

Testing / Debugging: We supply five sets of test programs, test scripts, and compare files. For each test program *Xxx.vm* we recommend following these steps:

0. Use the *XxxVME.tst* script to execute the test program *Xxx.vm* on the supplied VM emulator. This will familiarize you with the intended behavior of the test program. Inspect the simulated stack and virtual segments, and make sure that you understand what the test program is doing.
1. Use your partially implemented translator to translate the file *Xxx.vm* (test program). The result should be a text file named *Xxx.asm*, containing the Hack assembly code generated by your translator.
2. Inspect the generated *Xxx.asm* code produced by your translator. If there are visible errors, debug and fix your translator.
3. Use the supplied *Xxx.tst* and *Xxx.cmp* files to run and test your translated *Xxx.asm* program on the supplied *CPU emulator*. If there are any errors, debug and fix your translator.

When you are done with this project, be sure to save a copy of your VM translator. In the next chapter you will be asked to extend this program, adding the handling of more VM commands. If your project 8 modifications end up breaking the code developed in project 7, you'll be able to resort to your backup version.

A web-based version of project 7 is available at www.nand2tetris.org.

7.6 Perspective

In this chapter we began the process of developing a compiler for a high-level language. Following modern software engineering practices, we have chosen to base the compiler on a two-tier compilation model. In the *front-end* tier, covered in chapters 10 and 11, the high-level code is translated into an intermediate code designed to run on a virtual machine. In the *back-end* tier, covered in this and in the next chapter, the intermediate code is

translated further into the machine language of a target hardware platform (see [figure 7.1](#)).

Over the years, this two-stage compilation model has been used—implicitly and explicitly—in many compiler construction projects. In the late 1970s, IBM and Apple corporations introduced two pioneering and phenomenally successful personal computers, known as the IBM PC and the Apple II. One high-level language that became popular on these early PCs was Pascal. Alas, different Pascal compilers had to be developed, since the IBM and Apple machines used different processors, different machine languages, and different operating systems. Also, IBM and Apple were rival companies and had no interest in helping developers port their software to the other machine. As a result, software developers who wanted their Pascal apps to run on both lines of computers had to use different compilers, each designed to generate machine-specific binary code. Could there not be a better way to handle cross-platform compilation, so that programs could be written once and run everywhere?

One solution to this challenge was an early virtual machine framework called *p-code*. The basic idea was to compile Pascal programs into intermediate p-code (similar to our VM language) and then use one implementation for translating the abstract p-code into Intel's x86 instruction set, used by IBM PCs, and another implementation for translating the same p-code into Motorola's 68000 instruction set, used by Apple computers. Meanwhile, other companies developed highly optimized Pascal compilers that generated efficient p-code. The net result was that the same Pascal program could run as is on practically every machine in this nascent PC market: no matter which computer your customers used, you could ship them exactly the same p-code files, obviating the need to use multiple compilers. Of course, the whole scheme was based on the assumption that the customer's computer was equipped with a client-side p-code implementation (equivalent to our VM translator). To make this happen, the p-code implementations were distributed freely over the Internet, and customers were invited to download them to their computers. Historically, this was perhaps the first time that the notion of a cross-platform high-level language began realizing its full potential.

Following the explosive growth of the Internet and mobile devices in the mid-1990s, cross-platform compatibility became a universally vexing issue.

To address the problem, the Sun Microsystems company (later acquired by Oracle) sought to develop a new programming language whose compiled code could potentially run as is on any computer and digital device connected to the Internet. The language that emerged from this initiative—*Java*—was founded on an intermediate code execution model called the *Java Virtual Machine*, or JVM.

The JVM is a specification that describes an intermediate language called *bytecode*—the target VM language of Java compilers. Files written in bytecode are widely used for code distribution of Java programs over the Internet. In order to execute these portable programs, the client PCs, tablets, and cell phones that download them must be equipped with suitable JVM implementations, known as JREs (Java Runtime Environments). These programs are widely available for numerous processor and operating system combinations. Today, many personal computer and cell phone owners use these infrastructure programs (the JREs) routinely and implicitly, without ever noticing their existence on their devices.

The Python language, conceived in the late 1980s, is also based on a two-tier translation model, whose centerpiece is the PVM (Python Virtual Machine), which uses its own version of bytecode.

In the early 2000s, Microsoft launched its .NET Framework. The centerpiece of .NET is a virtual machine framework called *Common Language Runtime* (CLR). According to Microsoft’s vision, many programming languages, like C# and C++, can be compiled into intermediate code running on the CLR. This enables code written in different languages to interoperate and share the software libraries of a common run-time environment. Of course, single-tier compilers for C and C++ are still widely used, especially in high-performance applications requiring tight and optimized code.

Indeed, one issue that was completely sidestepped in this chapter is *efficiency*. Our contract calls for developing a VM translator, without requiring that the generated assembly code be efficient. Clearly, this is a serious oversight. The VM translator is a mission-critical enabling technology, lying at the core of your PC, tablet, or cell phone: if it will generate tight and efficient low-level code, apps will run on your machine swiftly, using as little hardware resources as possible. Therefore, optimizing the VM translator is a top priority.

In general, there are numerous opportunities to optimize the VM translator. For example, assignments like `let x = y` are prevalent in high-level code; these statements are translated by the compiler into VM commands like, for example, `push local 3` followed by `pop static 1`. Clever implementations of such pairs of VM commands can generate assembly code that sidesteps the stack completely, resulting in dramatic performance gains. Of course, this is one out of many examples of possible VM optimizations. Indeed, over the years, the VM implementations of Java, Python, and C# became dramatically more powerful and sophisticated.

We note in closing that a crucial ingredient that must be added to the virtual machine model before its full potential is unleashed is a common software library. Indeed the Java Virtual Machine comes with the *standard Java class library*, and the Microsoft’s .NET Framework comes with the *Framework Class Library*. These vast software libraries can be viewed as portable operating systems, providing numerous services like memory management, GUI toolkits, string functions, math functions, and so on. These extensions will be described and built in chapter 12.

8 Virtual Machine II: Control

If everything seems under control, you're just not going fast enough.

—Mario Andretti (b. 1940), race car champion

Chapter 7 introduced the notion of a *virtual machine* (VM), and project 7 began implementing our abstract virtual machine and VM language over the Hack platform. The implementation entailed developing a program that translates VM commands into Hack assembly code. Specifically, in the previous chapter we learned how to use and implement the VM's arithmetic-logical commands and push/pop commands; in this chapter we'll learn how to use and implement the VM's branching commands and function commands. As the chapter progresses, we'll extend the basic translator developed in project 7, ending with a full-scale VM translator over the Hack platform. This translator will serve as the back-end module of the compiler that we will build in chapters 10 and 11.

In any Gre at Gems in Applied Computer Science contest, stack processing will be a strong finalist. The previous chapter showed how arithmetic and Boolean expressions can be represented and evaluated by elementary operations on a stack. This chapter goes on to show how this remarkably simple data structure can also support remarkably complex tasks like nested function calling, parameter passing, recursion, and the various memory allocation and recycling tasks required to support program execution during run-time. Most programmers tend to take these services for granted, expecting the compiler and the operating system to deliver them, one way or another. We are now in a position to open up this black box and see how these fundamental programming mechanisms are actually realized.

Run-time system: Every computer system must specify a run-time model. This model answers essential questions without which programs cannot run: how to start a program’s execution, what the computer should do when a program terminates, how to pass arguments from one function to another, how to allocate memory resources to running functions, how to free memory resources when they are no longer needed, and so on.

In Nand to Tetris, these issues are addressed by the *VM language* specification, along with the *standard mapping on the Hack platform* specification. If a VM translator is developed according to these guidelines, it will end up realizing an executable run-time system. In particular, the VM translator will not only translate the VM commands (push, pop, add, and so on) into assembly instructions—it will also generate assembly code that realizes an envelope in which the program runs. All the questions mentioned above—how to start a program, how to manage the function call-and-return behavior, and so on—will be answered by generating enabling assembly code that wraps the code proper. Let’s see an example.

8.1 High-Level Magic

High-level languages allow writing programs in high-level terms. For example, an expression like $x = -b + \sqrt{b^2 - 4 \cdot a \cdot c}$ can be written as $x = -b + \text{sqrt}(\text{power}(b, 2) - 4 * a * c)$, which is almost as descriptive as the real thing. Note the difference between primitive operations like `+` and `-` and functions like `sqrt` and `power`. The former are built into the basic syntax of the high-level language. The latter are extensions of the basic language.

The unlimited capacity to extend the language at will is one of the most important features of high-level programming languages. Of course, at some point, someone must implement functions like `sqrt` and `power`. However, the story of implementing these abstractions is completely separate from the story of using them. Therefore, application programmers can assume that each one of these functions will get executed—somehow—and that following its execution, control will return—somehow—to the next operation in one’s code. Branching commands endow the language with additional expressive power, allowing writing conditional code like `if !(a == 0) { x = (-b + sqrt(power(b, 2) - 4 * a * c)) / (2 * a) } else { x = -c / b }`. Once

again, we see that high-level code enables the expression of high-level logic—in this case the algorithm for solving quadratic equations—almost directly.

Indeed, modern programming languages are programmer-friendly, offering useful and powerful abstractions. It is a sobering thought, though, that no matter how high we allow our high-level language to be, at the end of the day it must be realized on some hardware platform that can only execute primitive machine instructions. Thus, among other things, compiler and VM architects must find low-level solutions for realizing branching and function call-and-return commands.

Functions—the bread and butter of modular programming—are standalone programming units that are allowed to call each other for their effect. For example, `solve` can call `sqrt`, and `sqrt`, in turn, can call `power`. This calling sequence can be as deep as we please, as well as recursive. Typically, the calling function (the *caller*) passes arguments to the called function (the *callee*) and suspends its execution until the latter completes *its* execution. The callee uses the passed arguments to execute or compute something and then returns a value (which may be `void`) to the caller. The caller then snaps back into action, resuming its execution.

In general then, whenever one function (the *caller*) calls a function (the *callee*), someone must take care of the following overhead:

- Save the *return address*, which is the address within the caller’s code to which execution must return after the callee completes its execution;
- Save the memory resources of the caller;
- Allocate the memory resources required by the callee;
- Make the arguments passed by the caller available to the callee’s code;
- Start executing the callee’s code.

When the callee terminates and returns a value, someone must take care of the following overhead:

- Make the callee’s *return value* available to the caller’s code;
- Recycle the memory resources used by the callee;
- Reinstate the previously saved memory resources of the caller;

- Retrieve the previously saved *return address*;
- Resume executing the caller’s code, from the return address onward.

Blissfully, high-level programmers don’t have to ever think about all these nitty-gritty chores: the assembly code generated by the compiler handles them, stealthily and efficiently. And, in a two-tier compilation model, this housekeeping responsibility falls on the compiler’s back end, which is the VM translator that we are now developing. Thus, in this chapter, we will uncover, among other things, the run-time framework that enables what is probably the most important abstraction in the art of programming: *function call-and-return*. But first, let’s start with the easier challenge of handling branching commands.

8.2 Branching

The default flow of computer programs is sequential, executing one command after the other. For various reasons like embarking on a new iteration in a loop, this sequential flow can be redirected by branching commands. In low-level programming, branching is accomplished by *goto destination* commands. The destination specification can take several forms, the most primitive being the physical memory address of the instruction that should be executed next. A slightly more abstract specification is established by specifying a symbolic label (bound to a physical memory address). This variation requires that the language be equipped with a labeling directive, designed to assign symbolic labels to selected locations in the code. In our VM language, this is done using a labeling command whose syntax is `label symbol`.

With that in mind, the VM language supports two forms of branching. *Unconditional branching* is effected using a `goto symbol` command, which means: jump to execute the command just after the `label symbol` command in the code. *Conditional branching* is effected using the `if-goto symbol` command, whose semantics is: Pop the topmost value off the stack; if it’s not false, jump to execute the command just after the `label symbol` command; otherwise, execute the next command in the code. This contract implies that before specifying a conditional `goto` command, the VM code writer (for

example, a compiler) must first specify a condition. In our VM language, this is done by pushing a Boolean expression onto the stack. For example, the compiler that we'll develop in chapters 10–11 will translate `if ($n < 100$) goto LOOP` into `push n, push 100, lt, if-goto LOOP`.

Example: Consider a function that receives two arguments, x and y , and returns the product $x \cdot y$. This can be done by adding x repetitively to a local variable, say sum , y times, and then returning sum 's value. A function that implements this naïve multiplication algorithm is listed in [figure 8.1](#). This example illustrates how a typical looping logic can be expressed using the VM branching commands `goto`, `if-goto`, and `label`.

High-level code

```
// Returns x * y
int mult(int x, int y) {
    int sum = 0;
    int i = 0;
    while (i < y) {
        sum += x;
        i++;
    }
    return sum;
}
```

VM code

```
// Returns x * y
function mult(x,y)
    push 0
    pop sum
    push 0
    pop i
label WHILE_LOOP
    push i
    push y
    lt
    neg
    if-goto WHILE_END
    push sum
    push x
    add
    pop sum
    push i
    push 1
    add
    pop i
    goto WHILE_LOOP
label WHILE_END
    push sum
    return
```

Figure 8.1 Branching commands action. (The VM code on the right uses symbolic variable names instead of virtual memory segments, to make it more readable.)

Notice how the Boolean condition `!(i < y)`, implemented as `push i, push y, lt, neg`, is pushed onto the stack just before the `if-goto WHILE_END` command. In

chapter 7 we saw that VM commands can be used to express and evaluate any Boolean expression. As we see in [figure 8.1](#), high-level control structures like if and while can be easily realized using nothing more than goto and if-goto commands. In general, any flow of control structure found in high-level programming languages can be realized using our (rather minimal set of) VM logical and branching commands.

Implementation: Most low-level machine languages, including Hack, feature means for declaring symbolic labels and for effecting conditional and unconditional “goto label” actions. Therefore, if we base the VM implementation on a program that translates VM commands into assembly instructions, implementing the VM branching commands is a relatively simple matter.

Operating system: We end this section with two side comments. First, VM programs are not written by humans. Rather, they are written by compilers. [Figure 8.1](#) illustrates source code on the left and VM code on the right. In chapters 10–11 we’ll develop a *compiler* that translates the former into the latter. Second, note that the mult implementation shown in [figure 8-1](#) is inefficient. Later in the book we’ll present optimized multiplication and division algorithms that operate at the bit level. These algorithms will be used for realizing the Math.multiply and Math.divide functions, which are part of the operating system that we will build in chapter 12.

Our OS will be written in the Jack language, and translated by a Jack compiler into the VM language. The result will be a library of eight files named Math.vm, Memory.vm, String.vm, Array.vm, Output.vm, Screen.vm, Keyboard.vm, and Sys.vm (the OS API is given in appendix 6). Each OS file features a collection of useful functions that any VM function is welcome to call for their effect. For example, whenever a VM function needs multiplication or division services, it can call the Math.multiply or Math.divide function.

8.3 Functions

Every programming language is characterized by a fixed set of built-in operations. In addition, high-level and some low-level languages offer the

great freedom of extending this fixed repertoire with an open-ended collection of programmer-defined operations. Depending on the language, these canned operations are typically called *subroutines*, *procedures*, *methods*, or *functions*. In our VM language, all these programming units are referred to as *functions*.

In well-designed languages, built-in commands and programmer-defined functions have the same look and feel. For example, to compute $x + y$ on our stack machine, we push x , push y , and add. In doing so, we expect the add implementation to pop the two top values off the stack, add them up, and push the result onto the stack. Suppose now that either we, or someone else, has written a *power* function designed to compute x^y . To use this function, we follow exactly the same routine: we push x , push y , and call power. This consistent calling protocol allows composing primitive commands and function calls seamlessly. For example, expressions like $(x + y)^3$ can be evaluated using push x , push y , add, push 3, call power.

We see that the only difference between applying a primitive operation and invoking a function is the keyword `call` preceding the latter. Everything else is exactly the same: both operations require the caller to set the stage by pushing arguments onto the stack, both operations are expected to consume their arguments, and both operations are expected to push return values onto the stack. This calling protocol has an elegant consistency which, we hope, is not lost on the reader.

Example: [Figure 8.2](#) shows a VM program that computes the function $\sqrt{x^2 + y^2}$, also known as *hypot*. The program consists of three functions, with the following run-time behavior: `main` calls `hypot`, and then `hypot` calls `mult`, twice. There is also a call to a `sqrt` function, which we don't track, to reduce clutter.

```

0 function main()
// Computes hypot(3,4)
1   push 3
2   push 4
3   call hypot
4   return

5 function hypot(x,y)
// Computes sqrt(x*x + y*y)
6   push x
7   push x
8   call mult
9   push y
10  push y
11  call mult
12  add
13  call sqrt
14  return

15 function mult(x,y)
// Computes x * y (same as in figure 8.1)
16  push 0
17  pop sum
18  push 0
19  pop i
...
36  push sum
37  return

```

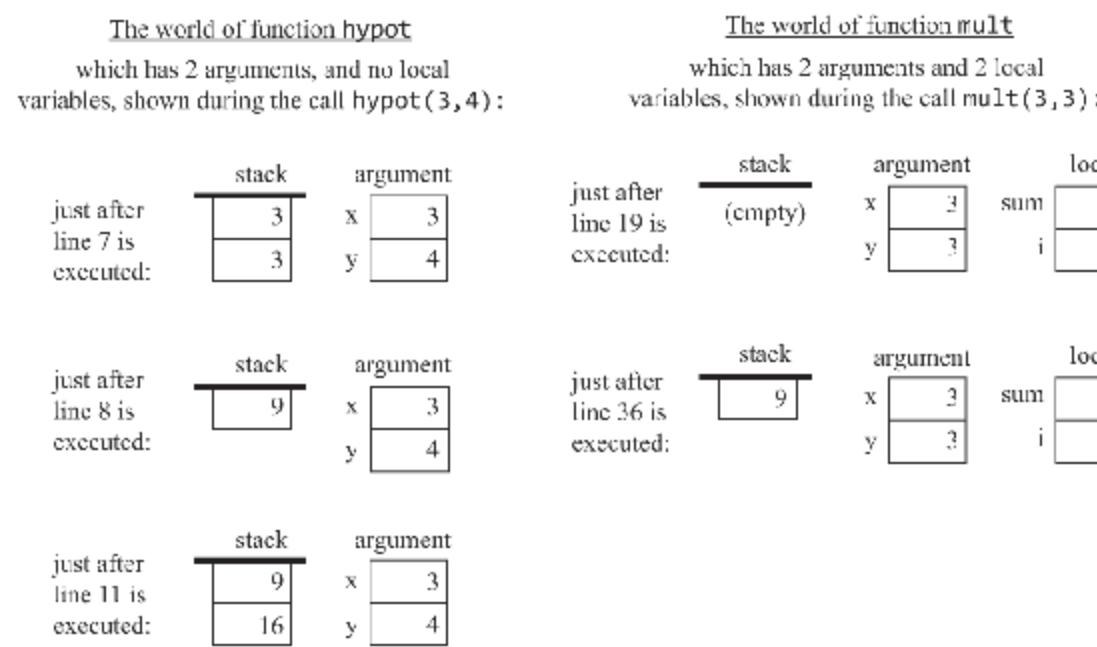


Figure 8.2 Run-time snapshots of selected stack and segment states during the execution of a three-function program. The line numbers are not part of the code and are given for reference only.

The bottom part of [figure 8.2](#) shows that during run-time, each function sees a private world, consisting of its own working stack and memory segments. These separate worlds are connected through two “wormholes”: when a function says call mult, the arguments that it pushed onto its stack prior to the call are somehow passed to the argument segment of the callee. Likewise, when a function says return, the last value that it pushed onto its stack just before returning is somehow copied onto the stack of the caller, replacing the previously pushed arguments. These hand-shaking actions are carried out by the VM implementation, as we now turn to describe.

Implementation: A computer program consists of typically several and possibly many functions. Yet at any given point during run-time, only a few of these functions are actually doing something. We use the term *calling chain* to refer, conceptually, to all the functions that are currently involved in the program’s execution. When a VM program starts running, the calling chain consists of one function only, say, `main`. At some point, `main` may call another function, say, `foo`, and that function may call yet another function, say, `bar`. At this point the calling chain is `main → foo → bar`. Each function in the calling chain waits for the function that it called to return. Thus, the only function that is truly active in the calling chain is the last one, which we call the *current function*, meaning the currently executing function.

In order to carry out their work, functions normally use *local* and *argument* variables. These variables are temporary: the memory segments that represent them must be allocated when the function starts executing and can be recycled when the function returns. This memory management task is complicated by the requirement that function calling is allowed to be arbitrarily nested, as well as recursive. During run-time, each function call must be executed independently of all the other calls and maintain its own stack, local variables, and argument variables. How can we implement this unlimited nesting mechanism and the memory management tasks associated with it?

The property that makes this housekeeping task tractable is the linear nature of the call-and-return logic. Although the function calling chain may be arbitrarily deep as well as recursive, at any given point in time only one function executes at the chain’s end, while all the other functions up the calling chain are waiting for it to return. This *Last-In-First-Out* processing model lends itself perfectly to the stack data structure, which is also LIFO. Let’s takes a closer look.

Assume that the current function is `foo`. Suppose that `foo` has already pushed some values onto its working stack and has modified some entries in its memory segments. Suppose that at some point `foo` wants to call another function, `bar`, for its effect. At this point we have to put `foo`’s execution on hold until `bar` will terminate *its* execution. Now, putting `foo`’s working stack on hold is not a problem: because the stack grows only in one direction, the working stack of `bar` will never override previously pushed values. Therefore, saving the working stack of the caller is easy—

we get it “for free” thanks to the linear and unidirectional stack structure. But how can we save *foo*’s memory segments? Recall that in chapter 7 we used the pointers LCL, ARG, THIS, and THAT to refer to the base RAM addresses of the local, argument, this, and that segments of the current function. If we wish to put these segments on hold, we can push their pointers onto the stack and pop them later, when we’ll want to bring *foo* back to life. In what follows, we use the term *frame* to refer, collectively, to the set of pointer values needed for saving and reinstating the function’s state.

We see that once we move from a single function setting to a multifunction setting, the humble stack begins to attain a rather formidable role in our story. Specifically, we now use the same data structure to hold both the working stacks as well as the frames of all the functions up the calling chain. To give it the respect that it deserves, from now on we’ll refer to this hard-working data structure as the *global stack*. See [figure 8.3](#) for the details.

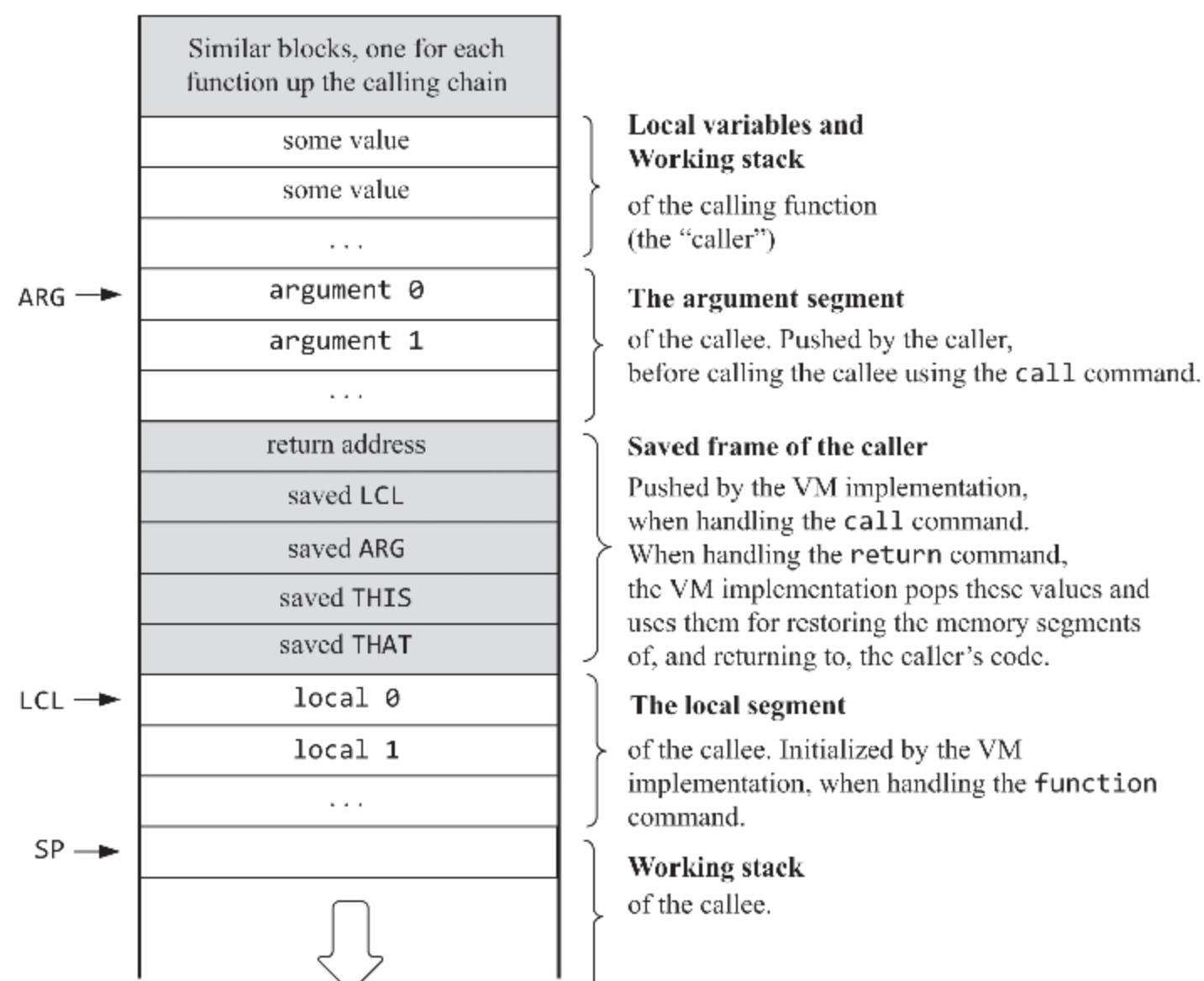


Figure 8.3 The global stack, shown when the callee is running. Before the callee terminates, it pushes a return value onto the stack (not shown). When the VM implementation handles the `return` command, it copies the return value onto `argument 0`, and sets `SP` to point to the address just

following it. This effectively frees the global stack area below the new value of SP. Thus, when the caller resumes its execution, it sees the return value at the top of its working stack.

As shown in [figure 8.3](#), when handling the `call functionName` command, the VM implementation pushes the caller’s frame onto the stack. At the end of this housekeeping, we are ready to jump to executing the callee’s code. This mega jump is not hard to implement. As we’ll see later, when handling a function `functionName` command, we use the function’s name to create, and inject into the generated assembly code stream, a unique symbolic label that marks where the function starts. Thus, when handling a “function `functionName`” command, we can generate assembly code that effects a “`goto functionName`” operation. When executed, this command will effectively transfer control to the callee.

Returning from the callee to the caller when the former terminates is trickier, since the VM `return` command specifies no return address. Indeed, the caller’s anonymity is inherent in the notion of a function call: functions like `mult` or `sqrt` are designed to serve any caller, implying that a return address cannot be specified a priori. Instead, a `return` command is interpreted as follows: redirect the program’s execution to the memory location holding the command just following the `call` command that invoked the current function.

The VM implementation can realize this contract by (i) saving the return address just before control is transferred to executing the caller and (ii) retrieving the return address and jumping to it just after the callee returns. But where shall we save the return address? Once again, the resourceful stack comes to the rescue. To remind you, the VM translator advances from one VM command to the next, generating assembly code as it goes along. When we encounter a `call foo` command in the VM code, we know exactly which command should be executed when `foo` terminates: it’s the assembly command just after the assembly commands that realize the `call foo` command. Thus, we can have the VM translator plant a label right there, in the generated assembly code stream, and push this label onto the stack. When we later encounter a `return` command in the VM code, we can pop the previously saved return address off the stack—let’s call it `returnAddress`—and effect the operation `goto returnAddress` in assembly. This is the low-

level trick that enables the run-time magic of redirecting control back to the right place in the caller's code.

The VM implementation in action: We now turn to give a step-by-step illustration of how the VM implementation supports the function call-and-return action. We will do it in the context of executing a factorial function, designed to compute $n!$ recursively. [Figure 8.4](#) gives the program's code, along with selected snapshots of the global stack during the execution of `factorial(3)`. A complete run-time simulation of this computation should also include the call-and-return action of the `mult` function, which, in this particular run-time example, is called twice: once before `factorial(2)` returns, and once before `factorial(3)` returns.

High-level code

```
// Tests the factorial function
int main() {
    return factorial(3);
}

// Computes n!
int factorial(int n) {
    if (n==1)
        return 1;
    else
        return n * factorial(n-1);
}
```

VM code

```
// Tests the factorial function
function main
    push 3
    call factorial
    return
// Computes n!
function factorial(n)
    push n
    push 1
    eq
    if-goto BASE_CASE
    push n
    push n
    push 1
    sub
    call factorial
    call mult
    return
label BASE_CASE
    push 1
    return
```

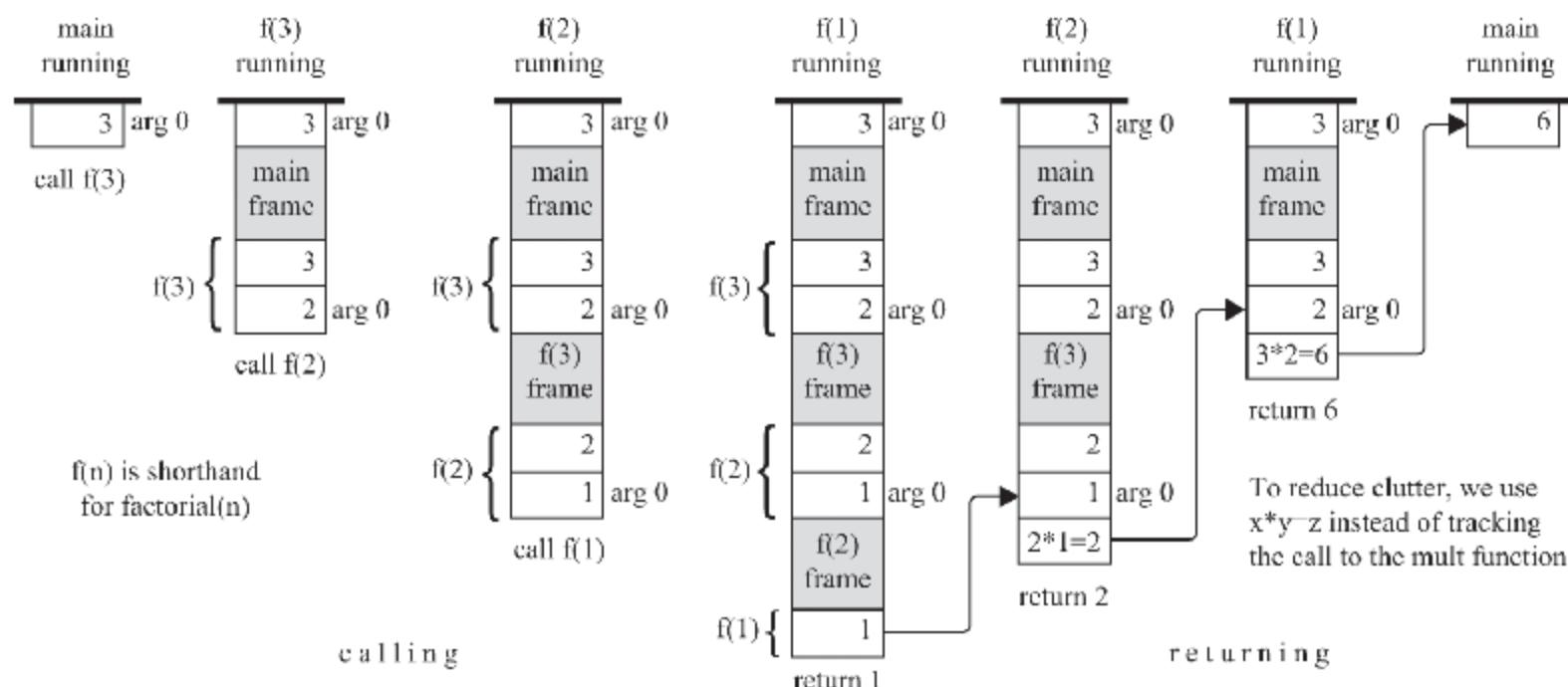


Figure 8.4 Several snapshots of the *global stack*, taken during the execution of the `main` function, which calls `factorial` to compute $3!$. The running function sees only its working stack, which is the unshaded area at the tip of the global stack; the other unshaded areas in the global stack are the working stacks of functions up the calling chain, waiting for the currently running function to return. Note that the shaded areas are not “drawn to scale,” since each frame consists of five words, as shown in [figure 8.3](#).

Focusing on the leftmost and rightmost parts of the bottom of [figure 8.4](#), here is what transpired from `main`'s perspective: “To set the stage, I pushed the constant 3 onto the stack, and then called `factorial` for its effect (see the leftmost stack snapshot). At this point I was put to sleep; at some later point in time I was woken up to find out that the stack now contains 6 (see the final and rightmost stack snapshot); I have no idea how this magic

happened, and I don't really care; all I know is that I set out to compute $3!$, and I got exactly what I asked for.” In other words, the caller is completely oblivious of the elaborate mini-drama that was unleashed by its call command.

As seen in [figure 8.4](#), the back stage on which this drama plays out is the global stack, and the choreographer who runs the show is the VM implementation: Each call operation is implemented by saving the frame of the caller on the stack and jumping to execute the callee. Each return operation is implemented by (i) using the most recently stored frame for getting the return address within the caller’s code and reinstating its memory segments, (ii) copying the topmost stack value (the return value) onto the stack location associated with argument 0, and (iii) jumping to execute the caller’s code from the return address onward. All these operations must be realized by generated assembly code.

Some readers may wonder why we have to get into all these details. There are at least three reasons why. First, we need them in order to implement the VM translator. Second, the implementation of the function call-and-return protocol is a beautiful example of low-level software engineering, so we can simply enjoy seeing it in action. Third, an intimate understanding of the virtual machine internals helps us become better and more informed high-level programmers. For example, tinkering with the stack provides an in-depth understanding of the benefits and pitfalls associated with recursion. Note that during run-time, each recursive call causes the VM implementation to add to the stack a memory block consisting of arguments, function frames, local variables, and a working stack for the callee. Therefore, unchecked use of recursion may well lead to the infamous *stack overflow* run-time debacle. This, as well as efficiency considerations, leads compiler writers to try to reexpress recursive code as sequential code, where possible. But that’s a different story that will be taken up in chapter 11.

8.4 VM Specification, Part II

So far in this chapter, we have presented general VM commands without committing to exact syntax and programming conventions. We now turn to specify formally the VM *branching* commands, the VM *function* commands, and the structure of VM *programs*. This completes the specification of the VM language that we began describing in VM Specification, Part I, in chapter 7.

It's important to reiterate that, normally, VM programs are not written by humans. Rather, they are generated by compilers. Therefore, the specifications described here are aimed at compiler developers. That is, if you write a compiler that is supposed to translate programs from some high-level language into VM code, the code that your compiler generates is expected to conform to the conventions described here.

Branching Commands

- `label label`: Labels the current location in the function's code. Only labeled locations can be jumped to. The scope of the label is the function in which it is defined. The *label* is a string composed of any sequence of letters, digits, underscore (_), dot (.), and colon (:) that does not begin with a digit. The `label` command can be located anywhere in the function, before or after the `goto` commands that refer to it.
- `goto label`: Effects an unconditional `goto` operation, causing execution to continue from the location marked by the label. The `goto` command and the labeled jump destination must be located in the same function.
- `if-goto label`: Effects a conditional `goto` operation. The stack's topmost value is popped; if the value is not zero, execution continues from the location marked by the label; otherwise, execution continues from the next command in the program. The `if-goto` command and the labeled jump destination must be located in the same function.

Function Commands

- function *functionName* *nVars*: Marks the beginning of a function named *functionName*. The command informs that the function has *nVars* local variables.
- call *functionName* *nArgs*: Calls the named function. The command informs that *nArgs* arguments have been pushed onto the stack before the call.
- return: Transfers execution to the command just following the call command in the code of the function that called the current function.

VM Program

VM programs are generated from high-level programs written in languages like Jack. As we'll see in the next chapter, a high-level Jack program is loosely defined as a collection of one or more .jack class files stored in the same folder. When applied to that folder, the Jack compiler translates each class file *FileName.jack* into a corresponding file named *FileName.vm*, containing VM commands.

Following compilation, each *constructor*, *function (static method)*, and *method* named *bar* in a Jack file *FileName.jack* is translated into a corresponding VM function, uniquely identified by the VM function name *FileName.bar*. The scope of VM function names is global: all the VM functions in all the .vm files in the program folder see each other and may call each other using the unique and full function name *FileName.functionName*.

Program entry point: One file in any Jack program must be named *Main.jack*, and one function in this file must be named *main*. Thus, following compilation, one file in any VM program is expected to be named *Main.vm*, and one VM function in this file is expected to be named *Main.main*, which is the application's entry point. This run-time convention is implemented as follows. When we start running a VM program, the first function that always executes is an argument-less VM function named *Sys.init*, which is part of the operating system. This OS function is programmed to call the entry point function in the user's program. In the case of Jack programs, *Sys.init* is programmed to call *Main.main*.

Program execution: There are several ways to execute VM programs, one of which is using the supplied *VM emulator* introduced in chapter 7. When you load a program folder containing one or more .vm files into the VM emulator, the emulator loads all the VM functions in all these files, one after the other (the order of the loaded VM functions is insignificant). The resulting code base is a sequence of all the VM functions in all the .vm files in the program folder. The notion of VM files ceases to exist, although it is implicit in the names of the loaded VM functions (*FileName.functionName*).

The Nand to Tetris VM emulator, which is a Java program, features a built-in implementation of the Jack OS, also written in Java. When the emulator detects a call to an OS function, for example, call Math.sqrt, it proceeds as follows. If it finds a corresponding function Math.sqrt command in the loaded VM code, the emulator executes the function's VM code. Otherwise, the emulator reverts to using its built-in implementation of the Math.sqrt method. This implies that as long as you use the supplied VM emulator for executing VM programs, there is no need to include OS files in your code. The VM emulator will service all the OS calls found in your code, using its built-in OS implementation.

8.5 Implementation

The previous section completed the specification of our VM language and framework. In this section we focus on implementation issues, leading up to the construction of a full-scale, VM-to-Hack translator. Section 8.5.1 proposes how to implement the function call-and-return protocol. Section 8.5.2 completes the standard mapping of the VM implementation over the Hack platform. Section 8.5.3 gives a proposed design and API for completing the VM translator that we began building in project 7.

8.5.1 Function Call and Return

The events of calling a function and returning from a function call can be viewed from two perspectives: that of the *calling function*, also referred to as *caller*, and that of the *called function*, also referred to as *callee*. Both the

caller and the callee have certain expectations, and certain responsibilities, regarding the handling of the call, function, and return commands. Fulfilling the expectations of one is the responsibility of the other. In addition, the VM implementation plays an important role in executing this contract. In what follows, the responsibilities of the VM implementation are marked by [†]:

The caller's view:	The callee's view:
<ul style="list-style-type: none"> • Before calling a function, I must push onto the stack as many arguments (<i>nArgs</i>) as the callee expects to get. • Next, I invoke the callee using the command <code>call fileName functionName nArgs</code>. • After the callee returns, the argument values that I pushed before the call have disappeared from the stack, and a <i>return value</i> (that always exists) appears at the top of the stack. Except for this change, my working stack is exactly the same as it was before the call [†]. • After the callee returns, all my memory segments are exactly the same as they were before the call [†], except that the contents of my static segment may have changed, and the temp segment is undefined. 	<ul style="list-style-type: none"> • Before I start executing, my argument segment has been initialized with the argument values passed by the caller, and my local variables segment has been allocated and initialized to zeros. My static segment has been set to the static segment of the VM file to which I belong, and my working stack is empty. The memory segments this, that, pointer, and temp are undefined upon entry [†]. • Before returning, I must push a return value onto the stack.

The VM implementation supports this contract by maintaining, and manipulating, the global stack structure described in [figure 8.3](#). In particular, every function, call, and return command in the VM code is handled by generating assembly code that manipulates the global stack as follows: A call command generates code that saves the frame of the caller on the stack and jumps to execute the callee. A function command generates code that initializes the local variables of the callee. Finally, a return command generates code that copies the return value to the top of the caller's working

stack, reinstates the segment pointers of the caller, and jumps to execute the latter from the return address onward. See [figure 8.5](#) for the details.

<i>VM command</i>	<i>Assembly (pseudo) code, generated by the VM translator</i>
<code>call f nArgs</code> <small>(calls a function <i>f</i>, informing that <i>nArgs</i> arguments were pushed to the stack before the call)</small>	<pre>push returnAddress // generates a label and pushes it to the stack push LCL // saves LCL of the caller push ARG // saves ARG of the caller push THIS // saves THIS of the caller push THAT // saves THAT of the caller ARG = SP-5-<i>nArgs</i> // repositions ARG LCL = SP // repositions LCL goto f // transfers control to the callee (<i>returnAddress</i>) // injects the return address label into the code</pre>
<code>function f nVars</code> <small>(declares a function <i>f</i>, informing that the function has <i>nVars</i> local variables)</small>	<pre>(<i>f</i>) repeat <i>nVars</i> times: push 0 // initializes the local variables to 0</pre>
<code>return</code> <small>(terminates the current function and returns control to the caller)</small>	<pre>frame = LCL // <i>frame</i> is a temporary variable retAddr = *(<i>frame</i>-5) // puts the return address in a temporary variable *ARG = pop() // repositions the return value for the caller SP = ARG+1 // repositions SP for the caller THAT = *(<i>frame</i>-1) // restores THAT for the caller THIS = *(<i>frame</i>-2) // restores THIS for the caller ARG = *(<i>frame</i>-3) // restores ARG for the caller LCL = *(<i>frame</i>-4) // restores LCL for the caller goto retAddr // go to the return address</pre>

Figure 8.5 Implementation of the function commands of the VM language. All the actions described on the right are realized by generated Hack assembly instructions.

8.5.2 Standard VM Mapping on the Hack Platform, Part II

Developers of the VM implementation on the Hack computer are advised to follow the conventions described here. These conventions complete the Standard VM Mapping on the Hack Platform, Part I, guidelines given in section 7.4.1.

The stack: On the Hack platform, RAM locations 0 to 15 are reserved for pointers and virtual registers, and RAM locations 16 to 255 are reserved for static variables. The stack is mapped on address 256 onward. To realize this mapping, the VM translator should start by generating assembly code that sets SP to 256. From this point onward, when the VM translator encounters

commands like pop, push, add, and so on in the source VM code, it generates assembly code that affects these operations by manipulating the address that SP points at, and modifying SP, as needed. These actions were explained in chapter 7 and implemented in project 7.

Special symbols: When translating VM commands into Hack assembly, the VM translator deals with two types of symbols. First, it manages predefined assembly-level symbols like SP, LCL, and ARG. Second, it generates and uses symbolic labels for marking return addresses and function entry points. To illustrate, let us revisit the PointDemo program presented in the introduction to part II. This program consists of two Jack class files, Main.jack ([figure II.1](#)) and Point.jack ([figure II.2](#)), stored in a folder named PointDemo. When applied to the PointDemo folder, the Jack compiler produces two VM files, named Main.vm and Point.vm. The first file contains a single VM function, Main.main, and the second file contains the VM functions Point.new, Point.getx, ..., Point.print.

When the VM translator is applied to this same folder, it produces a single assembly code file, named PointDemo.asm. At the assembly code level, the function abstractions no longer exist. Instead, for each function command, the VM translator generates an entry label in assembly; for each call command, the VM translator (i) generates an assembly goto instruction, (ii) creates a return address label and pushes it onto the stack, and (iii) injects the label into the generated code. For each return command, the VM translator pops the return address off the stack and generates a goto instruction. For example:

<u>VM code</u>	<u>Generated assembly code</u>
function Main.main	(Main.main)
...	...
call Point.new	goto Point.new
// Next VM command	(Main.main\$ret0)
...	// Next VM command (in assembly)
...	...
function Point.new	(Point.new)
...	...
return	goto Main.main\$ret0

Figure 8.6 specifies all the symbols that the VM translator handles and generates.

Symbol	Usage
SP	This predefined symbol points to the memory address within the host RAM just following the address containing the topmost stack value.
LCL, ARG, THIS, THAT	These predefined symbols point, respectively, to the base RAM addresses of the virtual segments <code>local</code> , <code>argument</code> , <code>this</code> , and that of the currently running VM function.
$Xxx.i$ symbols (represent static variables)	Each reference to <code>static i</code> appearing in file $Xxx.vm$ is translated to the assembly symbol $Xxx.i$. In the subsequent assembly process, the Hack assembler will allocate these symbolic variables to the RAM, starting at address 16.
$functionName$label$ (destinations of goto commands)	Let <code>foo</code> be a function within the file $Xxx.vm$. The handling of each <code>label bar</code> command within <code>foo</code> generates, and injects into the assembly code stream, the symbol $Xxx.foo$bar$. When translating <code>goto bar</code> and <code>if-goto bar</code> commands (within <code>foo</code>) into assembly, the label $Xxx.foo$bar$ must be used instead of <code>bar</code> .
$functionName$ (function entry point symbols)	The handling of each <code>function foo</code> command within the file $Xxx.vm$ generates, and injects into the assembly code stream, a symbol $Xxx.foo$ that labels the entry-point to the function's code. In the subsequent assembly process, the assembler translates this symbol into the physical address where the function code starts.
$functionName$ret.i$ (return address symbols)	Let <code>foo</code> be a function within the file $Xxx.vm$. The handling of each <code>call</code> command within <code>foo</code> 's code generates, and injects into the assembly code stream, a symbol $Xxx.foo$ret.i$, where i is a running integer (one such symbol is generated for each <code>call</code> command within <code>foo</code>). This symbol is used to mark the return address within the caller's code. In the subsequent assembly process, the assembler translates this symbol into the physical memory address of the command immediately following the <code>call</code> command.
R13 - R15	These predefined symbols can be used for any purpose. For example, if the VM translator generates assembly code that needs to use low-level variables for temporary storage, R13 - R15 can come handy.

Figure 8.6 The naming conventions described above are designed to support the translation of multiple `.vm` files and functions into a single `.asm` file, ensuring that the generated assembly symbols will be unique within the file.

Bootstrap code: The standard VM mapping on the Hack platform stipulates that the stack be mapped on the host RAM from address 256 onward, and that the first VM function that should start executing is the OS function `Sys.init`. How can we effect these conventions on the Hack platform? Recall that when we built the Hack computer in chapter 5, we wired it in such a way that upon reset, it will fetch and execute the instruction located in ROM address 0. Thus, if we want the computer to execute a predetermined code segment when it boots up, we can put this

code in the Hack computer's instruction memory, starting at address 0. Here is the code:

```
// Bootstrap (pseudo) code, should be expressed in machine language  
SP = 256  
call Sys.init
```

The `Sys.init` function, which is part of the operating system, is then expected to call the main function of the application, and enter an infinite loop. This action will cause the translated VM program to start running. Note that the notions of *application* and *main function* vary from one high-level language to another. In the Jack language, the convention is that `Sys.init` should call the VM function `Main.main`. This is similar to the Java setting: when we instruct the JVM to execute a given Java class, say, `Foo`, it looks for, and executes, the `Foo.main` method. In general, we can effect language-specific startup routines by using different versions of the `Sys.init` function.

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

```
prompt> VMTranslator source
```

where *source* is either a file name of the form `Xxx.vm` (the extension is mandatory) or the name of a folder (in which case there is no extension) containing one or more `.vm` files. The file/folder name may contain a file path. If no path is specified, the translator operates on the current folder. The output of the VM translator is a single assembly file, named *source.asm*. If *source* is a folder name, the single `.asm` file contains the translation of all the functions in all the `.vm` files in the folder, one after the other. 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.

8.5.3 Design Suggestions for the VM Implementation

In project 7 we proposed building the basic VM translator using three modules: `VMTranslator`, `Parser`, and `CodeWriter`. We now describe how to extend this basic implementation into a full-scale VM translator. This extension can be accomplished by adding the functionality described below to the

three modules already built in project 7. There is no need to develop additional modules.

The VMTranslator

If the translator's input is a single file, say `Prog.vm`, the `VMTranslator` constructs a `Parser` for parsing `Prog.vm` and a `CodeWriter` that starts by creating an output file named `Prog.asm`. Next, the `VMTranslator` enters a loop that uses the `Parser` services for iterating through the input file and parsing each line as a VM command, barring white space. For each parsed command, the `VMTranslator` uses the `CodeWriter` for generating Hack assembly code and emitting the generated code into the output file. All this was already done in project 7.

If the translator's input is a folder, named, say, `Prog`, the `VMTranslator` constructs a `Parser` for handling each `.vm` file in the folder, and a single `CodeWriter` for generating Hack assembly code into the single output file `Prog.asm`. Each time the `VMTranslator` starts translating a new `.vm` file in the folder, it must inform the `CodeWriter` that a new file is now being processed. This is done by calling a `CodeWriter` routine named `setFileName`, as we now turn to describe.

The Parser

This module is identical to the `Parser` developed in project 7.

The CodeWriter

The `CodeWriter` developed in project 7 was designed to handle the VM *arithmetic-logical* and *push / pop* commands. Here is the API of a complete `CodeWriter` that handles all the commands in the VM language:

<i>Routine</i>	<i>Arguments</i>	<i>Returns</i>	<i>Function</i>
Constructor / initializer	Output file / stream	—	Opens an output file / stream and gets ready to write into it.
			Writes the assembly instructions that effect the bootstrap code that starts the program's execution. This code must be placed at the beginning of the generated output file / stream.
			Comment: See the <i>Implementation Tips</i> at the end of section 8.6.
<code>setFileName</code>	<code>fileName</code> (string)	—	Informs that the translation of a new VM file has started (called by the <code>VMTranslator</code>).
<code>writeArithmetic</code> (developed in project 7)	<code>command</code> (string)	—	Writes to the output file the assembly code that implements the given arithmetic-logical command.
<code>writePushPop</code> (developed in project 7)	<code>command</code> (C_PUSH or C_POP), <code>segment</code> (string), <code>index</code> (int)	—	Writes to the output file the assembly code that implements the given push or pop command.
<code>writeLabel</code>	<code>label</code> (string)	—	Writes assembly code that effects the <code>label</code> command.
<code>writeGoto</code>	<code>label</code> (string)	—	Writes assembly code that effects the <code>goto</code> command.
<code>writeIf</code>	<code>label</code> (string)	—	Writes assembly code that effects the <code>if-goto</code> command.
<code>writeFunction</code>	<code>functionName</code> (string) <code>nVars</code> (int)	—	Writes assembly code that effects the <code>function</code> command.
<code>writeCall</code>	<code>functionName</code> (string) <code>nArgs</code> (int)	—	Writes assembly code that effects the <code>call</code> command.
<code>writeReturn</code>	—	—	Writes assembly code that effects the <code>return</code> command.
<code>close</code> (developed in project 7)	—	—	Closes the output file / stream.

8.6 Project

In a nutshell, we have to extend the basic translator developed in chapter 7 with the ability to handle multiple .vm files and the ability to translate VM *branching* commands and VM *function* commands into Hack assembly code. For each parsed VM command, the VM translator has to generate assembly code that implements the command's semantics on the host Hack platform. The translation of the three *branching* commands into assembly is not difficult. The translation of the three *function* commands is more challenging and entails implementing the pseudocode listed in [figure 8.5](#), using the symbols described in [figure 8.6](#). We repeat the suggestion given in the previous chapter: Start by writing the required assembly code on paper. Draw RAM and global stack images, keep track of the stack pointer and the relevant memory segment pointers, and make sure that your paper-based assembly code successfully implements all the low-level actions associated with handling the call, function, and return commands.

Objective: Extend the basic VM translator built in project 7 into a full-scale VM translator, designed to handle multi-file programs written in the VM language.

This version of the VM translator assumes that the source VM code is error-free. Error checking, reporting, and handling can be added to later versions of the VM translator but are not part of project 8.

Contract: Complete the construction of a VM-to-Hack translator, conforming to VM Specification, Part II (section 8.4) and to the Standard VM Mapping on the Hack Platform, Part II (section 8.5.2). Use your translator to translate the supplied VM test programs, yielding corresponding programs written in the Hack assembly language. When executed on the supplied CPU emulator along with the supplied test scripts, the assembly programs generated by your translator should deliver the results mandated by the supplied compare files.

Resources: You will need two tools: the programming language in which you will implement your VM translator and the *CPU emulator* supplied in the Nand to Tetris software suite. Use the CPU emulator to execute and test the assembly code generated by your translator. If the generated code runs

correctly, we will assume that your VM translator performs as expected. This partial test of the translator will suffice for our purposes.

Another tool that comes in handy in this project is the supplied *VM emulator*. Use this program to execute the supplied test VM programs, and watch how the VM code effects the simulated states of the stack and the virtual memory segments. This can help understand the actions that the VM translator must eventually realize in assembly.

Since the full-scale VM translator is implemented by extending the VM translator built in project 7, you will also need the source code of the latter.

Testing and Implementation Stages

We recommend completing the implementation of the VM translator in two stages. First, implement the *branching* commands, and then the *function* commands. This will allow you to unit-test your implementation incrementally, using the supplied test programs.

Testing the Handling of the VM Commands `label`, `if`, `if-goto`:

- `BasicLoop`: Computes $1 + 2 + \dots + \text{argument}[0]$, and pushes the result onto the stack. Tests how the VM translator handles the `label` and `if-goto` commands.
- `FibonacciSeries`: Computes and stores in memory the first n elements of the Fibonacci series. A more rigorous test of handling the `label`, `goto`, and `if-goto` commands.

Testing the Handling of the VM Commands `call`, `function`, `return`:

Unlike what we did in project 7, we now expect the VM translator to handle multi-file programs. We remind the reader that by convention, the first function that starts running in a VM program is `Sys.init`. Normally, `Sys.init` is programmed to call the program's `Main.main` function. For the purpose of this project, though, we use the supplied `Sys.init` functions for setting the stage for the various tests that we wish to perform.

- `SimpleFunction`: Performs a simple calculation and returns the result. Tests how the VM translator handles the `function` and `return` commands. Since this test entails the handling of a single file consisting of a single function, no `Sys.init` test function is needed.

- **FibonacciElement:** This test program consists of two files: Main.vm contains a single Fibonacci function that returns recursively the n -th element of the Fibonacci series; Sys.vm contains a single Sys.init function that calls Main.fibonacci with $n = 4$ and then enters an infinite loop (recall that the VM translator generates bootstrap code that calls Sys.init). The resulting setting provides a rigorous test of the VM translator’s handling of multiple .vm files, the VM function-call-and-return commands, the bootstrap code, and most of the other VM commands. Since the test program consists of two .vm files, the entire folder must be translated, producing a single FibonacciElement.asm file.
- **StaticsTest:** This test program consists of three files: Class1.vm and Class2.vm contain functions that set and get the values of several static variables; Sys.vm contains a single Sys.init function that calls these functions. Since the program consists of several .vm files, the entire folder must be translated, producing a single StaticsTest.asm file.

Implementation Tips

Since project 8 is based on extending the basic VM translator developed in project 7, we advise making a backup copy of the source code of the latter (if you haven’t done it already).

Start by figuring out the assembly code that is necessary to realize the logic of the VM commands label, goto, and if-goto. Next, proceed to implement the methods writeLabel, writeGoto, and writeIf of the CodeWriter. Test your evolving VM translator by having it translate the supplied BasicLoop.vm and FibonacciSeries.vm programs.

Bootstrapping code: In order for any translated VM program to start running, it must include startup code that forces the VM implementation to start executing the program on the host platform. Further, for any VM code to operate properly, the VM implementation must store the base addresses of the stack and the virtual segments in the correct locations in the host RAM. The first three test programs in this project (BasicLoop, FibonacciSeries, SimpleFunction) assume that the startup code was not yet implemented, and include test scripts that effect the necessary initializations *manually*, meaning that at this development stage you don’t have to worry about it.

The last two test programs (`FibonaciiElement` and `StaticsTest`) assume that the startup code is already part of the VM implementation.

With that in mind, the constructor of the `CodeWriter` must be developed in two stages. The first version of your constructor must not generate any bootstrapping code (that is, ignore the constructor's API guideline beginning with the text "Writes the assembly instructions ..."). Use this version of your translator for unit-testing the programs `BasicLoop`, `FibonaciiSeries`, and `SimpleFunction`. The second and final version of your `CodeWriter` constructor must write the bootstrapping code, as specified in the constructor's API. This version should be used for unit-testing `FibonaciiElement` and `StaticsTest`.

The supplied test programs were carefully planned to test the specific features of each stage in your VM implementation. We recommend implementing your translator in the proposed order, and testing it using the appropriate test programs at each stage. Implementing a later stage before an early one may cause the test programs to fail.

A web-based version of project 8 is available at www.nand2tetris.org.

8.7 Perspective

The notions of *branching* and *function calling* are fundamental to all high-level languages. This means that somewhere down the translation path from a high-level language to binary code, someone must take care of handling the intricate housekeeping chores related to their implementation. In Java, C#, Python, and Jack, this burden falls on the virtual machine level. And if the VM architecture is *stack-based*, it lends itself nicely to the job, as we have seen throughout this chapter.

To appreciate the expressive power of our stack-based VM model, take a second look at the programs presented in this chapter. For example, [figures 8.1](#) and [8.4](#) present high-level programs and their VM translations. If you do some line counting, you will note that each line of high-level code generates an average of about four lines of compiled VM code. As it turns out, this 1:4 translation ratio is quite consistent when compiling Jack programs into VM code. Even without knowing much about the art of

compilation, one can appreciate the brevity and readability of the VM code generated by the compiler. For example, as we will see when we build the compiler, a high-level statement like `let y = Math.sqrt(x)` is translated into `push x, call Math.sqrt, pop y`. The two-tier compiler can get away with so little work since it counts on the VM implementation for handling the rest of the translation. If we had to translate high-level statements like `let y = Math.sqrt(x)` directly into Hack code, without having the benefit of an intermediate VM layer, the resulting code would be far less elegant, and more cryptic.

That said, it would also be more efficient. Let us not forget that the VM code must be realized in machine language—that’s what projects 7 and 8 are all about. Typically, the final machine code resulting from a two-tier translation process is longer and less efficient than that generated from direct translation. So, which is more desirable: a two-tier Java program that eventually generates one thousand machine instructions or an equivalent one-tier C++ program that generates seven hundred instructions? The pragmatic answer is that each programming language has its pros and cons, and each application has different operational requirements.

One of the great virtues of the two-tier model is that the intermediate VM code (e.g., Java’s bytecode) can be *managed*, for example, by programs that test whether it contains malicious code, programs that monitor the code for business process modeling, and so on. In general, for most applications, the benefits of managed code justify the performance degradation caused by the VM level. Yet for high-performance programs like operating systems and embedded applications, the need to generate tight and efficient code typically mandates using C/C++, compiled directly to machine language.

For compiler writers, an obvious advantage of using an explicit interim VM language is that it simplifies the tasks of writing and maintaining compilers. For example, the VM implementation developed in this chapter frees the compiler from the significant tasks of handling the low-level implementation of the function call-and-return protocol. In general, the intermediate VM layer decouples the daunting challenge of building a high-level-to-low-level compiler into two far simpler challenges: building a high-level-to-VM compiler and building a VM-to-low-level translator. Since the latter translator, also called the compiler’s *back end*, was already developed in projects 7 and 8, we can be content that about half of the overall

compilation challenge has already been accomplished. The other half—developing the compiler’s *front end*—will be taken up in chapters 10 and 11.

We end this chapter with a general observation about the virtue of separating abstraction from implementation—an ongoing theme in Nand to Tetris and a crucial systems-building principle that goes far beyond the context of program compilation. Recall that VM functions can access their memory segments using commands like push argument 2, pop local 1, and so on while having no idea *how* these values are represented, saved, and reinstated during run-time. The VM implementation takes care of all the gory details. This complete separation of abstraction and implementation implies that developers of compilers that generate VM code don’t have to worry about how the code they generate will end up running; they have enough problems of their own, as you will soon realize.

So cheer up! You are halfway through writing a two-tier compiler for a high-level, object-based, Java-like programming language. The next chapter is devoted to describing this language. This will set the stage for chapters 10 and 11, in which we’ll complete the compiler’s development. We begin seeing Tetris bricks falling at the end of the tunnel.

9 High-Level Language

High thoughts need a high language.

—Aristophanes (427–386 B.C.)

The assembly and VM languages presented so far in this book are low-level, meaning that they are intended for controlling machines, not for developing applications. In this chapter we present a high-level language, called Jack, designed to enable programmers to write high-level programs. Jack is a simple object-based language. It has the basic features and flavor of mainstream languages like Java and C++, with a simpler syntax and no support for inheritance. Despite this simplicity, Jack is a general-purpose language that can be used to create numerous applications. In particular, it lends itself nicely to interactive games like Tetris, Snake, Pong, Space Invaders, and similar classics.

The introduction of Jack marks the beginning of the end of our journey. In chapters 10 and 11 we will write a compiler that translates Jack programs into VM code, and in chapter 12 we will develop a simple operating system for the Jack/Hack platform. This will complete the computer’s construction. With that in mind, it’s important to say at the outset that the goal of this chapter is not to turn you into a Jack programmer. Neither do we claim that Jack is an important language outside the Nand to Tetris context. Rather, we view Jack as a necessary scaffold for chapters 10–12, in which we will build a compiler and an operating system that make Jack possible.

If you have any experience with a modern object-oriented programming language, you will immediately feel at home with Jack. Therefore, we begin the chapter with a few representative examples of Jack programs. All these programs can be compiled by the Jack compiler supplied in `nand2tetris/tools`.

The VM code produced by the compiler can then be executed as is on any VM implementation, including the supplied VM emulator. Alternatively, you can translate the compiled VM code further into machine language, using the VM translator built in chapters 7–8. The resulting assembly code can then be executed on the supplied CPU emulator or translated further into binary code and executed on the hardware platform built in chapters 1–5.

Jack is a simple language, and this simplicity has a purpose. First, you can learn (and unlearn) Jack in about one hour. Second, the Jack language was carefully designed to lend itself nicely to common compilation techniques. As a result, you can write an elegant *Jack compiler* with relative ease, as we will do in chapters 10 and 11. In other words, the deliberately simple structure of Jack is designed to help uncover the software infrastructure of modern languages like Java and C#. Rather than taking the compilers and run-time environments of these languages apart, we find it more instructive to build a compiler and a run-time environment ourselves, focusing on the most important ideas underlying their construction. This will be done later, in the last three chapters of the book. Presently, let's take Jack out of the box.

9.1 Examples

Jack is mostly self-explanatory. Therefore, we defer the language specification to the next section and start with examples. The first example is the inevitable *Hello World* program. The second example illustrates procedural programming and array processing. The third example illustrates how abstract data types can be implemented in the Jack language. The fourth example illustrates a linked list implementation using the language's object-handling capabilities.

Throughout the examples, we discuss briefly various object-oriented idioms and commonly used data structures. We assume that the reader has a basic familiarity with these subjects. If not, read on—you'll manage.

Example 1: Hello World: The program shown in [figure 9.1](#) illustrates several basic Jack features. By convention, when we execute a compiled

Jack program, execution always starts with the `Main.main` function. Thus, each Jack program must include at least one class, named `Main`, and this class must include at least one function, named `Main.main`. This convention is illustrated in [figure 9.1](#).

```
/** Prints "Hello World". File name: Main.jack */
class Main {
    function void main() {
        do Output.printString("Hello World");
        do Output.println(); // New line
        return;           // The return statement is mandatory
    }
}
```

Figure 9.1 *Hello World*, written in the Jack language.

Jack comes with a *standard class library* whose complete API is given in appendix 6. This software library, also known as the *Jack OS*, extends the basic language with various abstractions and services such as mathematical functions, string processing, memory management, graphics, and input/output functions. Two such OS functions are invoked by the Hello World program, affecting the program's output. The program also illustrates the comment formats supported by Jack.

Example 2: Procedural programming and array handling: Jack features typical statements for handling assignment and iteration. The program shown in [figure 9.2](#) illustrates these capabilities in the context of array processing.

```

/** Inputs a sequence of integers, and computes their average. */
class Main {
    function void main() {
        var Array a;      // Jack arrays are not typed
        var int length;
        var int i, sum;
        let i = 0;
        let sum = 0;
        let length = Keyboard.readInt("How many numbers? ");
        let a = Array.new(length); // Constructs the array
        while (i < length) {
            let a[i] = Keyboard.readInt("Enter a number: ");
            let sum = sum + a[i];
            let i = i + 1;
        }
        do Output.printString("The average is: ");
        do Output.printInt(sum / length);
        do Output.println();
        return;
    }
}

```

Figure 9.2 Typical procedural programming and simple array handling. Uses the services of the OS classes `Array`, `Keyboard`, and `Output`.

Most high-level programming languages feature array declaration as part of the basic syntax of the language. In Jack, we have opted for treating arrays as instances of an `Array` class, which is part of the OS that extends the basic language. This was done for pragmatic reasons, as it simplifies the construction of Jack compilers.

Example 3: Abstract data types: Every programming language features a fixed set of primitive data types, of which Jack has three: `int`, `char`, and `boolean`. In object-based languages, programmers can introduce new types by creating classes that represent abstract data types as needed. For example, suppose we wish to endow Jack with the ability to handle rational numbers like $2/3$ and $314159/100000$ without loss of precision. This can be done by developing a standalone Jack class designed to create and manipulate fraction objects of the form x/y , where x and y are integers. This class can then provide a fraction abstraction to any Jack program that needs to represent and manipulate rational numbers. We now turn to describe how

a Fraction class can be used and developed. This example illustrates typical multi-class, object-based programming in Jack.

Using classes: Figure 9.3a lists a class skeleton (a set of method signatures) that specifies some of the services that one can reasonably expect to get from a fraction abstraction. Such a specification is often called an *Application Program Interface*. The client code at the bottom of the figure illustrates how this API can be used for creating and manipulating fraction objects.

```
/** Represents the Fraction type and related operations (class skeleton) */
class Fraction {
    /** Constructs a (reduced) fraction from x and y */
    constructor Fraction new(int x, int y)

    /** Returns the numerator of this fraction */
    method int getNumerator()

    /** Returns the denominator of this fraction */
    method int getDenominator()

    /** Returns the sum of this fraction and the other one */
    method Fraction plus(Fraction other)

    /** Prints this fraction in the format x/y */
    method void print()

    /** Disposes this fraction */
    method void dispose() {
        // More fraction-related methods:
        // minus, times, div, invert, etc.
    }
}

// Computes and prints the sum of 2/3 and 1/5
class Main {
    function void main() {
        // Creates 3 fraction variables (pointers to Fraction objects)
        var Fraction a, b, c;
        let a = Fraction.new(4,6); // a = 2/3
        let b = Fraction.new(1,5); // b = 1/5
        // Adds up the two fractions, and prints the result
        let c = a.plus(b); // c = a + b
        do c.print(); // Should print "13/15"
        return;
    }
}
```

Figure 9.3a Fraction API (top) and sample Jack class that uses it for creating and manipulating Fraction objects.

Figure 9.3a illustrates an important software engineering principle: users of an abstraction (like Fraction) don't have to know anything about its implementation. All they need is the class *interface*, also known as API. The API informs what functionality the class offers and how to use this functionality. That's all the client needs to know.

Implementing classes: So far, we have seen only the client perspective of the Fraction class—the view from which Fraction is used as a black box

abstraction. Figure 9.3b lists one possible implementation of this abstraction.

```
/** Represents the Fraction type and related operations. */
class Fraction {
    // Each Fraction object has a numerator and a denominator
    field int numerator, denominator;
    /* Constructs a (reduced) fraction from x and y */
    constructor Fraction new(int x, int y) {
        let numerator = x;
        let denominator = y;
        do reduce(); // Reduces this fraction
        return this; // Returns a reference to the new object
    }
    // Reduces this fraction
    method void reduce() {
        var int g;
        let g = Fraction.gcd(numerator, denominator);
        if (g > 1) {
            let numerator = numerator / g;
            let denominator = denominator / g;
        }
        return;
    }
    // Computes the greatest common divisor of two given integers
    function int gcd(int a, int b) {
        // Applies Euclid's algorithm
        var int r;
        while (~(b = 0)) {
            let r = a - (b * (a / b)); // r = remainder
            let a = b;
            let b = r;
        }
        return a;
    }
    // The Fraction class declaration continues on the top right
}

/** Accessors */
method int getNumerator() {
    return numerator;
}
method int getDenominator() {
    return denominator;
}
/** Returns the sum of this fraction and the other one */
method Fraction plus(Fraction other) {
    var int sum;
    let sum = (numerator * other.getDenominator()) +
              (other.getNumerator() * denominator);
    return Fraction.new(sum, denominator +
                        other.getDenominator());
}
/** Prints this fraction in the format x/y */
method void print() {
    do Output.printInt(numerator);
    do Output.printString("/");
    do Output.printInt(denominator);
    return;
}
/** Disposes this fraction */
method void dispose() {
    // Frees the memory held by this object
    do Memory.deAlloc(this);
    return;
}
// More fraction-related methods can come here:
// minus, times, div, invert, etc.
} // End of the Fraction class declaration.
```

Figure 9.3b A Jack implementation of the Fraction abstraction.

The Fraction class illustrates several key features of object-based programming in Jack. *Fields* specify object properties (also called *member variables*). *Constructors* are subroutines that create new objects, and *methods* are subroutines that operate on the current object (referred to using the keyword *this*). *Functions* are class-level subroutines (also called *static methods*) that operate on no particular object. The Fraction class also demonstrates all the statement types available in the Jack language: *let*, *do*, *if*, *while*, and *return*. The Fraction class is of course just one example of the unlimited number of classes that can be created in Jack to support any conceivable programming objective.

Example 4: Linked list implementation: The data structure *list* is defined recursively as a value, followed by a list. The value *null*—the definition’s base case—is also considered a list. Figure 9.4 shows a possible Jack implementation of a list of integers. This example illustrates how Jack can be used for realizing a major data structure, widely used in computer science.

```


/* Represents a list of integers. */
class List {
    field int data;      // A list consists of an int value,
    field List next;     // followed by a List

    /* Creates a list whose head is car and whose tail is cdr */
    // These identifiers are used in memory of the Lisp programming language
    constructor List new(int car, List cdr) {
        let data = car;
        let next = cdr;
        return this;
    }

    /* Accessors */
    method int getData() {return data;}
    method List getNext() {return next;}

    /* Prints the elements of this list */
    method void print() {
        // Initializes a pointer to the first element of this list
        var List current;
        let current = this;
        // Iterates through the list.
        while (~(current = null)) {
            do Output.printInt(current.getData());
            do Output.printChar(32); // Prints a space
            let current = current.getNext();
        }
        return;
    }

    /* Disposes this List */
    method void dispose() {
        // Disposes the tail of this list, recursively
        if (~(next = null)) {
            do next.dispose();
        }
        // Uses an OS routine to free the memory
        // held by this object.
        do Memory.deAlloc(this);
        return;
    }

    // More list-related methods can come here
} // End of the List class declaration.


```

```


// Client code example:
// Creates, prints, and disposes the list (2,3,5),
// which is shorthand for the list (2,(3,(5,null))). 
// (This code can appear in any Jack class):
...
var List v;
let v = List.new(5,null);
let v = List.new(2,List.new(3,v));
do v.print();           // Prints 2 3 5
do v.dispose();         // Disposes the list
...


```

Figure 9.4 Linked list implementation in Jack (left and top right) and sample usage (bottom right).

The operating system: Jack programs make extensive use of the Jack operating system, which will be discussed and developed in chapter 12. For now, suffice it to say that Jack programs use the OS services abstractly, without paying attention to their underlying implementation. Jack programs can use the OS services directly—there is no need to include or import any external code.

The OS consists of eight classes, summarized in figure 9.5 and documented in appendix 6.

<i>OS class</i>	<i>Services</i>
Math	Common mathematical operations: <code>max(int,int)</code> , <code>sqrt(int)</code> , ...
String	Represents strings and related operations: <code>length()</code> , <code>charAt(int)</code> , ...
Array	Represents arrays and related operations: <code>new(int)</code> , <code>dispose()</code>
Output	Facilitates text output to the screen: <code>printString(String)</code> , <code>printInt(int)</code> , <code>println()</code> , ...
Screen	Facilitates graphics output to the screen : <code>setColor(boolean)</code> , <code>drawPixel(int,int)</code> , <code>drawLine(int,int,int)</code> , ...
Keyboard	Facilitates input from the keyboard: <code>readLine(String)</code> , <code>readInt(String)</code> , ...
Memory	Facilitates access to the host RAM: <code>peek(int)</code> , <code>poke(int,int)</code> , <code>alloc(int)</code> , <code>deAlloc(Array)</code>
Sys	Facilitates execution-related services: <code>halt()</code> , <code>wait(int)</code> , ...

Figure 9.5 Operating system services (summary). The complete OS API is given in appendix 6.

9.2 The Jack Language Specification

This section can be read once and then used as a technical reference, to be consulted as needed.

9.2.1 Syntactic Elements

A Jack program is a sequence of tokens, separated by an arbitrary amount of white space and comments. Tokens can be symbols, reserved words, constants, and identifiers, as listed in [figure 9.6](#).

White space and comments	<p>Space characters, newline characters, and comments are ignored.</p> <p>The following comment formats are supported:</p> <pre>// Comment to end of line /* Comment until closing */ /** Aimed at software tools that extract API documentation. */</pre>												
Symbols	<ul style="list-style-type: none"> () Used for grouping arithmetic expressions and for enclosing argument-lists (in subroutine calls) and parameter-lists (in subroutine declarations) [] Used for array indexing { } Used for grouping program units and statements , : = Assignment and comparison operator . Class membership + - * / & ~ < > Operators 												
Reserved words	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">class, constructor, method, function</td> <td>Program components</td> </tr> <tr> <td>int, boolean, char, void</td> <td>Primitive types</td> </tr> <tr> <td>var, static, field</td> <td>Variable declarations</td> </tr> <tr> <td>let, do, if, else, while, return</td> <td>Statements</td> </tr> <tr> <td>true, false, null</td> <td>Constant values</td> </tr> <tr> <td>this</td> <td>Object reference</td> </tr> </table>	class, constructor, method, function	Program components	int, boolean, char, void	Primitive types	var, static, field	Variable declarations	let, do, if, else, while, return	Statements	true, false, null	Constant values	this	Object reference
class, constructor, method, function	Program components												
int, boolean, char, void	Primitive types												
var, static, field	Variable declarations												
let, do, if, else, while, return	Statements												
true, false, null	Constant values												
this	Object reference												
Constants	<ul style="list-style-type: none"> • <i>Integer constants</i> are values in the range 0 to 32767. Negative integers are not constants but rather expressions consisting of a unary minus operator applied to an integer constant. The resulting valid range of values is –32768 to 32767 (the former can be obtained using the expression <code>-32767-1</code>). • <i>String constants</i> are enclosed within double quote ("") characters and may contain any character except newline or double quote. These characters are supplied by the OS functions <code>String.newLine()</code> and <code>String.doubleQuote()</code>. • <i>Boolean constants</i> are <code>true</code> and <code>false</code>. • The <code>null</code> constant signifies a null reference. 												
Identifiers	<p>Identifiers are composed from arbitrarily long sequences of letters (A-Z, a-z), digits (0-9), and <code>_</code>. The first character must be a letter or <code>_</code>.</p> <p>The language is case sensitive: <code>x</code> and <code>X</code> are treated as different identifiers.</p>												

Figure 9.6 The syntax elements of the Jack language.

9.2.2 Program Structure

A Jack program is a collection of one or more classes stored in the same folder. One class must be named `Main`, and this class must have a function named `main`. The execution of a compiled Jack program always starts with the `Main.main` function.

The basic programming unit in Jack is a *class*. Each class *Xxx* resides in a separate file named *Xxx.jack* and is compiled separately. By convention, class names begin with an uppercase letter. The file name must be identical

to the class name, including capitalization. The class declaration has the following structure:

```
class name {  
    field variable declarations    // Must precede the subroutine declarations  
    static variable declarations   // Must precede the subroutine declarations  
    subroutine declarations        // Constructor, method and function declarations, in any order  
}
```

Each class declaration specifies a name through which the class services can be globally accessed. Next comes a sequence of zero or more field declarations and zero or more static variable declarations. Next comes a sequence of one or more subroutine declarations, each defining a *method*, a *function*, or a *constructor*.

Methods are designed to operate on the current object. *Functions* are class-level *static methods* that are not associated with any particular object. *Constructors* create and return new objects of the class type. A subroutine declaration has the following structure:

```
subroutine type name (parameter-list) {  
    local variable declarations  
    statements  
}
```

where *subroutine* is either constructor, method, or function. Each subroutine has a *name* through which it can be accessed and a *type* specifying the data type of the value returned by the subroutine. If the subroutine is designed to return no value, its type is declared void. The *parameter-list* is a comma-separated list of *<type identifier>* pairs, for example, (int x, boolean sign, Fraction g).

If the subroutine is a *method* or a *function*, its return type can be any of the primitive data types supported by the language (int, char, or boolean), any of the class types supplied by the standard class library (String or Array), or any of the types realized by other classes in the program (e.g., Fraction or List). If the subroutine is a *constructor*, it may have an arbitrary name, but its type must be the name of the class to which it belongs. A class can have 0,

1, or more constructors. By convention, one of the constructors is named `new`.

Following its interface specification, the subroutine declaration contains a sequence of zero or more local variable declarations (`var` statements) and then a sequence of one or more statements. Each subroutine must end with the statement `return expression`. In the case of a void subroutine, when there is nothing to return, the subroutine must end with the statement `return` (which can be viewed as a shorthand of `return void`, where `void` is a constant representing “nothing”). Constructors must terminate with the statement `return this`. This action returns the memory address of the newly constructed object, denoted `this` (Java constructors do the same, implicitly).

9.2.3 Data Types

The data type of a variable is either *primitive* (`int`, `char`, or `boolean`), or *ClassName*, where *ClassName* is either `String`, `Array`, or the name of a class residing in the program folder.

Primitive types: Jack features three primitive data types:

`int`: two’s-complement 16-bit integer

`char`: nonnegative, 16-bit integer

`boolean`: true or false

Each one of the three primitive data types `int`, `char`, and `boolean` is represented internally as a 16-bit value. The language is weakly typed: a value of any type can be assigned to a variable of any type without casting.

Arrays: Arrays are declared using the OS class `Array`. Array elements are accessed using the typical `arr[i]` notation, where the index of the first element is 0. A multidimensional array may be obtained by creating an array of arrays. The array elements are not typed, and different elements in the same array may have different types. The declaration of an array creates a reference, while the array proper is constructed by executing the constructor call `Array.new (arrayLength)`. For an example of working with arrays, see [figure 9.2](#).

Object types: A Jack class that contains at least one method defines an object type. As typical in object-oriented programming, object creation is a two-step affair. Here is an example:

```
// This client code example uses the Car and Employee classes, whose code is not shown here.  
// The Car class has two fields: model (a String) and licensePlate (a String).  
// The Employee class has two fields: name (a String) and car (a Car).  
...  
// Declares a Car object and two Employee objects (three pointer variables):  
var Car c;  
var Employee emp1, emp2;  
...  
// Constructs a new car:  
let c = Car.new("Aston Martin", "007"); // Sets c to the base address of a memory block  
                                         // containing the new car's data.  
// Constructs a new employee, and assigns a car to it:  
let emp1 = Employee.new("Bond", c);  
...  
// Creates an alias of Bond:  
let emp2 = emp1; // Only the reference (address) is copied, no new object is constructed.  
// We now have two Employee pointers referring to the same object.
```

Strings: Strings are instances of the OS class String, which implements strings as arrays of char values. The Jack compiler recognizes the syntax "foo" and treats it as a String object. The contents of a String object can be accessed using `charAt(index)` and other methods documented in the String class API (see appendix 6). Here is an example:

```
var String s; // An object variable  
var char c;   // A primitive variable  
...  
let s = "Hello World"; // Sets s to the String object "Hello World"  
let c = s.charAt(6); // Sets c to 87, the integer character code of 'W'
```

The statement `let s = "Hello World"` is equivalent to the statement `let s = String.new(11)`, followed by the eleven method calls `do s.appendChar(72), ..., do s.appendChar(100)`, where the argument of `appendChar` is the character's integer

code. In fact, that's exactly how the compiler handles the translation of `let s = "Hello World"`. Note that the single character idiom, for example, '`H`', is not supported by the Jack language. The only way to represent a character is to use its integer character code or a `charAt` method call. The Hack character set is documented in appendix 5.

Type conversions: Jack is weakly typed: the language specification does not define what happens when a value of one type is assigned to a variable of a different type. The decisions of whether to allow such casting operations, and how to handle them, are left to specific Jack compilers. This under-specification is intentional, allowing the construction of minimal compilers that ignore typing issues. Having said that, all Jack compilers are expected to support, and automatically perform, the following assignments.

- A character value can be assigned to an integer variable, and vice versa, according to the Jack character set specification (appendix 5). Example:

```
var char c;
let c = 33; // 'A'

// Equivalently:
var String s;
let s = "A";
let c = s.charAt(0);
```

- An integer can be assigned to a reference variable (of any object type), in which case it is interpreted as a memory address. Example:

```
var Array arr;      // Creates a pointer variable
let arr = 5000;     // Sets arr to 5000
let arr[100] = 17;  // Sets the contents of memory address 5100 to 17
```

- An object variable may be assigned to an `Array` variable, and vice versa. This allows accessing the object fields as array elements, and vice versa. Example:

```

// Creates the array [2,5]:
var Array arr;
let arr = Array.new(2);
let arr[0] = 2;
let arr[1] = 5;

// Creates the Fraction 2/5:
var Fraction x;
let x = arr; // sets x to the base address of the memory block
              // representing the array [2,5]

do Output.putInt(x.getNumerator()) // prints "2"
do x.print()                   // prints "2/5"

```

9.2.4 Variables

Jack features four kinds of variables. *Static variables* are defined at the class level and can be accessed by all the class subroutines. *Field variables*, also defined at the class level, are used to represent the properties of individual objects and can be accessed by all the class constructors and methods. *Local variables* are used by subroutines for local computations, and *parameter variables* represent the arguments that were passed to the subroutine by the caller. Local and parameter values are created just before the subroutine starts executing and are recycled when the subroutine returns. [Figure 9.7](#) gives the details. The *scope* of a variable is the region in the program in which the variable is recognized.

Kind	Description	Declared in	Scope
Class variables	<code>static type varName1, varName2, ... ;</code> One copy of each static variable exists, and this copy is shared by all the class subroutines (like <i>private static variables</i> in Java)	Class declaration	The class in which they are declared
Field variables	<code>field type varName1, varName2, ... ;</code> Every object (instance of the class) has a private copy of the field variables (like <i>member variables</i> in Java)	Class declaration	The class in which they are declared
Local variables	<code>var type varName1, varName2, ... ;</code> Created when the subroutine starts running and disposed when the subroutine returns.	Subroutine declaration	The subroutine in which they are declared
Parameter variables	Represent the arguments passed to the subroutine. Treated like local variables whose values are initialized by the subroutine caller.	Subroutine declaration	The subroutine in which they are declared

[Figure 9.7](#) Variable kinds in the Jack language. Throughout the table, *subroutine* refers to either a *function*, a *method*, or a *constructor*.

Variable initialization: Static variables are not initialized, and it is up to the programmer to write code that initializes them before using them. Field variables are not initialized; it is expected that they will be initialized by the class constructor, or constructors. Local variables are not initialized, and it is up to the programmer to initialize them. Parameter variables are initialized to the values of the arguments passed by the caller.

Variable visibility: Static and field variables cannot be accessed directly from outside the class in which they are defined. They can be accessed only through accessor and mutator methods, as facilitated by the class designer.

9.2.5 Statements

The Jack language features five statements, as seen in [figure 9.8](#).

Statement	Syntax	Description
let	<code>let varName = expression;</code> or: <code>let varName[expression1] = expression2;</code>	An assignment operation. The variable kind may be <i>static</i> , <i>local</i> , <i>field</i> , or <i>parameter</i> .
if	<code>if (expression) { statements1; } else { statements2; }</code>	Typical <i>if</i> statement, with an optional <i>else</i> clause. The curly brackets are mandatory, even if <i>statements</i> is a single statement.
while	<code>while (expression) { statements; }</code>	Typical <i>while</i> statement. The curly brackets are mandatory, even if <i>statements</i> is a single statement.
do	<code>do function-or-method-call;</code>	Used to call a function or a method for its effect, ignoring the returned value, if any.
return	<code>return expression;</code> or <code>return;</code>	Used to return a value from a subroutine. The second form must be used by <i>void</i> subroutines. Constructors must return the value <i>this</i> .

Figure 9.8 Statements in the Jack language.

9.2.6 Expressions

A Jack expression is one of the following:

- A *constant*
- A *variable name* in scope. The variable may be *static*, *field*, *local*, or *parameter*
- The *this* keyword, denoting the current object (cannot be used in functions)
- An *array element* using the syntax *arr[expression]*, where *arr* is a variable name of type *Array* in scope
- A *subroutine call* that returns a non-void type
- An expression prefixed by one of the unary operators - or ~:
 - *expression*: arithmetic negation
 - *expression*: Boolean negation (bitwise for integers)
- An expression of the form *expression op expression* where *op* is one of the following binary operators:
 - + - * /: integer arithmetic operators
 - & |: Boolean And and Boolean Or (bitwise for integers) operators
 - < > =: comparison operators
- (*expression*): an expression in parentheses

Operator priority and order of evaluation: Operator priority is *not* defined by the language, except that expressions in parentheses are evaluated first. Thus the value of the expression $2 + 3 * 4$ is unpredictable, whereas $2 + (3 * 4)$ is guaranteed to evaluate to 14. The Jack compiler supplied in Nand to Tetris (as well as the compiler that we'll develop in chapters 10–11) evaluates expressions left to right, so the expression $2 + 3 * 4$ evaluates to 20. Once again, if you wish to get the algebraically correct result, use $2 + (3 * 4)$.

The need to use parentheses for enforcing operator priority makes Jack expressions a bit cumbersome. This lack of formal operator priority is intentional, though, as it simplifies the implementation of Jack compilers. Different Jack compilers are welcome to specify an operator priority and add it to the language documentation, if so desired.

9.2.7 Subroutine Calls