In the context of this chapter, the source and the target are Jack and the VM language, respectively.

Modern high-level programming languages are rich and powerful. They allow defining and using elaborate abstractions like functions and objects, expressing algorithms using elegant statements, and building data structures of unlimited complexity. In contrast, the hardware platforms on which these programs ultimately run are spartan and minimal. Typically, they offer little more than a set of registers for storage and a set of primitive instructions for processing. Thus, the translation of programs from high level to low level is a challenging feat. If the target platform is a virtual machine and not the barebone hardware, life is somewhat easier since abstract VM commands are not as primitive as concrete machine instructions. Still, the gap between the expressiveness of a high-level language and

that of a VM language is wide and challenging.

The chapter begins with a general discussion of *code generation*, divided into six sections. First, we describe how compilers use *symbol tables* for mapping symbolic variables onto virtual memory segments. Next, we present algorithms for compiling *expressions* and *strings* of characters. We then present techniques for compiling *statements* like let, if, while, do, and return. Taken together, the ability to compile *variables*, *expressions*, and *statements* forms a foundation for building compilers for simple, procedural, C-like languages. This sets the stage for the remainder of section 11.1, in which we discuss the compilation of *objects* and *arrays*.

Section 11.2 (Specification) provides guidelines for mapping Jack programs on the VM platform and language, and section 11.3 (Implementation) proposes a software architecture and an API for completing the compiler's development. As usual, the chapter ends with a Project section, providing step-by-step guidelines and test programs for completing the compiler's construction, and a Perspective section that comments on various things that were left out from the chapter.

So what's in it for you? Although many professionals are eager to understand how compilers work, few end up getting their hands dirty and building a compiler from the ground up. That's because the cost of this experience—at least in academia—is typically a daunting, full-semester elective course. Nand to Tetris packs the key elements of this experience into four chapters and projects, culminating in the present chapter. In the process, we discuss and illustrate the key algorithms, data structures, and programming tricks underlying the construction of typical compilers. Seeing these clever ideas and techniques in action leads one to marvel, once again, at how human ingenuity can dress up a primitive switching machine to look like something approaching magic.

11.1 Code Generation

High-level programmers work with abstract building blocks like *variables*, *expressions*, *statements*, *subroutines*, *objects* and *arrays*. Programmers use these abstract building blocks for describing what they want the program to do. The job of the compiler is to translate this semantics into a language that a target computer understands.

In our case, the target computer is the virtual machine described in chapters 7–8. Thus, we have to figure out how to systematically translate *expressions*, *statements*, *subroutines*, and the handling of *variables*, *objects*, and *arrays* into sequences of stack-based VM commands that execute the desired semantics on the target virtual machine. We don't have to worry about the subsequent translation of VM programs into machine language, since we already took care of this royal headache in projects 7–8. Thank goodness for two-tier compilation, and for modular design.

Throughout the chapter, we present compilation examples of various parts of the Point class presented previously in the book. We repeat the class declaration in figure 11.1, which

illustrates most of the features of the Jack language. We advise taking a quick look at this Jack code now to refresh your memory about the Point class functionality. You will then be ready to delve into the illuminating journey of systematically reducing this high-level functionality—and any other similar object-based program—into VM code.

```
/** Represents a two-dimensional point.
    File name: Point.jack. */
class Point {

   // The coordinates of this point:
   field int x, y

   // The number of Point objects constructed so far:
   static int pointCount;

   /** Constructs a two-dimensional point and
       initializes it with the given coordinates. */
   constructor Point new(int ax, int ay) {
      let x = ax;
      let y = ay;
      let pointCount = pointCount + 1;
      return this;
   }

   /** Returns the x coordinate of this point. */
   method int getx() { return x; }

   /** Returns the y coordinate of this point. */
   method int gety() { return y; }

   /** Returns the number of Point  constructed so far. */
   function int getPointCount() {
      return pointCount;
   }

   // Class declaration continues on top right.
```

```
   /** Returns a point which is this point plus
       the other point. */
   method Point plus(Point other) {
      return Point.new(x + other.getx(),
                       y + other.gety());
   }

   /** Returns the Euclidean distance between
       this and the other point. */
   method int distance(Point other) {
      var int dx, dy;
      let dx = x - other.getx();
      let dy = y - other.gety();
      return Math.sqrt((dx*dx) + (dy*dy));
   }

   /** Prints this point, as "(x,y)" */
   method void print() {
      do Output.printString("(");
      do Output.printInt(x);
      do Output.printString(",");
      do Output.printInt(y);
      do Output.printString(")");
      return;
   }

} // End of Point class declaration.
```

Figure 11.1 The Point class. This class features all the possible variable kinds (field, static, local, and argument) and subroutine kinds (constructor, method, and function), as well as subroutines that return primitive types, object types, and void subroutines. It also illustrates function calls, constructor calls, and method calls on the current object (this) and on other objects.

11.1.1 Handling Variables

One of the basic tasks of compilers is mapping the variables declared in the source high-level program onto the host RAM of the target platform. For example, consider Java: int variables are designed to represent 32-bit values; long variables, 64-bit values; and so on. If the host RAM happens to be 32-bit wide, the compiler will map int and long variables on one memory word and on two consecutive memory words, respectively. In Nand to Tetris there are no mapping complications: all the primitive types in Jack (int, char, and boolean) are 16-bit wide, and so are the addresses and words of the Hack RAM. Thus, every Jack variable, including pointer variables holding 16-bit address values, can be mapped on exactly one word in memory.

The second challenge faced by compilers is that variables of different *kinds* have different life cycles. Class-level static variables are shared globally by all the subroutines in the class. Therefore, a single copy of each static variable should be kept alive during the complete duration of the program's execution. Instance-level field variables are treated differently: each object (instance of the class) must have a private set of its field variables, and, when the object is no longer needed, this memory must be freed. Subroutine-level local and argument variables are created each time a subroutine starts running and must be freed when the subroutine terminates.

That's the bad news. The good news is that we've already handled all these difficulties. In our two-tier compiler architecture, memory allocation and deallocation are delegated to the VM level. All we have to do now is map Jack *static* variables on static 0, static 1, static 2, …; *field* variables on this 0, this 1, …; *local* variables on local 0, local 1, …; and *argument* variables on argument 0, argument 1, …. The subsequent mapping of the virtual memory segments on the host RAM, as well as the intricate management of their run-time life cycles, are completely taken care of by the VM implementation.

Recall that this implementation was not achieved easily: we had to work hard to generate assembly code that dynamically maps the virtual memory segments on the host RAM as a side effect of realizing the function call-and-return protocol. Now we can reap the benefits of this effort: the only thing required from the compiler is mapping the high-level variables onto the virtual memory segments. All the subsequent gory details associated with managing these segments on the RAM will be handled by the VM implementation. That's why we sometimes refer to the latter as the compiler's *back end*.

To recap, in a two-tier compilation model, the handling of variables can be reduced to mapping high-level variables on virtual memory segments and using this mapping, as needed, throughout code generation. These tasks can be readily managed using a classical abstraction known as a *symbol table*.

Symbol table: Whenever the compiler encounters variables in a high-level statement, for example, let y = foo(x), it needs to know what the variables stand for. Is x a static variable, a field of an object, a local variable, or an argument of a subroutine? Does it represent an integer, a boolean, a char, or some class type? All these questions must be answered—for code generation—each time the variable x comes up in the source code. Of course, the variable y should be treated exactly the same way.

The variable properties can be managed conveniently using a *symbol table*. When a static, field, local, or argument variable is declared in the source code, the compiler allocates it to the next available entry in the corresponding static, this, local, or argument VM segment and records the mapping in the symbol table. Whenever a variable is encountered elsewhere in the code, the compiler looks up its name in the symbol table, retrieves its properties, and uses them, as needed, for code generation.

An important feature of high-level languages is *separate namespaces*: the same identifier can represent different things in different regions of the program. To enable separate namespaces, each identifier is implicitly associated with a *scope*, which is the region of the program in which it is recognized. In Jack, the scope of static and field variables is the class in which they are declared, and the scope of local and argument variables is the subroutine in which they are declared. Jack compilers can realize the scope abstractions by managing two separate symbol tables, as seen in figure 11.2.

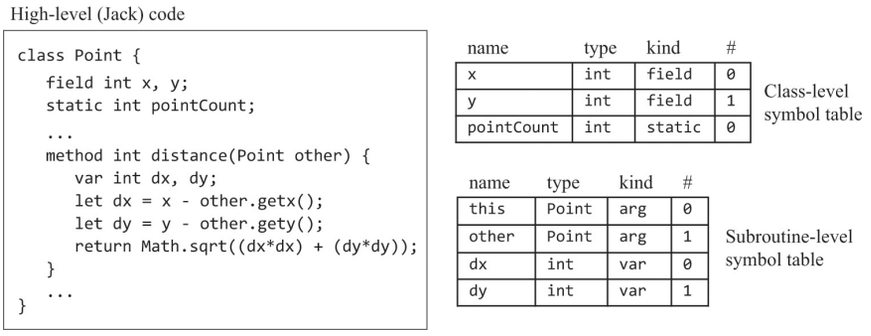

Figure 11.2 Symbol table examples. The this row in the subroutine-level table is discussed later in the chapter.

The scopes are nested, with inner scopes hiding outer ones. For example, when the Jack compiler encounters the expression `x + 17`, it first checks whether `x` is a subroutine-level variable (local or argument). Failing that, the compiler checks whether `x` is a static variable or a field. Some languages feature nested scoping of unlimited depth, allowing variables to be local in any block of code in which they are declared. To support unlimited nesting, the compiler can use a linked list of symbol tables, each reflecting a single scope nested within the next one in the list. When the compiler fails to find the variable in the table associated with the current scope, it looks it up in the next table in the list, from inner scopes outward. If the variable is not found in the list, the compiler can throw an "undeclared variable" error.

In the Jack language there are only two scoping levels: the subroutine that is presently being compiled, and the class in which the subroutine is declared. Therefore, the compiler can get away with managing two symbol tables only.

Handling variable declarations: When the Jack compiler starts compiling a class declaration, it creates a class-level symbol table and a subroutine-level symbol table. When the compiler parses a static or a field variable declaration, it adds a new row to the class-level symbol table. The row records the variable's *name*, *type* (integer, boolean, char, or class name), *kind* (static or field), and *index* within the kind.

When the Jack compiler starts compiling a subroutine (constructor, method, or function) declaration, it resets the subroutine-level symbol table. If the subroutine is a method, the compiler adds the row <this, *className*, arg, 0> to the subroutine-level symbol table (this initialization detail is explained in section 11.1.5.2 and can be ignored till then). When the compiler parses a local or an argument variable declaration, it adds a new row to the subroutine-level symbol table, recording the variable's name, type (integer, boolean, char, or class name), kind (var or arg), and index within the kind. The index of each kind (var or arg) starts at 0 and is incremented by 1 after each time a new variable of that kind is added to the table.

Handling variables in statements: When the compiler encounters a variable in a statement, it looks up the variable name in the subroutine-level symbol table. If the variable is not found, the compiler looks it up in the class-level symbol table. Once the variable has been found, the compiler can complete the statement's translation. For example, consider the

symbol tables shown in figure 11.2, and suppose we are compiling the high-level statement `let y = y + dy`. The compiler will translate this statement into the VM commands `push this 1`, `push local 1`, `add`, `pop this 1`. Here we assume that the compiler knows how to handle expressions and `let` statements, subjects which are taken up in the next two sections.

11.1.2 Compiling Expressions

Let's start by considering the compilation of simple expressions like `x + y – 7`. By "simple expression" we mean a sequence of *term operator term operator term* …, where each *term* is either a variable or a constant, and each *operator* is either +, −, *, or /.

In Jack, as in most high-level languages, expressions are written using *infix* notation: To add `x` and `y`, one writes `x + y`. In contrast, our compilation's target language is *postfix*: The same addition semantics is expressed in the stack-oriented VM code as `push x`, `push y`, `add`. In chapter 10 we introduced an algorithm that emits the parsed source code in infix using XML tags. Although the parsing logic of this algorithm can remain the same, the output part of the algorithm must now be modified for generating postfix commands. Figure 11.3 illustrates this dichotomy.

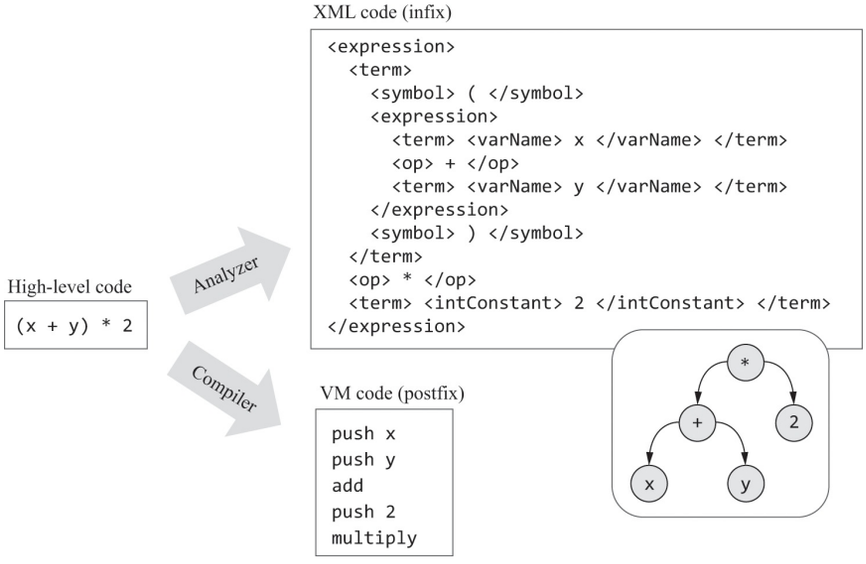

Figure 11.3 Infix and postfix renditions of the same semantics.

To recap, we need an algorithm that knows how to parse an infix expression and generate from it as output postfix code that realizes the same semantics on a stack machine. Figure 11.4 presents one such algorithm. The algorithm processes the input expression from left to right, generating VM code as it goes along. Conveniently, this algorithm also handles unary operators and function calls.

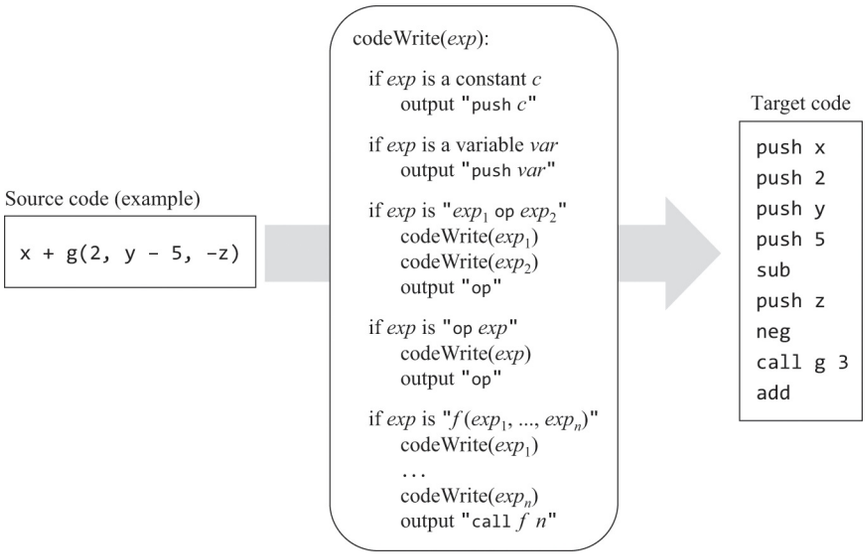

Figure 11.4 A VM code generation algorithm for expressions, and a compilation example. The algorithm assumes that the input expression is valid. The final implementation of this algorithm should replace the emitted symbolic variables with their corresponding symbol table mappings.

If we execute the stack-based VM code generated by the codeWrite algorithm (right side of figure 11.4), the execution will end up consuming all the expression's terms and putting the expression's value at the stack's top. That's exactly what's expected from the compiled code of an expression.

So far we have dealt with relatively simple expressions. Figure 11.5 gives the complete grammatical definition of Jack expressions, along with several examples of actual expressions consistent with this definition.

Definition (from the Jack grammar):

expression: *term* (*op term*) *

term: *integerConstant* | *stringConstant* | *keywordConstant* | *varName* |
varName '**[**' *expression* '**]**' | '**(**' *expression* '**)**' | (*unaryOp term*) | *subroutineCall*

subroutineCall: *subroutineName* '**(**' *expressionList* '**)**' |
(*className* | *varName*) '**.**' *subroutineName* '**(**' *expressionList* '**)**'

expressionList: (*expression* ('**,**' *expression*) *) ?

op: '**+**' | '**-**' | '*****' | '**/**' | '**&**' | '**|**' | '**<**' | '**>**' | '**=**'

unaryOp: '**-**' | '**~**'

keywordConstant: '**true**' | '**false**' | '**null**' | '**this**'

The definitions of *integerConstant*, *stringConstant*, *keywordConstant*, and other elements are given in the complete Jack grammar (figure 10.6), and are self-explanatory.

Examples:

```
5
x
x + 5
(-b + Math.sqrt(b*b - (4 * a * c))) / (2 * a)
arr[i] + foo(x)
foo(Math.abs(arr[x + foo(5)]))
```

Figure 11.5 Expressions in the Jack language.

The compilation of Jack expressions will be handled by a routine named compileExpression. The developer of this routine should start with the algorithm presented in figure 11.4 and extend it to handle the various possibilities specified in figure 11.5. We will have more to say about this implementation later in the chapter.

11.1.3 Compiling Strings

Strings—sequences of characters—are widely used in computer programs. Object-oriented languages typically handle strings as instances of a class named String (Jack's String class, which is part of the Jack OS, is documented in appendix 6). Each time a string constant comes up in a high-level statement or expression, the compiler generates code that calls the String constructor, which creates and returns a new String object. Next, the compiler initializes the new object with the string characters. This is done by generating a sequence of calls to the String method appendChar, one for each character listed in the high-level string constant.

This implementation of string constants can be wasteful, leading to potential memory leaks. To illustrate, consider the statement Output.printString("Loading … please wait"). Presumably, all the high-level programmer wants is to display a message; she certainly doesn't care if the compiler creates a new object, and she may be surprised to know that the object will persist in memory until the program terminates. But that's exactly what actually happens: a new String object will be created, and this object will keep lurking in the background, doing nothing.

Java, C#, and Python use a run-time *garbage collection* process that reclaims the memory used by objects that are no longer in play (technically, objects that have no variable referring to them). In general, modern languages use a variety of optimizations and specialized string classes for promoting the efficient use of string objects. The Jack OS features only one String class and no string-related optimizations.

Operating system services: In the handling of strings, we mentioned for the first time that the compiler can use OS services as needed. Indeed, developers of Jack compilers can assume that every constructor, method, and function listed in the OS API (appendix 6) is available *as a compiled VM function*. Technically speaking, any one of these VM functions can be called by the code generated by the compiler. This configuration will be fully realized in chapter 12, in which we will implement the OS in Jack and compile it into VM code.

11.1.4 Compiling Statements

The Jack programming language features five statements: let, do, return, if, and while. We now turn to discuss how the Jack compiler generates VM code that handles the semantics of these statements.

Compiling return statements: Now that we know how to compile expressions, the compilation of return *expression* is simple. First, we call the compileExpression routine, which

generates VM code designed to evaluate and put the expression's value on the stack. Next, we generate the VM command return.

Compiling let statements: Here we discuss the handling of statements of the form let *varName* = *expression*. Since parsing is a left-to-right process, we begin by remembering the *varName*. Next, we call compileExpression, which puts the expression's value on the stack. Finally, we generate the VM command pop *varName*, where *varName* is actually the symbol table mapping of *varName* (for example, local 3, static 1, and so on).

We'll discuss the compilation of statements of the form let *varName*[*expression*1] = *expression*2 later in the chapter, in a section dedicated to handling arrays.

Compiling do statements: Here we discuss the compilation of *function calls* of the form do *className.functionName* (*exp*1, *exp*2, … , *exp*n). The do abstraction is designed to call a subroutine for its effect, ignoring the return value. In chapter 10 we recommended compiling such statements as if their syntax were do *expression*. We repeat this recommendation here: to compile a do *className.functionName* (…) statement, we call compileExpression and then get rid of the topmost stack element (the expression's value) by generating a command like pop temp 0.

We'll discuss the compilation of *method calls* of the form do *varName.methodName* (…) and do *methodName* (…) later in the chapter, in a section dedicated to compiling method calls.

Compiling if and while statements: High-level programming languages feature a variety of *control flow statements* like if, while, for, and switch, of which Jack features if and while. In contrast, low-level assembly and VM languages control the flow of execution using two branching primitives: *conditional goto,* and *unconditional goto*. Therefore, one of the challenges faced by compiler developers is expressing the semantics of high-level control flow statements using nothing more than goto primitives. Figure 11.6 shows how this gap can be bridged systematically.

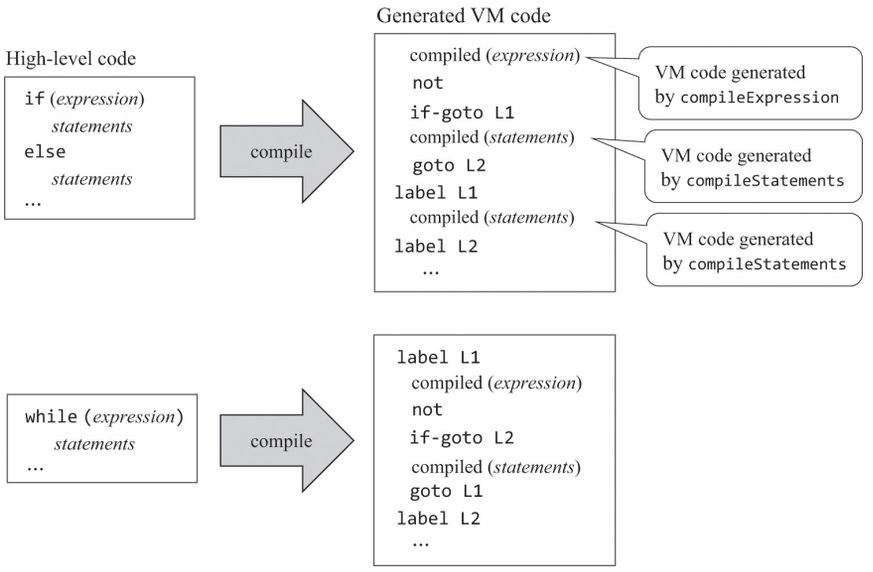

Figure 11.6 Compiling if and while statements. The L1 and L2 labels are generated by the compiler.

When the compiler detects an if keyword, it knows that it has to parse a pattern of the form if (*expression*) {*statements*} else {*statements*}. Hence, the compiler starts by calling compileExpression, which generates VM commands designed to compute and push the expression's value onto the stack. The compiler then generates the VM command not, designed to negate the expression's value. Next, the compiler creates a label, say L1, and uses it for generating the VM command if-goto L1. Next, the compiler calls compileStatements. This routine is designed to compile a sequence of the form *statement*; *statement*; … *statement*;, where each *statement* is either a let, a do, a return, an if, or a while. The resulting VM code is referred to conceptually in figure 11.6 as "compiled (*statements*)." The rest of the compilation strategy is self-explanatory.

A high-level program normally contains multiple instances of if and while. To handle this complexity, the compiler can generate labels that are globally unique, for example, labels whose suffix is the value of a running counter. Also, control flow statements are often nested —for example, an if within a while within another while, and so on. Such nestings are taken care of implicitly since the compileStatements routine is inherently recursive.

11.1.5 Handling Objects

So far in this chapter, we described techniques for compiling *variables*, *expressions*, *strings*, and *statements*. This forms most of the know-how necessary for building a compiler for a procedural, C-like language. In Nand to Tetris, though, we aim higher: building a compiler for an object-based, Java-like language. With that in mind, we now turn to discuss the handling of *objects*.

Object-oriented languages feature means for declaring and manipulating aggregate abstractions known as *objects*. Each object is implemented physically as a memory block which can be referenced by a static, field, local, or argument variable. The reference variable, also known as an *object variable*, or *pointer*, contains the memory block's base address. The

operating system realizes this model by managing a logical area in the RAM named *heap*. The *heap* is used as a memory pool from which memory blocks are carved, as needed, for representing new objects. When an object is no longer needed, its memory block can be freed and recycled back to the heap. The compiler stages these memory management actions by calling OS functions, as we'll see later.

At any given point during a program's execution, the heap can contain numerous objects. Suppose we want to apply a method, say foo, to one of these objects, say p. In an object-oriented language, this is done through the method call idiom p.foo(). Shifting our attention from the caller to the callee, we note that foo—like any other method—is designed to operate on a placeholder known as the *current object*, or this. In particular, when VM commands in foo's code make references to this 0, this 1, this 2, and so on, they should effect the fields of p, the object on which foo was called. Which begs the question: How do we align the this segment with p?

The virtual machine built in chapters 7–8 has a mechanism for realizing this alignment: the two-valued pointer segment, mapped directly onto RAM locations 3–4, also known as THIS and THAT. According to the VM specification, the pointer THIS (referred to as pointer 0) is designed to hold the base address of the memory segment this. Thus, to align the this segment with the object p, we can push p's value (which is an address) onto the stack and then pop it into pointer 0. Versions of this initialization technique are used conspicuously in the compilation of *constructors* and *methods*, as we now turn to describe.

11.1.5.1 Compiling Constructors

In object-oriented languages, objects are created by subroutines known as *constructors*. In this section we describe how to compile a constructor call (e.g., Java's new operator) from the *caller*'s perspective and how to compile the code of the constructor itself—the *callee*.

Compiling constructor calls: Object construction is normally a two-stage affair. First, one declares a variable of some class type, for example, var Point p. At a later stage, one can instantiate the object by calling a class constructor, for example, let p = Point.new(2,3). Or, depending on the language used, one can declare *and* construct objects using a single high-level statement. Behind the scenes, though, this action is always broken into two separate stages: declaration followed by construction.

Let's take a close look at the statement let p = Point.new(2,3). This abstraction can be described as "have the Point.new constructor allocate a two-word memory block for representing a new Point instance, initialize the two words of this block to 2 and 3, and have p refer to the base address of this block." Implicit in this semantics are two assumptions: First, the constructor knows how to allocate a memory block of the required size. Second, when the constructor—being a subroutine—terminates its execution, it returns to the caller the base address of the allocated memory block. Figure 11.7 shows how this abstraction can be realized.

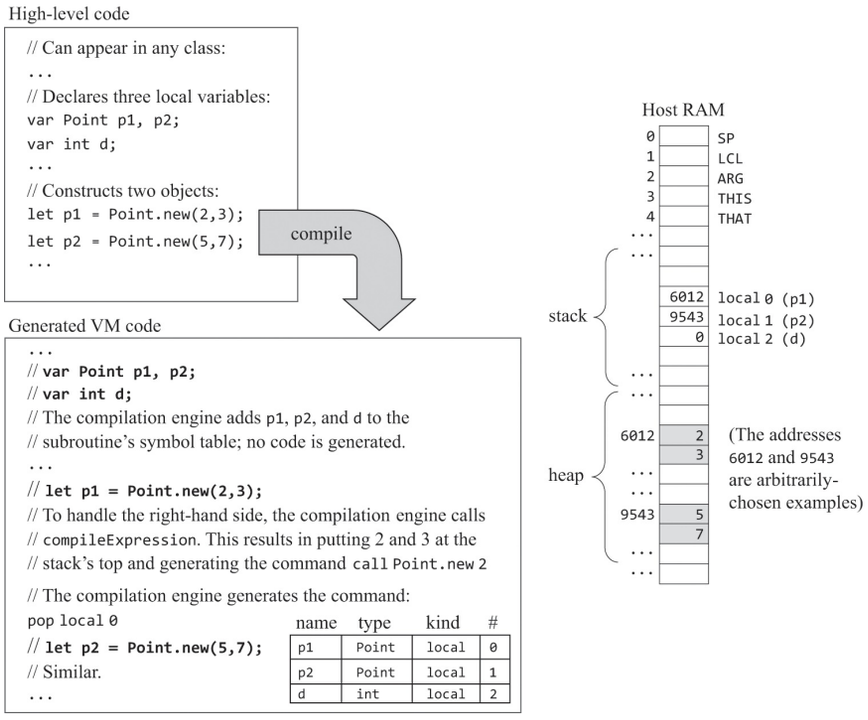

Figure 11.7 Object construction from the caller's perspective. In this example, the caller declares two object variables and then calls a class constructor for constructing the two objects. The constructor works its magic, allocating memory blocks for representing the two objects. The calling code then makes the two object variables refer to these memory blocks.

Three observations about figure 11.7 are in order. First, note that there is nothing special about compiling statements like `let p = Point.new(2,3)` and `let p = Point.new(5,7)`. We already discussed how to compile `let` statements and subroutine calls. The only thing that makes these calls special is the hocus-pocus assumption that—somehow—two objects will be constructed. The implementation of this magic is entirely delegated to the compilation of the *callee*—the constructor. As a result of this magic, the constructor creates the two objects seen in the RAM diagram in figure 11.7. This leads to the second observation: The physical addresses 6012 and 9543 are irrelevant; the high-level code as well as the compiled VM code have no idea where the objects are stored in memory; the references to these objects are strictly symbolic, via `p1` and `p2` in the high-level code and `local 0` and `local 1` in the compiled code. (As a side comment, this makes the program relocatable and safer.) Third, and stating the obvious, nothing of substance actually happens until the generated VM code is executed. In particular, during *compile-time*, the symbol table is updated, low-level code is generated, and that's it. The objects will be constructed and bound to the variables only during *run-time*, that is, if and when the compiled code will be executed.

Compiling constructors: So far, we have treated constructors as black box abstractions: we assume that they create objects, *somehow*. Figure 11.8 unravels this magic. Before inspecting the figure, note that a constructor is, first and foremost, a *subroutine*. It can have arguments, local variables, and a body of statements; thus the compiler treats it as such. What makes the compilation of a constructor special is that in addition to treating it as a regular subroutine, the compiler must also generate code that (i) creates a new object and (ii) makes the new

object the *current object* (also known as this), that is, the object on which the constructor's code is to operate.

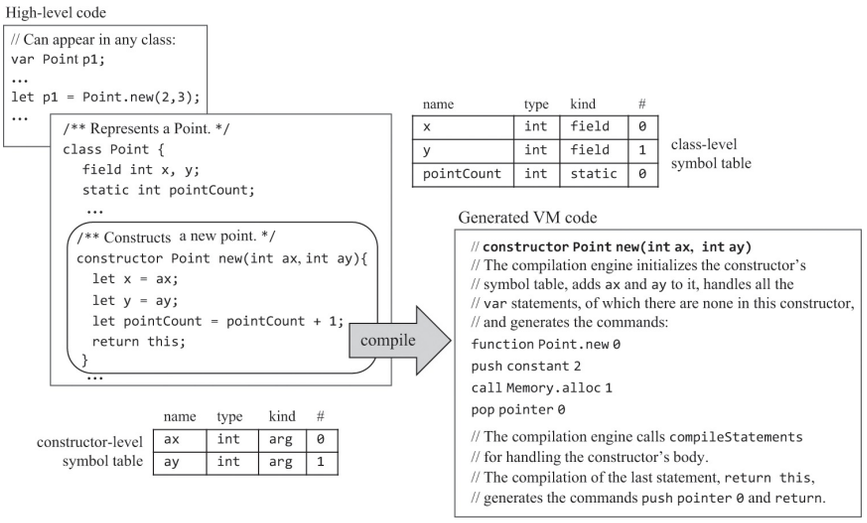

Figure 11.8 Object construction: the constructor's perspective.

The creation of a new object requires finding a free RAM block sufficiently large to accommodate the new object's data and marking the block as used. These tasks are delegated to the host operating system. According to the OS API listed in appendix 6, the OS function Memory.alloc (*size*) knows how to find an available RAM block of a given *size* (number of 16-bit words) and return the block's base address.

Memory.alloc and its sister function Memory.deAlloc use clever algorithms for allocating and freeing RAM resources efficiently. These algorithms will be presented and implemented in chapter 12, when we'll build the operating system. For now, suffice it to say that compilers generate low-level code that uses alloc (in constructors) and deAlloc (in destructors), abstractly.

Before calling Memory.alloc, the compiler determines the size of the required memory block. This can be readily computed from the class-level symbol table. For example, the symbol table of the Point class specifies that each Point object is characterized by two int values (the point's *x* and *y* coordinates). Thus, the compiler generates the commands push constant 2 and call Memory.alloc 1, effecting the function call Memory.alloc(2). The OS function alloc goes to work, finds an available RAM block of size 2, and pushes its base address onto the stack—the VM equivalent of returning a value. The next generated VM statement—pop pointer 0—sets THIS to the base address returned by alloc. From this point onward, the constructor's this segment will be aligned with the RAM block that was allocated for representing the newly constructed object.

Once the this segment is properly aligned, we can proceed to generate code easily. For example, when the compileLet routine is called to handle the statement let x = ax, it searches the symbol tables, resolving x to this 0 and ax to argument 0. Thus, compileLet generates the commands push argument 0, followed by pop this 0. The latter command rests on the assumption that the this segment is properly aligned with the base address of the new object, as indeed

was done when we had set pointer 0 (actually, THIS) to the base address returned by alloc. This one-time initialization ensures that all the subsequent push / pop this *i* commands will end up hitting the right targets in the RAM (more accurately, in the *heap*). We hope that the intricate beauty of this contract is not lost on the reader.

According to the Jack language specification, every constructor must end with a return this statement. This convention forces the compiler to end the constructor's compiled version with push pointer 0 and return. These commands push onto the stack the value of THIS, the base address of the constructed object. In some languages, like Java, constructors don't have to end with an explicit return this statement. Nonetheless, the compiled code of Java constructors performs exactly the same action at the VM level, since that's what constructors are expected to do: create an object and return its handle to the caller.

Recall that the elaborate low-level drama just described was unleashed by the caller-side statement let *varName* = *className.constructorName* (…). We now see that, by design, when the constructor terminates, *varName* ends up storing the base address of the new object. When we say "by design," we mean by the syntax of the high-level object construction idiom and by the hard work that the compiler, the operating system, the VM translator, and the assembler have to do in order to realize this abstraction. The net result is that high-level programmers are spared from all the gory details of object construction and are able to create objects easily and transparently.

11.1.5.2 Compiling Methods

As we did with constructors, we'll describe how to compile method calls and then how to compile the methods themselves.

Compiling method calls: Suppose we wish to compute the Euclidean distance between two points in a plane, p1 and p2. In C-style procedural programming, this could have been implemented using a function call like distance(p1,p2), where p1 and p2 represent composite data types. In object-oriented programming, though, p1 and p2 will be implemented as instances of some Point class, and the same computation will be done using a method call like p1.distance(p2). Unlike functions, *methods* are subroutines that always operate on a given object, and it's the caller's responsibility to specify this object. (The fact that the distance method takes another Point object as an argument is a coincidence. In general, while a method is always designed to operate on an object, the method can have 0, 1, or more arguments, of any type).

Observe that distance can be described as a *procedure* for computing the distance from a given point to another, and p1 can be described as the *data* on which the procedure operates. Also, note that both idioms distance(p1,p2) and p1.distance(p2) are designed to compute and return the same value. Yet while the C-style syntax puts the focus on distance, in the object-oriented syntax, the object comes first, literally speaking. That's why C-like languages are sometimes called *procedural*, and object-oriented languages are said to be *data-driven*. Among other things, the object-oriented programming style is based on the assumption that

objects know how to take care of themselves. For example, a Point object knows how to compute the distance between it and another Point object. Said otherwise, the distance operation is *encapsulated* within the definition of being a Point.

The agent responsible for bringing all these fancy abstractions down to earth is, as usual, the hard-working compiler. Because the target VM language has no concept of objects or methods, the compiler handles object-oriented method calls like p1.distance (p2) as if they were procedural calls like distance (p1,p2). Specifically, it translates p1.distance (p2) into push p1, push p2, call distance. Let us generalize: Jack features two kinds of method calls:

// Applies a method to the object referred to by *varName*:
varName.*methodName*(*exp*$_1$, *exp*$_2$, …, *exp*$_n$)

// Applies a method to the *current object*:
methodName(*exp*$_1$, *exp*$_2$, …, *exp*$_n$) // Same as `this.` *methodName*(*exp*$_1$, *exp*$_2$, …, *exp*$_n$)

To compile the method call *varName.methodName*(*exp*$_1$, *exp*$_2$, …, *exp*$_n$), we start by generating the command push *varName*, where *varName* is the symbol table mapping of *varName*. If the method call mentions no *varName*, we push the symbol table mapping of this. Next, we call compileExpressionList. This routine calls compileExpression n times, once for each expression in the parentheses. Finally, we generate the command call *className.methodName* $n+1$, informing that $n+1$ arguments were pushed onto the stack. The special case of calling an argument-less method is translated into call *className.methodName* 1. Note that *className* is the symbol table *type* of the *varName* identifier. See figure 11.9 for an example.

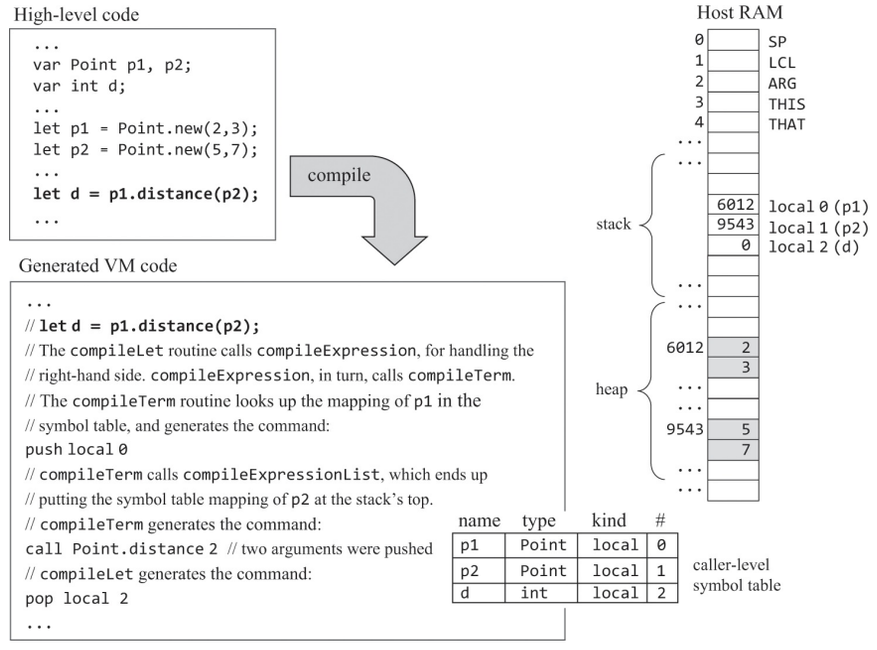

Figure 11.9 Compiling method calls: the caller's perspective.

Compiling methods: So far we discussed the distance method abstractly, from the caller's perspective. Consider how this method could be implemented, say, in Java:

```
/** A Point class method: returns the distance between this Point and the other one. */
int distance(Point other) {
   int dx, dy;
   dx = x - other.x;
   dy = y - other.x;
   return Math.sqrt((dx*dx) + (dy*dy));
}
```

Like any method, distance is designed to operate on the *current object*, represented in Java (and in Jack) by the built-in identifier this. As the above example illustrates, though, one can write an entire method without ever mentioning this. That's because the friendly Java compiler handles statements like `dx=x-other.x` as if they were `dx=this.x-other.x`. This convention makes high-level code more readable and easier to write.

We note in passing, though, that in the Jack language, the idiom *object.field* is not supported. Therefore, fields of objects other than the current object can be manipulated only using accessor and mutator methods. For example, expressions like x – other.x are implemented in Jack as x – other.getx(), where getx is an accessor method in the Point class.

So how does the Jack compiler handle expressions like x – other.getx()? Like the Java compiler, it looks up x in the symbol tables and finds that it represents the first field in the current object. But *which* object in the pool of so many objects out there does the *current object* represent? Well, according to the method call contract, it must be the first argument that was passed by the method's caller. Therefore, from the callee's perspective, the current object must be the object whose base address is the value of argument 0. This, in a nutshell, is the low-level compilation trick that makes the ubiquitous abstraction "apply a method to an object" possible in languages like Java, Python, and, of course, Jack. See figure 11.10 for the details.

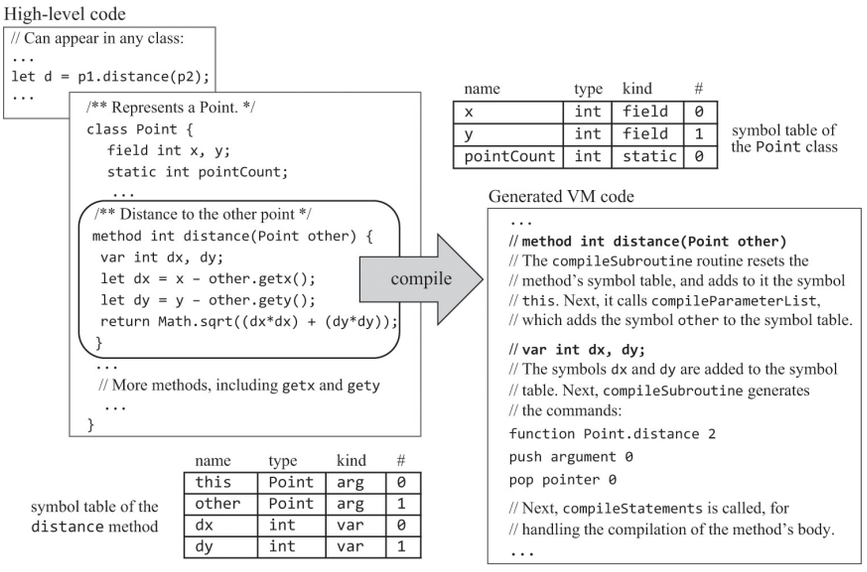

Figure 11.10 Compiling methods: the callee's perspective.

The example starts at the top left of figure 11.10, where the caller's code makes the method call p1.distance(p2). Turning our attention to the compiled version of the callee, note that the code proper starts with push argument 0, followed by pop pointer 0. These commands set the method's THIS pointer to the value of argument 0, which, by virtue of the method calling contract, contains the base address of the object on which the method was called to operate. Thus, from this point onward, the method's this segment is properly aligned with the base address of the target object, making every push / pop this *i* command properly aligned as well. For example, the expression x—other.getx () will be compiled into push this 0, push argument 1, call Point.getx 1, sub. Since we started the compiled method code by setting THIS to the base address of the called object, we are guaranteed that this 0 (and any other reference this *i*) will hit the mark, targeting the right field of the right object.

11.1.6 Compiling Arrays

Arrays are similar to objects. In Jack, arrays are implemented as instances of an Array class, which is part of the operating system. Thus, arrays and objects are declared, implemented, and stored exactly the same way; in fact, arrays *are* objects, with the difference that the array abstraction allows accessing array elements using an index, for example, let arr[3] = 17. The agent that makes this useful abstraction concrete is the compiler, as we now turn to describe.

Using pointer notation, observe that arr[i] can be written as *(arr + i), that is, memory address arr + i. This insight holds the key for compiling statements like let x = arr[i]. To compute the physical address of arr[i], we execute push arr, push i, add, which results in pushing the target address onto the stack. Next, we execute pop pointer 1. According to the VM specification, this action stores the target address in the method's THAT pointer (RAM[4]), which has the effect of aligning the base address of the virtual segment that with the target address. Thus we can now execute push that 0 and pop x, completing the low-level translation of let x = arr[i]. See figure 11.11 for the details.

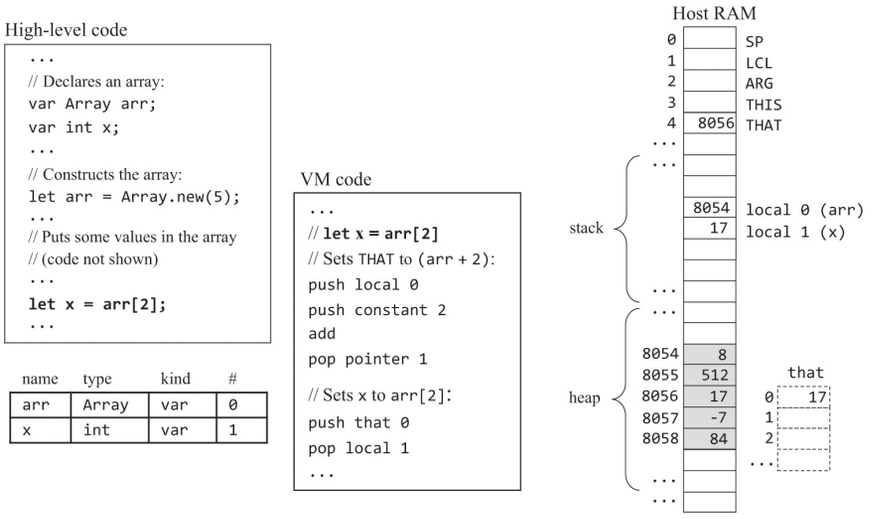

Figure 11.11 Array access using VM commands.

This nice compilation strategy has only one problem: it doesn't work. More accurately, it works with statements like let `a=b[j]` but fails with statements in which the left-hand side of the assignment is indexed, as in let `a[i]=b[j]`. See figure 11.12.

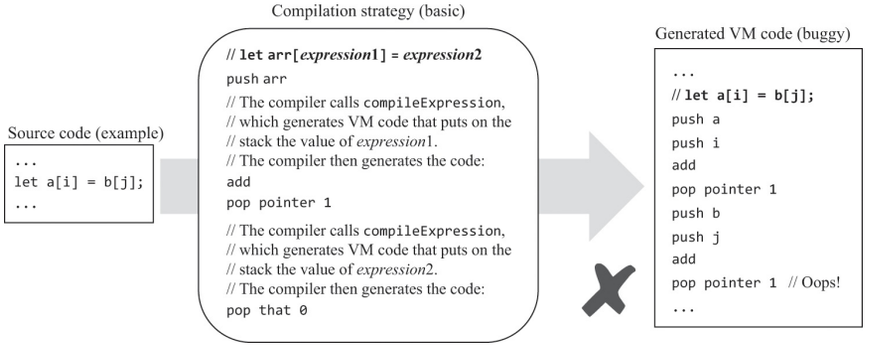

Figure 11.12 Basic compilation strategy for arrays, and an example of the bugs that it can generate. In this particular case, the value stored in pointer 1 is overridden, and the address of a[i] is lost.

The good news is that this flawed compilation strategy can be easily fixed to compile correctly any instance of let arr[*expression*1] = *expression*2. As before, we start by generating the command push arr, calling compileExpression, and generating the command add. This sequence puts the target address (arr + *expression*1) at the stack's top. Next, we call compileExpression, which will end up putting at the stack's top the value of *expression*2. At this point we save this value—we can do it using pop temp 0. This operation has the nice side effect of making (arr + *expression*1) the top stack element. Thus we can now pop pointer 1, push temp 0, and pop that 0. This little fix, along with the recursive nature of the compileExpression routine, makes this compilation strategy capable of handling let arr[*expression*1] = *expression*2 statements of any recursive complexity, such as, say, let `a[b[i]+a[j+b[a[3]]]]=b[b[j]+2]`.

In closing, several things make the compilation of Jack arrays relatively simple. First, Jack arrays are not typed; rather, they are designed to store 16-bit values, with no restrictions.

Second, all primitive data types in Jack are 16-bit wide, all addresses are 16-bit wide, and so is the RAM's word width. In strongly typed programming languages, and in languages where this one-to-one correspondence cannot be guaranteed, the compilation of arrays requires more work.

11.2 Specification

The compilation challenges and solutions that we have described so far can be generalized to support the compilation of any object-based programming language. We now turn from the general to the specific: from here to the end of the chapter, we describe the *Jack compiler*. The Jack compiler is a program that gets a Jack program as input and generates executable VM code as output. The VM code realizes the program's semantics on the virtual machine specified in chapters 7–8.

Usage: The compiler accepts a single command-line argument, as follows,

```
prompt> JackCompiler source
```

where *source* is either a file name of the form *Xxx*.jack (the extension is mandatory) or the name of a folder (in which case there is no extension) containing one or more .jack files. The file/folder name may contain a file path. If no path is specified, the compiler operates on the current folder. For each *Xxx*.jack file, the compiler creates an output file *Xxx*.vm and writes the VM commands into it. The output file is created in the same folder as the input file. If there is a file by this name in the folder, it will be overwritten.

11.3 Implementation

We now turn to provide guidelines, implementation tips, and a proposed API for extending the syntax analyzer built in chapter 10 into a full-scale Jack compiler.

11.3.1 Standard Mapping over the Virtual Machine

Jack compilers can be developed for different target platforms. This section provides guidelines on how to map various constructs of the Jack language on one specific platform: the virtual machine specified in chapters 7–8.

Naming Files and Functions

- A Jack class file *Xxx*.jack is compiled into a VM class file named *Xxx*.vm
- A Jack subroutine *yyy* in file *Xxx*.jack is compiled into a VM function named *Xxx.yyy*

Mapping Variables

- The first, second, third, … *static* variable declared in a class declaration is mapped on the virtual segment entry static 0, static 1, static 2, …
- The first, second, third, … *field* variable declared in a class declaration is mapped on this 0, this 1, this 2, …
- The first, second, third, … *local* variable declared in the var statements of a subroutine is mapped on local 0, local 1, local 2, …
- The first, second, third, … *argument* variable declared in the parameter list of a *function* or a *constructor* (but not a *method*) is mapped on argument 0, argument 1, argument 2, …
- The first, second, third, … *argument* variable declared in the parameter list of a *method* is mapped on argument 1, argument 2, argument 3, …

Mapping Object Fields

To align the virtual segment this with the object passed by the caller of a *method*, use the VM commands push argument 0, pop pointer 0.

Mapping Array Elements

The high-level reference arr[*expression*] is compiled by setting pointer 1 to (arr + *expression*) and accessing that 0.

Mapping Constants

- References to the Jack constants null and false are compiled into push constant 0.
- References to the Jack constant true are compiled into push constant 1, neg. This sequence pushes the value −1 onto the stack.
- References to the Jack constant this are compiled into push pointer 0. This command pushes the base address of the current object onto the stack.

11.3.2 Implementation Guidelines

Throughout this chapter we have seen many conceptual compilation examples. We now give a concise and formal summary of all these compilation techniques.

Handling Identifiers

The identifiers used for naming variables can be handled using symbol tables. During the compilation of valid Jack code, any identifier not found in the symbol tables may be assumed to be either a subroutine name or a class name. Since the Jack syntax rules suffice for distinguishing between these two possibilities, and since the Jack compiler performs no

"linking," there is no need to keep these identifiers in a symbol table.

Compiling Expressions

The compileExpression routine should process the input as the sequence *term op term op term* …. To do so, compileExpression should implement the codeWrite algorithm (figure 11.4), extended to handle all the possible *terms* specified in the Jack grammar (figure 11.5). Indeed, an inspection of the grammar rules reveals that most of the action in compiling *expressions* occurs in the compilation of their underlying *terms*. This is especially true following our recommendation that the compilation of subroutine calls be handled directly by the compilation of *terms* (implementation notes following the CompilationEngine API, section 10.3).

The *expression* grammar and thus the corresponding compileExpression routine are inherently recursive. For example, when compileExpression detects a left parenthesis, it should recursively call compileExpression to handle the inner expression. This recursive descent ensures that the inner expression will be evaluated first. Except for this priority rule, the Jack language supports *no operator priority*. Handling operator priority is of course possible, but in Nand to Tetris we consider it an optional compiler-specific extension, not a standard feature of the Jack language.

The expression `x * y` is compiled into `push x`, `push y`, `call Math.multiply 2`. The expression `x / y` is compiled into `push x`, `push y`, `call Math.divide 2`. The Math class is part of the OS, documented in appendix 6. This class will be developed in chapter 12.

Compiling Strings

Each string constant "*ccc … c*" is handled by (i) pushing the string length onto the stack and calling the String.new constructor, and (ii) pushing the character code of *c* on the stack and calling the String method appendChar, once for each character *c* in the string (the Jack character set is documented in appendix 5). As documented in the String class API in appendix 6, both the new constructor and the appendChar method return the string as the return value (i.e., they push the string object onto the stack). This simplifies compilation, avoiding the need to re-push the string each time appendChar is called.

Compiling Function Calls and Constructor Calls

The compiled version of calling a function or calling a constructor that has *n* arguments must (i) call compileExpressionList, which will call compileExpression *n* times, and (ii) make the call informing that *n* arguments were pushed onto the stack before the call.

Compiling Method Calls

The compiled version of calling a method that has *n* arguments must (i) push a reference to the object on which the method is called to operate, (ii) call compileExpressionList, which will call compileExpression *n* times, and (iii) make the call, informing that $n+1$ arguments were

pushed onto the stack before the call.

Compiling `do` Statements

We recommend compiling `do` *subroutineCall* statements as if they were `do` *expression* statements, and then yanking the topmost stack value using `pop temp 0`.

Compiling Classes

When starting to compile a class, the compiler creates a class-level symbol table and adds to it all the *field* and *static* variables declared in the class declaration. The compiler also creates an empty subroutine-level symbol table. No code is generated.

Compiling Subroutines

- When starting to compile a subroutine (*constructor*, *function*, or *method*), the compiler initializes the subroutine's symbol table. If the subroutine is a *method*, the compiler adds to the symbol table the mapping <`this`, *className*, `arg`, `0`>.
- Next, the compiler adds to the symbol table all the parameters, if any, declared in the subroutine's parameter list. Next, the compiler handles all the `var` declarations, if any, by adding to the symbol table all the subroutine's local variables.
- At this stage the compiler starts generating code, beginning with the command `function` *className.subroutineName nVars*, where *nVars* is the number of local variables in the subroutine.
- If the subroutine is a *method*, the compiler generates the code `push argument 0`, `pop pointer 0`. This sequence aligns the virtual memory segment `this` with the base address of the object on which the method was called.

Compiling Constructors

- First, the compiler performs all the actions described in the previous section, ending with the generation of the command `function` *className.constructorName nVars*.
- Next, the compiler generates the code `push constant` *nFields*, `call Memory.alloc 1`, `pop pointer 0`, where *nFields* is the number of fields in the compiled class. This results in allocating a memory block of *nFields* 16-bit words and aligning the virtual memory segment `this` with the base address of the newly allocated block.
- The compiled constructor must end with `push pointer 0`, `return`. This sequence returns to the caller the base address of the new object created by the constructor.

Compiling Void Methods and Void Functions

Every VM function is expected to push a value onto the stack before returning. When

compiling a void Jack method or function, the convention is to end the generated code with `push constant 0`, `return`.

Compiling Arrays

Statements of the form `let arr[`*expression*`1] =` *expression*`2` are compiled using the technique described at the end of section 11.1.6. *Implementation tip*: When handling arrays, there is never a need to use `that` entries whose index is greater than `0`.

The Operating System

Consider the high-level expression `Math.sqrt((dx*dx)+(dy*dy))`. The compiler compiles it into the VM commands `push` *dx*, `push` *dx*, `call Math.multiply 2`, `push` *dy*, `push` *dy*, `call Math.multiply 2`, `add`, `call Math.sqrt 1`, where *dx* and *dy* are the symbol table mappings of `dx` and `dy`. This example illustrates the two ways in which operating system services come into play during compilation. First, some high-level abstractions, like the expression `x * y`, are compiled by generating code that calls OS subroutines like `Math.multiply`. Second, when a Jack expression includes a high-level call to an OS routine, for example, `Math.sqrt(x)`, the compiler generates VM code that makes exactly the same call using VM postfix syntax.

The OS features eight classes, documented in appendix 6. Nand to Tetris provides two different implementations of this OS—*native* and *emulated*.

Native OS Implementation

In project 12 you will develop the OS class library in Jack and compile it using a Jack compiler. The compilation will yield eight `.vm` files, comprising the native OS implementation. If you put these eight `.vm` files in the same folder that stores the `.vm` files resulting from the compilation of *any* Jack program, all the OS functions will become accessible to the compiled VM code since they belong to the same code base.

Emulated OS Implementation

The supplied VM emulator, which is a Java program, features a Java-based implementation of the Jack OS. Whenever the VM code loaded into the emulator calls an OS function, the emulator checks whether a VM function by that name exists in the loaded code base. If so, it executes the VM function. Otherwise, it calls the built-in implementation of this OS function. The bottom line is this: If you use the supplied VM emulator for executing the VM code generated by your compiler, as we do in project 11, you need not worry about the OS configuration; the emulator will service all the OS calls without further ado.

11.3.3 Software Architecture

The proposed compiler architecture builds upon the syntax analyzer described in chapter 10.

Specifically, we propose to gradually evolve the syntax analyzer into a full-scale compiler, using the following modules:

- JackCompiler: main program, sets up and invokes the other modules
- JackTokenizer: tokenizer for the Jack language
- SymbolTable: keeps track of all the variables found in the Jack code
- VMWriter: writes VM code
- CompilationEngine: recursive top-down compilation engine

The JackCompiler

This module drives the compilation process. It operates on either a file name of the form *Xxx*.jack or on a folder name containing one or more such files. For each source *Xxx*.jack file, the program

1. creates a JackTokenizer from the *Xxx*.jack input file;
2. creates an output file named *Xxx*.vm; and
3. uses a CompilationEngine, a SymbolTable, and a VMWriter for parsing the input file and emitting the translated VM code into the output file.

We provide no API for this module, inviting you to implement it as you see fit. Remember that the first routine that must be called when compiling a .jack file is compileClass.

The JackTokenizer

This module is identical to the tokenizer built in project 10. See the API in section 10.3.

The SymbolTable

This module provides services for building, populating, and using symbol tables that keep track of the symbol properties *name*, *type*, *kind*, and a running *index* for each kind. See figure 11.2 for an example.

Routine	*Arguments*	*Returns*	*Function*
Constructor / initializer	—	—	Creates a new symbol table.
`reset`	—	—	Empties the symbol table, and resets the four indexes to 0. Should be called when starting to compile a subroutine declaration.
`define`	`name` (string) `type` (string) `kind` (`STATIC`, `FIELD`, `ARG`, or `VAR`)	—	Defines (adds to the table) a new variable of the given `name`, `type`, and `kind`. Assigns to it the index value of that `kind`, and adds 1 to the index.
`varCount`	`kind` (`STATIC`, `FIELD`, `ARG`, or `VAR`)	int	Returns the number of variables of the given `kind` already defined in the table.
`kindOf`	`name` (string)	(`STATIC`, `FIELD`, `ARG`, `VAR`, `NONE`)	Returns the `kind` of the named identifier. If the identifier is not found, returns `NONE`.
`typeOf`	`name` (string)	string	Returns the `type` of the named variable.
`indexOf`	`name` (string)	int	Returns the `index` of the named variable.

Implementation note: During the compilation of a Jack class file, the Jack compiler uses two instances of SymbolTable.

The VMWriter

This module features a set of simple routines for writing VM commands into the output file.

Routine	*Arguments*	*Returns*	*Function*
Constructor / initializer	Output file / stream	—	Creates a new output `.vm` file / stream, and prepares it for writing.
`writePush`	`segment` (`CONSTANT`, `ARGUMENT`, `LOCAL`, `STATIC`, `THIS`, `THAT`, `POINTER`, `TEMP`) `index` (int)	—	Writes a VM `push` command.
`writePop`	`segment` (`ARGUMENT`, `LOCAL`, `STATIC`, `THIS`, `THAT`, `POINTER`, `TEMP`) `index` (int)	—	Writes a VM `pop` command.
`writeArithmetic`	`command` (`ADD`, `SUB`, `NEG`, `EQ`, `GT`, `LT`, `AND`, `OR`, `NOT`)	—	Writes a VM arithmetic-logical command.
`writeLabel`	`label` (string)	—	Writes a VM `label` command.
`writeGoto`	`label` (string)	—	Writes a VM `goto` command.
`writeIf`	`label` (string)		Writes a VM `if-goto` command.
`writeCall`	`name` (string) `nArgs` (int)	—	Writes a VM `call` command.
`writeFunction`	`name` (string) `nVars` (int)	—	Writes a VM `function` command.
`writeReturn`	—	—	Writes a VM `return` command.
`close`	—	—	Closes the output file / stream.

The CompilationEngine

This module runs the compilation process. Although the CompilationEngine API is almost identical the API presented in chapter 10, we repeat it here for ease of reference.

The CompilationEngine gets its input from a JackTokenizer and uses a VMWriter for writing the VM code output (instead of the XML produced in project 10). The output is generated by a series of compile*xxx* routines, each designed to handle the compilation of a specific Jack language construct *xxx* (for example, compileWhile generates the VM code that realizes while statements). The contract between these routines is as follows: Each compile*xxx* routine gets from the input and handles all the tokens that make up *xxx*, advances the tokenizer exactly beyond these tokens, and emits to the output VM code effecting the semantics of *xxx*. If *xxx* is a part of an expression, and thus has a value, the emitted VM code should compute this

value and leave it at the top of the stack. As a rule, each compile*xxx* routine is called only if the current token is *xxx*. Since the first token in a valid .jack file must be the keyword class, the compilation process starts by calling the routine compileClass.

Routine	*Arguments*	*Returns*	*Function*
Constructor / initializer	Input file / stream Output file / stream	–	Creates a new compilation engine with the given input and output. The next routine called must be `compileClass`.
`compileClass`	–	–	Compiles a complete class.
`compileClassVarDec`	–	–	Compiles a static variable declaration, or a field declaration.
`compileSubroutine`	–	–	Compiles a complete method, function, or constructor.
`compileParameterList`	–	–	Compiles a (possibly empty) parameter list. Does not handle the enclosing parenthesis tokens `(`and`)`.
`compileSubroutineBody`	–	–	Compiles a subroutine's body.
`compileVarDec`	–	–	Compiles a `var` declaration.
`compileStatements`	–	–	Compiles a sequence of statements. Does not handle the enclosing curly bracket tokens `{`and `}`.
`compileLet`	–	–	Compiles a `let` statement.
`compileIf`	–	–	Compiles an `if` statement, possibly with a trailing `else` clause.
`compileWhile`	–	–	Compiles a `while` statement.
`compileDo`	–	–	Compiles a `do` statement.
`compileReturn`	–	–	Compiles a `return` statement.
`compileExpression`	–	–	Compiles an expression.
`compileTerm`	–	–	Compiles a *term.* If the current token is an *identifier*, the routine must resolve it into a *variable*, an *array element*, or a *subroutine call.* A single lookahead token, which may be `[`, `(`, or `.`, suffices to distinguish between the possibilities. Any other token is not part of this term and should not be advanced over.
`compileExpressionList`	–	int	Compiles a (possibly empty) comma-separated list of expressions. Returns the number of expressions in the list.

Note: The following Jack grammar rules have no corresponding compile xxx routines in the CompilationEngine: *type*, *className*, *subroutineName*, *varName*, *statement*, *subroutineCall*.

The parsing logic of these rules should be handled by the routines that implement the rules that refer to them. The Jack language grammar is presented in section 10.2.1.

Token lookahead: The need for token lookahead, and the proposed solution for handling it, are discussed in section 10.3, just after the CompilationEngine API.

11.4 Project

Objective: Extend the syntax analyzer built in chapter 10 into a full-scale Jack compiler. Apply your compiler to all the test programs described below. Execute each translated program, and make sure that it operates according to its given documentation.

This version of the compiler assumes that the source Jack code is error-free. Error checking, reporting, and handling can be added to later versions of the compiler but are not part of project 11.

Resources: The main tool that you need is the programming language in which you will implement the compiler. You will also need the supplied VM emulator for testing the VM code generated by your compiler. Since the compiler is implemented by extending the syntax analyzer built in project 10, you will also need the analyzer's source code.

Implementation Stages

We propose morphing the syntax analyzer built in project 10 into the final compiler. In particular, we propose to gradually replace the routines that generate passive XML output with routines that generate executable VM code. This can be done in two main development stages.

(Stage 0: Make a backup copy of the syntax analyzer code developed in project 10.)

Stage 1: Symbol table: Start by building the compiler's SymbolTable module, and use it for extending the syntax analyzer built in Project 10, as follows. Presently, whenever an identifier is encountered in the source code, say foo, the syntax analyzer outputs the XML line <identifier> foo </identifier>. Instead, extend your syntax analyzer to output the following information about each identifier:

- *name*
- *category* (field, static, var, arg, class, subroutine)

- *index*: if the identifier's category is field, static, var, or arg, the running index assigned to the identifier by the symbol table
- *usage*: whether the identifier is presently being *declared* (for example, the identifier appears in a static / field / var Jack variable declaration) or *used* (for example, the identifier appears in a Jack expression)

Have your syntax analyzer output this information as part of its XML output, using markup tags of your choice.

Test your new SymbolTable module and the new functionality just described by running your extended syntax analyzer on the test Jack programs supplied in project 10. If your extended syntax analyzer outputs the information described above correctly it means that you've developed a complete executable capability to understand the semantics of Jack programs. At this stage you can make the switch to developing the full-scale compiler and start generating VM code instead of XML output. This can be done gradually, as we now turn to describe.

(Stage 1.5: Make a backup copy of the extended syntax analyzer code).

Stage 2: Code generation: We provide six application programs, designed to gradually unit-test the code generation capabilities of your Jack compiler. We advise developing, and testing, your evolving compiler on the test programs in the given order. This way, you will be implicitly guided to build the compiler's code generation capabilities in sensible stages, according to the demands presented by each test program.

Normally, when one compiles a high-level program and runs into difficulties, one concludes that the program is screwed up. In this project the setting is exactly the opposite. All the supplied test programs are error-free. Therefore, if their compilation yields any errors, it's the compiler that you have to fix, not the programs. Specifically, for each test program, we recommend going through the following routine:

1. Compile the program folder using the compiler that you are developing. This action should generate one .vm file for each source .jack file in the given folder.
2. Inspect the generated VM files. If there are visible problems, fix your compiler and go to step 1. Remember: All the supplied test programs are error-free.
3. Load the program folder into the VM emulator, and run the loaded code. Note that each one of the six supplied test programs contains specific execution guidelines; test the compiled program (translated VM code) according to these guidelines.
4. If the program behaves unexpectedly, or if an error message is displayed by the VM emulator, fix your compiler and go to step 1.

Test Programs

Seven: Tests how the compiler handles a simple program containing an arithmetic expression with integer constants, a do statement, and a return statement. Specifically, the program

computes the expression `1+(2*3)` and prints its value at the top left of the screen. To test whether your compiler has translated the program correctly, run the translated code in the VM emulator, and verify that it displays `7` correctly.

ConvertToBin: Tests how the compiler handles all the procedural elements of the Jack language: expressions (without arrays or method calls), functions, and the statements `if`, `while`, `do`, `let`, and `return`. The program does not test the handling of methods, constructors, arrays, strings, static variables, and field variables. Specifically, the program gets a 16-bit decimal value from `RAM[8000]`, converts it to binary, and stores the individual bits in `RAM[8001…8016]` (each location will contain `0` or `1`). Before the conversion starts, the program initializes `RAM[8001…8016]` to `–1`. To test whether your compiler has translated the program correctly, load the translated code into the VM emulator, and proceed as follows:

- Put (interactively, using the emulator's GUI) some decimal value in `RAM[8000]`.
- Run the program for a few seconds, then stop its execution.
- Check (by visual inspection) that memory locations `RAM[8001…8016]` contain the correct bits and that none of them contains `–1`.

Square: Tests how the compiler handles the object-based features of the Jack language: constructors, methods, fields, and expressions that include method calls. Does not test the handling of static variables. Specifically, this multiclass program stages a simple interactive game that enables moving a black square around the screen using the keyboard's four arrow keys.

While moving, the size of the square can be increased and decreased by pressing the `z` and `x` keys, respectively. To quit the game, press the `q` key. To test whether your compiler has translated the program correctly, run the translated code in the VM emulator, and verify that the game works as expected.

Average: Tests how the compiler handles arrays and strings. This is done by computing the average of a user-supplied sequence of integers. To test whether your compiler has translated the program correctly, run the translated code in the VM emulator, and follow the instructions displayed on the screen.

Pong: A complete test of how the compiler handles an object-based application, including the handling of objects and static variables. In the classical Pong game, a ball is moving randomly, bouncing off the edges of the screen. The user tries to hit the ball with a small paddle that can be moved by pressing the keyboard's left and right arrow keys. Each time the paddle hits the ball, the user scores a point and the paddle shrinks a little, making the game increasingly more challenging. If the user misses and the ball hits the bottom the game is over. To test whether your compiler has translated this program correctly, run the translated code in the VM emulator and play the game. Make sure to score some points to test the part of the program that displays the score on the screen.

ComplexArrays: Tests how the compiler handles complex array references and expressions. To that end, the program performs five complex calculations using arrays. For each such calculation, the program prints on the screen the expected result along with the result computed by the compiled program. To test whether your compiler has translated the program correctly, run the translated code in the VM emulator, and make sure that the expected and actual results are identical.

A web-based version of project 11 is available at www.nand2tetris.org.

11.5 Perspective

Jack is a general-purpose, object-based programming language. By design, it was made to be a relatively simple language. This simplicity allowed us to sidestep several thorny compilation issues. For example, while Jack looks like a typed language, that is hardly the case: all of Jack's data types—int, char, and boolean—are 16 bits wide, allowing Jack compilers to ignore almost all type information. In particular, when compiling and evaluating expressions, Jack compilers need not determine their types. The only exception is the compilation of method calls of the form x.m(), which requires determining the class type of x. Another aspect of the Jack type simplicity is that array elements are not typed.

Unlike Jack, most programming languages feature rich type systems, which place additional demands on their compilers: different amounts of memory must be allocated for different types of variables; conversion from one type into another requires implicit and explicit casting operations; the compilation of a simple expression like x+y depends strongly on the types of x and y; and so on.

Another significant simplification is that the Jack language does not support *inheritance*. In languages that support inheritance, the handling of method calls like x.m() depends on the class membership of the object x, which can be determined only during run-time. Therefore, compilers of object-oriented languages that feature inheritance must treat all methods as virtual and resolve their class memberships according to the run-time type of the object on which the method is applied. Since Jack does not support inheritance, all method calls can be compiled statically during compile time.

Another common feature of object-oriented languages not supported by Jack is the distinction between private and public class members. In Jack, all static and field variables are private (recognized only within the class in which they are declared), and all subroutines are public (can be called from any class).

The lack of real typing, inheritance, and public fields allows a truly independent compilation of classes: a Jack class can be compiled without accessing the code of any other class. The fields of other classes are never referred to directly, and all linking to methods of other classes is "late" and done just by name.

Many other simplifications of the Jack language are not significant and can be relaxed with

little effort. For example, one can easily extend the language with `for` and `switch` statements. Likewise, one can add the capability to assign character constants like `'c'` to `char` type variables, which is presently not supported by the language.

Finally, our code generation strategies paid no attention to optimization. Consider the high-level statement `c++`. A naïve compiler will translate it into the series of low-level VM operations `push c`, `push 1`, `add`, `pop c`. Next, the VM translator will translate each one of these VM commands further into several machine-level instructions, resulting in a considerable chunk of code. At the same time, an optimized compiler will notice that we are dealing with a simple increment and translate it into, say, the two machine instructions `@c` followed by `M=M+1`. Of course, this is only one example of the finesse expected from industrial-strength compilers. In general, compiler writers invest much effort and ingenuity to ensure that the generated code is time- and space-efficient.

In Nand to Tetris, efficiency is rarely an issue, with one major exception: the operating system. The Jack OS is based on efficient algorithms and optimized data structures, as we'll elaborate in the next chapter.