11 Compiler II: Code Generation

The syntactic component of a grammar must specify, for each sentence, a deep structure that determines its semantic interpretation.
—Noam Chomsky (b. 1928), mathematical linguist

Most programmers take compilers for granted. But if you'll stop to think about it for a moment, the ability to translate a high-level program into binary code is almost like magic. In this book we demystify this transformation by writing a compiler for *Jack*—a simple yet modern object-based language. As with Java and C#, the overall Jack compiler is based on two tiers: a virtual machine *back-end*, developed in chapters 7–8, and a typical *front-end* module, designed to bridge the gap between the high-level language and the VM language. The compiler's front-end module consists of a *syntax analyzer*, developed in chapter 10, and a *code generator*—the subject of this chapter.

Although the compiler's front-end comprises two conceptual modules, they are usually combined into a single program, as we will do here. Specifically, in chapter 10 we built a syntax analyzer capable of "understanding"—parsing—source Jack programs. In this chapter we extend the analyzer into a full-scale compiler that converts each "understood" high-level construct into an equivalent series of VM operations. This approach follows the modular *analysis-synthesis* paradigm underlying the construction of most compilers.

Modern high-level programming languages are rich and powerful. They allow defining and using elaborate abstractions such as objects and functions, implementing algorithms using elegant flow of control statements, and building data structures of unlimited complexity. In contrast, the target platforms on which these programs eventually run are spartan and minimal. Typically, they offer nothing more than a vector of registers for storage and a primitive instruction set for processing. Thus, the translation of programs from high-level to low-level is an interesting brain teaser.

If the target platform is a virtual machine, life is somewhat easier, but still the gap between the expressiveness of a high-level language and that of a virtual machine is wide and challenging.

The chapter begins with a Background section covering the minimal set of topics necessary for completing the compiler's development: managing a symbol table; representing and generating code for variables, objects, and arrays; and translating control flow commands into low-level instructions. The Specification section defines how to map the semantics of Jack programs on the VM platform and language, and the Implementation section proposes an API for a code generation module that performs this transformation. The chapter ends with the usual Project section, providing step-by-step guidelines and test programs for completing the compiler's construction.

So what's in it for *you*? Typically, students who don't take a formal compilation course don't have an opportunity to develop a full-scale compiler. Thus readers who follow our instructions and build the Jack compiler from scratch will gain an important lesson for a relatively small effort (of course, their knowledge of compilation theory will remain limited unless they take a course on the subject). Further, some of the tricks and techniques used in the code generation part of the compiler are rather clever. Seeing these tricks 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 Background

A program is essentially a series of operations that manipulate data. Thus, the compilation of high-level programs into a low-level language focuses on two main issues: *data translation* and *command translation*.

The overall compilation task entails translation all the way to binary code. However, since we are focusing on a two-tier compiler architecture, we assume throughout this chapter that the compiler generates VM code. Therefore, we do not touch low-level issues that have already been dealt with at the Virtual Machine level (chapters 7 and 8).

11.1.1 Data Translation

Programs manipulate many *types* of variables, including simple types like integers and booleans and complex types like arrays and objects. Another dimension of

interest is the variables' *kind* of life cycle and *scope*—namely, whether it is local, global, an argument, an object field, and so forth.

For each variable encountered in the program, the compiler must map the variable on an equivalent representation suitable to accommodate its *type* in the target platform. In addition, the compiler must manage the variable's life cycle and scope, as implied by its *kind*. This section describes how compilers handle these tasks, beginning with the notion of a *symbol table*.

Symbol Table High-level programs introduce and manipulate many *identifiers*. Whenever the compiler encounters an identifier, say `xxx`, it needs to know what `xxx` stands for. Is it a variable name, a class name, or a function name? If it's a variable, is `xxx` a field of an object, or an argument of a function? What type of variable is it—an integer, a boolean, a char, or perhaps some class type? The compiler must resolve these questions before it can represent `xxx`'s semantics in the target language. Further, all these questions must be answered (for code generation) each time `xxx` is encountered in the source code.

Clearly, there is a need to keep track of all the identifiers introduced by the program, and, for each one, to record what the identifier stands for in the source program and on which construct it is mapped in the target language. Most compilers maintain this information using a *symbol table* abstraction. Whenever a new identifier is encountered in the source code for the first time (e.g., in a variable declaration), the compiler adds its description to the table. Whenever an identifier is encountered elsewhere in the code, the compiler looks it up in the symbol table and gets all the necessary information about it. Here is a typical example:

Symbol table (of some hypothetical subroutine)

Name	Type	Kind	#
nAccounts	int	static	0
id	int	field	0
name	String	field	1
balance	int	field	2
sum	int	argument	0
status	boolean	local	0

The symbol table is the "Rosetta stone" that the compiler uses when translating high-level code involving identifiers. For example, consider the statement `balance=`

`balance+sum`. Using the symbol table, the compiler can translate this statement into code reflecting the facts that `balance` is field number 2 of the current object, while `sum` is argument number 0 of the running subroutine. Other details of this translation will depend on the target language.

The basic symbol table abstraction is complicated slightly due to the fact that most languages permit different program units to use the same identifiers to represent completely different things. In order to enable this freedom of expression, each identifier is implicitly associated with a *scope*, namely, the region of the program in which the identifier is recognized. The scopes are typically nested, the convention being that inner-scoped definitions hide outer ones. For example, if the statement `x++` appears in some C function, the C compiler first checks whether the identifier `x` is declared locally in the current function, and if so, generates code that increments the local variable. Otherwise, the compiler checks whether `x` is declared globally in the file, and if so, generates code that increments the global variable. The depth of this scoping convention is potentially unlimited, since some languages permit defining variables which are local only to the block of code in which they are declared.

Thus, we see that in addition to all the relevant information that must be kept about each identifier, the symbol table must also record in some way the identifier's *scope*. The classic data structure for this purpose is a *list of hash tables*, each reflecting a single scope nested within the next one in the list. When the compiler fails to find the identifier in the table associated with the current scope, it looks it up in the next table in the list, from inner scopes outward. Thus if `x` appears undeclared in a certain code segment (e.g., a method), it may be that `x` is declared in the code segment that owns the current segment (e.g., a class), and so on.

Handling Variables One of the basic challenges faced by every compiler is how to map the various types of variables declared in the source program onto the memory of the target platform. This is not a trivial task. First, different *types* of variables require different sizes of memory chunks, so the mapping is not one-to-one. Second, different *kinds* of variables have different life cycles. For example, a single copy of each static variable should be kept alive during the complete duration of the program's run-time. In contrast, each object instance of a class should have a different copy of all its instance variables (*fields*), and, when disposed, the object's memory should be recycled. Also, each time a subroutine is being called, new copies of its local and argument variables must be created—a need that is clearly seen in recursion.

That's the bad news. The good news is that we have already handled all these difficulties. In our two-tier compiler architecture, memory allocation of variables was

delegated to the VM back-end. In particular, the virtual machine that we built in chapters 7–8 includes built-in mechanisms for accommodating the standard kinds of variables needed by most high-level languages: *static*, *local*, and *argument* variables, as well as *fields* of objects. All the allocation and de-allocation details of these variables were already handled at the VM level, using the global stack and the virtual memory segments.

Recall that this functionality was not achieved easily. In fact, we had to work rather hard to build a VM implementation that maps the global stack and the virtual memory segments on the ultimate hardware platform. Yet this effort was worth our while: For any given language *L*, any *L*-to-VM compiler is now completely relieved from low-level memory management. The only thing required from the compiler is mapping the variables found in the source program on the virtual memory segments and expressing the high-level commands that manipulate them using VM commands—a rather simple translation task.

Handling Arrays *Arrays* are almost always stored as sequences of consecutive memory locations (multi-dimensional arrays are flattened into one-dimensional ones). The array name is usually treated as a pointer to the base address of the RAM block allocated to store the array in memory. In some languages like Pascal, the entire memory space necessary to represent the array is allocated when the array is declared. In other languages like Java, the array declaration results in the allocation of a single pointer only, which, eventually, may point to the array's base address. The array proper is created in memory later, if and when the array is actually constructed at run-time. This type of *dynamic memory allocation* is done from the heap, using the memory management services of the operating system. Typically, the OS has an `alloc(size)` function that knows how to find an available memory block of size `size` and return its base address to the caller. Thus, when compiling a high-level statement like `bar=new int[10]`, the compiler generates low-level code that effects the operation `bar=alloc(10)`. This results in assigning the base-address of the array's memory block to `bar`, which is exactly what we want. Figure 11.1 offers a snapshot of this practice.

Let us consider how the compiler translates the statement `bar[k]=19`. Since the symbol `bar` points to the array's base-address, this statement can be also expressed using the C-language notation `*(bar+k)=19`, that is, "store 19 in the memory cell whose address is `bar+k`." In order to implement this operation, the target language must be equipped with some sort of an indirect addressing mechanism. Specifically, instead of storing a value in some memory location *y*, we need to be able to store the value in the memory location whose address is the current contents of *y*. Different

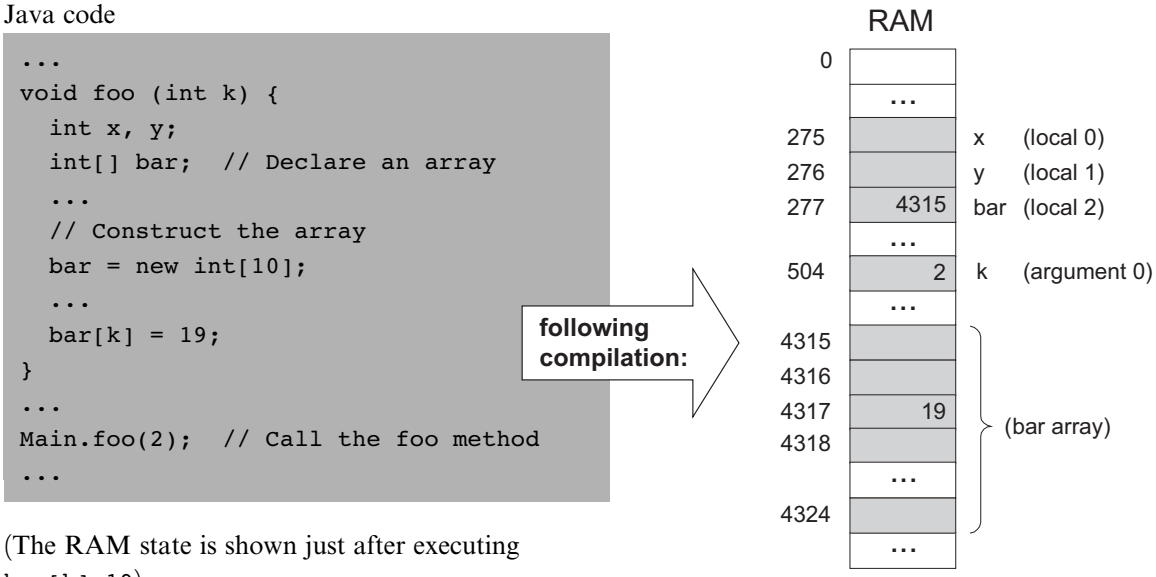


Figure 11.1 Array handling. Since memory allocations are run-time dependent, all the shown addresses are arbitrary examples.

languages have different means to carry out this pointer arithmetic, and figure 11.2 shows two possibilities.

Handling Objects Object instances of a certain class, say *Employee*, are said to encapsulate data items like *name* and *salary*, as well as a set of operations (methods) that manipulate them. The data and the operations are handled quite differently by the compiler. Let's start with the data.

The low-level handling of object data is quite similar to that of arrays, storing the fields of each object instance in consecutive memory locations. In most object-oriented languages, when a class-type variable is declared, the compiler only allocates a pointer variable. The memory space for the object proper is allocated later, if and when the object is actually created via a call to a class constructor. Thus, when compiling a constructor of some class `Xxx`, the compiler first uses the number and type of the class fields to determine how many words—say *n*—are necessary to represent an object instance of type `Xxx` on the host RAM. Next, the compiler generates the code necessary for allocating memory for the newly constructed object, for example, `this=alloc(n)`. This operation sets the `this` pointer to the base address of

Pseudo VM code

```
// bar[k]=19, or *(bar+k)=19
push bar
push k
add
// Use a pointer to access x[k]
pop addr // addr points to bar[k]
push 19
pop *addr // Set bar[k] to 19
```

Final VM code

```
// bar[k]=19, or *(bar+k)=19
push local 2
push argument 0
add
// Use the that segment to access x[k]
pop pointer 1
push constant 19
pop that 0
```

Figure 11.2 Array processing. The Hack VM code (right) follows the conventions described in section 7.2.6.

the memory block that represents the new object, which is exactly what we want. Figure 11.3 illustrates these operations in a Java context.

Since each object is represented by a pointer variable that contains its base-address, the data encapsulated by the object can be accessed linearly, using an index relative to its base. For example, suppose that the `Complex` class includes the following method:

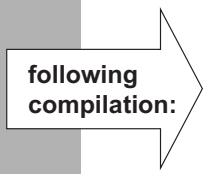
```
Public void mult (int c) {
  re = re * c;
  im = im * c;
}
```

How should the compiler handle the statement `im = im * c`? Well, an inspection of the symbol table will tell the compiler that `im` is the second field of `this` object and that `c` is the first argument of the `mult` method. Using this information, the compiler can translate `im = im * c` into code effecting the operation `*(this + 1) = *(this + 1) times (argument 0)`. Of course, the generated code will have to accomplish this operation using the target language.

Suppose now that we wish to apply the `mult` method to the `b` object, using a method call like `b.mult(5)`. How should the compiler handle this method call? Unlike the fields data (e.g., `re` and `im`), of which *different copies* are kept for each object instance, only *one copy* of each method (e.g., `mult`) is actually kept at the target code level for *all* the object instances derived from this class. In order to make it look as if each object encapsulates its own code, the compiler must force this single method to always operate on the desired object. The standard compilation

Java code

```
class Complex {
  // Properties (fields):
  int re;  // Real part
  int im;  // Imaginary part
  ...
  /** Constructs a new Complex object. */
  public Complex(int aRe, int aIm) {
    re = aRe;
    im = aIm;
  }
  ...
}
  // The following code can be in any class:
  public void bla() {
    Complex a, b, c;
    ...
    a = new Complex(5,17);
    b = new Complex(12,192);
    ...
    c = a;  // Only the reference is copied
    ...
  }
```



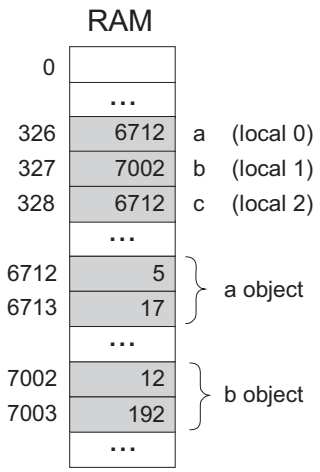


Figure 11.3 Objects handling. Since memory allocations are run-time dependent, all the shown addresses are arbitrary examples.

trick that accomplishes this abstraction is to pass a reference to the manipulated object as a hidden argument of the called method, compiling `b.mult(5)` as if it were written as `mult(b,5)`. In general then, each object-based method call `foo.bar(v1,v2,...)` is translated into the VM code `push foo`, `push v1`, `push v2`, ..., `call bar`. This way, the compiler can force the same method to operate on any desired object for instance, creating the high-level perception that each object encapsulates its own code.

However, the compiler's job is not done yet. Since the language allows different methods in different classes to have the same name, the compiler must ensure that the right method is applied to the right object. Further, due to the possibility of method overriding in a subclass, compilers of object-oriented languages must do this deter-

mination at run-time. When run-time typing is out of the picture, for example, in languages like Jack, this determination can be done at compile-time. Specifically, in each method call like `x.m(y)`, the compiler must ensure that the called method `m()` belongs to the class from which the `x` object was derived.

11.1.2 Commands Translation

We now describe how high-level commands are translated into the target language. Since we have already discussed the handling of variables, objects, and arrays, there are only two more issues to consider: *expression evaluation* and *flow control*.

Evaluating Expressions How should we generate code for evaluating high-level expressions like `x+g(2,y,-z)*5`? First, we must "understand" the syntactic structure of the expression, for example, convert it into a parse tree like the one depicted in figure 11.4. This parsing was already handled by the *syntax analyzer* described in chapter 10. Next, as seen in the figure, we can traverse the parse tree and generate from it the equivalent VM code.

The choice of the code generation algorithm depends on the target language into which we are translating. For a stack-based target platform, we simply need to print the tree in *postfix* notation, also known as *Right Polish Notation* (RPN). In RPN

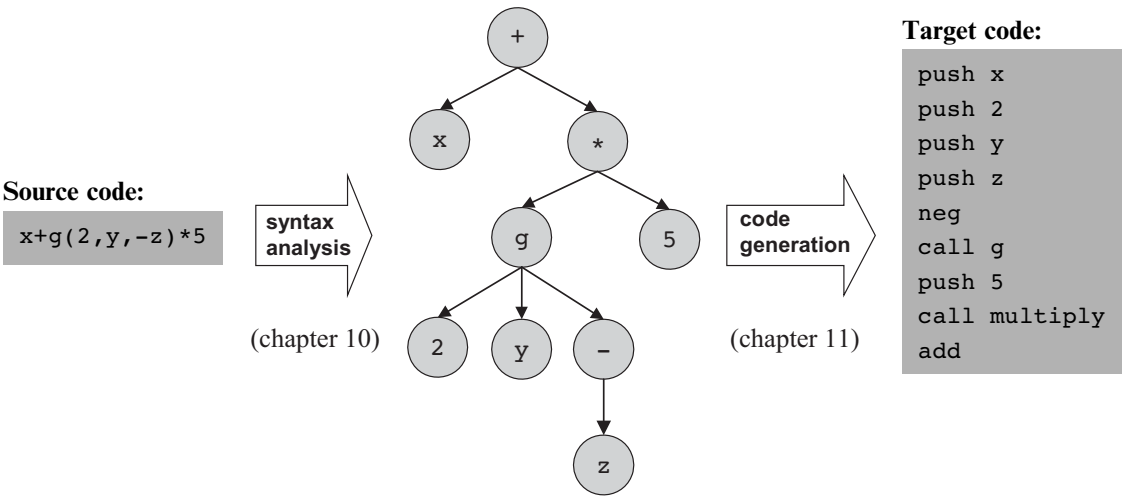


Figure 11.4 Code generation.

syntax, an operation like $f(x, y)$ is expressed as x, y, f (or, in the VM language syntax, `push x`, `push y`, `call f`). Likewise, an operation like $x + y$, which is $+(x, y)$ in prefix notation, is stated as $x, y, +$ (i.e., `push x`, `push y`, `add`). The strategy for translating expressions into stack-based VM code is straightforward and is based on recursive post-order traversal of the underlying parse tree, as follows:

codeWrite(exp):	
if *exp* is a number *n*	then output "`push n`"
if *exp* is a variable *v*	then output "`push v`"
if *exp* = (*exp1 op exp2*)	then codeWrite(*exp1*), codeWrite(*exp2*), output "`op`"
if *exp* = *op*(*exp1*)	then codeWrite(*exp1*), output "`op`"
if *exp* = *f*(*exp1* ... *expN*)	then codeWrite(*exp1*), ..., codeWrite(*expN*), output "`call f`"

The reader can verify that when applied to the tree in figure 11.4, this algorithm generates the stack-machine code shown in the figure.

Translating Flow Control High-level programming languages are equipped with a variety of control flow structures like `if`, `while`, `for`, `switch`, and so on. In contrast, low-level languages typically offer two basic control primitives: *conditional goto* and *unconditional goto*. Therefore, one of the challenges faced by the compiler writer is to translate structured code segments into target code utilizing these primitives only. As shown in figure 11.5, the translation logic is rather simple.

Two features of high-level languages make the compilation of control structures slightly more challenging than that shown in figure 11.5. First, a program normally contains multiple instances of `if` and `while` statements. The compiler can handle this multiplicity by generating and using unique label names. Second, control structures can be nested, for example, `if` within `while` within another `while` and so on. This complexity can be dealt with easily using a recursive compilation strategy.

11.2 Specification

Usage The Jack compiler accepts a single command line parameter, as follows:

```
prompt> JackCompiler source
```

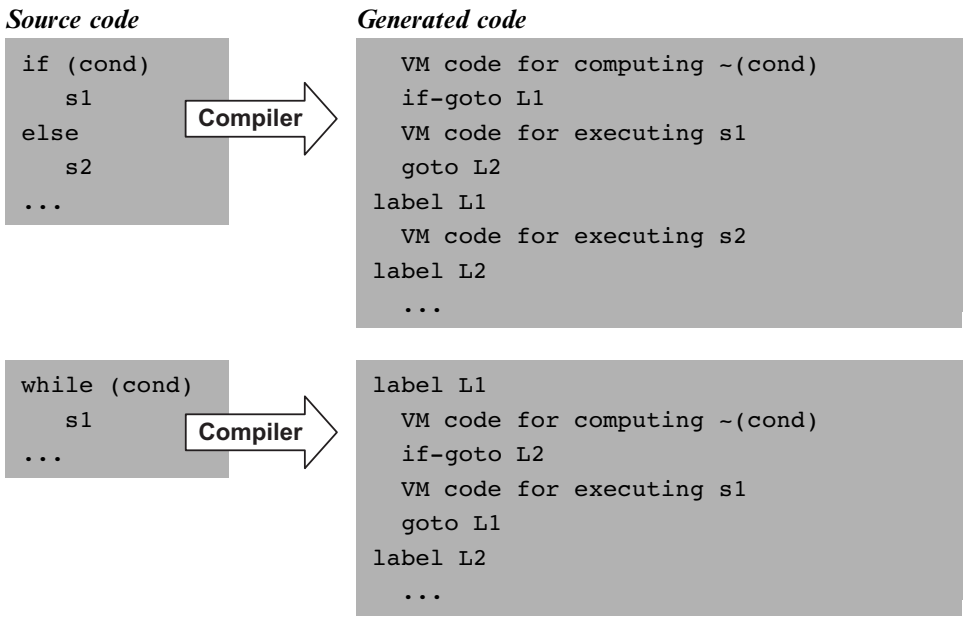


Figure 11.5 Compilation of control structures.

Where *source* is either a file name of the form `Xxx.jack` (the extension is mandatory) or a directory name containing one or more `.jack` files (in which case there is no extension). The compiler compiles each `Xxx.jack` file into a file named `Xxx.vm`, created in the same directory in which the source file is located. If *source* is a directory name, each `.jack` file located in it is compiled, creating a corresponding `.vm` file in the same directory.

11.2.1 Standard Mapping over the Virtual Machine

The compiler translates each `.jack` file into a `.vm` file containing one VM function for each constructor, function, and method found in the `.jack` file (see figure 7.8). In doing so, every Jack-to-VM compiler must follow the following code generation conventions.

File and Function Naming Each `.jack` class file is compiled into a separate `.vm` file. The Jack subroutines (functions, methods, and constructors) are compiled into VM functions as follows:

- A Jack subroutine `xxx()` in a Jack class `Yyy` is compiled into a VM function called `Yyy.xxx`.
- A Jack *function* or *constructor* with k arguments is compiled into a VM function that operates on k arguments.
- A Jack *method* with k arguments is compiled into a VM function that operates on $k + 1$ arguments. The first argument (argument number 0) always refers to the `this` object.

Memory Allocation and Access

- The *local variables* of a Jack subroutine are allocated to, and accessed via, the virtual `local` segment.
- The *argument variables* of a Jack subroutine are allocated to, and accessed via, the virtual `argument` segment.
- The *static variables* of a `.jack` class file are allocated to, and accessed via, the virtual `static` segment of the corresponding `.vm` file.
- Within a VM function corresponding to a Jack *method* or a Jack *constructor*, access to the fields of the `this` object is obtained by first pointing the virtual `this` segment to the current object (using `pointer 0`) and then accessing individual fields via `this` *index* references, where *index* is an non-negative integer.
- Within a VM function, access to array entries is obtained by first pointing the virtual `that` segment (using `pointer 1`) to the address of the desired array entry and then accessing the array entry via `that` 0 references.

Subroutine Calling

- Before calling a VM function, the caller (itself a VM function) must push the function's arguments onto the stack. If the called VM function corresponds to a Jack *method*, the first pushed argument must be a reference to the object on which the method is supposed to operate.
- When compiling a Jack *method* into a VM function, the compiler must insert VM code that sets the base of the `this` segment properly. Similarly, when compiling a Jack *constructor*, the compiler must insert VM code that allocates a memory block for the new object and then sets the base of the `this` segment to point at its base.

Returning from Void Methods and Functions High-level void subroutines don't return values. This abstraction is handled as follows:

- VM functions corresponding to *void* Jack methods and functions must return the constant 0 as their return value.
- When translating a `do sub` statement where `sub` is a void method or function, the caller of the corresponding VM function must pop (and ignore) the returned value (which is always the constant 0).

Constants

- `null` and `false` are mapped to the constant 0. `True` is mapped to the constant −1 (this constant can be obtained via `push constant 1` followed by `neg`).

Use of Operating System Services The basic Jack OS is implemented as a set of VM files named `Math.vm`, `Array.vm`, `Output.vm`, `Screen.vm`, `Keyboard.vm`, `Memory.vm`, and `Sys.vm` (the API of these compiled class files was given in chapter 9). All these files must reside alongside the VM files generated by the compiler. This way, any VM function can call any OS VM function for its effect. In particular, when needed, the compiler should generate code that uses the following OS functions:

- Multiplication and division are handled using the OS functions `Math.multiply()` and `Math.divide()`.
- String constants are created using the OS constructor `String.new(length)`. String assignments like `x="cc...c"` are handled using a series of calls to the OS routine `String.appendChar(nextChar)`.
- Constructors allocate space for new objects using the OS function `Memory.alloc(size)`.

11.2.2 Compilation Example

Compiling a Jack program (one or more `.jack` class files) involves two main tasks: parsing the code using the compilation engine developed in the previous chapter, and generating code according to the guidelines and specifications given above. Figure 11.6 gives a "live example" of many of the code generation issues mentioned in this chapter.

High-level code (*BankAccount.jack* class file)

```
/* Some common sense was sacrificed in this banking example in order to
   create a nontrivial and easy-to-follow compilation example. */
class BankAccount {
   // Class variables
   static int nAccounts;
   static int bankCommission;  // As a percentage, e.g., 10 for 10 percent
   // account properties
   field int id;
   field String owner;
   field int balance;

   method int commission(int x) { /* Code omitted */ }

   method void transfer(int sum, BankAccount from, Date when) {
      var int i, j;  // Some local variables
      var Date due;  // Date is a user-defined type
      let balance = (balance + sum) - commission(sum * 5);
      // More code ...
      return;

   }
   // More methods ...
}
```

Class-scope symbol table

Name	Type	Kind	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

Method-scope (transfer) symbol table

Name	Type	Kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

Figure 11.6 Code generation example focusing on the translation of the statement `let balance = (balance + sum) - commission(sum * 5)`.

Pseudo VM code

```
function BankAccount.commission
  // Code omitted
function BankAccount.transfer
  // Code for setting "this" to point
  // to the passed object (omitted)
  push balance
  push sum
  add
  push this
  push sum
  push 5
  call multiply
  call commission
  sub
  pop balance
  // More code ...
  push 0
  return
```

Final VM code

```
function BankAccount.commission 0
  // Code omitted
function BankAccount.transfer 3
  push argument 0
  pop pointer 0
  push this 2
  push argument 1
  add
  push argument 0
  push argument 1
  push constant 5
  call Math.multiply 2
  call BankAccount.commission 2
  sub
  pop this 2
  // More code ...
  push 0
  return
```

Figure 11.6 (continued)

11.3 Implementation

We now turn to propose a software architecture for the overall compiler. This architecture builds upon the *syntax analyzer* described in chapter 10. In fact, the current architecture is based on gradually evolving the syntax analyzer into a full-scale compiler. The overall compiler can thus be constructed using five modules:

- `JackCompiler`: top-level driver that sets up and invokes the other modules;
- `JackTokenizer`: tokenizer;
- `SymbolTable`: symbol table;
- `VMWriter`: output module for generating VM code;
- `CompilationEngine`: recursive top-down compilation engine.

11.3.1 The *JackCompiler* Module

The compiler operates on a given *source*, where *source* is either a file name of the form `Xxx.jack` or a directory name containing one or more such files. For each `Xxx.jack` input file, the compiler creates a *JackTokenizer* and an output `Xxx.vm` file. Next, the compiler uses the *CompilationEngine*, *SymbolTable*, and *VMWriter* modules to write the output file.

11.3.2 The *JackTokenizer* Module

The tokenizer API was given in section 10.3.2.

11.3.3 The *SymbolTable* Module

This module provides services for creating and using a *symbol table*. Recall that each symbol has a scope from which it is visible in the source code. The symbol table implements this abstraction by giving each symbol a running number (index) within the scope. The index starts at 0, increments by 1 each time an identifier is added to the table, and resets to 0 when starting a new scope. The following kinds of identifiers may appear in the symbol table:

Static: Scope: class.

Field: Scope: class.

Argument: Scope: subroutine (method/function/constructor).

Var: Scope: subroutine (method/function/constructor).

When compiling error-free Jack code, any identifier not found in the symbol table may be assumed to be a subroutine name or a class name. Since the Jack language syntax rules suffice for distinguishing between these two possibilities, and since no "linking" needs to be done by the compiler, there is no need to keep these identifiers in the symbol table.

SymbolTable: Provides a symbol table abstraction. The symbol table associates the identifier names found in the program with identifier properties needed for compilation: *type*, *kind*, and running index. The symbol table for Jack programs has two nested scopes (class/subroutine).

Routine	Arguments	Returns	Function
`Constructor`	—	—	Creates a new empty symbol table.
`startSubroutine`	—	—	Starts a new subroutine scope (i.e., resets the subroutine's symbol table).
`Define`	`name` (String) `type` (String) `kind` (`STATIC`, `FIELD`, `ARG`, or `VAR`)	—	Defines a new identifier of a given *name*, *type*, and *kind* and assigns it a running index. `STATIC` and `FIELD` identifiers have a class scope, while `ARG` and `VAR` identifiers have a subroutine scope.
`VarCount`	`kind` (`STATIC`, `FIELD`, `ARG`, or `VAR`)	int	Returns the number of variables of the given *kind* already defined in the current scope.
`KindOf`	`name` (String)	(`STATIC`, `FIELD`, `ARG`, `VAR`, `NONE`)	Returns the *kind* of the named identifier in the current scope. If the identifier is unknown in the current scope, returns `NONE`.
`TypeOf`	`name` (String)	String	Returns the *type* of the named identifier in the current scope.
`IndexOf`	`name` (String)	int	Returns the *index* assigned to the named identifier.

Implementation Tip The symbol table abstraction and API can be implemented using two separate hash tables: one for the class scope and another one for the subroutine scope. When a new subroutine is started, the subroutine scope table can be cleared.

11.3.4 The *VMWriter* Module

VMWriter: Emits VM commands into a file, using the VM command syntax.

Routine	Arguments	Returns	Function
`Constructor`	Output file/stream	—	Creates a new file and prepares it for writing.
`writePush`	`Segment` (`CONST`, `ARG`, `LOCAL`, `STATIC`, `THIS`, `THAT`, `POINTER`, `TEMP`) `Index` (int)	—	Writes a VM `push` command.
`writePop`	`Segment` (`CONST`, `ARG`, `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 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) `nLocals` (int)	—	Writes a VM `function` command.
`writeReturn`	—	—	Writes a VM `return` command.
`close`	—	—	Closes the output file.

11.3.5 The *CompilationEngine* Module

This class does the compilation itself. It reads its input from a `JackTokenizer` and writes its output into a `VMWriter`. It is organized as a series of `compilexxx()` routines, where `xxx` is a syntactic element of the Jack language. The contract between these routines is that each `compilexxx()` routine should read the syntactic construct `xxx` from the input, `advance()` the tokenizer exactly beyond `xxx`, and emit to the output VM code effecting the semantics of `xxx`. Thus `compilexxx()` may only be called if indeed `xxx` is the next syntactic element of the input. If `xxx` is a part of an expression and thus has a value, the emitted code should compute this value and leave it at the top of the VM stack.

The API of this module is identical to that of the syntax analyzer's *compilationEngine* module from chapter 10, and thus we suggest gradually morphing the syntax analyzer into a full compiler. Section 11.5 provides step-by-step instructions and test programs for this construction.

11.4 Perspective

The fact that Jack is a relatively simple language permitted us to sidestep several thorny compilation issues. For example, while Jack looks like a typed language, this is hardly the case. All of Jack's data types are 16-bits long, and the language semantics allows Jack compilers to ignore almost all type information. As a result, when compiling and evaluating expressions, Jack compilers need not determine their types (with the single exception that compiling a method call `x.m()` requires determining the class type of `x`). Likewise, array entries in Jack are not typed. In contrast, most programming languages feature rich type systems that have significant implications on their compilers: Different amounts of memory must be allocated for different types of variables; conversion from one type into another requires specific language 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. This implies that all method calls can be handled statically, at compile-time. In contrast, compilers of languages with inheritance must treat methods as virtual, and determine their locations according to the run-time type of the underlying object. For example, consider the method call `x.m()`. If the language supports inheritance, `x` can be derived from more than one class, and we cannot know *which*

until run-time. Thus, if the definition of the method `m` is not found in the class from which `x` was derived, it may still be found in a class that supersedes it, and so on.

Another common feature of object-oriented languages not supported by Jack is public class fields. For example, if `circ` is an object of type `Circle` with a property `radius`, one cannot write statements like `r=circ.radius`. Instead, the programmer must equip the `Circle` class with accessor methods, allowing only statements like `r=circ.getRadius()` (which is good programming practice anyway).

The lack of real typing, inheritance, and public class fields allows a truly independent compilation of classes. In particular, 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 may easily extend the language with `for` and `switch` statements. Likewise, one can add the capability to assign constants like `'c'` to `char` type variables, which is presently not supported by the language. (To assign the constant `'c'` to a Jack `char` variable `x`, one must first assign `"c"` to a `String` variable, say `s`, and then use `let x=s.charAt(0)`. Clearly, it would be nicer to simply say `let x='c'`, as in Java).

Finally, as usual, we did not pay any attention to optimization. Consider the high-level statement `c++`. A naïve compiler may translate it into the series of low-level VM operations `push c`, `push 1`, `add`, `pop c`. Next, the VM implementation will translate each one of these VM commands 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 nothing more than a simple increment, and translate it into, say, the two machine instructions `@c` followed by `M=M+1` on the Hack platform. Of course this is just one example of the finesse expected from industrial-strength compilers. Therefore, time and space efficiency play an important role in the code generation part of compilers and compilation courses.

11.5 Project

Objective Extend the *syntax analyzer* built in chapter 10 into a full-scale Jack compiler. In particular, gradually replace the software modules that generate passive XML code with software modules that generate executable VM code.

Resources The main tool that you need is the programming language in which you will implement the compiler. You will also need an executable copy of the Jack

operating system, as explained below. Finally, you will need the supplied VM Emulator, to test the code generated by your compiler on a set of test programs supplied by us.

Contract Complete the Jack compiler implementation. The output of the compiler should be VM code designed to run on the virtual machine built in the projects in chapters 7 and 8. Use your compiler to compile all the Jack programs given here. Make sure that each translated program executes according to its documentation.

Stage 1: Symbol Table

We suggest that you start by building the compiler's symbol table module and using it to extend the syntax analyzer built in Project 10. Presently, whenever an *identifier* is encountered in the program, say `foo`, the syntax analyzer outputs the XML line `<identifier> foo </identifier>`. Instead, have your analyzer output the following information as part of its XML output (using some format of your choice):

- the identifier category (*var*, *argument*, *static*, *field*, *class*, *subroutine*);
- whether the identifier is presently being *defined* (e.g., the identifier stands for a variable declared in a `var` statement) or *used* (e.g., the identifier stands for a variable in an expression);
- whether the identifier represents a variable of one of the four kinds (*var*, *argument*, *static*, *field*), and the running index assigned to the identifier by the symbol table.

You may test your symbol table module and the preceding capability by running your (extended) syntax analyzer on the test Jack programs supplied in Project 10. Once the output of your extended syntax analyzer includes this information, it means that you have developed a complete executable capability to understand the semantics of Jack programs. At this stage you can make the switch to a full-scale compiler and start generating VM code instead of XML output. This can be done by gradually morphing the code of the extended syntax analyzer into a full compiler.

Stage 2: Code Generation

We don't provide specific guidelines on how to develop the code generation features of the compiler, though the examples spread throughout the chapter are quite instructive. Instead, we provide a set of six application programs designed to unit-test these

features incrementally. We strongly suggest to test your compiler on these programs in the given order. This way, you will be implicitly guided to build the compiler's code generation capabilities in stages, according to the demands of each test program.

The Operating System The Jack OS—the subject of chapter 12—was written in the Jack language. The source OS code was then translated (by an error-free Jack compiler) into a set of VM files, collectively known as the *Jack OS*. Each time we want to run an application program on the VM emulator, we must load into the emulator not only the application's `.vm` files, but also all the OS `.vm` files. This way, when an application-level VM function calls some OS-level VM function, they will find each other in the same environment.

Testing Method Normally, when you compile a program and run into some problems, you conclude that the program is screwed up and proceed to debug it. In this project the setting is exactly the opposite. All the test programs that we supply are error-free. Therefore, if their compilation yields any errors, it's the *compiler* that you have to fix, not the test programs. For each test program, we recommend going through the following routine:

1. Copy all the supplied OS `.vm` files from `tools/OS` into the program directory, together with the supplied `.jack` file(s) comprising the test program.
2. Compile the program directory using your compiler. This operation should compile only the `.jack` files in the directory, which is exactly what we want.
3. If there are any compilation errors, fix your compiler and return to step 2 (note that all the supplied test programs are error-free).
4. At this point, the program directory should contain one `.vm` file for each source `.jack` file, as well as all the supplied OS `.vm` files. If this is not the case, fix your compiler and return to step 2.
5. Execute the translated VM program in the VM Emulator, loading the entire directory and using the "no animation" mode. Each one of the six test programs contains specific execution guidelines, as listed here.
6. If the program behaves unexpectedly or some error message is displayed by the VM emulator, fix your compiler and return to step 2.

Test Programs

We supply six test programs. Each program is designed to gradually unit-test specific language handling capabilities of your compiler.

Seven This program computes the value of `(3*2)+1` and prints the result 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 make sure that it displays 7 correctly. Purpose: Tests how your compiler handles a simple program containing an arithmetic expression with integer constants (without variables), a `do` statement, and a `return` statement.

Decimal-to-Binary Conversion This program converts a 16-bit decimal number into its binary representation. The program takes a decimal number 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 go through the following routine:

- Put (interactively) a 16-bit decimal value in `RAM[8000]`.
- Run the program for a few seconds, then stop its execution.
- Check (interactively) that `RAM[8001..8016]` contain the correct results, and that none of them contains `-1`.

Purpose: Tests how your compiler handles all the procedural elements of the Jack language, namely, *expressions* (without arrays or method calls), *functions*, and all the language *statements*. The program does not test the handling of methods, constructors, arrays, strings, static variables, and field variables.

Square Dance This program is a trivial 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 if your compiler has translated the program correctly, run the translated code in the VM emulator and make sure that it works according to this description. Purpose: Tests how your compiler handles the object-oriented constructs of the Jack language: *constructors*, *methods*, *fields* and expressions that include *method calls*. It does not test the handling of static variables.

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

Pong A ball is moving randomly on the screen, bouncing off the screen "walls." The user can move a small bat horizontally by pressing the keyboard's left and right arrow keys. Each time the bat hits the ball, the user scores a point and the bat shrinks a little, to make the game harder. If the user misses and the ball hits the bottom horizontal line, 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). Purpose: Provides a complete test of how your compiler handles *objects*, including the handling of *static variables*.

Complex Arrays Performs five complex calculations using arrays. For each such calculation, the program prints on the screen the expected result versus the actual result (as performed 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 actual results are identical to the expected results. Purpose: Tests how your compiler handles complex *array references* and *expressions*.