the symbol table is updated, low-level code is generated, and that's it. The objects will be constructed and bound to the variables only during *run-time*, that is, if and when the compiled code will be executed.

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

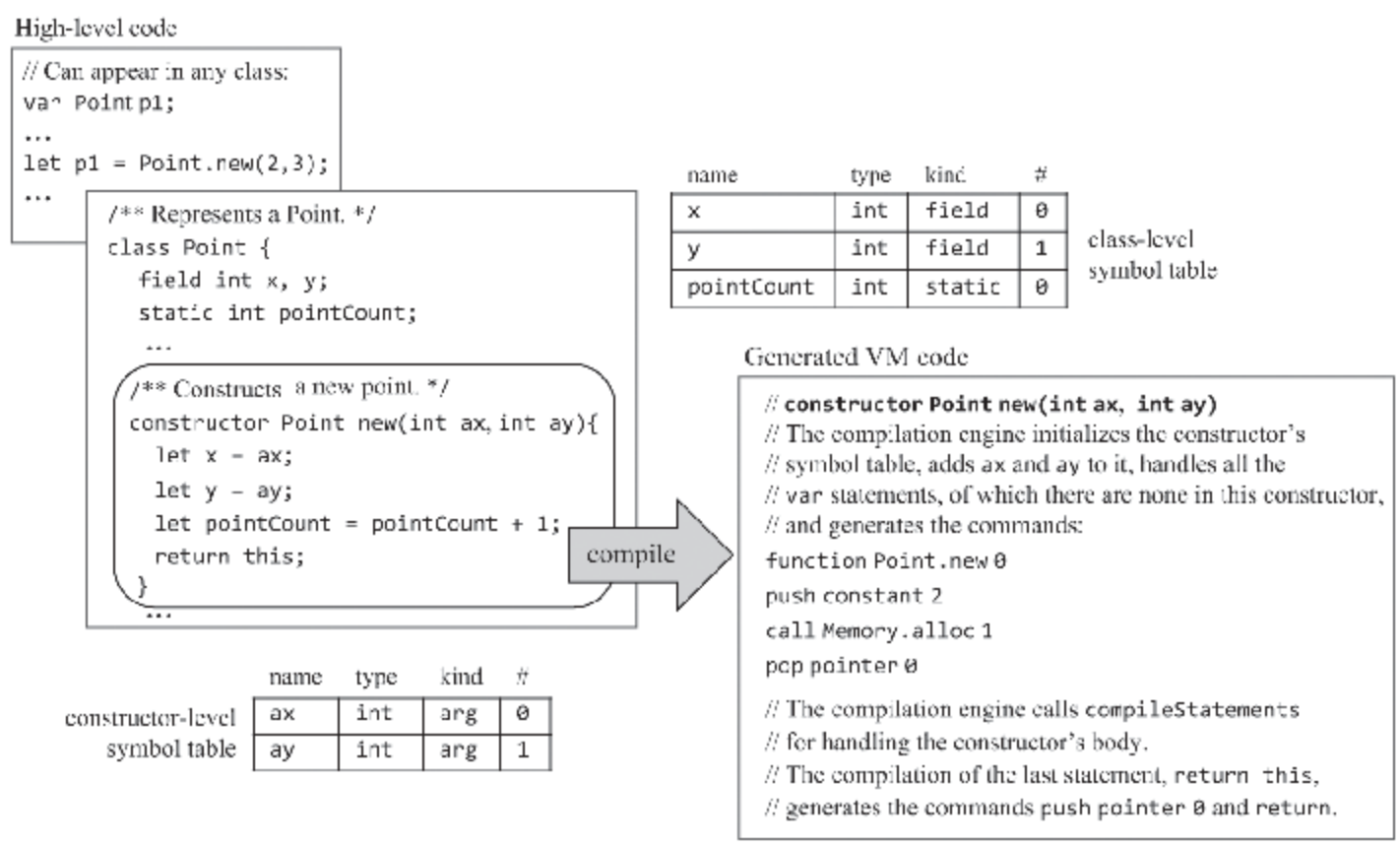


Figure 11.8 Object construction: the constructor's perspective.

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

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

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

Once the this segment is properly aligned, we can proceed to generate code easily. For example, when the compileLet routine is called to handle the statement let x = ax, it searches the symbol tables, resolving x to this 0 and ax to argument 0. Thus, compileLet generates the commands push argument 0, followed by pop this 0. The latter command rests on the assumption that the this segment is properly aligned with the base address of the new object, as indeed was done when we had set pointer 0 (actually, THIS) to the base address returned by alloc. This one-time initialization ensures that all the subsequent push / pop this *i* commands will end up hitting the right targets in the RAM (more accurately, in the *heap*). We hope that the intricate beauty of this contract is not lost on the reader.

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

that's what constructors are expected to do: create an object and return its handle to the caller.

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

11.1.5.2 Compiling Methods

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

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

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

are said to be *data-driven*. Among other things, the object-oriented programming style is based on the assumption that objects know how to take care of themselves. For example, a Point object knows how to compute the distance between it and another Point object. Said otherwise, the distance operation is *encapsulated* within the definition of being a Point.

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

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

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

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

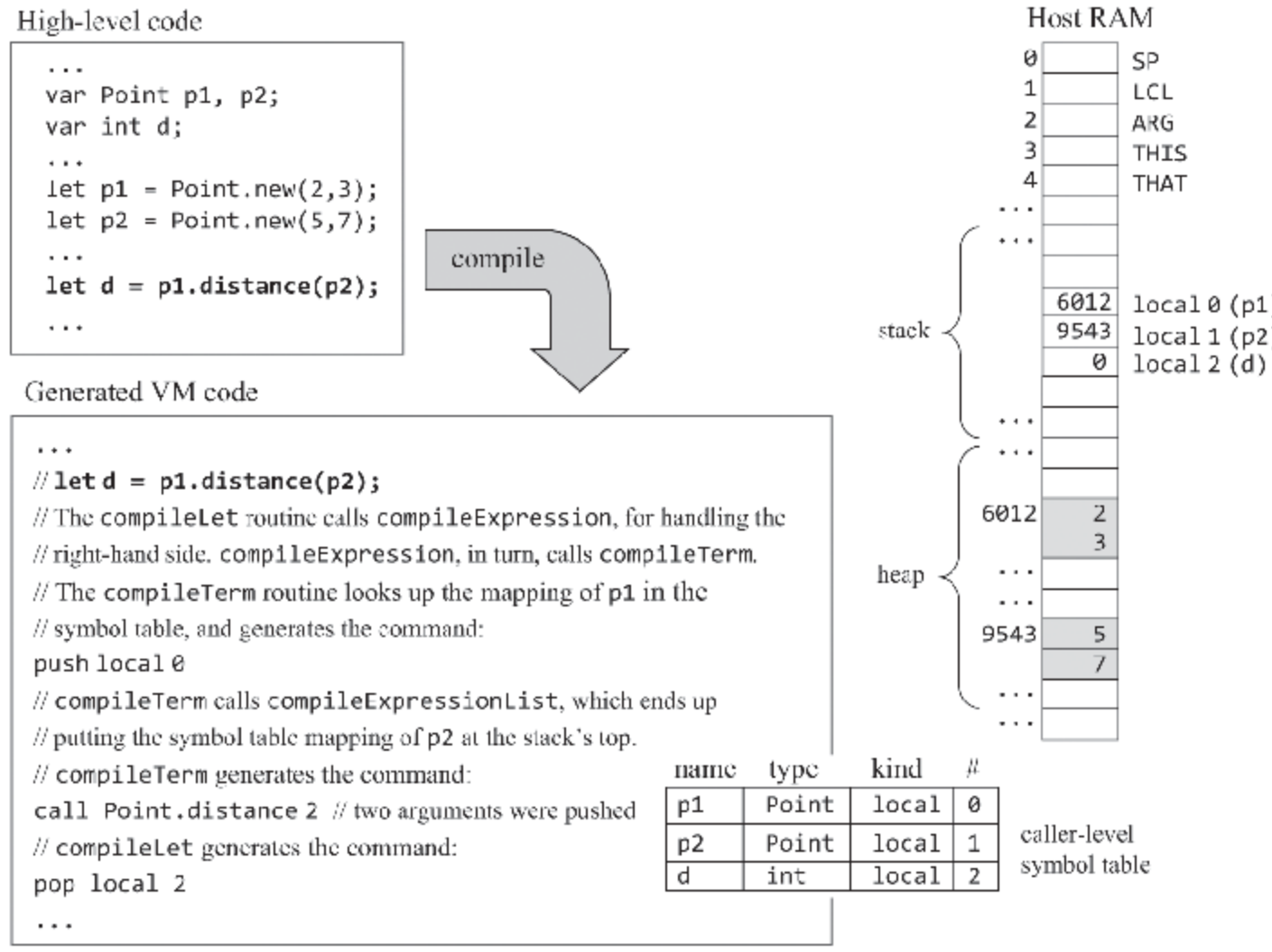


Figure 11.9 Compiling method calls: the caller's perspective.

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

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

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

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

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

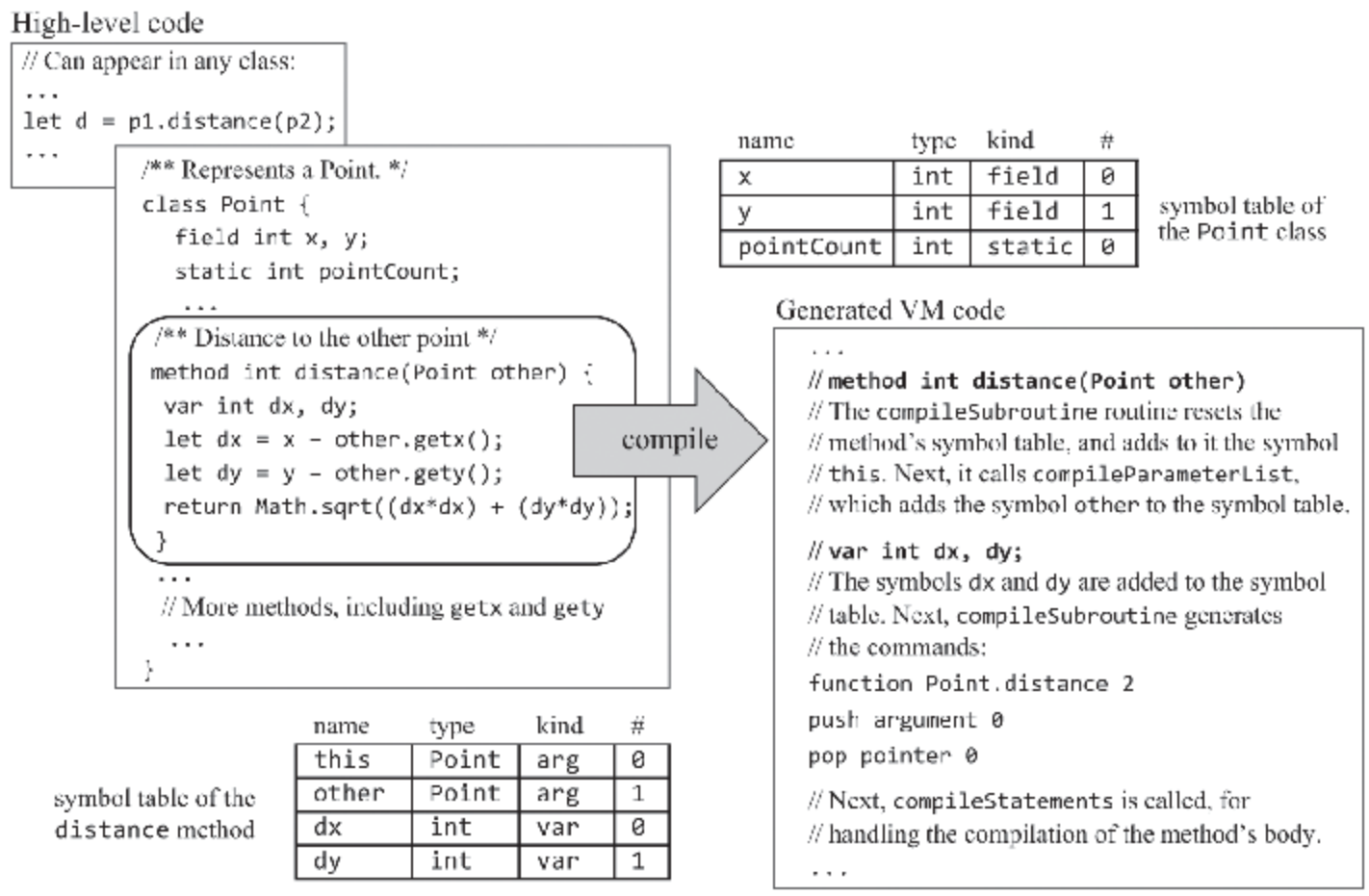


Figure 11.10 Compiling methods: the callee's perspective.

The example starts at the top left of figure 11.10, where the caller's code makes the method call `p1.distance(p2)`. Turning our attention to the compiled version of the callee, note that the code proper starts with `push argument 0`, followed by `pop pointer 0`. These commands set the method's `THIS` pointer to

the value of argument 0, which, by virtue of the method calling contract, contains the base address of the object on which the method was called to operate. Thus, from this point onward, the method's this segment is properly aligned with the base address of the target object, making every push / pop this *i* command properly aligned as well. For example, the expression x—other.getx () will be compiled into push this 0, push argument 1, call Point.getx 1, sub. Since we started the compiled method code by setting THIS to the base address of the called object, we are guaranteed that this 0 (and any other reference this *i*) will hit the mark, targeting the right field of the right object.

11.1.6 Compiling Arrays

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

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

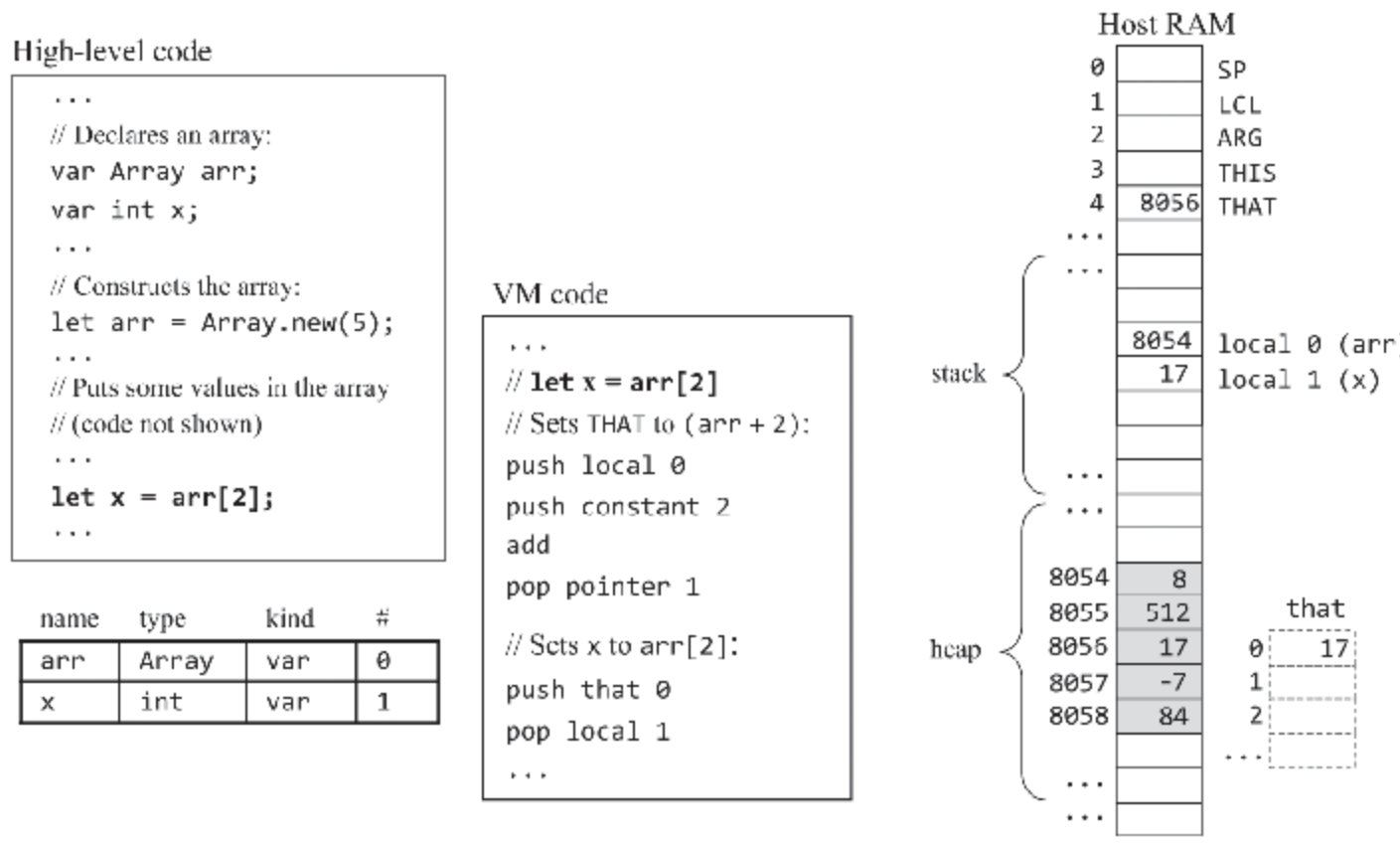


Figure 11.11 Array access using VM commands.

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

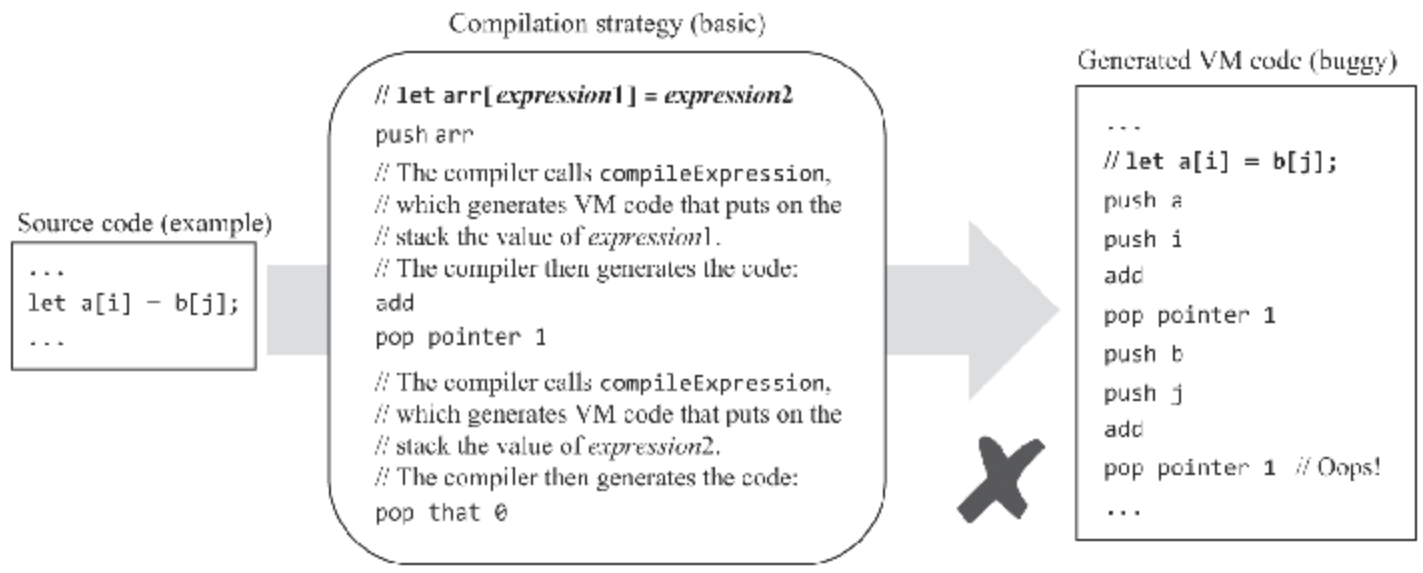


Figure 11.12 Basic compilation strategy for arrays, and an example of the bugs that it can generate. In this particular case, the value stored in pointer 1 is overridden, and the address of a[i] is lost.

The good news is that this flawed compilation strategy can be easily fixed to compile correctly any instance of let arr[*expression*1] = *expression*2. As before, we start by generating the command push arr, calling compileExpression, and generating the command add. This sequence puts the target address (arr + *expression*1) at the stack's top. Next, we call

compileExpression, which will end up putting at the stack's top the value of *expression*2. At this point we save this value—we can do it using pop temp 0. This operation has the nice side effect of making (arr + *expression*1) the top stack element. Thus we can now pop pointer 1, push temp 0, and pop that 0. This little fix, along with the recursive nature of the compileExpression routine, makes this compilation strategy capable of handling let arr[*expression*1] = *expression*2 statements of any recursive complexity, such as, say, let a[b[i]+a[j+b[a[3]]]]=b[b[j]+2].

In closing, several things make the compilation of Jack arrays relatively simple. First, Jack arrays are not typed; rather, they are designed to store 16-bit values, with no restrictions. Second, all primitive data types in Jack are 16-bit wide, all addresses are 16-bit wide, and so is the RAM's word width. In strongly typed programming languages, and in languages where this one-to-one correspondence cannot be guaranteed, the compilation of arrays requires more work.

11.2 Specification

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

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

prompt> JackCompiler *source*

where *source* is either a file name of the form *Xxx*.jack (the extension is mandatory) or the name of a folder (in which case there is no extension) containing one or more .jack files. The file/folder name may contain a file path. If no path is specified, the compiler operates on the current folder. For each *Xxx*.jack file, the compiler creates an output file *Xxx*.vm and writes the

VM commands into it. The output file is created in the same folder as the input file. If there is a file by this name in the folder, it will be overwritten.

11.3 Implementation

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

11.3.1 Standard Mapping over the Virtual Machine

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

Naming Files and Functions

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

Mapping Variables

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

Mapping Object Fields

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

Mapping Array Elements

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

Mapping Constants

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

11.3.2 Implementation Guidelines

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

Handling Identifiers

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

Compiling Expressions

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

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

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

Compiling Strings

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

Compiling Function Calls and Constructor Calls

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

Compiling Method Calls

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

Compiling do Statements

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

Compiling Classes

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

Compiling Subroutines

- When starting to compile a subroutine (*constructor*, *function*, or *method*), the compiler initializes the subroutine's symbol table. If the subroutine is a *method*, the compiler adds to the symbol table the mapping <this, *className*, arg, 0>.
- Next, the compiler adds to the symbol table all the parameters, if any, declared in the subroutine's parameter list. Next, the compiler handles all the var declarations, if any, by adding to the symbol table all the subroutine's local variables.

- At this stage the compiler starts generating code, beginning with the command function *className.subroutineName nVars*, where *nVars* is the number of local variables in the subroutine.
- If the subroutine is a *method*, the compiler generates the code push argument 0, pop pointer 0. This sequence aligns the virtual memory segment this with the base address of the object on which the method was called.

Compiling Constructors

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

Compiling Void Methods and Void Functions

Every VM function is expected to push a value onto the stack before returning. When compiling a void Jack method or function, the convention is to end the generated code with push constant 0, return.

Compiling Arrays

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

The Operating System

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

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

Native OS Implementation

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

Emulated OS Implementation

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

11.3.3 Software Architecture

The proposed compiler architecture builds upon the syntax analyzer described in chapter 10. Specifically, we propose to gradually evolve the syntax analyzer into a full-scale compiler, using the following modules:

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

The JackCompiler

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

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

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

The JackTokenizer

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

The SymbolTable

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

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

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

The VMWriter

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

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

The CompilationEngine

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

The CompilationEngine gets its input from a JackTokenizer and uses a VMWriter for writing the VM code output (instead of the XML produced in project 10). The output is generated by a series of compile*xxx* routines, each designed to handle the compilation of a specific Jack language construct *xxx* (for example, compileWhile generates the VM code that realizes while statements). The contract between these routines is as follows: Each

compile*xxx* routine gets from the input and handles all the tokens that make up *xxx*, advances the tokenizer exactly beyond these tokens, and emits to the output VM code effecting the semantics of *xxx*. If *xxx* is a part of an expression, and thus has a value, the emitted VM code should compute this value and leave it at the top of the stack. As a rule, each compile*xxx* routine is called only if the current token is *xxx*. Since the first token in a valid .jack file must be the keyword class, the compilation process starts by calling the routine compileClass.

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

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

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

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

11.4 Project

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

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

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

Implementation Stages

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

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

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

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

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

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

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

Stage 2: Code generation: We provide six application programs, designed to gradually unit-test the code generation capabilities of your Jack compiler. We advise developing, and testing, your evolving compiler on the test programs in the given order. This way, you will be implicitly guided to

build the compiler's code generation capabilities in sensible stages, according to the demands presented by each test program.

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

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

Test Programs

Seven: Tests how the compiler handles a simple program containing an arithmetic expression with integer constants, a `do` statement, and a `return` statement. Specifically, the program computes the expression `1+(2*3)` and prints its value at the top left of the screen. To test whether your compiler has translated the program correctly, run the translated code in the VM emulator, and verify that it displays `7` correctly.

ConvertToBin: Tests how the compiler handles all the procedural elements of the Jack language: expressions (without arrays or method calls), functions, and the statements `if`, `while`, `do`, `let`, and `return`. The program does not test the handling of methods, constructors, arrays, strings, static variables, and field variables. Specifically, the program gets a 16-bit decimal value from

RAM[8000], converts it to binary, and stores the individual bits in RAM[8001…8016] (each location will contain 0 or 1). Before the conversion starts, the program initializes RAM[8001…8016] to –1. To test whether your compiler has translated the program correctly, load the translated code into the VM emulator, and proceed as follows:

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

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

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

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

Pong: A complete test of how the compiler handles an object-based application, including the handling of objects and static variables. In the classical Pong game, a ball is moving randomly, bouncing off the edges of the screen. The user tries to hit the ball with a small paddle that can be moved by pressing the keyboard's left and right arrow keys. Each time the paddle hits the ball, the user scores a point and the paddle shrinks a little, making the game increasingly more challenging. If the user misses and the

ball hits the bottom the game is over. To test whether your compiler has translated this program correctly, run the translated code in the VM emulator and play the game. Make sure to score some points to test the part of the program that displays the score on the screen.

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

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

11.5 Perspective

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

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

Another significant simplification is that the Jack language does not support *inheritance*. In languages that support inheritance, the handling of

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

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

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

Many other simplifications of the Jack language are not significant and can be relaxed with little effort. For example, one can easily extend the language with for and switch statements. Likewise, one can add the capability to assign character constants like 'c' to char type variables, which is presently not supported by the language.

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

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

12 Operating System

Civilization progresses by extending the number of operations that we can perform without thinking about them.

—Alfred North Whitehead, *Introduction to Mathematics* (1911)

In chapters 1–6 we described and built a general-purpose hardware architecture. In chapters 7–11 we developed a software hierarchy that makes the hardware usable, culminating in the construction of a modern, object-based language. Other high-level programming languages can be specified and implemented on top of the hardware platform, each requiring its own compiler.

The last major piece missing in this puzzle is an *operating system*. The OS is designed to close gaps between the computer's hardware and software, making the computer system more accessible to programmers, compilers, and users. For example, to display the text `Hello World` on the screen, several hundred pixels must be drawn at specific screen locations. This can be done by consulting the hardware specification and writing code that turns bits on and off in selected RAM locations. Clearly, high-level programmers expect a better interface. They want to write `print ("Hello World")` and let someone else worry about the details. That's where the operating system enters the picture.

Throughout this chapter, the term *operating system* is used rather loosely. Our OS is minimal, aiming at (i) encapsulating low-level hardware-specific services in high-level programmer-friendly software services and (ii) extending high-level languages with commonly used functions and abstract data types. The dividing line between an *operating system* in this sense and a *standard class library* is not clear. Indeed, modern programming

languages pack many standard operating system services like graphics, memory management, multitasking, and numerous other extensions into what is known as the language's *standard class library*. Following this model, the Jack OS is packaged as a collection of supporting classes, each providing a set of related services via Jack subroutine calls. The complete OS API is given in appendix 6.

High-level programmers expect the OS to deliver its services through well-designed interfaces that hide the gory hardware details from their application programs. To do so, the OS code must operate close to the hardware, manipulating memory, input/output, and processing devices almost directly. Further, because the OS supports the execution of every program that runs on the computer, it must be highly efficient. For example, application programs create and dispose objects and arrays all the time. Therefore, we better do it quickly and economically. Any gain in the time- and space-efficiency of an enabling OS service can impact dramatically the performance of all the application programs that depend on it.

Operating systems are usually written in a high-level language and compiled into binary form. Our OS is no exception—it is written in Jack, just like Unix was written in C. Like the C language, Jack was designed with sufficient "lowness" in it, permitting an intimate closeness to the hardware when needed.

The chapter starts with a relatively long Background section that presents key algorithms normally used is OS implementations. These include mathematical operations, string manipulations, memory management, text and graphics output, and keyboard input. This algorithmic introduction is followed by a Specification section describing the Jack OS, and an Implementation section that offers guidance on how to build the OS using the algorithms presented. As usual, the Project section provides the necessary guidelines and materials for gradually constructing and unit-testing the entire OS.

The chapter embeds key lessons in system-oriented software engineering and in computer science. On the one hand, we describe programming techniques for developing low-level system services, as well as "programming at the large" techniques for integrating and streamlining the OS services. On the other hand, we present a set of elegant and highly efficient algorithms, each being a computer science gem.

12.1 Background

Computers are typically connected to a variety of input/output devices such as a keyboard, screen, mouse, mass storage, network interface card, microphone, speakers, and more. Each of these I/O devices has its own electromechanical idiosyncrasies; thus, reading and writing data on them involves many technical details. High-level languages abstract away these details by offering high-level abstractions like `let n = Keyboard.readInt("Enter a number:")`. Let's delve into what should be done in order to realize this seemingly simple data-entry operation.

First, we engage the user by displaying the prompt `Enter a number:`. This entails creating a `String` object and initializing it to the array of `char` values `'E'`, `'n'`, `'t'`, …, and so on. Next, we have to render this string on the screen, one character at a time, while updating the *cursor* position for keeping track of where the next character should be physically displayed. After displaying the `Enter a number:` prompt, we have to stage a loop that waits until the user will oblige to press some keys on the keyboard—hopefully keys that represent digits. This requires knowing how to (i) capture a keystroke, (ii) get a single character input, (iii) append these characters to a string, and (iv) convert the string into an integer value.

If what has been elaborated so far sounds arduous, the reader should know that we were actually quite gentle, sweeping many gory details under the rug. For example, what exactly is meant by "creating a string object," "displaying a character on the screen," and "getting a multicharacter input"?

Let's start with "creating a string object." String objects don't pop out of thin air, fully formed. Each time we want to create an object, we must find available space for representing the object in the RAM, mark this space as used, and remember to free it when the object is no longer needed. Proceeding to the "display a character" abstraction, note that characters cannot be displayed. The only things that can be physically displayed are individual pixels. Thus, we have to figure out what is the character's *font*, compute where the bits that represent the font image can be found in the screen memory map, and then turn these bits on and off as needed. Finally, to "get a multicharacter input," we have to enter a loop that not only listens to the keyboard and accumulates characters as they come along but also

allows the user to backspace, delete, and retype characters, not to mention the need to echo each of these actions on the screen for visual feedback.

The agent that takes care of this elaborate behind-the-scenes work is the operating system. The execution of the statement `let n = Keyboard.readInt("Enter a number:")` entails many OS function calls, dealing with diverse issues like memory allocation, input driving, output driving, and string processing. Compilers use the OS services abstractly by injecting OS function calls into the compiled code, as we saw in the previous chapter. In this chapter we explore *how these functions are actually realized*. Of course, what we have surveyed so far is just a small subset of the OS responsibilities. For example, we didn't mention mathematical operations, graphical output, and other commonly needed services. The good news is that a well-written OS can integrate these diverse and seemingly unrelated tasks in an elegant and efficient way, using cool algorithms and data structures. That's what this chapter is all about.

12.1.1 Mathematical Operations

The four arithmetic operations *addition*, *subtraction*, *multiplication*, and *division* lie at the core of almost every computer program. If a loop that executes a million times contains expressions that use some of these operations, they'd better be implemented efficiently.

Normally, addition is implemented in hardware, at the ALU level, and subtraction is gained freely, courtesy of the two's complement method. Other arithmetic operations can be handled either by hardware or by software, depending on cost/performance considerations. We now turn to present efficient algorithms for computing multiplication, division, and square roots. These algorithms lend themselves to both software and hardware implementations.

Efficiency First

Mathematical algorithms are made to operate on n-bit values, n typically being 16, 32, or 64 bits, depending on the operands' data types. As a rule, we seek algorithms whose running time is a polynomial function of this

word size n. Algorithms whose running time depends on the *values* of n-bit numbers are unacceptable, since these values are exponential in n. For example, suppose we implement the multiplication operation $x \times y$ naïvely, using the repeated addition algorithm `for i = 1 … y {sum = sum + x}`. If y is 64-bit wide, its value may well be greater than 9,000,000,000,000,000,000, implying that the loop may run for billions of years before terminating.

In sharp contrast, the running times of the multiplication, division, and square root algorithms that we present below depend not to the n-bit *values* on which they are called to operate, which may be as large as 2^n, but rather on n, the number of their bits. When it comes to efficiency of arithmetic operations, that's the best that we can possibly hope for.

We will use the *Big-O* notation, $O(n)$, to describe a running time which is "in the order of magnitude of n." The running time of all the arithmetic algorithms that we present in this chapter is $O(n)$, where n is the bit width of the inputs.

Multiplication

Consider the standard multiplication method taught in elementary school. To compute 356 times 73, we line up the two numbers one on top of the other, right-justified. Next, we multiply 356 by 3. Next, we shift 356 to the left one position, and multiply 3560 by 7 (which is the same as multiplying 356 by 70). Finally, we sum up the columns and obtain the result. This procedure is based on the insight that $356 \times 73 = 356 \times 70 + 356 \times 3$. The binary version of this procedure is illustrated in figure 12.1, using another example.

$x = 27 =$	... 0 0 0 0 **1 1 0 1 1**	
$y = 9 =$	... 0 0 0 0 0 **1 0 0 1**	i-th bit of y
	... 0 0 0 0 1 1 0 1 1	1
	... 0 0 0 1 1 0 1 1 0	0
	... 0 0 1 1 0 1 1 0 0	0
	... 0 1 1 0 1 1 0 0 0	1
$x * y = 243 =$	... 0 **1 1 1 1 0 0 1 1**	sum

```
// Returns x * y, where x, y ≥ 0.
multiply(x, y):
  sum = 0
  shiftedx = x
  for i = 0 ... n – 1 do
      if ((i-th bit of y) == 1)
          sum = sum + shiftedx
      shiftedx = 2 * shiftedx
  return sum
```

Figure 12.1 Multiplication algorithm.

Notation note: The algorithms presented in this chapter are written in a self-explanatory pseudocode syntax. We use indentation to mark blocks of code, obviating the need for curly brackets or begin/end keywords. For example, in figure 12.1, $sum = sum + shiftedx$ belongs to the single-statement body of the if logic, and $shiftedx = 2 * shiftedx$ ends the two-statement body of the for logic.

Let's inspect the multiplication procedure illustrated at the left of figure 12.1. For each i-th bit of y, we shift x i times to the left (same as multiplying x by 2^i). Next, we look at the i-th bit of y: If it is 1, we add the shifted x to an accumulator; otherwise, we do nothing. The algorithm shown on the right formalizes this procedure. Note that 2 * *shiftedx* can be computed efficiently either by left-shifting the bitwise representation of *shiftedx* or by adding *shiftedx* to itself. Either operation lends itself to primitive hardware operations.

Running time: The multiplication algorithm performs n iterations, where n is the bit width of the y input. In each iteration, the algorithm performs a few addition and comparison operations. It follows that the total running time of the algorithm is $a + b \cdot n$, where a is the time it takes to initialize a few variables, and b is the time it takes to perform a few addition and comparison operations. Formally, the algorithm's running time is $O(n)$, where n is the bit width of the inputs.

To reiterate, the running time of this $x \times y$ algorithm does not depend on the *values* of the x and y inputs; rather, it depends on the *bit width* of the inputs. In computers, the bit width is normally a small fixed constant like 16 (short), 32 (int), or 64 (long), depending on the data types of the inputs. In the Hack platform, the bit width of all data types is 16. If we assume that each iteration of the multiplication algorithm entails about ten Hack machine instructions, it follows that each multiplication operation will require at most 160 clock cycles, irrespective of the size of the inputs. In contrast, algorithms whose running time is proportional not to the bit width but rather to the values of the inputs will require $10 \cdot 2^{16} = 655{,}360$ clock cycles.

Division

The naïve way to compute the division of two n-bit numbers x / y is to count how many times y can be subtracted from x until the remainder becomes less than y. The running time of this algorithm is proportional to the value of the dividend x and thus is unacceptably exponential in the number of bits n.

To speed things up, we can try to subtract large chunks of y's from x in each iteration. For example, suppose we have to divide 175 by 3. We start by asking: What is the largest number, $x = (90, 80, 70, \ldots, 20, 10)$, so that $3 \cdot x \leq 175$? The answer is 50. In other words, we managed to subtract fifty 3's from 175, shaving fifty iterations from the naïve approach. This accelerated subtraction leaves a remainder of $175 - 3 \cdot 50 = 25$. Moving along, we now ask: What is the largest number, $x = (9, 8, 7, \ldots 2, 1)$, so that $3 \cdot x \leq 25$? The answer is 8, so we managed to make eight additional subtractions of 3, and the answer, so far, is $50 + 8 = 58$.The remainder is $25 - 3 \cdot 8 = 1$, which is less than 3, so we stop the process and announce that $175/3 = 58$ with a remainder of 1.

This technique is the rationale behind the dreaded school procedure known as *long division*. The binary version of this algorithm is identical, except that instead of accelerating the subtraction using powers of 10 we use powers of 2. The algorithm performs n iterations, n being the number of digits in the dividend, and each iteration entails a few multiplication (actually, shifting), comparison, and subtraction operations. Once again, we have an x/yalgorithm whose running time does not depend on the values of x and y. Rather, the running time is $O(n)$, where n is the bit width of the inputs.

Writing down this algorithm as we have done for multiplication is an easy exercise. To make things interesting, figure 12.2 presents another division algorithm which is as efficient, but more elegant and easier to implement.

```
// Returns the integer division x / y,
// where x ≥ 0 and y > 0.
divide(x, y):
    if (y > x) return 0
    q = divide (x, 2 * y)
    if ((x – 2 * q * y) < y)
        return 2 * q
    else
        return 2 * q + 1
```

Figure 12.2 Division algorithm.

Suppose we have to divide 480 by 17. The algorithm shown in figure 12.2 is based on the insight $480/17 = 2 \cdot (240/17) = 2 \cdot (2 \cdot (120/17)) = 2 \cdot (2 \cdot (2 \cdot (60/17))) = \cdots$, and so on. The depth of this recursion is bounded by the number of times y can be multiplied by 2 before reaching x. This also happens to be, at most, the number of bits required to represent x. Thus, the running time of this algorithm is $O(n)$, where n is the bit width of the inputs.

One snag in this algorithm is that each multiplication operation also requires $O(n)$ operations. However, an inspection of the algorithm's logic reveals that the value of the expression $(2 * q * y)$ can be computed without multiplication. Instead, it can be obtained from its value in the previous recursion level, using addition.

Square Root

Square roots can be computed efficiently in a number of different ways, for example, using the Newton-Raphson method or a Taylor series expansion. For our purpose, though, a simpler algorithm will suffice. The square root function $y = \sqrt{x}$ has two attractive properties. First, it is monotonically increasing. Second, its inverse function, $y = x^2$, is a function that we already know how to compute efficiently—multiplication. Taken together, these properties imply that we have all we need to compute square roots efficiently, using a form of *binary search*. Figure 12.3 gives the details.

```
// Computes the integer part of y = √x
// Strategy: finds an integey such that y^2 ≤ x < (y+1)^2  (for 0 ≤ x < 2^n)
// by performing binary search in the range 0 ... 2^(n/2) – 1
sqrt(x):
   y = 0
   for  j = (n/2 – 1) ... 0  do
      if (y+2^j)^2 ≤ x then y = y+2^j
   return y
```

Figure 12.3 Square root algorithm.

Since the number of iterations in the binary search that the algorithm performs is bound by $n / 2$ where n is the number of bits in x, the algorithm's running time is $O(n)$.

To sum up this section about mathematical operations, we presented algorithms for computing multiplication, division, and square root. The running time of each of the algorithms is $O(n)$, where n is the bit width of the inputs. We also observed that in computers, n is a small constant like 16, 32, or 64. Therefore, every addition, subtraction, multiplication, and division operation can be carried out swiftly, in a predictable time that is unaffected by the magnitude of the inputs.

12.1.2 Strings

In addition to primitive data types, most programming languages feature a *string* type designed to represent sequences of characters like "Loading game …" and "QUIT". Typically, the string abstraction is supplied by a String class that is part of the standard class library that supports the language. This is also the approach taken by Jack.

All the string constants that appear in Jack programs are implemented as String objects. The String class, whose API is documented in appendix 6, features various string processing methods like appending a character to the string, deleting the last character, and so on. These services are not difficult to implement, as we'll describe later in the chapter. The more challenging String methods are those that convert integer values to strings and strings of digit characters to integer values. We now turn to discuss algorithms that carry out these operations.

String representation of numbers: Computers represent numbers internally using binary codes. Yet humans are used to dealing with numbers that are written in decimal notation. Thus, when humans have to read or input numbers, *and only then*, a conversion to or from decimal notation must be performed. When such numbers are captured from an input device like a keyboard, or rendered on an output device like a screen, they are cast as strings of characters, each representing one of the digits 0 to 9. The subset of relevant characters is:

Character:	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
Character code:	48	49	50	51	52	53	54	55	56	57

(The complete Hack character set is given in appendix 5). We see that digit characters can be easily converted into the integers that they represent, and vice versa. The integer value of character c, where $48 \le c \le 57$, is $c - 48$. Conversely, the character code of the integer x, where $0 \le x \le 9$, is $x + 48$.

Once we know how to handle single-digit characters, we can develop algorithms for converting any integer into a string and any string of digit characters into the corresponding integer. These conversion algorithms can be based on either iterative or recursive logic, so figure 12.4 presents one of each.

int to string:

```
// Returns the string representation of
// a nonnegative integer.
int2String(val):
   lastDigit = val % 10
   c = character representing lastDigit
   if (val < 10)
     return  c  (as a string)
   else
     return  int2String(val / 10).appendChar(c)
```

string to int:

```
// Returns the integer value of a string
// of digit characters, assuming that str[0]
// represents the most significant digit.
string2Int(str):
    val = 0
    for ( i = 0 . . . str.length() ) do
        d = integer value of str.charAt(i)
        val = val * 10 + d
    return  val
```

Figure 12.4 String-integer conversions. (appendChar, length, and charAt are String class methods.)

It is easy to infer from figure 12.4 that the running times of the int2String and string2Int algorithms are $O(n)$, where n is the number of the digit-characters in the input.

12.1.3 Memory Management

Each time a program creates a new array or a new object, a memory block of a certain size must be allocated for representing the new array or object. And when the array or object is no longer needed, its RAM space may be recycled. These chores are done by two classical OS functions called alloc and deAlloc. These functions are used by compilers when generating low-level code for handling constructors and destructors, as well as by high-level programmers, as needed.

The memory blocks for representing arrays and objects are carved from, and recycled back into, a designated RAM area called a *heap*. The agent responsible for managing this resource is the operating system. When the OS starts running, it initializes a pointer named heapBase, containing the heap's base address in the RAM (in Jack, the heap starts just after the stack's end, with heapBase=2048). We'll present two heap management algorithms: basic and improved.

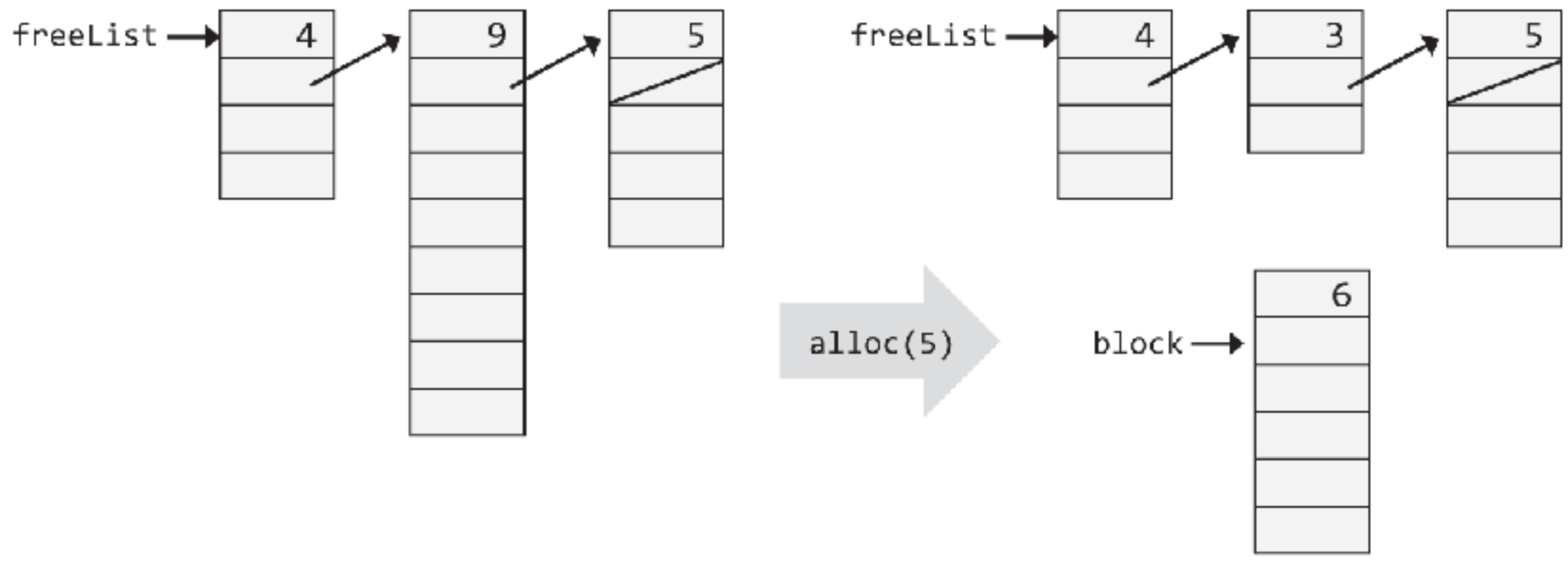
Memory allocation algorithm (basic): The data structure that this algorithm manages is a single pointer, named free, which points to the beginning of the heap segment that was not yet allocated. See figure 12.5a for the details.

```
init():
    free = heapBase

// Allocates a memory block of size words.
alloc(size):
    block = free
    free = free + size
    return block

// Frees the memory space of the given object.
deAlloc(object):
    do nothing
```

Figure 12.5a Memory allocation algorithm (basic).

The basic heap management scheme is clearly wasteful, as it never reclaims any memory space. But, if your application programs use only a few small objects and arrays, and not too many strings, you may get away with it.

Memory allocation algorithm (improved): This algorithm manages a linked list of available memory segments, called freeList (see figure 12.5b). Each segment in the list begins with two housekeeping fields: the segment's *length* and a pointer to the *next* segment in the list.




```
init():
    freeList = heapBase
    freeList.size = heapSize
    freeList.next = 0

// Allocates a memory block of size words.
alloc(size):
    search freeList using best-fit or first-fit heuristics
        to obtain a segment with segment.size ≥ size + 2
    if no such segment is found, return failure
        (or attempt defragmentation)
    block = base address of the found space
    update the freeList and the fields of block
        to account for the allocation
    return block

// Frees the memory space of the given object.
deAlloc(object):
    append object to the end of the freeList
```

Figure 12.5b Memory allocation algorithm (improved).

When asked to allocate a memory block of a given size, the algorithm has to search the freeList for a suitable segment. There are two heuristics for doing this search. *Best-fit* finds the shortest segment that is long enough for representing the required size, while *first-fit* finds the first segment that is long enough. Once a suitable segment has been found, the required memory block is carved from it (the location just before the beginning of the returned block, block[−1], is reserved to hold its length, to be used during deallocation).

Next, the *length* of this segment is updated in the freeList, reflecting the length of the part that remained after the allocation. If no memory was left

in the segment, or if the remaining part is practically too small, the entire segment is eliminated from the freeList.

When asked to reclaim the memory block of an unused object, the algorithm appends the deallocated block to the end of the freeList.

Dynamic memory allocation algorithms like the one shown in figure 12.5b may create block fragmentation problems. Hence, a *defragmentation* operation should be considered, that is, merging memory areas that are physically adjacent in memory but logically split into different segments in the freeList. The defragmentation can be done each time an object is deallocated, when alloc() fails to find a block of the requested size, or according to some other, periodical ad hoc condition.

Peek and poke: We end the discussion of memory management with two simple OS functions that have nothing to do with resource allocation. Memory.peek(*addr*) returns the value of the RAM at address *addr*, and Memory.poke(*addr*,*value*) sets the word in RAM address *addr* to *value*. These functions play a role in various OS services that manipulate the memory, including graphics routines, as we now turn to discuss.

12.1.4 Graphical Output

Modern computers render graphical output like animation and video on high-resolution color screens, using optimized graphics drivers and dedicated graphical processing units (GPUs). In Nand to Tetris we abstract away most of this complexity, focusing instead on fundamental graphics-drawing algorithms and techniques.

We assume that the computer is connected to a physical black-and-white screen arranged as a grid of rows and columns, and at the intersection of each lies a pixel. By convention, the columns are numbered from left to right and the rows are numbered from top to bottom. Thus pixel (0,0) is located at the screen's top-left corner.

We assume that the screen is connected to the computer system through a *memory map*—a dedicated RAM area in which each pixel is represented by one bit. The screen is refreshed from this memory map many times per second by a process that is external to the computer. Programs that simulate the computer's operations are expected to emulate this refresh process.

The most basic operation that can be performed on the screen is drawing an individual pixel specified by (*x*,*y*) coordinates. This is done by turning the corresponding bit in the memory map on or off. Other operations like drawing a line and drawing a circle are built on top of this basic operation. The graphics package maintains a *current color* that can be set to *black* or *white*. All the drawing operations use the current color.

Pixel drawing (drawPixel): Drawing a selected pixel in screen location (*x*,*y*) is achieved by locating the corresponding bit in the memory map and setting it to the current color. Since the RAM is an *n*-bit device, this operation requires reading and writing an *n*-bit value. See figure 12.6.

```
// Sets pixel (x,y) to the current color.
drawPixel(x,y):
    Using x and y, compute the RAM address where
        the pixel is represented;
    Using Memory.peek, get the 16-bit value of
        this address;
    Using some bitwise operation, set (only) the bit
        that corresponds to the pixel to the current color;
    Using Memory.poke, write the modified 16-bit
        value "back" to the RAM address.
```

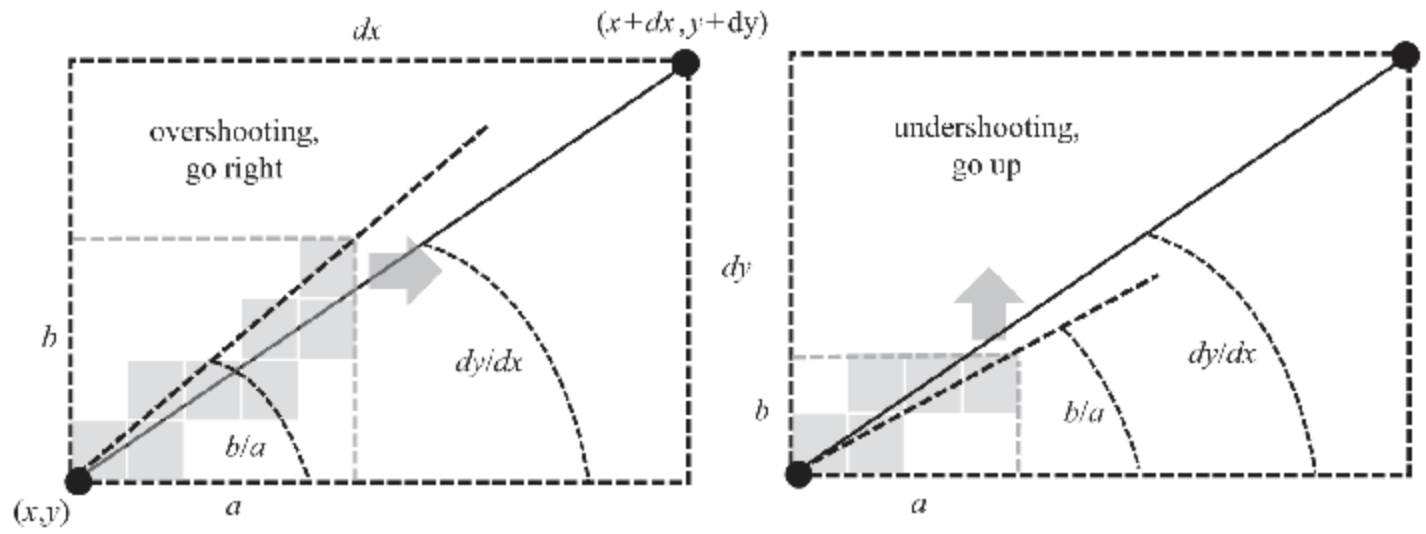
Figure 12.6 Drawing a pixel.

The memory map interface of the Hack screen is specified in section 5.2.4. This mapping should be used in order to realize the drawPixel algorithm.

Line drawing (drawLine): When asked to render a continuous "line" between two "points" on a grid made of discrete pixels, the best that we can possibly do is approximate the line by drawing a series of pixels along the imaginary line connecting the two points. The "pen" that we use for drawing the line can move in four directions only: up, down, left, and right.

Thus, the drawn line is bound to be jagged, and the only way to make it look good is to use a high-resolution screen with the tiniest possible pixels. Note, though, that the human eye, being yet another machine, also has a limited image-capturing capacity, determined by the number and type of receptor cells in the retina. Thus, high-resolution screens can fool the human brain to believe that the lines made of pixels are visibly smooth. In fact they are always jagged.

The procedure for drawing a line from ($x1$,$y1$) to ($x2$,$y2$) starts by drawing the ($x1$,$y1$) pixel and then zigzagging in the direction of ($x2$,$y2$) until that pixel is reached. See figure 12.7.




```
// Draws a line between two given points.
drawLine(x1, y1, x2, y2):
    set x to x1, y to y1, compute dx and dy.
    // a and b track how many times we
    // went right, and up, so far.
    a = 0, b = 0
    while ((a ≤ dx) and (b ≤ dy))
        drawPixel (x + a, y + b)
        // Goes right, or up:
        if (b/a > dy/dx)   { a=a+1 }
        else               { b=b+1 }
```

```
// Draws a line between two points (improved).
drawLine(x1, y1, x2, y2):
    set x to x1, y to y1, compute dx and dy.
    // a and b track how many times we
    // went right, and up, so far.
    a = 0, b = 0
    diff = 0
    while ((a < dx) and (b < dy))
        drawPixel (x + a, y + b)
        // Goes right, or up:
        if (diff < 0)  { a=a+1, diff = diff − dy }
        else           { b=b+1, diff = diff − dx }
```

involves only addition and subtraction operations

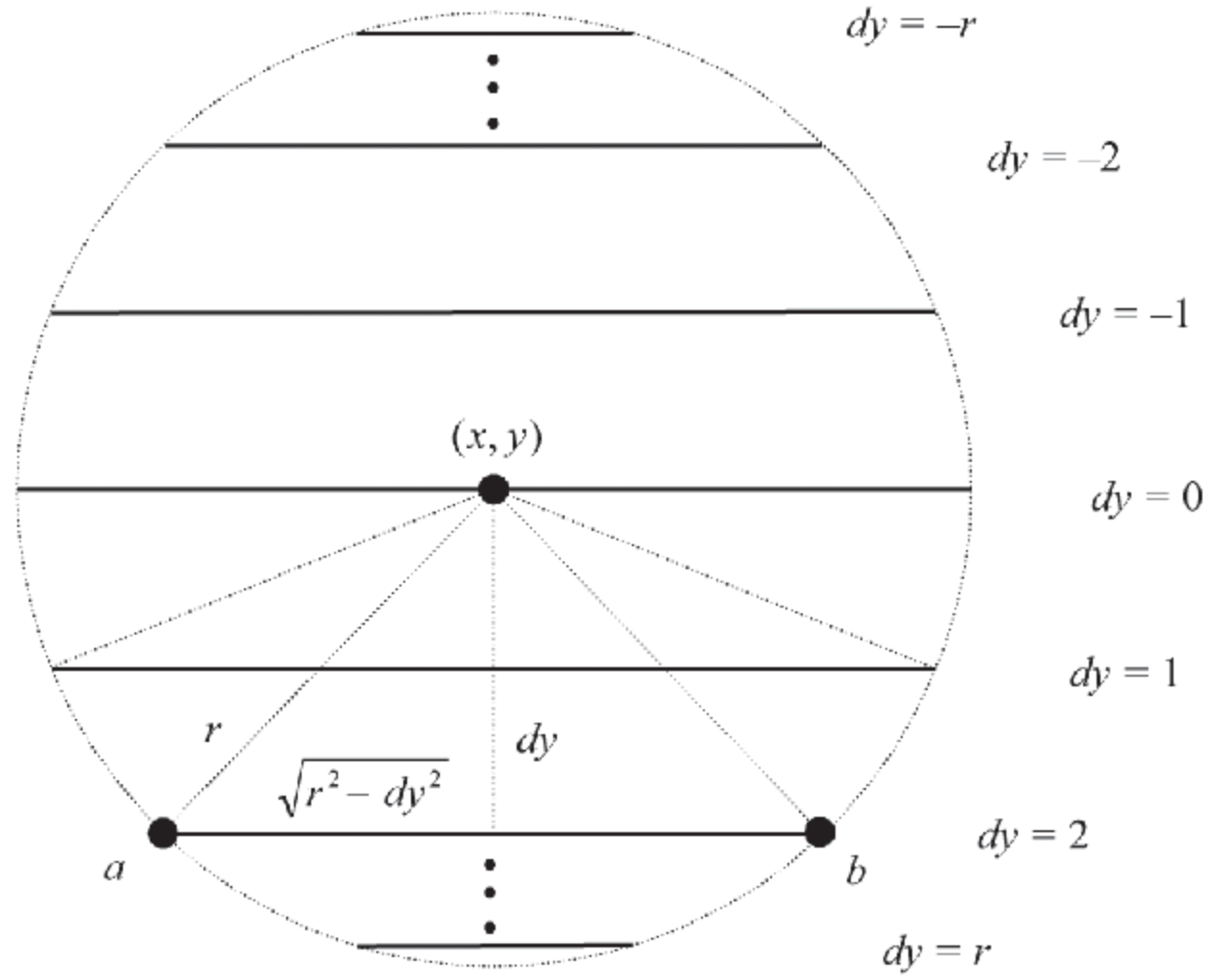
Figure 12.7 Line-drawing algorithm: basic version (bottom, left) and improved version (bottom, right).

The use of two division operations in each loop iteration makes this algorithm neither efficient nor accurate. The first obvious improvement is replacing the $b / a > dy / dx$ condition with the equivalent $a \cdot dy < b \cdot dx$, which requires only integer multiplication. Careful inspection of the latter condition reveals that it may be checked without any multiplication. As shown in the improved algorithm in figure 12.7, this may be done

efficiently by maintaining a variable that updates the value of $(a \cdot dy - b \cdot dx)$ each time a or b is incremented.

The running time of this line-drawing algorithm is $O(n)$, where n is the number of pixels along the drawn line. The algorithm uses only addition and subtraction operations and can be implemented efficiently in either software or hardware.

Circle drawing (drawCircle): Figure 12.8 presents an algorithm that uses three routines that we've already implemented: multiplication, square root, and line drawing.



point $a = (x - \sqrt{r^2 - dy^2}, y + dy)$ point $b = (x + \sqrt{r^2 - dy^2}, y + dy)$

```
// Draws a filled circle of radius r, centered at (x,y).
drawCircle(x, y, r):
    for each dy = -r to r do:
        drawLine((x - √(r² - dy²), y + dy), (x + √(r² - dy²), y + dy))
```

Figure 12.8 Circle-drawing algorithm.

The algorithm is based on drawing a sequence of horizontal lines (like the typical line *ab* in the figure), one for each row in the range $y-r$ to $y+r$. Since r is specified in pixels, the algorithm ends up drawing a line in every row along the circle's north-south diameter, resulting in a completely filled circle. A simple tweak can cause this algorithm to draw only the circle's outline, if so desired.

12.1.5 Character Output

To develop a capability for displaying characters, we first turn our physical, pixel-oriented screen into a logical, character-oriented screen suitable for rendering fixed, bitmapped images that represent characters. For example, consider a physical screen that features 256 rows of 512 pixels each. If we allocate a grid of 11 rows by 8 columns for drawing a single character, then our screen can display 23 lines of 64 characters each, with 3 extra rows of pixels left unused.

Fonts: The character sets that computers use are divided into *printable* and *non-printable* subsets. For each printable character in the Hack character set (see appendix 5), an 11-row-by-8-column bitmap image was designed, to the best of our limited artistic abilities. Taken together, these images are called a *font*. Figure 12.9 shows how our font renders the uppercase letter N. To handle character spacing, each character image includes at least a 1-pixel space before the next character in the row and at least a 1-pixel space between adjacent rows (the exact spacing varies with the size and squiggles of individual characters). The Hack font consists of ninety-five such bitmap images, one for each printable character in the Hack character set.

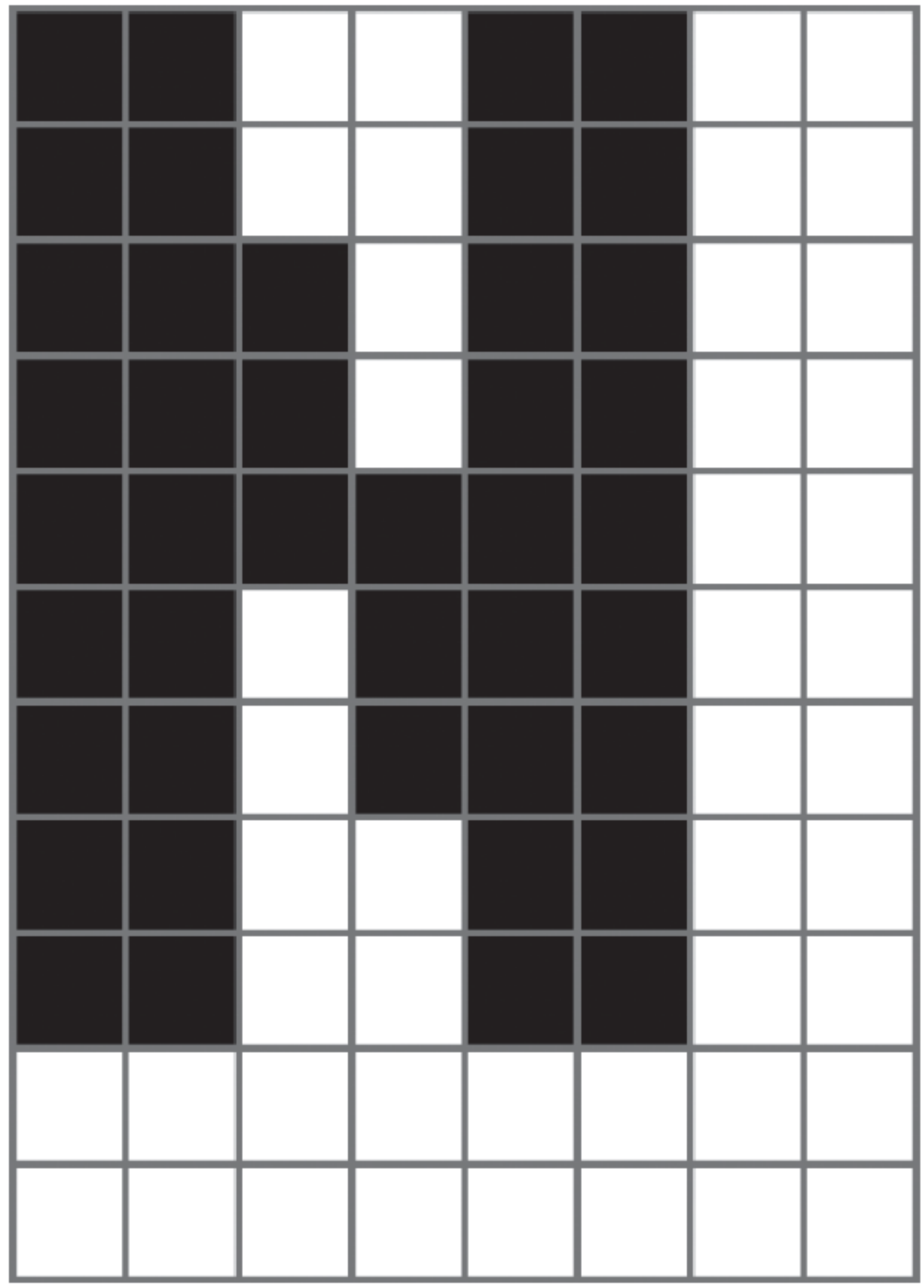


Figure 12.9 Example of a character bitmap.

Font design is an ancient yet vibrant art. The oldest fonts are as old as the art of writing, and new ones are routinely introduced by type designers who wish to make an artistic statement or solve a technical or functional objective. In our case, the small physical screen, on the one hand, and the wish to display a reasonable number of characters in each row, on the other, led to the pragmatic choice of a frugal image area of 11×8 pixels. This economy forced us to design a crude font, which nonetheless serves its purpose well.

Cursor: Characters are usually displayed one after the other, from left to right, until the end of the line is reached. For example, consider a program in which the statement print("a") is followed (perhaps not immediately) by the statement print ("b"). This implies that the program wants to display ab on the screen. To effect this continuity, the character-writing package maintains a global *cursor* that keeps track of the screen location where the next character should be drawn. The cursor information consists of column and row counts, say, cursor.col and cursor.row. After a character has been displayed, we do cursor.col++. At the end of the row we do cursor.row++ and cursor.col = 0. When the bottom of the screen is reached, there is a question of what to do next. Two possible actions are effecting a scrolling operation or clearing the screen and starting over by setting the cursor to (0,0).

To recap, we described a scheme for writing individual characters on the screen. Writing other types of data follows naturally from this basic capability: strings are written character by character, and numbers are first converted to strings and then written as strings.

12.1.6 Keyboard Input

Capturing inputs that come from the keyboard is more intricate than meets the eye. For example, consider the statement let name = Keyboard.readLine("enter your name:"). By definition, the execution of the readLine function depends on the dexterity and collaboration of an unpredictable entity: a human user. The function will not terminate until the user has pressed some keys on the keyboard, ending with an ENTER. The problem is that humans press and release keyboard keys for variable and unpredictable durations of time, and often take a coffee break in the middle. Also, humans are fond of backspacing, deleting, and retyping characters. The implementation of the readLine function must handle all these irregularities.

This section describes how keyboard input is managed, in three levels of abstraction: (i) detecting which key is currently pressed on the keyboard, (ii) capturing a single-character input, and (iii) capturing a multicharacter input.

Detecting keyboard input (keyPressed): Detecting which key is presently pressed is a hardware-specific operation that depends on the keyboard

interface. In the Hack computer, the keyboard continuously refreshes a 16-bit memory register whose address is kept in a pointer named KBD. The interaction contract is as follows: If a key is currently pressed on the keyboard, that address contains the key's character code (the Hack character set is given in appendix 5); otherwise, it contains 0. This contract is used for implementing the keyPressed function shown in figure 12.10.

Listens to the keyboard

```
// Returns the character code of the
// currently pressed key on the keyboard,
// or 0 if no key is pressed.
keyPressed():
  // uses Memory.peek
```

Gets a character

```
// Waits until a key is pressed and released;
// echoes the character on the screen, advances
// the cursor, and returns the character code.
readChar():
   display the cursor
   // waits until a key is pressed:
   while (keyPressed() = 0)
      do nothing
   c = code of the currently pressed key
   // waits until the key is released:
   while (keyPressed() ≠ 0)
      do nothing
   display c at the current cursor location
   advance the cursor
   return c
```

Gets a string

```
// Displays the message on the screen, gets the next
// line (until a newLine character) from the keyboard,
// and returns its value, as a string.
readLine(message):
   display the message
   str = empty string
   repeat
      c = readChar()
      if (c == newLine):
         display newLine
         return str
      else if (c == backSpace):
            remove the last character from str
            move the cursor back
         else
            str.appendChar(c)
   return str
```

Figure 12.10 Handling input from the keyboard.

Reading a single character (readChar): The elapsed time between the *key pressed* and the subsequent *key released* events is unpredictable. Hence, we have to write code that neutralizes this uncertainty. Also, when users press keys on the keyboard, we want to give feedback as to which keys have been pressed (something that you have probably grown to take for granted). Typically, we want to display some graphical cursor at the screen location where the next input goes, and, after some key has been pressed, we want to echo the inputted character by displaying its bitmap on the screen at the cursor location. All these actions are implemented by the readChar function.

Reading a string (readLine): A multicharacter input typed by the user is considered final after the ENTER key has been pressed, yielding the newLine character. Until the ENTER key is pressed, the user should be allowed to

backspace, delete, and retype previously typed characters. All these actions are accommodated by the readLine function.

As usual, our input-handling solutions are based on a cascading series of abstractions: The high-level program relies on the readLine abstraction, which relies on the readChar abstraction, which relies on the keyPressed abstraction, which relies on the Memory.peek abstraction, which relies on the hardware.

12.2 The Jack OS Specification

The previous section presented algorithms that address various classical operating system tasks. In this section we turn to formally specify one particular operating system—the Jack OS. The Jack operating system is organized in eight classes:

- Math: provides mathematical operations
- String: implements the String type
- Array: implements the Array type
- Memory: handles memory operations
- Screen: handles graphics output to the screen
- Output: handles character output to the screen
- Keyboard: handles input from the keyboard
- Sys: provides execution-related services

The complete OS API is given in appendix 6. This API can be viewed as the OS specification. The next section describes how this API can be implemented using the algorithms presented in the previous section.

12.3 Implementation

Each OS class is a collection of subroutines (constructors, functions, and methods). Most of the OS subroutines are simple to implement and are not discussed here. The remaining OS subroutines are based on the algorithms

presented in section 12.2. The implementation of these subroutines can benefit from some tips and guidelines, which we now turn to present.

Init **functions**: Some OS classes use data structures that support the implementation of some of their subroutines. For each such *OSClass*, these data structures can be declared statically at the class level and initialized by a fucntion which, by convention, we call *OSClass*.init. The init functions are for internal purposes and are not documented in the OS API.

Math

multiply: In each iteration *i* of the *multiplication* algorithm (see figure 12.1), the *i*-th bit of the second multiplicand is extracted. We suggest encapsulating this operation in a helper function bit (x,i) that returns true if the i-th bit of the integer x is 1, and false otherwise. The bit(x,i) function can be easily implemented using shifting operations. Alas, Jack does not support shifting. Instead, it may be convenient to define a fixed static array of length 16, say twoToThe, and set each element *i* to 2 raised to the power of *i*. The array can then be used to support the implementation of bit (x,i). The twoToThe array can be built by the Math.init function.

divide: The *multiplication* and *division* algorithms presented in figures 12.1 and 12.2 are designed to operate on nonnegative integers. Signed numbers can be handled by applying the algorithms to absolute values and setting the sign of the return values appropriately. For the multiplication algorithm, this is not needed: since the multiplicands are given in two's complement, their product will be correct with no further ado.

In the *division* algorithm, y is multiplied by a factor of 2, until $y > x$. Thus y can overflow. The overflow can be detected by checking when y becomes negative.

sqrt: In the *square root* algorithm (figure 12.3), the calculation of $(y+2^j)^2$ can overflow, resulting in an abnormally negative result. This problem can be addressed by changing efficiently the algorithm's if logic to: if $(y+2^j)^2 \le x$ and $(y+2^j)^2 > 0$ then $y = y+2^j$.

String

All the string constants that appear in a Jack program are realized as objects of the String class, whose API is documented in appendix 6. Specifically, each string is implemented as an object consisting of an array of char values, a maxLength property that holds the maximum length of the string, and a length property that holds the actual length of the string.

For example, consider the statement `let str="scooby"`. When the compiler handles this statement, it calls the String constructor, which creates a char array with `maxLength=6` and `Length=6`. If we later call the String method str.eraseLastChar(), the length of the array will become 5, and the string will effectively become "scoob". In general, then, array elements beyond length are not considered part of the string.

What should happen when an attempt is made to add a character to a string whose length equals maxLength? This issue is not defined by the OS specification: the String class may act gracefully and resize the array—or not; this is left to the discretion of individual OS implementations.

intValue, setInt: These subroutines can be implemented using the algorithms presented in figure 12.4. Note that neither algorithm handles negative numbers—a detail that must be handled by the implementation.

newLine, backSpace, doubleQuote: As seen in appendix 5, the codes of these characters are 128, 129, and 34.

The remaining String methods can be implemented straightforwardly by operating on the char array and on the length field that characterizes each String object.

Array

new: In spite of its name, this subroutine is not a constructor but rather a function. Therefore, the implementation of this function must allocate memory space for the new array by explicitly calling the OS function Memory.alloc.

dispose: This void method is called by statements like do arr.dispose(). The dispose implementation deallocates the array by calling the OS function Memory.deAlloc.