8

Virtual Machine II: Program 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 ended with the construction of a basic VM implementation over the Hack platform. In this chapter we continue to develop the VM abstraction, language, and implementation. In particular, we design stack-based mechanisms for handling nested subroutine calls (procedures, functions, methods) of procedural or object-oriented languages. As the chapter progresses, we extend the previously built basic VM implementation, ending with a full-scale VM translator. This translator will serve as the backend of the compiler that we will build in chapters 10 and 11, following the introduction of a high-level object-based language in chapter 9.

In any Great Gems in Computer Science contest, stack processing will be a strong finalist. The previous chapter showed how arithmetic and Boolean expressions can be calculated by elementary stack operations. This chapter goes on to show how this remarkably simple data structure can also support remarkably complex tasks like nested subroutine calling, parameter passing, recursion, and the associated memory allocation techniques. Most programmers tend to take these capabilities for granted, expecting the compiler to deliver them, one way or another. We are now in a position to open this black box and see how these fundamental programming mechanisms are actually implemented by a stack-based virtual machine.

8.1 Background

High-level languages allow writing programs in high-level terms. For example, $x = -b + \sqrt{b^2 - 4 \cdot a \cdot c}$ can be expressed as x=-b+sqrt(power(b,2)-4*a*c), which is almost as descriptive as the real thing. High-level languages support this power of expression through three conventions. First, one is allowed to freely define high-level operations like sqrt and power, as needed. Second, one is allowed to freely use (call) these subroutines as if they were elementary operations like + and *. Third, one is allowed to assume that each called subroutine will get executed—somehow—and that following its termination control will return—somehow—to the next command in one's code. Flow of control commands take this freedom one step further, allowing writing, say, if ~(a=0) {x=(-b+sqrt(power(b,2)-4*a*c))/ (2*a)} else {x=-c/b}.

The ability to compose such expressions freely permits us to write abstract code, closer to the world of algorithmic thought than to that of machine execution. Of course the more abstract the high level, the more work we have to do at the low level. In particular, the low level must manage the delicate interplay between the calling subroutine (the caller) and the called subroutines—the program units that implement system- and user-defined operations like sqrt and power. For each subroutine call during runtime, the low level must handle the following details behind the scene:

- Passing parameters from the caller to the called subroutine
- Saving the state of the caller before switching to execute the called subroutine
- Allocating space for the local variables of the called subroutine
- Jumping to execute the called subroutine
- Returning values from the called subroutine back to the caller
- Recycling the memory space occupied by the called subroutine, when it returns
- Reinstating the state of the caller
- Jumping to execute the code of the caller immediately following the spot where we left it

Taking care of these housekeeping chores is a major headache, and high-level programmers are fortunate that the compiler relieves them from this duty. So how does the compiler do it? Well, if we choose to base our low level implementation on a stack machine, the job will be surprisingly manageable. In fact, the stack structure lends itself perfectly well to supporting all the housekeeping tasks mentioned above.

With that in mind, the remainder of this section describes how program flow and subroutine calling commands can be implemented on a stack machine. We begin with the implementation of program flow commands, which is rather simple and requires no memory management, and continue to describe the more challenging implementation of subroutine calling commands.

8.1.1 Program Flow

The default execution of computer programs is linear, one command after the other. This sequential flow is occasionally broken by branching commands, for example, embarking on a new iteration in a loop. In low-level programming, the branching logic is accomplished by instructing the machine to continue execution at some destination in the program other than the next instruction, using a goto destination command. The destination specification can take several forms, the most primitive being the physical address of the instruction that should be executed next. A slightly more abstract redirection command is established by describing the jump destination using a symbolic label. This variation requires that the language be equipped with some labeling directive, designed to assign symbols to selected points in the code.

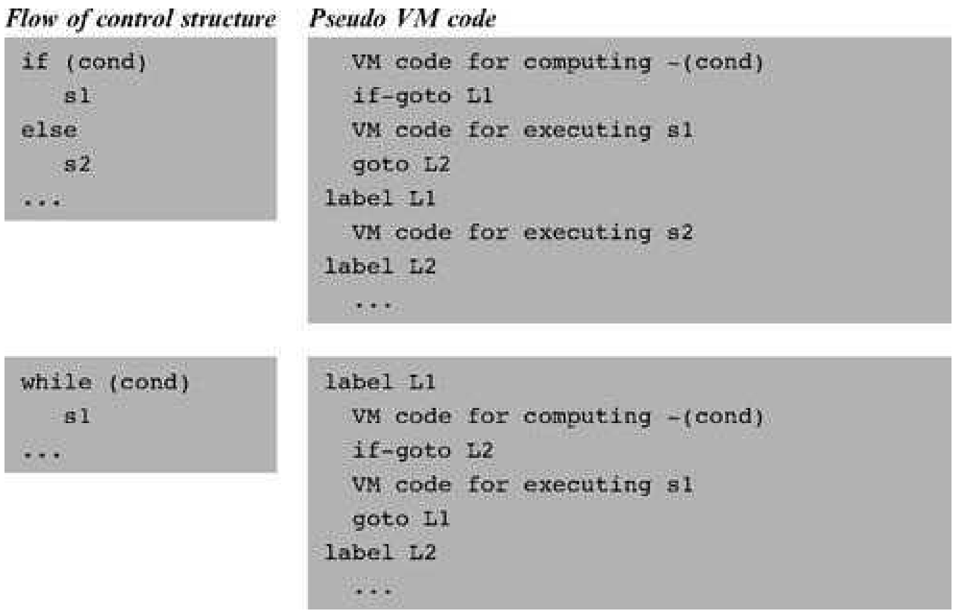
This basic *goto* mechanism can easily be altered to effect conditional branching as well. For example, an if-goto destination command can instruct the machine to take the jump only if a given Boolean condition is true; if the condition is false, the regular program flow should continue, executing the next command in the code. How should we introduce the Boolean condition into the language? In a stack machine paradigm, the most natural approach is conditioning the jump on the value of the stack's topmost element: if it's not zero, jump to the specified destination; otherwise, execute the next command in the program.

In chapter 7 we saw how primitive VM operations can be used to compute any Boolean expression, leaving its truth-value at the stack's topmost element. This power of expression, combined with the goto and if-goto commands just described, can be used to express any flow of control structure found in any programming language. Two typical examples appear in figure 8.1.

The low-level implementation of the VM commands label, goto label, and if-goto label is straightforward. All programming languages, including the "lowest" ones, feature branching commands of some sort. For example, if our low-level implementation is based on translating the VM commands into assembly code, all we have to do is reexpress these goto commands using the branching logic of the assembly language.

8.1.2 Subroutine Calling

Each programming language is characterized by a fixed set of built-in commands. The key abstraction mechanism provided by modern languages is the freedom to extend this basic repertoire with high-level, programmer-defined operations. In procedural languages, the high-level operations are called subroutines, procedures, or functions, and in object-oriented languages they are usually called methods. Throughout this chapter, all these high-level program units are referred to as subroutines.



Flow of control structure	*Pseudo VM code*

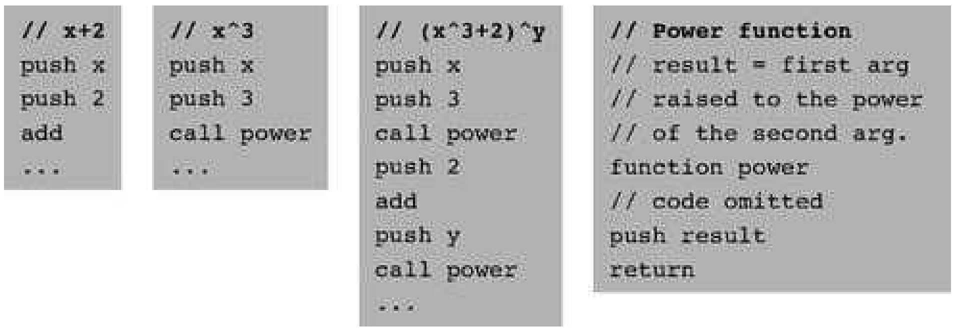
`if (cond)`
`   s1`
`else`
`   s2`
`...`
	`  VM code for computing ~(cond)`
	`  if-goto L1`
	`  VM code for executing s1`
	`  goto L2`
	`label L1`
	`  VM code for executing s2`
	`label L2`
	`  ...`

`while (cond)`
`   s1`
`...`
	`label L1`
	`  VM code for computing ~(cond)`
	`  if-goto L2`
	`  VM code for executing s1`
	`  goto L1`
	`label L2`
	`  ...`

Figure 8.1 Low-level flow of control using goto commands.

In well-designed programming languages, the use of a high-level operation (implemented by a subroutine) has the same “look and feel” as that of built-in commands. For example, consider the functions add and raise to a power. Most languages feature the former as a built-in operation, while the latter may be written as a subroutine. In spite of these different implementations, both functions should ideally look alike from the caller’s perspective. This would allow the caller to weave the two operations together naturally, yielding consistent and readable code. A stack language implementation of this principle is illustrated in figure 8.2.

We see that the only difference between invoking a built-in command and calling a user-defined subroutine is the keyword call preceding the latter. Everything else is exactly the same: Both operations require the caller to set up their arguments, both operations are expected to remove their arguments from the stack, and both operations are expected to return a value which becomes the topmost stack element. The uniformity of this protocol has a subtle elegance that, we hope, is not lost on the reader.



```
// x+2
push x
push 2
add
...
```

```
// x^3
push x
push 3
call power
...
```

```
// (x^3+2)^y
push x
push 3
call power
push 2
add
push y
call power
...
```

```
// Power function
// result = first arg
// raised to the power
// of the second arg.
function power
// code omitted
push result
return
```

Figure 8.2 Subroutine calling. Elementary commands (like add) and high-level operations (like power) have the same look and feel in terms of argument handling and return values.

Subroutines like power usually use local variables for temporary storage. These local variables must be represented in memory during the subroutine's lifetime, namely, from the point the subroutine starts executing until a return command is encountered. At this point, the memory space occupied by the subroutine's local variables can be freed. This scheme is complicated by allowing subroutines to be arbitrarily nested: One subroutine may call another subroutine, which may then call another one, and so on. Further, subroutines should be allowed to call themselves recursively; each recursive call must be executed independently of all the other calls and maintain its own set of local and argument variables. How can we implement this nesting mechanism and the memory management tasks implied by it?

The property that makes this housekeeping task tractable is the hierarchical nature of the call-and-return logic. Although the subroutine calling chain may be arbitrarily deep as well as recursive, at any given point in time only one subroutine executes at the top of the chain, while all the other subroutines down the calling chain are waiting for it to terminate. This *Last-In-First-Out* (LIFO) processing model lends itself perfectly well to a stack data structure, which is also LIFO. When subroutine xxx calls subroutine yyy, we can push (save) xxx's world on the stack and branch to execute yyy. When yyy returns, we can pop (reinstate) xxx's world off the stack, and continue executing xxx as if nothing happened. This execution model is illustrated in figure 8.3.

We use the term *frame* to refer, conceptually, to the subroutine's local variables, the arguments on which it operates, its working stack, and the other memory segments that support its operation. In chapter 7, the term stack referred to the working memory that supports operations like pop, push, add, and so on. From now on, when we say *stack* we mean *global stack*—the memory area containing the frames of the current subroutine and all the subroutines waiting for it to return. These two stack notions are closely related, since the working stack of the current subroutine is located at the very tip of the global stack.

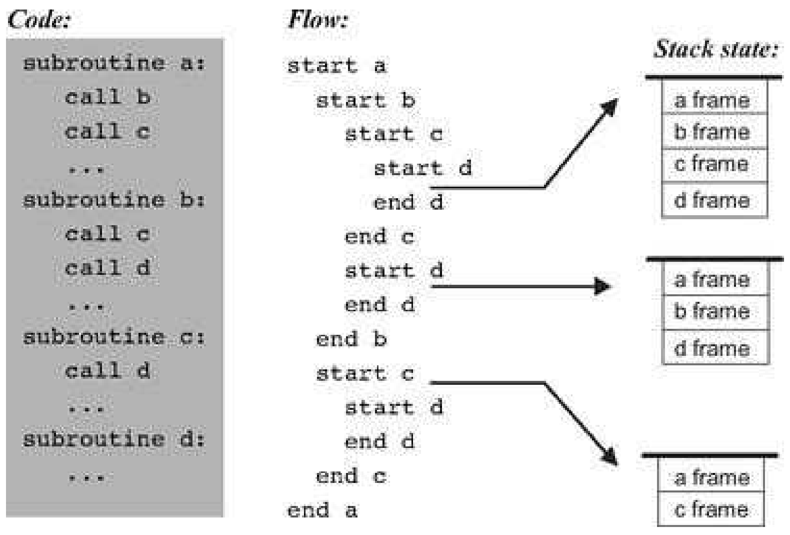


Figure 8.3 Subroutine calls and stack states associated with three representative points in the program's life cycle. All the layers in the stack are waiting for the current layer to complete its execution, at which point the stack becomes shorter and execution resumes at the level just below the current layer. (Following convention, the stack is drawn as if it grows downward.)

To recap, the low-level implementation of the call xxx operation entails saving the caller's frame on the stack, allocating stack space for the local variables of the called subroutine (xxx), then jumping to execute its code. This last "mega jump" is not hard to implement. Since the name of the target subroutine is specified in the call command, the implementation can resolve the symbolic name to a memory address, then jump to execute the code starting at that address. Returning from the called subroutine via a return command is trickier, since the command specifies no return address. Indeed, the caller's anonymity is inherent in the very notion of a subroutine call. For example, subroutines like power(x,y) or sqrt(x) are designed to serve any caller, implying that the return address cannot be part of their code. Instead, a return command should be interpreted as follows: Redirect the program's execution to the command following the call command that called the current subroutine, wherever this command may be. The memory location of this command is called *return address*.

A glance at figure 8.3 suggests a stack-based solution to implementing this return logic. When we encounter a call xxx operation, we know exactly what the return address should be: It's the address of the next command in the caller's code. Thus, we can push this return address on the stack and proceed to execute the code of the called subroutine. When we later encounter a return command, we can pop the saved return address and simply goto it. In other words, the return address can also be placed in the caller's frame.

8.2 VM Specification, Part II

This section extends the basic VM specification from chapter 7 with program flow and *function calling* commands, thereby completing the overall VM specification.

8.2.1 Program Flow Commands

The VM language features three program flow commands:

■ label *label* This command labels the current location in the function’s code.

Only labeled locations can be jumped to from other parts of the program. The scope of the label is the function in which it is defined. The label is an arbitrary string composed of any sequence of letters, digits, underscore (_), dot (.), and colon (:) that does not begin with a digit.

■ goto *label* This command effects an unconditional goto operation, causing execution to continue from the location marked by the label. The jump destination must be located in the same function.

■ if-goto *label* This command 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 jump destination must be located in the same function.

8.2.2 Function Calling Commands

Different high-level languages have different names for program units including functions, procedures, methods, and subroutines. In our overall compilation model (elaborated in chapters 10-11), each such high-level program unit is translated into a low-level program unit called *VM function*, or simply function.

A function has a symbolic name that is used globally to call it. The function name is an arbitrary string composed of any sequence of letters, digits, underscore (_), dot (.), and colon (:) that does not begin with a digit. (We expect that a method bar in class Foo in some high-level language will be translated by the compiler to a VM function named Foo.bar). The scope of the function name is global: All functions in all files are seen by each other and may call each other using the function name.

The VM language features three function-related commands:

■ function *f n* Here starts the code of a function named f that has n local variables;

■ call *f m* Call function *f*, stating that *m* arguments have already been pushed onto the stack by the caller;

■ return Return to the calling function.

8.2.3 The Function Calling Protocol

The events of calling a function and returning from a function can be viewed from two different perspectives: that of the calling function and that of the called function.

The calling function view:

- Before calling the function, the caller must push as many arguments as necessary onto the stack;

- Next, the caller invokes the function using the call command;

- After the called function returns, the arguments that the caller has pushed before the call have disappeared from the stack, and a return value (that always exists) appears at the top of the stack;

- After the called function returns, the caller's memory segments argument, local, static, this, that, and pointer are the same as before the call, and the temp segment is undefined.

The called function view:

- When the called function starts executing, its argument segment has been initialized with actual argument values passed by the caller and its local variables segment has been allocated and initialized to zeros. The static segment that the called function sees has been set to the static segment of the VM file to which it belongs, and the working stack that it sees is empty. The segments this, that, pointer, and temp are undefined upon entry.

- Before returning, the called function must push a value onto the stack.

To repeat an observation made in the previous chapter, we see that when a VM function starts running (or resumes its previous execution), it assumes that it is surrounded by a private world, all of its own, consisting of its memory segments and stack, waiting to be manipulated by its commands. The agent responsible for building this virtual worldview for every VM function is the VM implementation, as we elaborate in section 8.3.

8.2.4 Initialization

A VM program is a collection of related VM functions, typically resulting from the compilation of some high-level program. When the VM implementation starts running (or is reset), the convention is that it always executes an argument-less VM function called Sys.init. Typically, this function then calls the main function in the user's program. Thus, compilers that generate VM code must ensure that each translated program will have one such Sys.init function.

8.3 Implementation

This section describes how to complete the VM implementation that we started building in chapter 7, leading to a full-scale virtual machine implementation. Section 8.3.1 describes the stack structure that must be maintained, along with its standard mapping over the Hack platform. Section 8.3.2 gives an example, and section 8.3.3 provides design suggestions and a proposed API for actually building the VM implementation.

Some of the implementation details are rather technical, and dwelling on them may distract attention from the overall VM operation. This big picture is restored in section 8.3.2, which illustrates the VM implementation in action. Therefore, one may want to consult 8.3.2 for motivation while reading 8.3.1.

8.3.1 Standard VM Mapping on the Hack Platform, Part II

The Global Stack The memory resources of the VM are implemented by maintaining a global stack. Each time a function is called, a new block is added to the global stack. The block consists of the arguments that were set for the called function, a set of pointers used to save the state of the calling function, the local variables of the called function (initialized to 0), and an empty working stack for the called function. Figure 8.4 shows this generic stack structure.

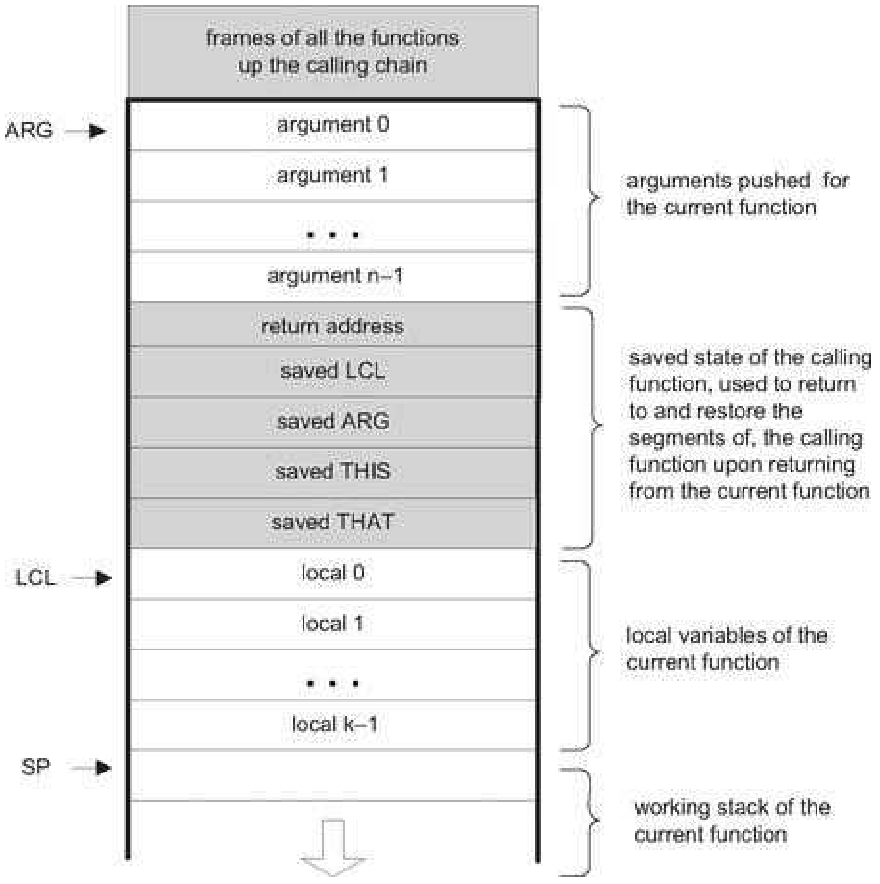

frames of all the functions up the calling chain
ARG
argument 0
argument 1
. . .
argument n–1
arguments pushed for the current function
return address
saved LCL
saved ARG
saved THIS
saved THAT
saved state of the calling function, used to return to and restore the segments of, the calling function upon returning from the current function
LCL
local 0
local 1
. . .
local k–1
local variables of the current function
SP
working stack of the current function

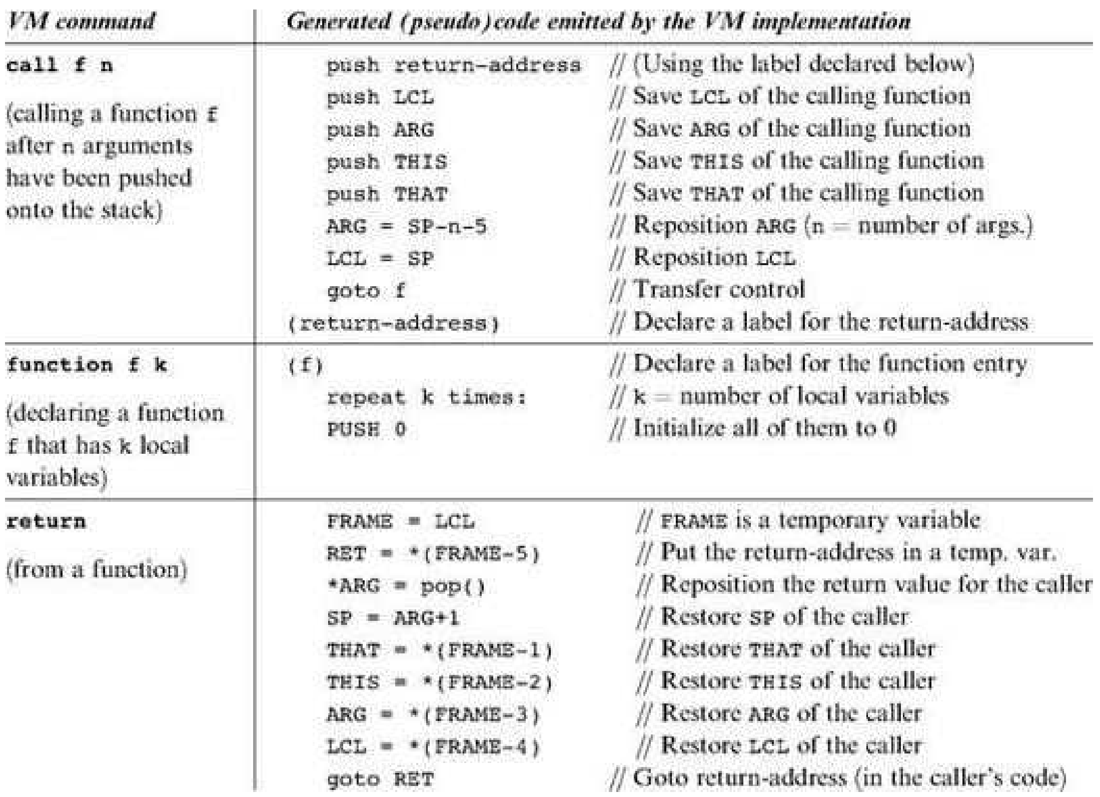
Figure 8.4 The global stack structure.

Note that the shaded areas in figure 8.4 as well as the ARG, LCL, and SP pointers are never seen by VM functions. Rather, they are used by the VM implementation to implement the function call-and-return protocol behind the scene.

How can we implement this model on the Hack platform? Recall that the standard mapping specifies that the stack should start at RAM address 256, meaning that the VM implementation can start by generating assembly code that sets SP=256. From this point onward, when the VM implementation encounters commands like pop, push, add, and so forth, it can emit assembly code that effects these operations by manipulating SP and relevant words in the host RAM. All this was already done in chapter 7. Likewise, when the VM implementation encounters commands like call, function, and return, it can emit assembly code that maintains the stack structure shown in figure 8.4 on the host RAM. This code is described next.

Function Calling Protocol Implementation The function calling protocol and the global stack structure implied by it can be implemented on the Hack platform by effecting (in Hack assembly) the pseudo-code given in figure 8.5.

Recall that the VM implementation is a translator program, written in some high-level language. It accepts VM code as input and emits assembly code as output. Hence, each pseudo-operation described in the right column of figure 8.5 is actually implemented by emitting assembly language instructions. Note that some of these "instructions" entail planting label declarations in the generated code stream.



VM command	*Generated (pseudo)code emitted by the VM implementation*	
`call f n`	`push return-address`	// (Using the label declared below)
(calling a function `f` after `n` arguments have been pushed onto the stack)	`push LCL`	// Save `LCL` of the calling function
	`push ARG`	// Save `ARG` of the calling function
	`push THIS`	// Save `THIS` of the calling function
	`push THAT`	// Save `THAT` of the calling function
	`ARG = SP-n-5`	// Reposition `ARG` (n = number of args.)
	`LCL = SP`	// Reposition `LCL`
	`goto f`	// Transfer control
	`(return-address)`	// Declare a label for the return-address
`function f k`	`(f)`	// Declare a label for the function entry
(declaring a function `f` that has `k` local variables)	`repeat k times:`	// k = number of local variables
	`PUSH 0`	// Initialize all of them to 0
`return`	`FRAME = LCL`	// `FRAME` is a temporary variable
(from a function)	`RET = *(FRAME-5)`	// Put the return-address in a temp. var.
	`*ARG = pop()`	// Reposition the return value for the caller
	`SP = ARG+1`	// Restore `SP` of the caller
	`THAT = *(FRAME-1)`	// Restore `THAT` of the caller
	`THIS = *(FRAME-2)`	// Restore `THIS` of the caller
	`ARG = *(FRAME-3)`	// Restore `ARG` of the caller
	`LCL = *(FRAME-4)`	// Restore `LCL` of the caller
	`goto RET`	// Goto return-address (in the caller's code)

Figure 8.5 VM implementation of function commands. The parenthetical (return address) and (f) are label declarations, using Hack assembly syntax convention.

Assembly Language Symbols As we have seen earlier, the implementation of program flow and function calling commands requires the VM implementation to create and use special symbols at the assembly level. These symbols are summarized in figure 8.6. For completeness of presentation, the first three rows of the table document the symbols described and implemented in chapter 7.

Symbol	*Usage*
`SP, LCL, ARG, THIS, THAT`	These predefined symbols point, respectively, to the stack top and to the base addresses of the virtual segments `local`, `argument`, `this`, and `that`.
`R13-R15`	These predefined symbols can be used for any purpose.
`Xxx.j`	Each static variable `j` in a VM file `Xxx.vm` is translated into the assembly symbol `Xxx.j`. In the subsequent assembly process, these symbolic variables will be allocated RAM space by the Hack assembler.
`functionName$label`	Each `label b` command in a VM function `f` should generate a globally unique symbol "`f$b`" where "`f`" is the function name and "`b`" is the label symbol within the VM function's code. When translating `goto b` and `if-goto b` VM commands into the target language, the full label specification "`f$b`" must be used instead of "`b`".
`(FunctionName)`	Each VM function f should generate a symbol "`f`" that refers to its entry point in the instruction memory of the target computer.
return-address	Each VM function call should generate and insert into the translated code stream a unique symbol that serves as a return address, namely the memory location (in the target platform's memory) of the command following the function call.

Figure 8.6 All the special assembly symbols prescribed by the VM-on-Hack standard mapping.

Bootstrap Code When applied to a VM program (a collection of one or more .vm files), the VM-to-Hack translator produces a single .asm file, written in the Hack assembly language. This file must conform to certain conventions. Specifically, the standard mapping specifies that (i) the VM stack should be mapped on location RAM[256] onward, and (ii) the first VM function that starts executing should be Sys.init (see section 8.2.4).

How can we effect this initialization in the .asm file produced by the VM translator? Well, when we built the Hack computer hardware in chapter 5, we wired it in such a way that upon reset, it will fetch and execute the word located in ROM[0]. Thus, the code segment that starts at ROM address 0, called bootstrap code, is the first thing that gets executed when the computer "boots up." Therefore, in view of the previous paragraph, the computer's bootstrap code should effect the following operations (in machine language):

```
SP=256          // Initialize the stack pointer to 0x0100
call Sys.init   // Start executing (the translated code of) Sys.init
```

Sys.init is then expected to call the main function of the main program and then enter an infinite loop. This action should cause the translated VM program to start running.

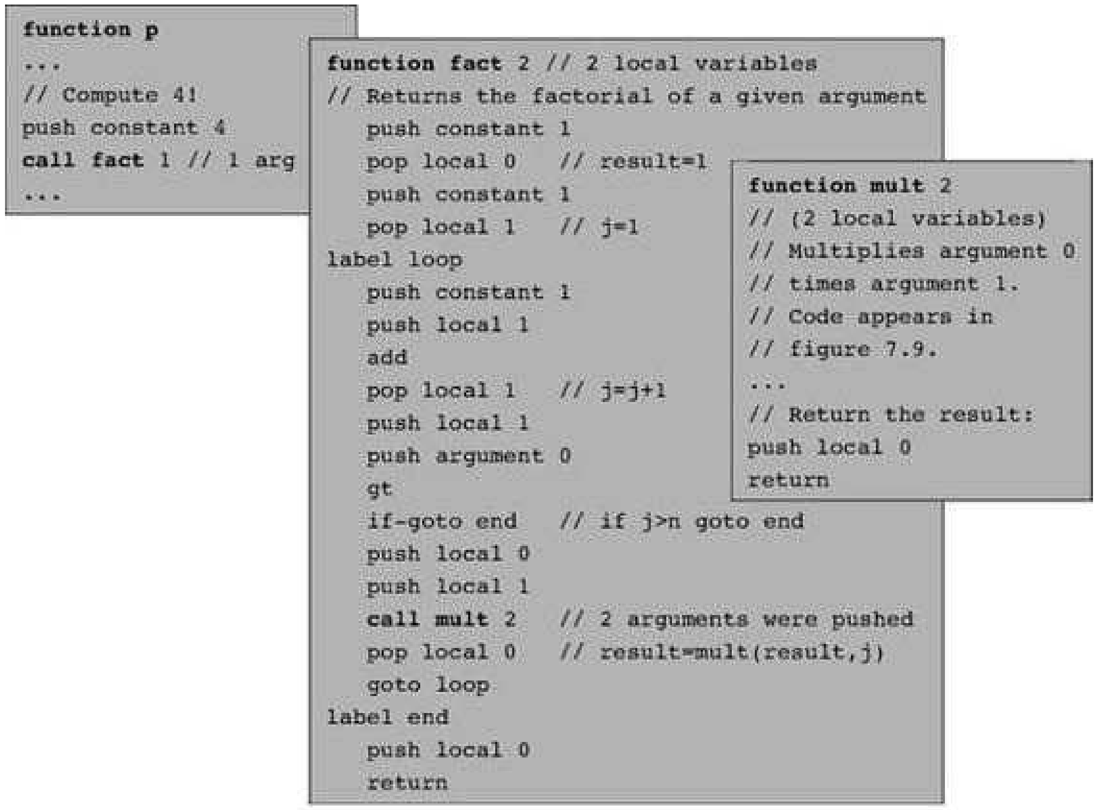
The notions of "program," "main program," and "main function" are compilation-specific and vary from one high-level language to another. For example, in the Jack language, the default is that the first program unit that starts running automatically is the main method of a class named Main. In a similar fashion, when we tell the JVM to execute a given Java class, say Foo, it looks for, and executes, the Foo.main method. Each language compiler can effect such “automatic” startup routines by programming Sys.init appropriately.

8.3.2 Example

The factorial of a positive number n can be computed by the iterative formula n! = 1 ·2·... ·$(n$ - 1) · n. This algorithm is implemented in figure 8.7.

Let us focus on the call mult command highlighted in the fact function code from figure 8.7. Figure 8.8 shows three stack states related to this call, illustrating the function calling protocol in action.

If we ignore the middle stack instance in figure 8.8, we observe that fact has set up some arguments and called mult to operate on them (left stack instance). When mult returns (right stack instance), the arguments of the called function have been replaced with the function's return value. In other words, when the dust clears from the function call, the calling function has received the service that it has requested, and processing resumes as if nothing happened: The drama of mult's processing (middle stack instance) has left no trace whatsoever on the stack, except for the return value.



```
function p
...
// Compute 4!
push constant 4
call fact 1 // 1 arg
...
```

```
function fact 2 // 2 local variables
// Returns the factorial of a given argument
   push constant 1
   pop local 0    // result=1
   push constant 1
   pop local 1    // j=1
label loop
   push constant 1
   push local 1
   add
   pop local 1    // j=j+1
   push local 1
   push argument 0
   gt
   if-goto end    // if j>n goto end
   push local 0
   push local 1
   call mult 2    // 2 arguments were pushed
   pop local 0    // result=mult(result,j)
   goto loop
label end
   push local 0
   return
```

```
function mult 2
// (2 local variables)
// Multiplies argument 0
// times argument 1.
// Code appears in
// figure 7.9.
...
// Return the result:
push local 0
return
```

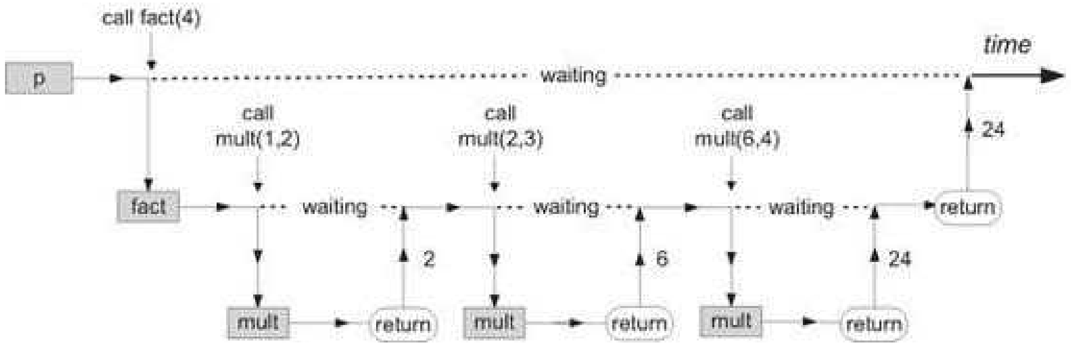


Figure 8.7 The life cycle of function calls. An arbitrary function p calls function fact, which then calls

mult several times. Vertical arrows depict transfer of control from one function to another. At any given point in time, only one function is running, while all the functions up the calling chain are waiting for it to return. When a function returns, the function that called it resumes its execution.

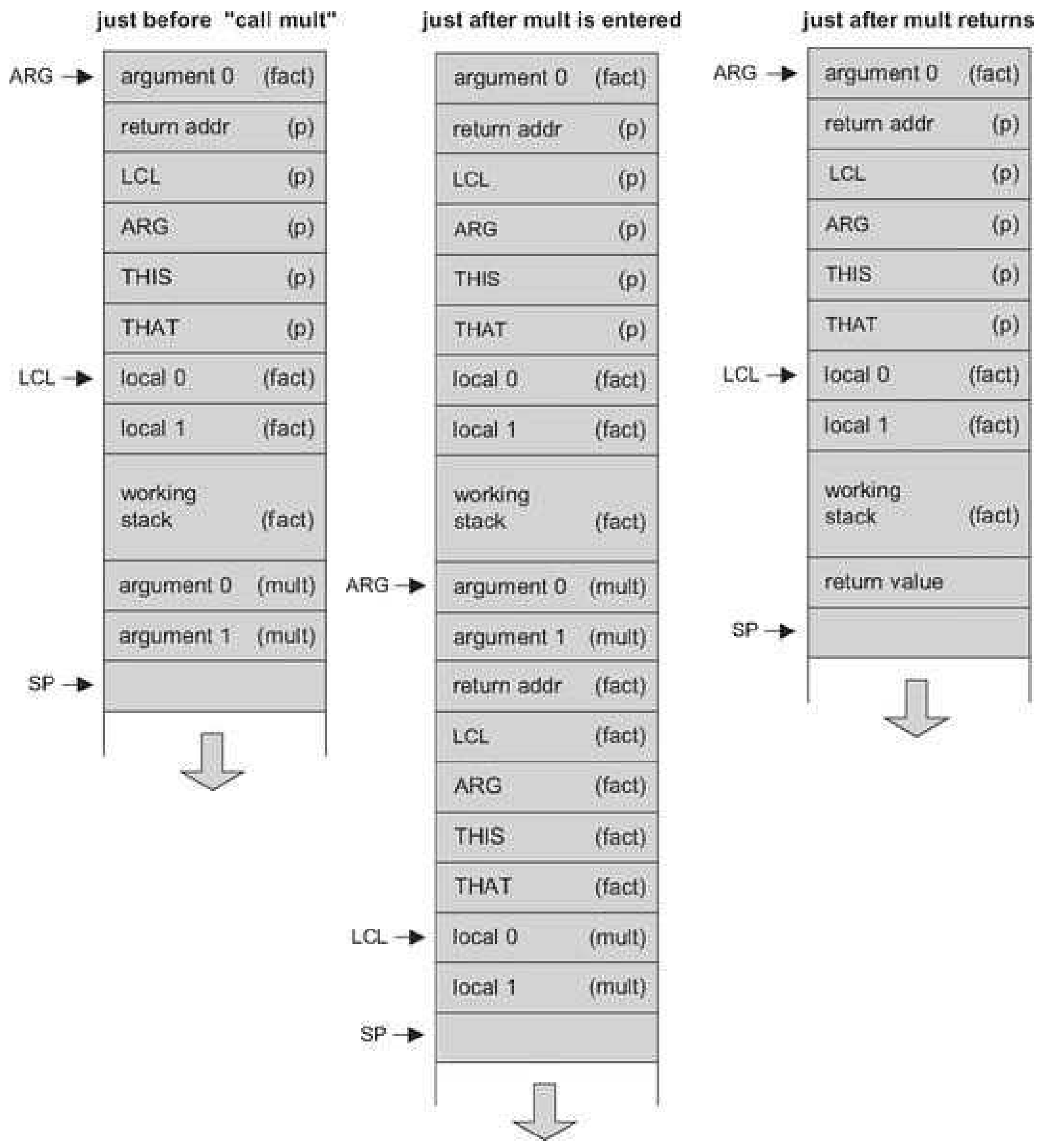


Figure 8.8 Global stack dynamics corresponding to figure 8.7, focusing on the call mult event. The pointers SP, ARG, and LCL are not part of the VM abstraction and are used by the VM implementation to map the stack on the host RAM.

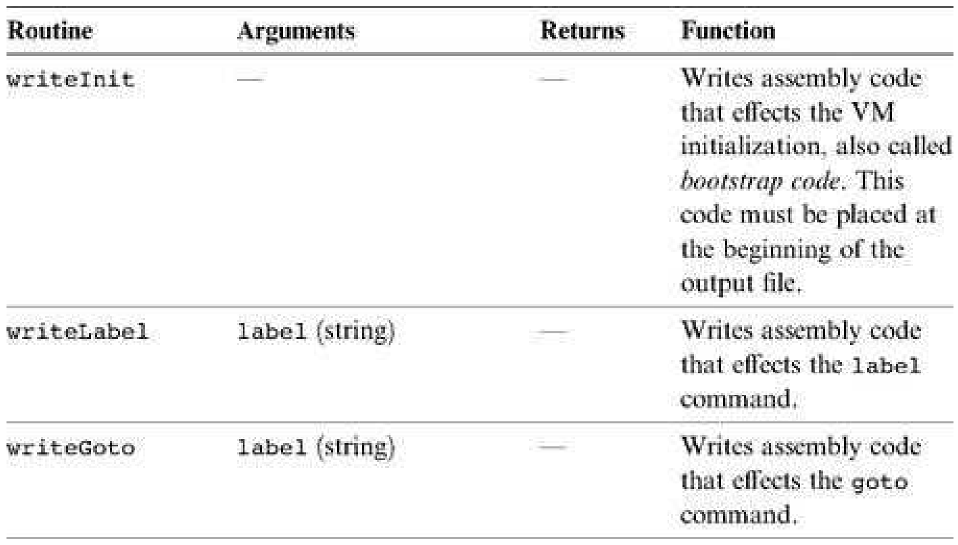
8.3.3 Design Suggestions for the VM Implementation

The basic VM translator built in Project 7 was based on two modules: parser and code writer. This translator can be extended into a full-scale VM implementation by extending these modules with the functionality described here.

The *Parser* Module If the basic parser that you built in Project 7 does not already parse the six VM commands specified in this chapter, then add their parsing now. Specifically, make sure that the commandType method developed in Project 7 also returns the constants corresponding to the six VM commands described in this chapter: C_LABEL, C_GOTO, C_IF, C_FUNCTION, C_RETURN, and C_CALL.

The *CodeWriter* Module The basic CodeWriter specified in chapter 7 should be augmented with the following methods.

CodeWriter: Translates VM commands into Hack assembly code. The routines listed here should be added to the CodeWriter module API given in chapter 7.



Routine	Arguments	Returns	Function
`writeInit`	—	—	Writes assembly code that effects the VM initialization, also called *bootstrap code*. This code must be placed at the beginning of the output file.
`writeLabel`	`label` (string)	—	Writes assembly code that effects the `label` command.
`writeGoto`	`label` (string)	—	Writes assembly code that effects the `goto` command.

Routine	Arguments	Returns	Function
`writeIf`	`label` (string)	—	Writes assembly code that effects the `if-goto` command.
`writeCall`	`functionName` (string) `numArgs` (int)	—	Writes assembly code that effects the `call` command.
`writeReturn`	—	—	Writes assembly code that effects the `return` command.
`writeFunction`	`functionName` (string) `numLocals` (int)	—	Writes assembly code that effects the `function` command.

8.4 Perspective

The notions of subroutine calling and program flow are fundamental to all high-level languages. This means that somewhere down the translation path to binary code, someone must take care of the intricate housekeeping chores related to their implementation. In Java, C#, and Jack, this burden falls on the VM level. And if the VM is stack-based, it lends itself nicely to the job, as we have seen throughout this chapter. In general then, virtual machines that implement subroutine calls and recursion as a primitive feature deliver a significant and useful abstraction.

Of course this is just one implementation option. Some compilers handle the details of subroutine calling directly, without using a VM at all. Other compilers use various forms of VMs, but not necessarily for managing subroutine calling. Finally, in some architectures most of the subroutine calling functionality is handled directly by the hardware.

In the next two chapters we will develop a Jack-to-VM compiler. Since the back-end of this compiler was already developed—it is the VM implementation built in chapters 7-8—the compiler's development will be a relatively easy task.

8.5 Project

Objective Extend the basic VM translator built in Project 7 into a full-scale VM translator. In particular, add the ability to handle the program flow and function calling commands of the VM language.

Resources (same as Project 7) You will need two tools: the programming language in which you will implement your VM translator, and the CPU emulator supplied with the book. This emulator will allow you to execute the machine code generated by your VM translator—an indirect way to test the correctness of the latter. Another tool that may come in handy in this project is the visual VM emulator supplied with the book. This program allows experimenting with a working VM implementation before you set out to build one yourself. For more information about this tool, refer to the VM emulator tutorial.

Contract Write a full-scale VM-to-Hack translator, extending the translator developed in Project 7, and conforming to the VM Specification, Part II (section 8.2) and to the Standard VM Mapping on the Hack Platform (section 8.3.1). Use it to translate the VM programs supplied below, yielding corresponding programs written in the Hack assembly language. When executed on the supplied CPU emulator, these assembly programs should deliver the results mandated by the supplied test scripts and compare files.

Testing Programs

We recommend completing the implementation of the translator in two stages. First implement the program flow commands, then the function calling commands. This will allow you to unit-test your implementation incrementally, using the test programs supplied here.

For each program Xxx, the XxxVME.tst script allows running the program on the supplied VM emulator, so that you can gain familiarity with the program's intended operation. After translating the program using your VM translator, the supplied Xxx.tst and Xxx.cmp scripts allow testing the translated assembly code on the CPU emulator.

Test Programs for Program Flow Commands

■ BasicLoop: computes $1 + 2 + \cdots + n$ and pushes the result onto the stack. This program tests the implementation of the VM language's goto and if-goto commands.

■ Fibonacci: computes and stores in memory the first n elements of the Fibonacci series. This typical array manipulation program provides a more challenging test of the VM's branching commands.

Test Programs for Function Calling Commands

■ SimpleFunction: performs a simple calculation and returns the result. This program provides a basic test of the implementation of the function and return commands.

■ FibonacciElement: This program provides a full test of the implementation of the VM's function calling commands, the bootstrap section, and most of the other VM commands.

The program directory consists of two .vm files:

● Main.vm contains one function called fibonacci. This recursive function returns the *n*-th element of the Fibonacci series;

● Sys.vm contains one function called init. This function calls the Main.fibonacci function with *n* = 4, then loops infinitely.

Since the overall program consists of two .vm files, the entire directory must be compiled in order to produce a single FibonacciElement.asm file. (compiling each ● vm file separately will yield two separate .asm files, which is not desired here).

■ StaticsTest: A full test of the VM implementation's handling of static variables. Consists of two .vm files, each representing the compilation of a stand-alone class file, plus a Sys.vm file. The entire directory should be compiled in order to produce a single StaticsTest.asm file.

(Recall that according to the VM Specification, the bootstrap code generated by the VM implementation must include a call to the Sys.init function).

Tips

Initialization In order for any translated VM program to start running, it must include a preamble startup code that forces the VM implementation to start executing it on the host platform. In addition, in order for any VM code to operate properly, the VM implementation must store the base addresses of the virtual segments in selected locations in the host RAM. The first three test programs in this project assume that the startup code was not yet implemented and include test scripts that effect the necessary initializations "manually." The last two programs assume that the startup code is already part of the VM implementation.

Testing/Debugging For each one of the five test programs, follow these steps:

1. Run the program on the supplied VM emulator, using the XxxVME.tst test script, to get acquainted with the intended program's behavior.

2. Use your partial translator to translate the .vm file(s), yielding a single .asm text file that contains a translated program written in the Hack assembly language.

3. Inspect the translated .asm program. If there are visible syntax (or any other) errors, debug and fix your translator.

4. Use the supplied .tst and .cmp files to run your translated .asm program on the CPU emulator. If there

are run-time errors, debug and fix your translator.

Note: The supplied test programs were carefully planned to unit-test the specific features of each stage in your VM implementation. Therefore, it’s important to implement your translator in the proposed order and to test it using the appropriate test programs at each stage. Implementing a later stage before an early one may cause the test programs to fail.

Tools Same as in Project 7.

9

High-Level Language

High thoughts need a high language.

—Aristophanes (448-380 BC)

All the hardware and software systems presented so far in the book were low-level, meaning that humans are not expected to interact with them directly. In this chapter we present a high-level language, called Jack, designed to enable human programmers write high-level programs. Jack is a simple object-based language. It has the basic features and flavor of modern languages like Java and C#, with a much simpler syntax and no support for inheritance. In spite of this simplicity, Jack is a general-purpose language that can be used to create numerous applications. In particular, it lends itself nicely to simple interactive games like Snake, Tetris, and Pong—a program whose complete Jack code is included in the book's software suite.

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, written in Jack. 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. Instead, our hidden agenda is to prepare you to develop the compiler and operating system that lie ahead.

If you have any experience with a modern object-oriented programming language, you will immediately feel at home with Jack. Therefore, the Background section starts the chapter with some typical programming examples, and the Specification section proceeds with a full functional description of the language and its standard library. The Implementation section gives some screen shots of typical Jack applications and offers general guidelines on how to write similar programs over the Hack platform. The final Project section provides additional details about compiling and debugging Jack programs.

All the programs shown in the chapter can be compiled by the Jack compiler supplied with the book. The resulting VM code can then run as is on the supplied VM emulator. Alternatively, one can further translate the compiled VM code into binary code, using the VM translator and the assembler built in chapters 7-8 and 6, respectively. The resulting machine code can then be executed as is on the hardware platform that we built in chapters 1-5.

It's important to reiterate that in and by itself, Jack is a rather uninteresting and simple-minded language. However, this simplicity has a purpose. First, you can learn (and unlearn) Jack very quickly—in about an hour. Second, the Jack language was carefully planned to lend itself nicely to simple compilation techniques. As a result, one 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 engineering principles underlying modern languages like Java and C#. Rather than taking the compilers and run-time environments of these languages for granted, we will build a Jack compiler and a run-time environment ourselves, beginning in the next chapter. For now, let’s take Jack out of the box.

9.1 Background

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

9.1.1 Example 1: Hello World

When we tell the Jack run-time environment to run a given 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.

Jack is equipped with a standard library whose complete API is given in section 9.2.7. This library extends the basic language with various abstractions and services such as arrays, strings, mathematical functions, memory management, and input/output functions. Two such functions are invoked by the program in figure 9.1, effecting the "Hello world" printout. The program also demonstrates the three comment formats supported by Jack.

```
/** Hello World program. */
class Main {
  function void main() {
    /* Prints some text using the standard library. */
    do Output.printString("Hello World");
    do Output.println();    // New line
    return;
  }
}
```

Figure 9.1 Hello World.

9.1.2 Example 2: Procedural Programming and Array Handling

Jack is equipped with typical language constructs for procedural programming. It also includes basic commands for declaring and manipulating arrays. Figure 9.2 illustrates both of these features, in the context of inputting and computing the average of a series of numbers.

Jack programs declare and construct arrays using the built-in Array class, which is part of the standard Jack library. Note that Jack arrays are not typed and can include anything—integers, objects, and so forth.

9.1.3 Example 3: Abstract Data Types

Every programming language has a fixed set of primitive data types, of which Jack supports three: int, char, and boolean. Programmers can extend this basic repertoire by creating new classes that represent abstract data types, as needed. For example, suppose we wish to endow Jack with the ability to handle rational numbers, namely, objects of the form *n/m* where *n* and *m* are integers. This can be done by creating a stand-alone class, designed to provide a fraction abstraction for Jack programs. Let us call this class Fraction.

Defining a Class Interface A reasonable way to get started is to specify the set of properties and services expected from a fraction abstraction. One such Application Program Interface (API), is given in figure 9.3a.

```
/** Computes the average of a sequence of integers. */
class Main {
  function void main() {
    var Array a;
    var int length;
    var int i, sum;

    let length = Keyboard.readInt("How many numbers? ");
    let a = Array.new(length); // Constructs the array
    let i = 0;

    while (i < length) {
      let a[i] = Keyboard.readInt("Enter the next 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 Procedural programming and array handling.

In Jack, operations on the current object (referred to as this) are represented by methods, whereas class-level operations (equivalent to static methods in Java) are represented by *functions*. Operations that create new objects are called constructors.

Using Classes APIs mean different things to different people. If you are the programmer who has to implement the fraction class, you can view its API as a contract that must be implemented, one way or another. Alternatively, if you are a programmer who needs to use fractions in your work, you can view the API as a documentation of a fraction server, designed to generate fraction objects and supply fraction-related operations. Taking this latter view, consider the Jack code listed in figure 9.3b.

Figure 9.3b illustrates an important software engineering principle: Users of any given abstraction don't have to know anything about its underlying implementation. Rather, they can be given access only to the abstraction's interface, or class API, and then use it as a black box server of abstraction-related operations.

```
// A Fraction is an object representation of n/m where n and m are integers.
field int numerator, denominator        // Fraction object properties
constructor Fraction new(int a, int b)  // Returns a new Fraction object
method int getNumerator()               // Returns the numerator of this
                                        // fraction
method int getDenominator()             // Returns the denominator of this
                                        // fraction
method Fraction plus(Fraction other)    // Returns the sum of this fraction
                                        // and another fraction, as a
                                        // fraction
method void print()                     // Prints this fraction in the
                                        // format "numerator/denominator"
// Additional fraction-related services are specified here, as needed.
```

Figure 9.3a Fraction class API.

```
// Computes the sum of 2/3 and 1/5.
class Main {
  function void main() {
    var Fraction a, b, c;
    let a = Fraction.new(2,3);
    let b = Fraction.new(1,5);
    let c = a.plus(b);  // Compute c = a + b
    do c.print();  // Should print the text "13/15"
    return;
  }
}
```

Figure 9.3b Using the Fraction abstraction.

```
/** Provides the Fraction type and related services. */
class Fraction {
  field int numerator, denominator;

  /** Constructs a new (and reduced) fraction from given
   *  numerator and denominator. */
  constructor Fraction new(int a, int b) {
    let numerator = a;  let denominator = b;
    do reduce();  // If a/b is not reduced, reduce it
    return this;
  }

  /** 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 denominator of a and b. */
  function int gcd(int a, int b){
    var int r;
    while (~(b = 0)) {           // Apply Euclid's algorithm.
      let r = a - (b * (a / b));  // r=remainder of a/b
      let a = b; let b = r; }
    return a;
  }

  /** Accessors. */
  method int getNumerator() { return numerator; }
  method int getDenominator() { return denominator; }
```

Figure 9.3c A possible Fraction class implementation.

```
  /** Returns the sum of this fraction and another one.
  method Fraction plus(Fraction other){
    var int sum;
    let sum = (numerator * other.getDenominator()) +
              (other.getNumerator() * denominator());
    return Fraction.new(sum, denominator *
           other.getDenominator());
  }

  // More fraction-related methods: minus, times, div, etc.

  /** Prints this fraction. */
  method void print() {
    do Output.printInt(numerator);
    do Output.printString("/");
    do Output.printInt(denominator);
    return;
  }
} // Fraction class
```

Implementing the Class We now turn to the other player in our story—the programmer who has to actually implement the fraction abstraction. A possible Jack implementation is given in figure 9.3c.

Figure 9.3c illustrates the typical Jack program structure: classes, methods, constructors, and functions. It also demonstrates all the statement types available in the language: let, do, if, while, and return.

9.1.4 Example 4: Linked List Implementation

A *linked list* (or simply *list*) is a chain of objects, each consisting of a data element and a reference (pointer) to the rest of the list. Figure 9.4 shows a possible Jack class implementation of the linked list abstraction. The purpose of this example is to illustrate typical object handling in the Jack language.

```
/** The List class provides a linked list abstraction. */
class List {
  field int data;
  field List next;

  /* Creates a new List object. */
  constructor List new(int car, List cdr) {
    let data = car;
    let next = cdr;
    return this;
  }

  /* Disposes this List by recursively disposing its tail. */
  method void dispose() {
    if (~(next = null)) {
       do next.dispose();
    }
    // Use an OS routine to recycle the memory held by this
    // object.
    do Memory.deAlloc(this);
    return;
  }

  // More List-related methods come here

 } // class List
```

```
/* Creates a list holding the numbers (2,3,5).
   (this code can appear in any class). */
function void create235() {
  var List v;
  let v = List.new(5,null);
  let v = List.new(2,List.new(3,v));
  ... // Does something with the list
  do v.dispose();
  return;
}
```

Figure 9.4 Object handling in a linked list context.

9.2 The Jack Language Specification

We now turn to a formal and complete description of the Jack language, organized by its syntactic elements, program structure, variables, expressions, and statements. This language specification should be viewed 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, which are ignored. Tokens can be symbols, reserved words, constants, and identifiers, as listed in figure 9.5.

9.2.2 Program Structure

The basic programming unit in Jack is a class. Each class resides in a separate file and can be compiled separately. Class declarations have the following format:

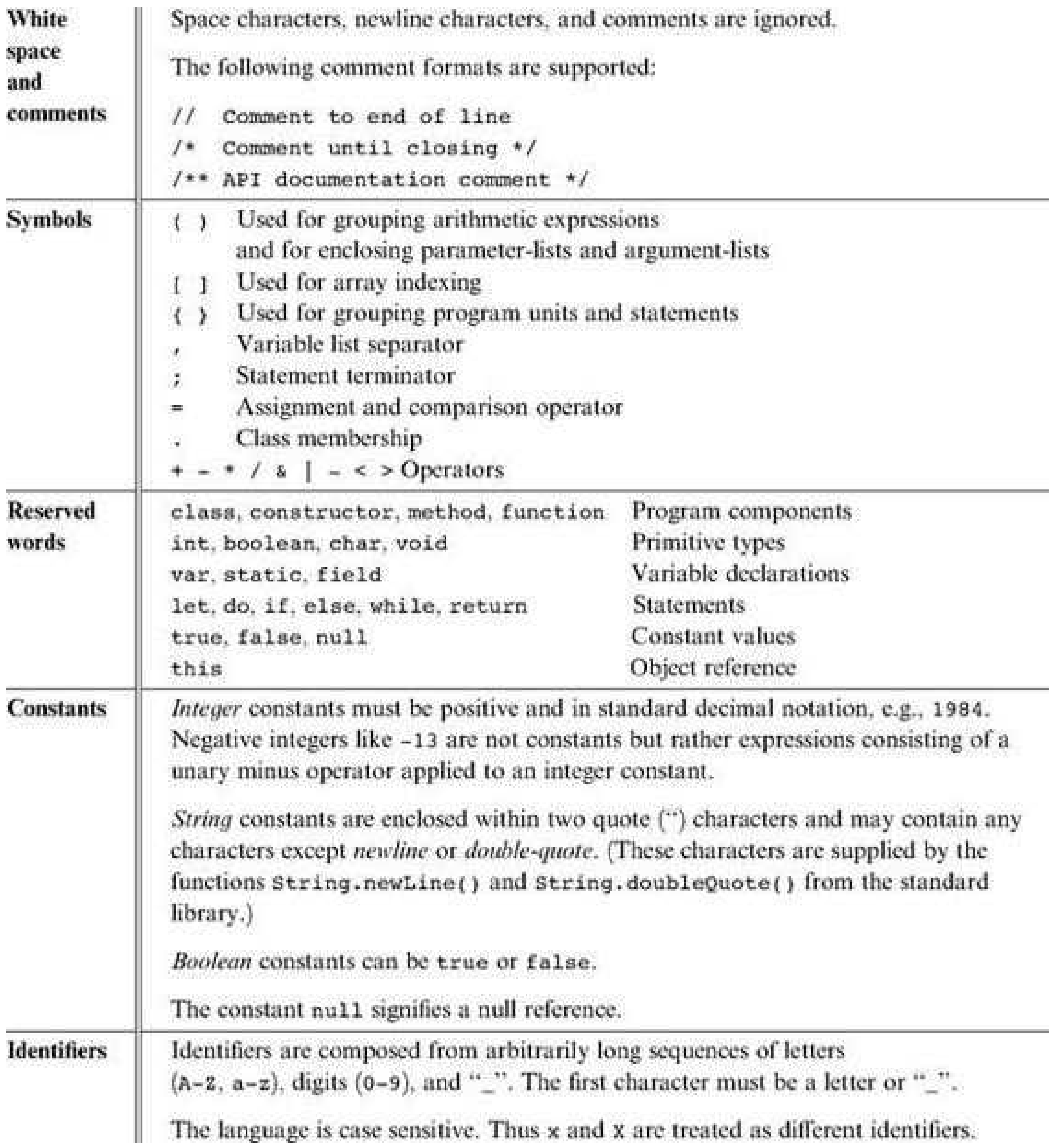
```
class name {
    Field and static variable declarations  // Must precede subroutine declarations.
    Subroutine declarations  // Constructor, method and function declarations.
}
```

Each class declaration specifies a name through which the class can be globally accessed. Next comes a sequence of zero or more field and static variable declarations. Then comes a sequence of one or more subroutine declarations, each defining a method, a *function*, or a constructor. Methods "belong to" objects and provide their functionality, while functions "belong to" the class in general and are not associated with a particular object (similar to Java's static *methods*). A constructor "belongs to" the class and, when called, generates object instances of this class.

All subroutine declarations have the following format:

```
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 describing the value returned by the subroutine. If the subroutine returns no value, the type is declared void; otherwise, it can be any of the primitive data types supported by the language, or any of the class types supplied by the standard library, or any of the class types supplied by other classes in the application. Constructors may have arbitrary names, but they must return an object of the class type. Therefore the type of a constructor must always be the name of the class to which it belongs.



White space and comments	Space characters, newline characters, and comments are ignored.	
	The following comment formats are supported:	
	`// Comment to end of line`	
	`/* Comment until closing */`	
	`/** API documentation comment */`	
Symbols	`( )`	Used for grouping arithmetic expressions and for enclosing parameter-lists and argument-lists
	`[ ]`	Used for array indexing
	`{ }`	Used for grouping program units and statements
	`,`	Variable list separator
	`;`	Statement terminator
	`=`	Assignment and comparison operator
	`.`	Class membership
	`+ - * / & | ~ < >`	Operators
Reserved words	`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	*Integer* constants must be positive and in standard decimal notation, e.g., `1984`. Negative integers like `-13` are not constants but rather expressions consisting of a unary minus operator applied to an integer constant.	
	String constants are enclosed within two quote (") characters and may contain any characters except *newline* or *double-quote*. (These characters are supplied by the functions `String.newLine()` and `String.doubleQuote()` from the standard library.)	
	Boolean constants can be `true` or `false`.	
	The constant `null` signifies a null reference.	
Identifiers	Identifiers are composed from arbitrarily long sequences of letters (`A-Z`, `a-z`), digits (`0-9`), and "_". The first character must be a letter or "_".	
	The language is case sensitive. Thus `x` and `X` are treated as different identifiers.	

Figure 9.5 Jack syntactic elements.

Following its header specification, the subroutine declaration contains a sequence of zero or more local variable declarations, then a sequence of zero or more statements.

As in Java, a *Jack program* is a collection of one or more classes. One class must be named Main, and this class must include at least one function named main. When instructed to execute a Jack program that resides in some directory, the Jack run-time environment will automatically start running the Main.main function.

9.2.3 Variables

Variables in Jack must be explicitly declared before they are used. There are four kinds of variables: field, static, local, and parameter variables, each with its associated scope. Variables must be typed.

Data Types Each variable can assume either a primitive data type (int, char, boolean), as predefined in the Jack language specification, or an object type, which is the name of a class. The class that implements this type can be either part of the Jack standard library (e.g., String or Array), or it may be any other class residing in the program directory.

Primitive Types Jack features three primitive data types:

- int: 16-bit 2’s complement
- boolean: false and true
- char: unicode character

Variables of primitive types are allocated to memory when they are declared. For example, the declarations var int age; var boolean gender; cause the compiler to create the variables age and gender and to allocate memory space to them.

Object Types Every class defines an object type. As in Java, the declaration of an object variable only causes the creation of a reference variable (pointer). Memory for storing the object itself is allocated later, if and when the programmer actually *constructs* the object by calling a constructor. Figure 9.6 gives an example.

```
// This code assumes the existence of Car and Employee classes.
// Car objects have model and licensePlate fields.
// Employee objects have name and Car fields.
var Employee e, f;  // Creates variables e, f that contain null references
var Car c;          // Creates a variable c that contains a null reference
...
let c = Car.new("Jaguar","007")    // Constructs a new Car object
let e = Employee.new("Bond",c)     // Constructs a new Employee object
// At this point c and e hold the base addresses of the memory segments
// allocated to the two objects.
let f = e;  // Only the reference is copied - no new object is constructed.
```

Figure 9.6 Object types (example).

The Jack standard library provides two built-in object types (classes) that play a role in the language syntax: Array and String.

Arrays Arrays are declared using a built-in class called Array. Arrays are one-dimensional and the first index is always 0 (multi-dimensional arrays may be obtained as arrays of arrays). Array entries do not have a declared type, and different entries in the same array may have different types. The declaration of an array only creates a reference, while the actual construction of the array is done by calling the Array.new(length) constructor. Access to array elements is done using the a [j] notation. Figure 9.2 illustrates working with arrays.

Strings Strings are declared using a built-in class called String. The Jack compiler recognizes the syntax "xxx" and treats it as the contents of some String object. The contents of String objects can be accessed and modified using the methods of the String class, as documented in its API. Example:

```
var String s;
var char c;
...
let s = "Hello World";
let c = s.charAt(6);      // "W"
```

Type Conversions The Jack language is weakly typed. The language specification does not define the results of attempted assignment or conversion from one type to another, and different Jack compilers may allow or forbid them. (This under-specification is intentional, allowing the construction of minimal Jack compilers that ignore typing issues.)

Having said that, all Jack compilers are expected to allow, and automatically perform, the following assignments:

■ Characters and integers are automatically converted into each other as needed, according to the Unicode specification. Example:

```
var char c;  var String s;
let c = 33;  // 'A'
// Equivalently:
let s = "A"; let c = s.charAt(0);
```

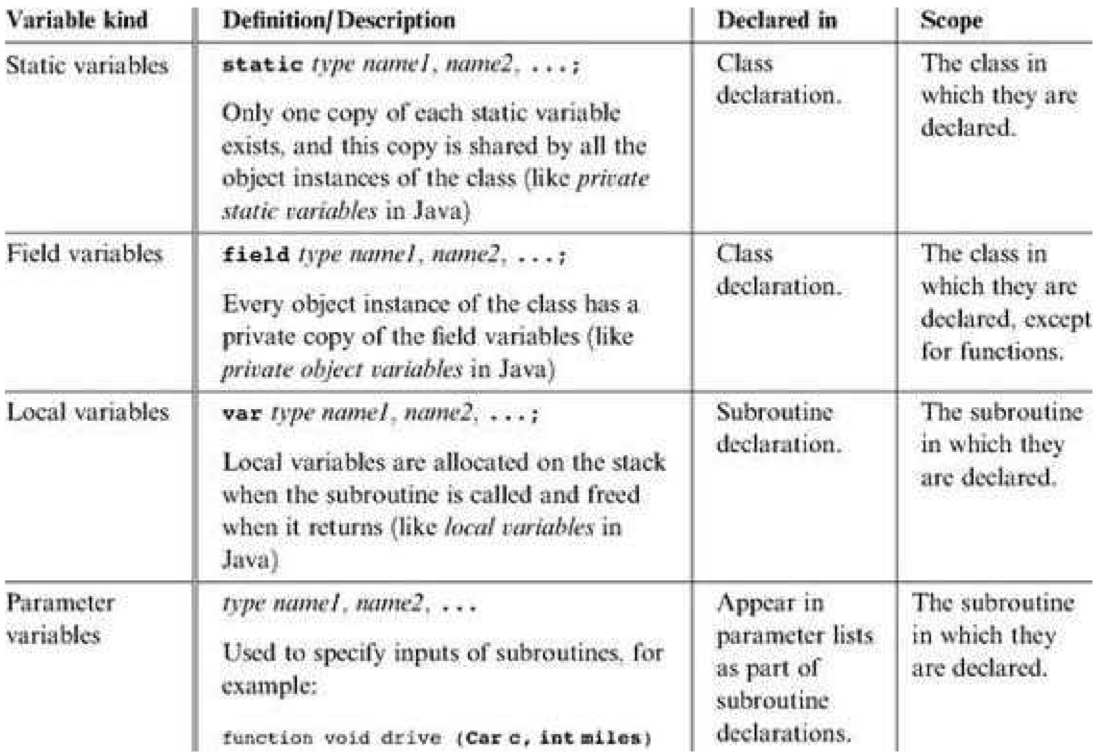
■ An integer can be assigned to a reference variable (of any object type), in which case it is treated as an address in memory. Example:

```
var Array a;
let a = 5000;
let a[100] = 77; // Memory address 5100 is set to 77
```

■ An object variable (whose type is a class name) may be converted into an Array variable, and vice versa. The conversion allows accessing the object fields as array entries, and vice versa. Example:

```
// Assume that class Complex has two int fields: re and im.
var Complex c; var Array a;
let a = Array.new(2);
let a[0] = 7; let a[1] = 8;
let c = a; // c==Complex(7,8)
```

Variable Kinds and Scope Jack features four kinds of variables. Static variables are defined at the class level and are shared by all the objects derived from the class. For example, a BankAccount class may have a totalBalance static variable holding the sum of balances of all the bank accounts, each account being an object derived from the BankAccount class. Field variables are used to define the properties of individual objects of the class, for example, account owner and balance. Local variables, used by subroutines, exist only as long as the subroutine is running, and parameter variables are used to pass arguments to subroutines. For example, our BankAccount class may include the method signature method void transfer- (BankAccount from, int sum), declaring the two parameters from and sum. Thus, if joeAccount and janeAccount were two variables of type BankAccount, the command joeAccount.transfer(janeAccount, 100) will effect a transfer of 100 from Jane to Joe.



Variable kind	Definition/Description	Declared in	Scope
Static variables	**static** *type name1, name2, ...;* Only one copy of each static variable exists, and this copy is shared by all the object instances of the class (like *private static variables* in Java)	Class declaration.	The class in which they are declared.
Field variables	**field** *type name1, name2, ...;* Every object instance of the class has a private copy of the field variables (like *private object variables* in Java)	Class declaration.	The class in which they are declared, except for functions.
Local variables	**var** *type name1, name2, ...;* Local variables are allocated on the stack when the subroutine is called and freed when it returns (like *local variables* in Java)	Subroutine declaration.	The subroutine in which they are declared.
Parameter variables	*type name1, name2, ...* Used to specify inputs of subroutines, for example: `function void drive (Car c, int miles)`	Appear in parameter lists as part of subroutine declarations.	The subroutine in which they are declared.

Figure 9.7 Variable kinds in the Jack language (throughout the table, *subroutine* is either a *function, a method, or a constructor*).

Figure 9.7 gives a formal description of all the variable kinds supported by the Jack language. The scope of a variable is the region in the program in which the variable name is recognized.

Statement	Syntax	Description
`let`	**`let`** *variable* `=` *expression*`;` or **`let`** *variable* `[`*expression*`]` `=` *expression*`;`	An assignment operation (where *variable* is either single-valued or an array). The variable kind may be *static*, *local*, *field*, or *parameter*.
`if`	**`if`** `(`*expression*`)` `{` *statements* `}` **`else`** `{` *statements* `}`	Typical *if* statement with an optional *else* clause. The curly brackets are mandatory even if *statements* is a single statement.
`while`	**`while`** `(`*expression*`)` `{` *statements* `}`	Typical *while* statement. The curly brackets are mandatory even if *statements* is a single statement.
`do`	**`do`** *function-or-method-call*`;`	Used to call a function or a method for its effect, ignoring the returned value.
`return`	**`Return`** *expression*`;` or **`return;`**	Used to return a value from a subroutine. The second form must be used by functions and methods that return a void value. Constructors must return the expression `this`.

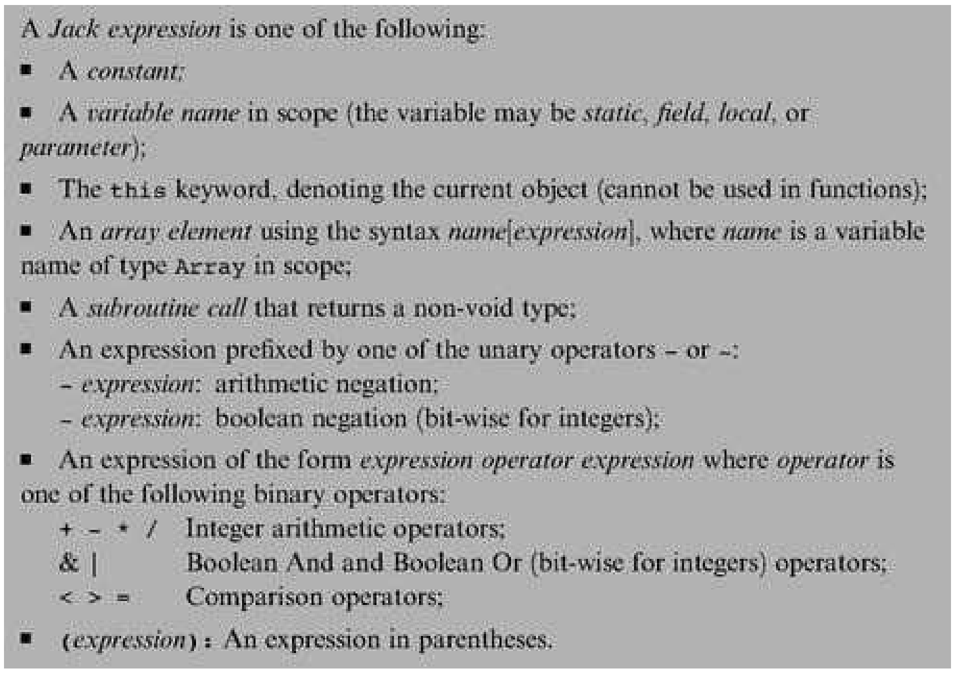
Figure 9.8 Jack statements.

9.2.4 Statements

The Jack language features five generic statements. They are defined and described in figure 9.8.

9.2.5 Expressions

Jack expressions are defined recursively according to the rules given in figure 9.9.



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 *name*[*expression*], where *name* 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 (bit-wise for integers);
- An expression of the form *expression operator expression* where *operator* is one of the following binary operators:

+ – * /	Integer arithmetic operators;
& |	Boolean And and Boolean Or (bit-wise for integers) operators;
< > =	Comparison operators;

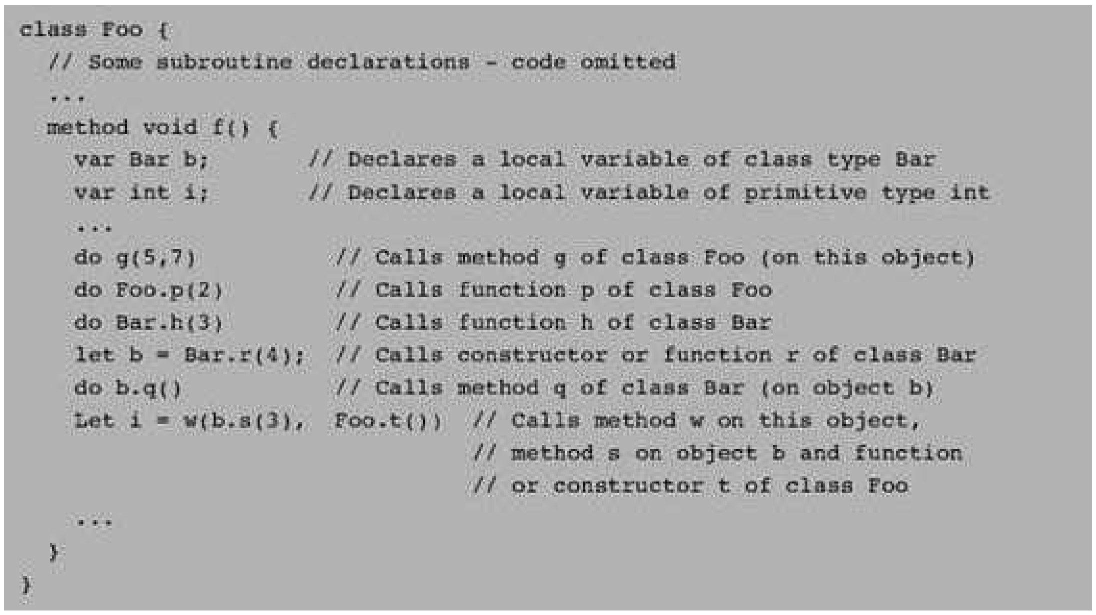
- (*expression*) : An expression in parentheses.

Figure 9.9 Jack expressions.

Operator Priority and Order of Evaluation Operator priority is not defined by the language, except that expressions in parentheses are evaluated first. Thus an expression like 2+3*4 may yield either 20 or 14, whereas 2+(3*4) is guaranteed to yield 14. The need to use parentheses in such expressions makes Jack programming a bit cumbersome. However, the lack of formal operator priority is intentional, since it simplifies the writing of Jack compilers. Of course, different language implementations (compilers) can specify an operator priority and add it to the language documentation, if so desired.

9.2.6 Subroutine Calls

Subroutine calls invoke methods, functions, and constructors for their effect, using the general syntax *subroutineName*(*argument-list*). The number and type of the arguments must match those of the subroutine's parameters, as defined in its declaration. The parentheses must appear even if the argument list is empty. Each argument may be an expression of unlimited complexity. For example, the Math class, which is part of Jack's standard library, contains a square root function whose declaration is function int sqrt(int n). Such a function can be invoked using calls like Math.sqrt (17), or Math.sqrt ((a * Math.sqrt (c-17) + 3), and so on.



```
class Foo {
  // Some subroutine declarations - code omitted
  ...
  method void f() {
    var Bar b;       // Declares a local variable of class type Bar
    var int i;       // Declares a local variable of primitive type int
    ...
    do g(5,7)          // Calls method g of class Foo (on this object)
    do Foo.p(2)        // Calls function p of class Foo
    do Bar.h(3)        // Calls function h of class Bar
    let b = Bar.r(4);  // Calls constructor or function r of class Bar
    do b.q()           // Calls method q of class Bar (on object b)
    Let i = w(b.s(3),  Foo.t())  // Calls method w on this object,
                                 // method s on object b and function
                                 // or constructor t of class Foo

    ...
  }
}
```

Figure 9.10 Subroutine call examples.

Within a class, methods are called using the syntax *methodName(argument-list*), while functions and constructors must be called using their full names, namely, *className.subroutineName*(*argument-list*). Outside a class, the class functions and constructors are also called using their full names, while methods are called using the syntax varName.methodName(argument-list), where varName is a previously defined object variable. Figure 9.10 gives some examples.

Object Construction and Disposal Object construction is a two-stage affair. When a program declares a variable of some object type, only a reference (pointer) variable is created and allocated memory. To complete the object's construction (if so desired), the program must call a constructor from the object's class. Thus, a class that implements a type (e.g., Fraction from figure 9.3c) must contain at least one constructor. Constructors may have arbitrary names, but it is customary to call one of them new. Constructors are called just like any other class function using the format:

```
let varName = className.constructorName(parameter-list);
```

For example, let c = Circle.new(x,y,50) where x, y, and 50 are the screen location of the circle's center and its radius. When a constructor is called, the compiler requests the operating system to allocate enough memory space to hold the new object in memory. The OS returns the base address of the allocated memory segment, and the compiler assigns it to this (in the circle example, the value of this is assigned to c). Next, the constructed object is typically initialized to some valid state, effected by the Jack commands found in the constructor's body.

When an object is no longer needed in a program, it can be disposed. In particular, objects can be de-allocated from memory and their space reclaimed using the Memory.deAlloc (object) function from the standard library. Convention calls for every class to contain a dispose() method that properly encapsulates this de-allocation. For example, see figure 9.4.

9.2.7 The Jack Standard Library

The Jack language comes with a collection of built-in classes that extend the language's capabilities. This standard library, which can also be viewed as a basic operating system, must be provided in every Jack language implementation. The standard library includes the following classes, all implemented in Jack:

- *Math:* provides basic mathematical operations;
- *String:* implements the String type and string-related operations;
- *Array:* implements the Array type and array-related operations;
- *Output:* handles text output to the screen;
- *Screen:* handles graphic output to the screen;
- *Keyboard:* handles user input from the keyboard;
- *Memory:* handles memory operations;
- *Sys:* provides some execution-related services.

Math This class enables various mathematical operations.

- function void **init** (): for internal use only.
- function int **abs** (int x): returns the absolute value of x.
- function int **multiply**(int x, int y): returns the product of x and y.
- function int **divide**(int x, int y): returns the integer part of x/y.
- function int **min**(int x, int y): returns the minimum of x and y.
- function int **max**(int x, int y): returns the maximum of x and y.
- function int **sqrt**(int x): returns the integer part of the square root of x.

String This class implements the String data type and various string-related operations.

■ constructor String **new**(int maxLength): constructs a new empty string (of length zero) that can contain at most maxLength characters;

■ method void **dispose**(): disposes this string;

■ method int **length**(): returns the length of this string;

■ method char **charAt**(int j): returns the character at location j of this string;

■ method void **setCharAt**(int j, char c): sets the j-th element of this string to c;

■ method String **appendChar**(char c): appends c to this string and returns this string;

■ method void **eraseLastChar**(): erases the last character from this string;

■ method int **intValue**(): returns the integer value of this string (or of the string prefix until a non-digit character is detected);

■ method void **setInt**(int j): sets this string to hold a representation of j;

■ function char **backSpace**(): returns the backspace character;

■ function char **doubleQuote**(): returns the double quote (“) character;

■ function char **newLine**(): returns the newline character.

Array This class enables the construction and disposal of arrays.

■ function Array **new**(int size): constructs a new array of the given size;

■ method void **dispose**(): disposes this array.

Output This class allows writing text on the screen.

■ function void **init**(): for internal use only;

■ function void **moveCursor**(int i, int j): moves the cursor to the j-th column of the i-th row, and erases the character displayed there;

■ function void **printChar**(char c): prints c at the cursor location and advances the cursor one column forward;

■ function void **printString**(String s): prints s starting at the cursor location and advances the cursor appropriately;

■ function void **printInt**(int i): prints i starting at the cursor location and advances the cursor appropriately;

■ function void **println**(): advances the cursor to the beginning of the next line;

■ function void **backSpace**(): moves the cursor one column back.

Screen This class allows drawing graphics on the screen. Column indices start at 0 and are left-to-right. Row indices start at 0 and are top-to-bottom. The screen size is hardware-dependant (in the Hack platform: 256 rows by 512 columns).

■ function void **init**(): for internal use only;

■ function void **clearScreen**(): erases the entire screen;

■ function void **setColor**(boolean b): sets a color (white = false, black = true) to be used for all further drawXXX commands;

■ function void **drawPixel**(int x, int y): draws the (x,y) pixel;

■ function void **drawLine**(int x1, int y1, int x2, int y2): draws a line from pixel (x1,y1) to pixel (x2,y2);

■ function void **drawRectangle**(int x1, int y1, int x2, int y2): draws a filled rectangle whose top left corner is (x1,y1) and bottom right corner is (x2,y2);

■ function void **drawCircle**(int x, int y, int r): draws a filled circle of radius r <= 181 around (x,y).

Keyboard This class allows reading inputs from a standard keyboard.

■ function void **init**(): for internal use only;

■ function char **keyPressed**(): returns the character of the currently pressed key on the keyboard; if no key is currently pressed, returns 0;

■ function char **readChar**(): waits until a key is pressed on the keyboard and released, then echoes the key to the screen and returns the character of the pressed key;

■ function String **readLine**(String message): prints the message on the screen, reads the line (text until a newline character is detected) from the keyboard, echoes the line to the screen, and returns its value. This function also handles user backspaces;

■ function int **readInt**(String message): prints the message on the screen, reads the line (text until a newline character is detected) from the keyboard, echoes the line to the screen, and returns its integer value (until the first nondigit character in the line is detected). This function also handles user backspaces.

Memory This class allows direct access to the main memory of the host platform.

■ function void **init**(): for internal use only;

■ function int **peek**(int address): returns the value of the main memory at this address;

■ function void **poke**(int address, int value): sets the contents of the main memory at this address to value;

■ function Array **alloc**(int size): finds and allocates from the heap a memory block of the specified size and returns a reference to its base address;

■ function void **deAlloc**(Array o): De-allocates the given object and frees its memory space.

Sys This class supports some execution-related services.

■ function void **init**(): calls the init functions of the other OS classes and then calls the Main.main () function. For internal use only;

■ function void **halt**(): halts the program execution;

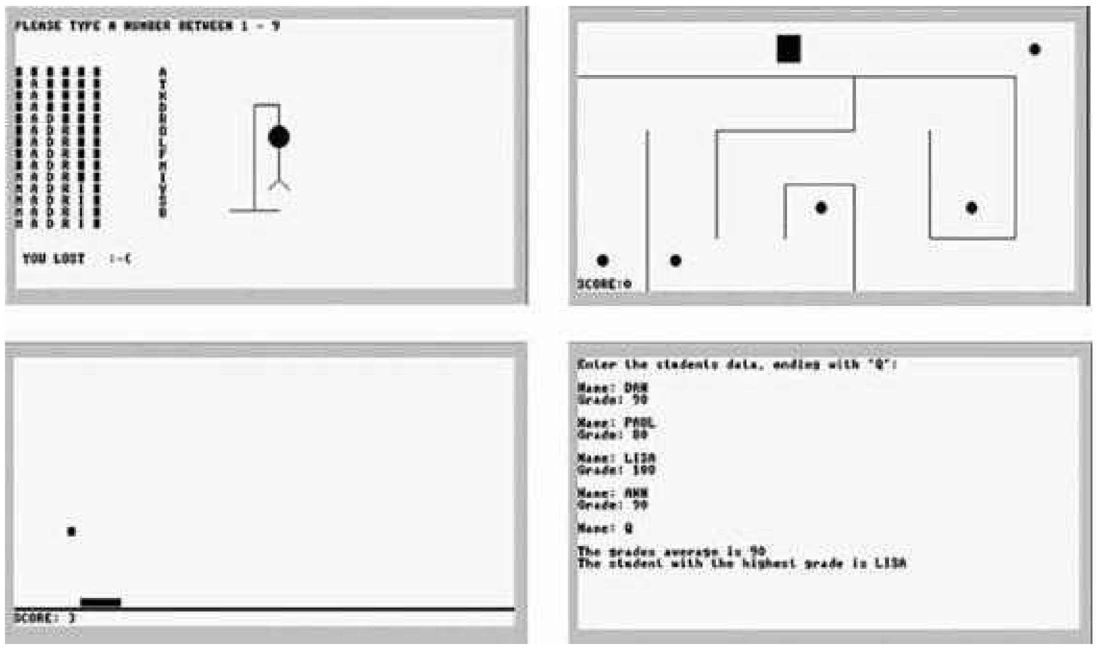
■ function void **error**(int errorCode): prints the error code on the screen and halts;

■ function void **wait**(int duration): waits approximately duration milliseconds and returns.

9.3 Writing Jack Applications

Jack is a general-purpose programming language that can be implemented over different hardware platforms. In the next two chapters we will develop a Jack compiler that ultimately generates binary Hack code, and thus it is natural to discuss Jack applications in the Hack context. This section illustrates briefly three such applications and provides general guidelines about application development on the Jack-Hack platform.

Examples Four sample applications are illustrated in figure 9.11. The Pong game, whose Jack code is supplied with the book, provides a good illustration of Jack programming over the Hack platform. The Pong code is not trivial, requiring several hundred lines of Jack code organized in several classes. Further, the program has to carry out some nontrivial mathematical calculations in order to compute the direction of the ball's movements. The program must also animate the movement of graphical objects on the screen, requiring extensive use of the language's graphics drawing services. And, in order to do all of the above quickly, the program must be efficient, meaning that it has to do as few real-time calculations and screen drawing operations as possible.



```
Enter the students data, ending with 'Q':

Name: DAN
Grade: 90

Name: PAUL
Grade: 80

Name: LISA
Grade: 100

Name: ANN
Grade: 90

Name: Q

The grades average is 90
The student with the highest grade is LISA
```

Figure 9.11 Screen shots of sample Jack applications, running on the Hack computer. Hangman, Maze, Pong, and a simple data processing program.

Application Design and Implementation The development of Jack applications over a hardware platform like Hack requires careful planning (as always). First, the application designer must consider the physical limitations of the hardware, and plan accordingly. For example, the dimensions of the computer's screen limit the size of the graphical images that the program can handle. Likewise, one must consider the language's range of input/output commands and the platform's execution speed, to gain a realistic expectation of what can and cannot be done.

As usual, the design process normally starts with a conceptual description of the application's behavior. In the case of graphical and interactive programs, this may take the form of hand-written drawings of typical screens. In simple applications, one can proceed to implementation using procedural programming. In more complex tasks, it is advisable to first create an object-based design of the application. This entails the identification of classes, fields, and subroutines, possibly leading to the creation of some API document (e.g., figure 9.3a).

Next, one can proceed to implement the design in Jack and compile the class files using a Jack compiler. The testing and debugging of the code generated by the compiler depend on the details of the target platform. In the Hack platform supplied with the book, testing and debugging are normally done using the supplied VM emulator. Alternatively, one can translate the Jack program all the way to binary code and run it directly on the Hack hardware, or on the CPU emulator supplied with the book.

The Jack OS Jack programs make an extensive use of the various abstractions and services supplied by the language's standard library, also called the Jack OS. This OS is itself implemented in Jack, and thus its executable version is a set of compiled .vm files—just like the user program (following compilation). Therefore, before running any Jack program, you must first copy into the program directory the .vm files comprising the Jack OS (supplied with the book). The chain of command is as follows: The computer is programmed to first run the Sys.init. This OS function, in turn, is programmed to start running your Main.main function. This function will then call various subroutines from both the user program and from the OS, and so on.

Although the standard library of the Jack language can be extended, readers will perhaps want to hone their programming skills elsewhere. After all, we don't expect Jack to be part of your life beyond this book. Therefore, it is best to view the Jack/ Hack platform as a given environment and make the best out of it. That's precisely what programmers do when they write software for embedded devices and dedicated processors that operate in restricted environments. Instead of viewing the constrains imposed by the host platform as a problem, professionals view it as an opportunity to display their resourcefulness and ingenuity. That's why some of the best programmers in the trade were first trained on primitive computers.

9.4 Perspective

Jack is an “object-based” language, meaning that it supports objects and classes, but not inheritance. In this respect it is located somewhere between procedural languages like Pascal or C and object-oriented languages like Java or C++. Jack is certainly more “clunky” than any of these industrial-strength programming languages. However, its basic syntax and semantics are not very different from those of modern languages.

Some features of the Jack language leave much to be desired. For example, its primitive type system is, well, rather primitive. Moreover, it is a weakly typed language, meaning that type conformity in assignments and operations is not strictly enforced. Also, one may wonder why the Jack syntax includes keywords like do and let, why curly brackets must be used even in single statement blocks, and why the language does not enforce a formal operator priority.

Well, all these deviations from normal programming languages were introduced into Jack with one purpose: to allow the development of elegant and simple Jack compilers, as we will do in the next two chapters. For example, when parsing a statement (in any language), it is much easier to handle the code if the first token of the statement indicates which statement we’re in. That’s why the Jack syntax includes the do and let keywords, and so on. Thus, although Jack’s simplicity may be a nuisance when writing a Jack application, you will probably be quite grateful for it while writing the Jack compiler in the next two chapters.

Most modern languages are deployed with standard libraries, and so is Jack. As in Java and C#, this library can also be viewed as an interface to a simple and portable operating system. In the Jack-Hack platform, the services supplied by this OS are extremely minimal. They include no concurrency to support multi-threading or multi-processing, no file system to support permanent storage, and no communication. At the same time, the Jack OS provides some classical OS services like graphic and textual I/O (in very basic forms), standard implementation of strings, and standard memory allocation and de-allocation. Additionally, the Jack OS implements various mathematical functions, including multiplication and division, normally implemented in hardware. We return to these issues in chapter 12, where we will build this simple operating system as the last module in our computer system.

9.5 Project

Objective The hidden agenda of this project is to get acquainted with the Jack language, for two purposes: writing the Jack compiler in Projects 10 and 11, and writing the Jack operating system in Project 12.

Contract Adopt or invent an application idea, for example, a simple computer game or some other interactive program. Then design and build the application.

Resources You will need three tools: the Jack compiler, to translate your program into a set of .vm files, the VM emulator, to run and test your translated program, and the Jack Operating System.

The Jack OS The Jack Operating System is available as a set of .vm files. These files constitute an implementation of the standard library of the Jack programming language. In order for any Jack program to execute properly, the compiled .vm files of the program must reside in a directory that also contains all the .vm files of the Jack OS. When an OS-oriented error is detected by the Jack OS, it displays a numeric error code (rather than text, which wastes precious memory space). A list of all the currently supported error codes and their textual descriptions can be found in the file projects/09/OSerrors.txt.

Compiling and Running a Jack Program

0. Each program must be stored in a separate directory, say Xxx. Start by creating this directory, then copy all the files from tools/OS into it.

1. Write your Jack program—a set of one or more Jack classes—each stored in a separate ClassName.jack text file. Put all these . jack files in the Xxx directory.

2. Compile your program using the supplied Jack compiler. This is best done by applying the compiler to the name of the program directory (Xxx). This will cause the compiler to translate all the . jack classes found in the directory into corresponding . vm files. If a compilation error is reported, debug the program and recompile Xxx until no error messages are issued.

3. At this point the program directory should contain three sets of files: (i) your source . jack files, (ii) the compiled .vm files, one for each of your . jack class files, and (iii) additional .vm files, comprising the supplied Jack OS. To test the compiled program, invoke the VM emulator and load the entire Xxx program directory. Then run the program. In case of run-time errors or undesired program behavior, fix the program and go to stage 2.

A Sample Jack Program The book’s software suite includes a complete example of a Jack application, stored in projects/09/Square. This directory contains the source Jack code of three classes comprising a simple interactive game.