A subroutine call invokes a function, a constructor, or a method for its effect, using the general syntax *subroutineName* (*exp*$_1$, *exp*$_2$, …, *exp*$_n$), where each argument *exp* is an expression. The number and type of the arguments must match those of the subroutine's parameters, as specified in the subroutine's declaration. The parentheses must appear even if the argument list is empty.

Subroutines can be called from the class in which they are defined, or from other classes, according to the following syntax rules:

Function calls / Constructor calls:

- *className.functionName* (*exp*$_1$, *exp*$_2$, … , *exp*$_n$)
- *className.constructorName* (*exp*$_1$, *exp*$_2$, … , *exp*$_n$)

The *className* must always be specified, even if the function/constructor is in the same class as the caller.

Method calls:

- *varName.methodName* (*exp*$_1$, *exp*$_2$, … , *exp*$_n$)

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

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

Here are subroutine call examples:

```
class Foo {
   ...
   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 Foo.g()          // Calls function g of the current class
      do Bar.h()          // Calls function h of class Bar
      do m()              // Calls method m of the current class, on the this object
      do b.q()            // Calls method q of class Bar, on object b
      let i = w(b.s(), Foo.t())  // Calls method w on the this object,
                                 // Calls method s of class Bar on object b,
                                 // Calls function or constructor t of class Foo.
    }
}
```

9.2.8 Object Construction and Disposal

Object construction is done in two stages. First, a reference variable (pointer to an object) is declared. 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) must feature at least one constructor. Jack constructors may have arbitrary names; by convention, one of them is named new.

Objects are constructed and assigned to variables using the idiom let *varName* = *className.constructorName* (*exp*$_1$, *exp*$_2$, … , *exp*$_n$), for example, let c = Circle.new (x,y,50). Constructors typically include code that initializes the fields of the new object to the argument values passed by the caller.

When an object is no longer needed, it can be disposed, to free the memory that it occupies. For example, suppose that the object that c points at is no longer needed. The object can be deallocated from memory by calling the OS function Memory.deAlloc (c). Since Jack has no garbage collection, the best-practice advice is that every class that represents an object must feature a dispose() method that properly encapsulates this de-allocation. Figures 9.3 and 9.4 give examples. To avoid memory leaks, Jack programmers are advised to dispose objects when they are no longer needed.

9.3 Writing Jack Applications

Jack is a general-purpose language that can be implemented over different hardware platforms. In Nand to Tetris we develop a *Jack compiler over the Hack platform*, and thus it is natural to discuss Jack applications in the Hack context.

Examples: Figure 9.9 shows screenshots of four sample Jack programs. Generally speaking, the Jack/Hack platform lends itself nicely to simple interactive games like Pong, Snake, Tetris, and similar classics. Your projects/09/Square folder includes the full Jack code of a simple interactive program that allows the user to move a square image on the screen using the four keyboard arrow keys.

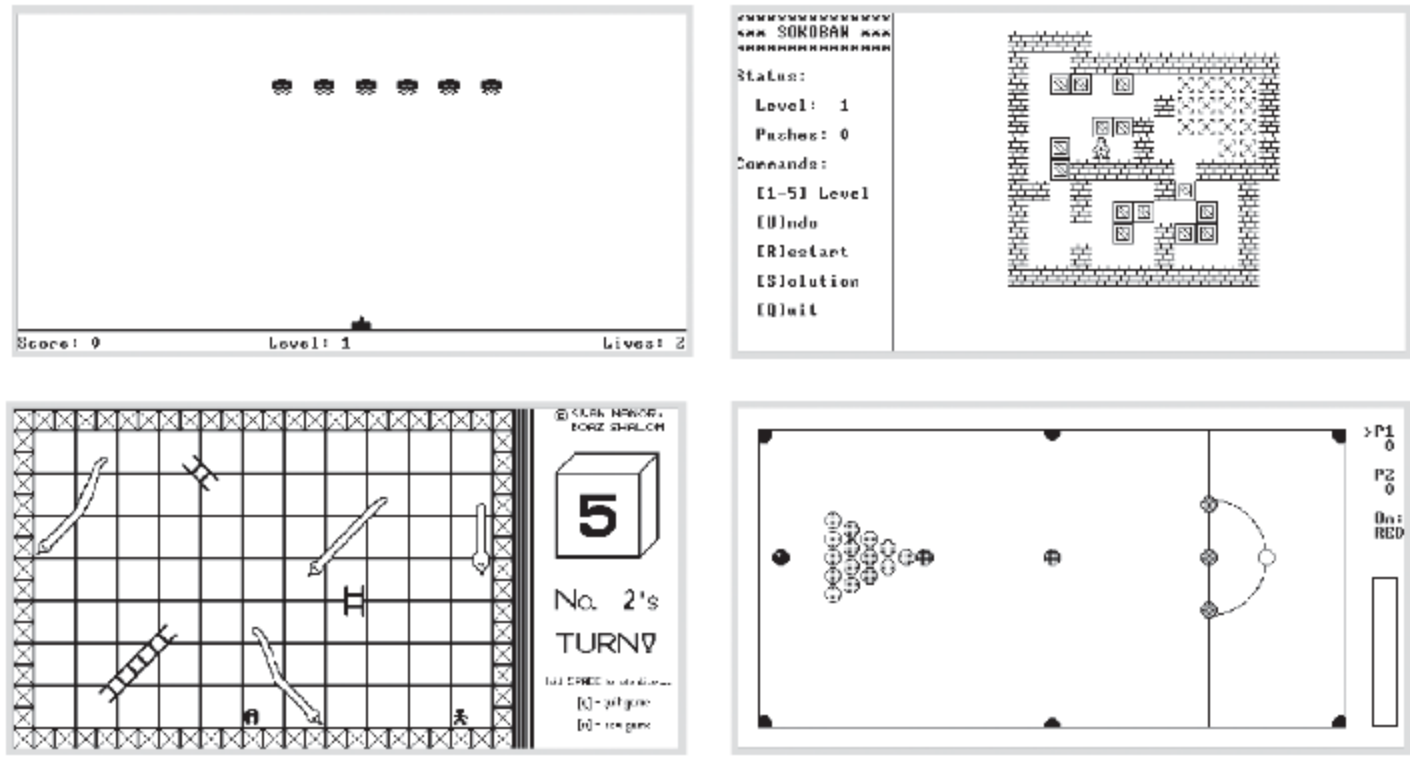

Figure 9.9 Screenshots of Jack applications running on the Hack computer.

Executing this program while reviewing its Jack source code is a good way for learning how to use Jack to write interactive graphical applications. Later in the chapter we describe how to compile and execute Jack programs using the supplied tools.

Application design and implementation: Software development should always rest on careful planning, especially when done over a spartan hardware platform like the Hack computer. First, the program designer must

consider the physical limitations of the hardware and plan accordingly. To start with, 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.

The design process normally starts with a conceptual description of the desired program's behavior. In the case of graphical and interactive programs, this may take the form of handwritten drawings of typical screens. Next, one normally designs an object-based architecture of the program. This entails the identification of *classes*, *fields*, and *subroutines*. For example, if the program is supposed to allow the user to create square objects and move them around the screen using the keyboard's arrow keys, it will make sense to design a Square class that encapsulates these operations using methods like moveRight, moveLeft, moveUp, and moveDown, as well as a constructor subroutine for creating squares and a disposer subroutine for disposing them. In addition, it will make sense to create a SquareGame class that carries out the user interaction and a Main class that gets things started. Once the APIs of these classes are carefully specified, one can proceed to implement, compile, and test them.

Compiling and executing Jack programs: All the .jack files comprising the program must reside in the same folder. When you apply the Jack compiler to the program folder, each source .jack file will be translated into a corresponding .vm file, stored in the same program folder.

The simplest way to execute or debug a compiled Jack program is to load the program folder into the VM emulator. The emulator will load all the VM functions in all the .vm files in the folder, one after the other. The result will be a (possibly long) stream of VM functions, listed in the VM emulator's code pane using their full *fileName.functionName* names. When you instruct the emulator to execute the program, the emulator will start executing the OS Sys.init function, which will then call the Main.main function in your Jack program.

Alternatively, you can use a VM translator (like the one built in projects 7–8) for translating the compiled VM code, as well as the eight supplied tools/OS/*.vm OS files, into a single .asm file written in the Hack machine

language. The assembly code can then be executed on the supplied CPU emulator. Or, you can use an assembler (like the one built in project 6) for translating the .asm file further into a binary code .hack file. Next, you can load a Hack computer chip (like the one built in projects 1–5) into the hardware simulator or use the built-in Computer chip, load the binary code into the ROM chip, and execute it.

The operating system: Jack programs make extensive use of the language's *standard class library*, which we also refer to as the *Operating System*. In project 12 you will develop the OS class library in Jack (like Unix is written in C) and compile it using a Jack compiler. The compilation will yield eight .vm files, comprising the OS implementation. If you put these eight .vm files in your program folder, all the OS functions will become accessible to the compiled VM code, since they belong to the same code base (by virtue of belonging to the same folder).

Presently, though, there is no need to worry about the OS implementation. The supplied VM emulator, which is a Java program, features a built-in Java implementation of the Jack OS. When the VM code loaded into the emulator calls an OS function, say Math.sqrt, one of two things happens. If the OS function is found in the loaded code base, the VM emulator executes it, just like executing any other VM function. If the OS function is not found in the loaded code base, the emulator executes its built-in implementation.

9.4 Project

Unlike the other projects in this book, this one does not require building a hardware of software module. Rather, you have to pick some application of your choice and build it in Jack over the Hack platform.

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 like a simple computer game or some interactive program. Then design and build the application.

Resources: You will need the supplied tools/JackCompiler for translating your program into a set of .vm files, and the supplied tools/VMEmulator for running and testing the compiled code.

Compiling and Running a Jack Program

0. Create a folder for your program. Let's call it the *program folder*.
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 program folder.
2. Compile the program folder using the supplied Jack compiler. This will cause the compiler to translate all the .jack classes found in the folder into corresponding .vm files. If a compilation error is reported, debug the program and recompile until no error messages are issued.
3. At this point the program folder should contain your source .jack files along with the compiled .vm files. To test the compiled program, load the program folder into the supplied VM emulator, and run the loaded code. In case of run-time errors or undesired program behavior, fix the relevant file and go back to step 2.

Program examples: Your nand2tetris/project/09 folder includes the source code of a complete, three-class interactive Jack program (Square). It also includes the source code of the Jack programs discussed in this chapter.

Bitmap editor: If you develop a program that needs high-speed graphics, it is best to design *sprites* for rendering the key graphical elements of the program. For example, the output of the Sokoban application depicted in figure 9.9 consists of several repeating sprites. If you wish to design such sprites and write them directly into the screen memory map (bypassing the services of the OS Screen class, which may be too slow), you will find the projects/09/BitmapEditor tool useful.

A web-based version of project 9 is available at www.nand2tetris.org.

9.5 Perspective

Jack is an *object-based* language, meaning that it supports objects and classes but not inheritance. In this respect it is positioned somewhere between procedural languages like Pascal or C and object-oriented languages like Java or C++. Jack is certainly more simple-minded than any of these industrial strength programming languages. However, its basic syntax and semantics are similar to 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, you may wonder why the Jack syntax includes clunky keywords like `do` and `let`, why every subroutine must end with a `return` statement, why the language does not enforce operator priority, and so on—you may add your favorite complaint to the list.

All these somewhat tedious idiosyncrasies were introduced into Jack with one purpose: allowing the development of simple and minimal Jack compilers, as we will do in the next two chapters. For example, the parsing of a statement (in any language) is significantly easier if the first token of the statement reveals which statement we're in. That's why Jack uses a `let` keyword for prefixing assignment statements. Thus, although Jack's simplicity may be a nuisance when writing Jack *applications*, you'll be grateful for this simplicity when writing the Jack *compiler*, as we'll do in the next two chapters.

Most modern languages are deployed with a set of *standard classes*, and so is Jack. Taken together, these classes can be viewed as a portable, language-oriented operating system. Yet unlike the standard libraries of industrial-strength languages, which feature numerous classes, the Jack OS provides a minimal set of services, which is nonetheless sufficient for developing simple interactive applications.

Clearly, it would be nice to extend the Jack OS to provide concurrency for supporting multi-threading, a file system for permanent storage, sockets for communications, and so on. Although all these services can be added to the OS, readers will perhaps want to hone their programming skills elsewhere. After all, we don't expect Jack to be part of your life beyond

Nand to Tetris. 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 constraints imposed by the host platform as a problem, professionals view it as an opportunity to display their resourcefulness and ingenuity. That's what you are expected to do in project 9.

10 Compiler I: Syntax Analysis

Neither can embellishments of language be found without arrangement and expression of thoughts, nor can thoughts be made to shine without the light of language.

—Cicero (106–43 B.C.)

The previous chapter introduced Jack—a simple, object-based programming language with a Java-like syntax. In this chapter we start building a compiler for the Jack language. A *compiler* is a program that translates programs from a source language into a target language. The translation process, known as *compilation*, is conceptually based on two distinct tasks. First, we have to understand the syntax of the source program and, from it, uncover the program's semantics. For example, the parsing of the code can reveal that the program seeks to declare an array or manipulate an object. Once we know the semantics, we can reexpress it using the syntax of the target language. The first task, typically called *syntax analysis*, is described in this chapter; the second task—*code generation*—is taken up in the next chapter.

How can we tell that a compiler is capable of "understanding" programs? Well, as long as the code generated by the compiler is doing what it's supposed to be doing, we can optimistically assume that the compiler is operating properly. Yet in this chapter we build only the syntax analyzer module of the compiler, with no code generation capabilities. If we wish to unit-test the syntax analyzer in isolation, we have to contrive a way to demonstrate that it understands the source program. Our solution is to have the syntax analyzer output an XML file whose marked-up content reflects the syntactic structure of the source code. By inspecting the generated XML

output, we'll be able to ascertain that the analyzer is parsing input programs correctly.

Writing a compiler from the ground up is an exploit that brings to bear several fundamental topics in computer science. It requires the use of parsing and language translation techniques, application of classical data structures like trees and hash tables, and use of recursive compilation algorithms. For all these reasons, writing a compiler is also a challenging feat. However, by splitting the compiler's construction into two separate projects (actually *four*, counting chapters 7 and 8 as well) and by allowing the modular development and unit-testing of each part in isolation, we turn the compiler's development into a manageable and self-contained activity.

Why should you go through the trouble of building a compiler? Aside from the benefits of feeling competent and accomplished, a hands-on grasp of compilation internals will turn you into a better high-level programmer. Further, the same rules and grammars used for describing programming languages are also used in diverse fields like computer graphics, communications and networks, bioinformatics, machine learning, data science, and blockchain technology. And, the vibrant area of *natural language processing*—the enabling science and practice behind intelligent chatbots, robotic personal assistants, language translators, and many artificial intelligence applications—requires abilities for analyzing texts and synthesizing semantics. Thus, while most programmers don't develop compilers in their regular jobs, many programmers have to parse and manipulate texts and data sets of complex and varying structures. These tasks can be done efficiently and elegantly using the algorithms and techniques described in this chapter.

We start with a Background section that surveys the minimal set of concepts necessary for building a syntax analyzer: lexical analysis, context-free grammars, parse trees, and recursive descent parsing algorithms. This sets the stage for a Specification section that presents the Jack language grammar and the output that a Jack analyzer is expected to generate. The Implementation section proposes a software architecture for constructing a Jack analyzer, along with a suggested API. As usual, the Project section gives step-by-step instructions and test programs for building a syntax analyzer. In the next chapter, this analyzer will be extended into a full-scale compiler.

10.1 Background

Compilation consists of two main stages: *syntax analysis* and *code generation*. The syntax analysis stage is usually divided further into two substages: *tokenizing*, the grouping of input characters into language atoms called *tokens*, and *parsing*, the grouping of tokens into structured statements that have a meaning.

The tokenizing and parsing tasks are completely independent of the target language into which we seek to translate the source input. Since in this chapter we don't deal with code generation, we have chosen to have the syntax analyzer output the parsed structure of the input program as an XML file. This decision has two benefits. First, the output file can be readily inspected, demonstrating that the syntax analyzer is parsing source programs correctly. Second, the requirement to output this file explicitly forces us to write the syntax analyzer in an architecture that can be later morphed into a full-scale compiler. Indeed, as figure 10.1 shows, in the next chapter we will extend the syntax analyzer developed in this chapter into a full-scale compilation engine capable of generating executable VM code rather than passive XML code.

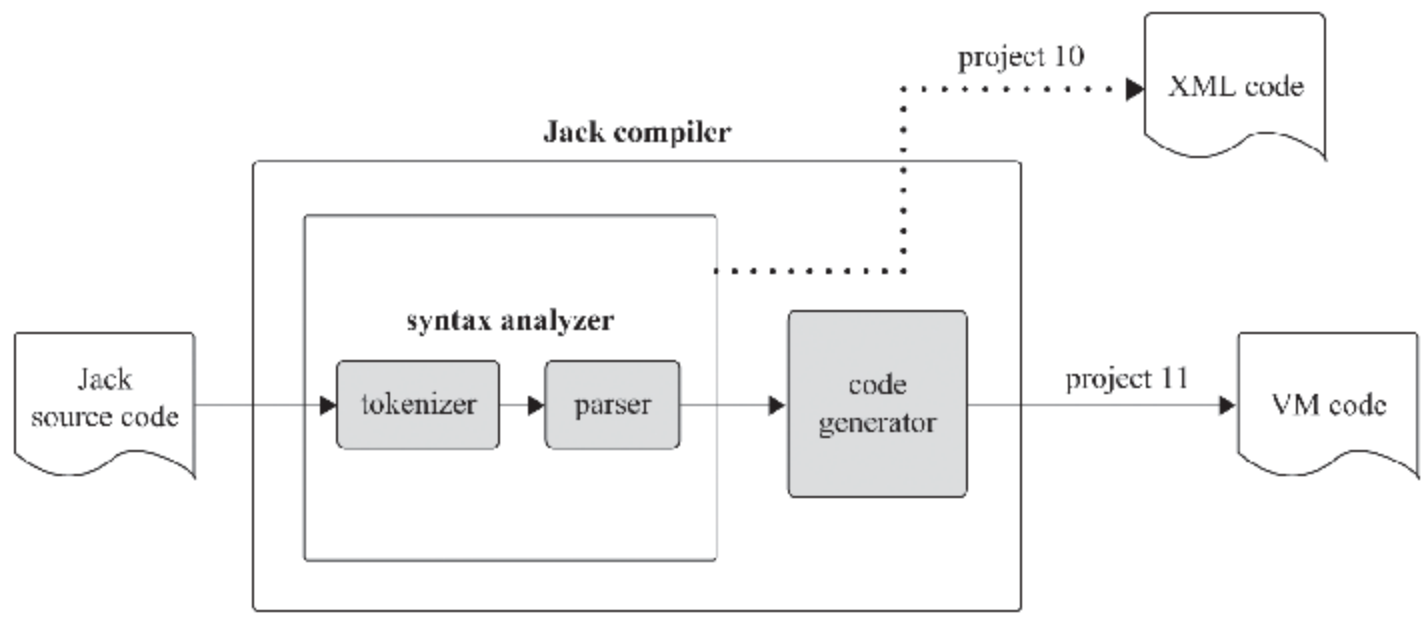

Figure 10.1 Staged development plan of the Jack compiler.

In this chapter we focus only on the syntax analyzer module of the compiler, whose job is *understanding the structure of a program*. This notion needs explanation. When humans read the source code of a computer program, they can immediately relate to the program's structure. They can

do so since they have a mental image of the language's *grammar*. In particular, they sense which program constructs are valid, and which are not. Using this grammatical insight, humans can identify where classes and methods begin and end, what are declarations, what are statements, what are expressions and how they are built, and so on. In order to recognize these language constructs, which may well be nested, humans recursively map them on the range of textual patterns accepted by the language grammar.

Syntax analyzers can be developed to perform similarly by building them according to a given *grammar*—the set of rules that define the syntax of a programming language. To understand—*parse*—a given program means to determine the exact correspondence between the program's text and the grammar's rules. To do so, we must first transform the program's text into a list of *tokens*, as we now turn to describe.

10.1.1 Lexical Analysis

Each programming language specification includes the types of *tokens*, or words, that the language recognizes. In the Jack language, tokens fall into five categories: *keywords* (like `class` and `while`), *symbols* (like `+` and `<`), *integer constants* (like `17` and `314`), *string constants* (like `"FAQ"` and `"Frequently Asked Questions"`), and *identifiers*, which are the textual labels used for naming variables, classes, and subroutines. Taken together, the tokens defined by these lexical categories can be referred to as the language *lexicon*.

In its plainest form, a computer program is a stream of characters stored in a text file. The first step in analyzing the program's syntax is grouping the characters into tokens, as defined by the language lexicon, while ignoring white space and comments. This task is called *lexical analysis*, *scanning*, or *tokenizing*—all meaning exactly the same thing.

Once a program has been tokenized, the tokens, rather than the characters, are viewed as its basic atoms. Thus, the token stream becomes the compiler's main input.

Figure 10.2 presents the Jack language lexicon and illustrates the tokenization of a typical code segment. This version of the tokenizer outputs the tokens as well as their lexical classifications.

Prog.jack (character stream)

```
...
if (x < 0) {
   // handles the sign
   let sign = "negative";
}
...
```

Tokenizing

Output (token stream)

```
...
<keyword>     if        </keyword>
<symbol>      (         </symbol>
<identifier>  x         </identifier>
<symbol>      <         </symbol>
<intConst>    0         </intConst>
<symbol>      )         </symbol>
<symbol>      {         </symbol>
<keyword>     let       </keyword>
<identifier>  sign      </identifier>
<symbol>      =         </symbol>
<stringConst> negative  </stringConst>
<symbol>      ;         </symbol>
<symbol>      }         </symbol>
...
```

keyword: 'class'|'constructor'|'function'|
'method'|'field'|'static' 'var'|
'int'|'char'|'boolean'|'void'|
'true'|'false'|'null'|'this'|'let'|
'do'|'if' 'else'|'while'|'return'

symbol: '{'|'}'|'('|')'|'['|']'|'.'|','|';'|
'+'|'-'|'*'|'/'|'&'|'|'|'<'|'>'|'='|'~'

integerConstant: a decimal integer in the range 0 ... 32767

stringConstant: '"' a sequence of characters, not including double quote or newline '"'

identifier: a sequence of letters, digits, and underscore ('_'), not starting with a digit.

Figure 10.2 Definition of the Jack lexicon, and lexical analysis of a sample input.

Tokenizing is a simple and important task. Given a language lexicon, it is not difficult to write a program that turns any given character stream into a stream of tokens. This capability provides the first stepping stone toward developing a syntax analyzer.

10.1.2 Grammars

Once we develop the ability to access a given text as a stream of tokens, or words, we can proceed to attempt grouping the words into valid sentences. For example, when we hear that "Bob got the job" we nod approvingly, while inputs like "Got job the Bob" or "Job Bob the got" sound weird. We perform these parsing tasks without thinking about them, since our brains have been trained to map sequences of words on patterns that are either accepted or rejected by the English grammar. The grammars of programming languages are much simpler than those of natural languages. See figure 10.3 for an example.

Jack grammar (subset)

statements: *statement**

statement: *letStatement* |
ifStatement |
whileStatement

letStatement: '**let**' *varName* '=' *expression* '**;**'

ifStatement: '**if**' '**(**' *expression* '**)**'
'**{**' *statements* '**}**'

whileStatement: '**while**' '**(**' *expression* '**)**'
'**{**' *statements* '**}**'

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

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a non-negative integer

op: '+' | '-' | '=' | '>' | '<'

Input examples

```
let x = 100;
```

```
let x = x + 1;
```

```
if (x = 1)
   let x = 100;
   let x = x + 1;
}
```

```
while (lim < 100) {
  if (x = 1) {
    let z = 100;
    while (z > 0) {
      let z = z - 1;
    }
  }
  let lim = lim + 10;
}
```

Figure 10.3 A subset of the Jack language grammar, and Jack code segments that are either accepted or rejected by the grammar.

A grammar is written in a *meta-language*: a language describing a language. Compilation theory is rife with formalisms for specifying, and reasoning about, grammars, languages, and meta-languages. Some of these formalisms are, well, painfully formal. Trying to keep things simple, in Nand to Tetris we view a grammar as a set of rules. Each rule consists of a left side and a right side. The left side specifies the rule's name, which is not part of the language. Rather, it is made up by the person who describes the grammar, and thus it is not terribly important. For example, if we replace a rule's name with another name throughout the grammar, the grammar will be just as valid (though it may be less readable).

The rule's right side describes the lingual pattern that the rule specifies. This pattern is a left-to-right sequence consisting of three building blocks: *terminals*, *nonterminals*, and *qualifiers*. Terminals are tokens, nonterminals are names of other rules, and qualifiers are represented by the five symbols |, *, ?, (, and). Terminal elements, like 'if', are specified in bold font and enclosed within single quotation marks; nonterminal elements, like *expression*, are specified using italic font; qualifiers are specified using

regular font. For example, the rule *ifStatement*: 'if' '(' *expression* ')' '{' *statements* '}' stipulates that every valid instance of an *ifStatement* must begin with the token if, followed by the token (, followed by a valid instance of an *expression* (defined elsewhere in the grammar), followed by the token), followed by the token {, followed by a valid instance of *statements* (defined elsewhere in the grammar), followed by the token}.

When there is more than one way to parse a pattern, we use the qualifier | to list the alternatives. For example, the rule *statement*: *letStatement* | *ifStatement* | *whileStatement* stipulates that a *statement* can be either a *letStatement*, an *ifStatement*, or a *whileStatement*.

The qualifier * is used to denote "0, 1, or more times." For example, the rule *statements*: *statement** stipulates that *statements* stands for 0, 1, or more instances of *statement*. In a similar vein, the qualifier ? is used to denote "0 or 1 times." For example, the rule *expression*: *term* (*op term)?* stipulates that *expression* is a *term* that may or may not be followed by the sequence *op term*. This implies that, for example, `x` is an *expression*, and so are `x+17`, `5*7`, and `x<y`. The qualifiers (and) are used for grouping grammar elements. For example, (*op term*) stipulates that, in the context of this rule, *op* followed by *term* should be treated as one grammatical element.

10.1.3 Parsing

Grammars are inherently recursive. Just like the sentence "Bob got the job that Alice offered" is considered valid, so is the statement `if (x<0){if(y>0){...}}`. How can we tell that this input is accepted by the grammar? After getting the first token and realizing that we have an if pattern, we focus on the rule *ifStatement*: 'if' '(' *expression* ')' '{' *statements* '}'. The rule informs that following the token if there ought to be the token (, followed by an *expression*, followed by the token). And indeed, these requirements are satisfied by the input element `(x<0)`. Back to the rule, we see that we now have to anticipate the token {, followed by *statements*, followed by the token}. Now, *statements* is defined as 0 or more instances of *statement*, and *statement*, in turn, is either a *letStatement*, an *ifStatement*, or a *whileStatement*. This expectation is met by the inner input element if `(y>0){...}`, which is an *ifStatement*.

We see that the grammar of a programming language can be used to ascertain, without ambiguity, whether given inputs are accepted or rejected.[1] As a side effect of this parsing act, the parser produces an exact correspondence between the given input, on the one hand, and the syntactic patterns admitted by the grammar rules, on the other. The correspondence can be represented by a data structure called a *parse tree*, also called a *derivation tree*, like the one shown in figure 10.4a. If such a tree can be constructed, the parser renders the input valid; otherwise, it can report that the input contains syntax errors.

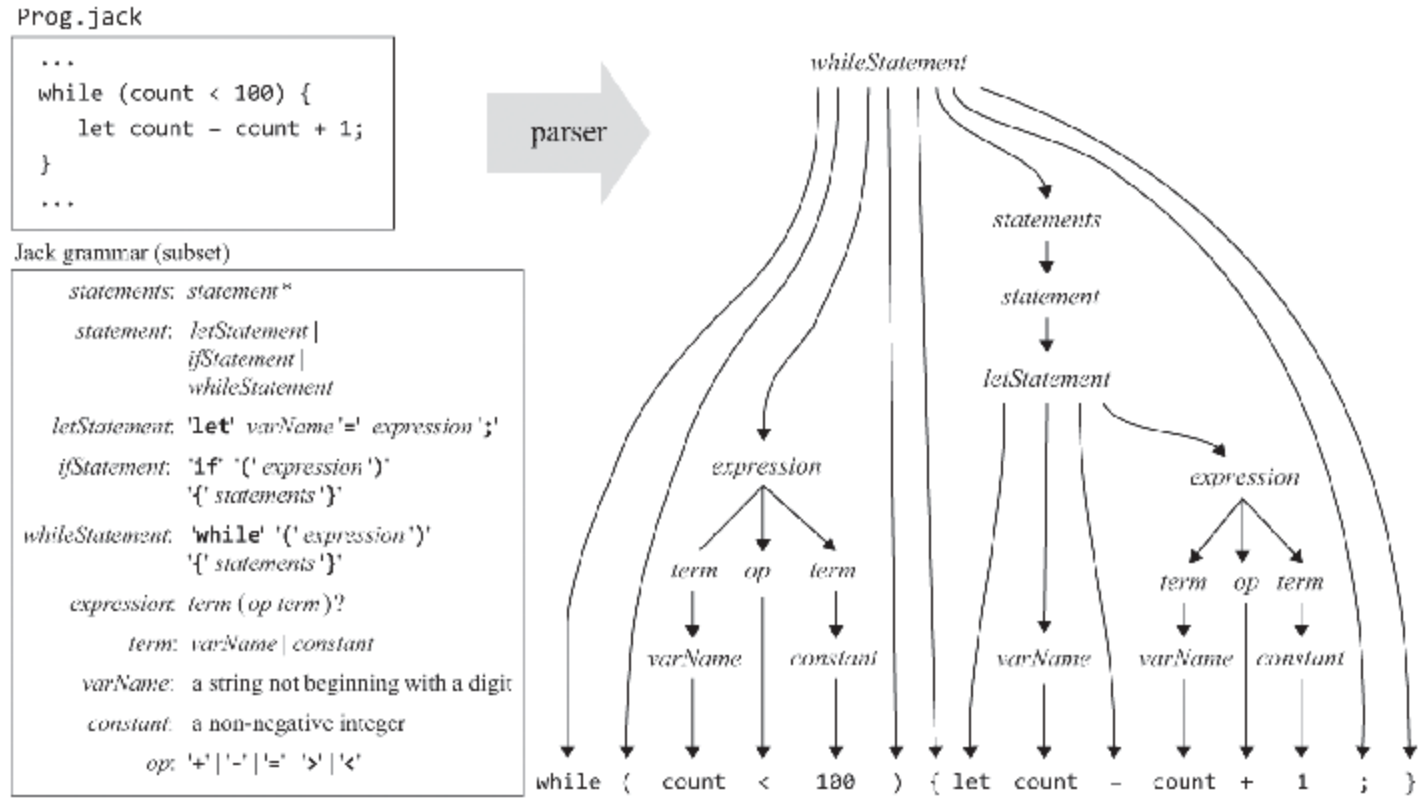

Figure 10.4a Parse tree of a typical code segment. The parsing process is driven by the grammar rules.

How can we represent parse trees textually? In Nand to Tetris, we decided to have the parser output an XML file whose marked-up format reflects the tree structure. By inspecting this XML output file, we can ascertain that the parser is parsing the input correctly. See figure 10.4b for an example.

Prog.xml

```
...
<whileStatement>
   <keyword> while </keyword>
   <symbol> ( </symbol>
   <expression>
      <term> <varName> count </varName> </term>
      <op> <symbol> < </symbol> </op>
      <term> <constant> 100 </constant> </term>
   </expression>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <statements>
      <statement> <letStatement>
         <keyword> let </keyword>
         <varName> count </varName>
         <symbol> = </symbol>
         <expression>
            <term> <varName> count </varName> </term>
            <op> <symbol> + </symbol> </op>
            <term> <constant> 1 </constant> </term>
         </expression>
         <symbol> ; </symbol>
      </letStatement> </statement>
   </statements>
   <symbol> } </symbol>
</whileStatement>
...
```

Figure 10.4b Same parse tree, in XML.

10.1.4 Parser

A parser is an agent that operates according to a given grammar. The parser accepts as input a stream of tokens and attempts to produce as output the parse tree associated with the given input. In our case, the input is expected to be structured according to the Jack grammar, and the output is written in XML. Note, though, that the parsing techniques that we now turn to describe are applicable to handling any programming language and structured file format.

There are several algorithms for constructing parse trees. The top-down approach, also known as *recursive descent parsing*, attempts to parse the tokenized input recursively, using the nested structures admitted by the language grammar. Such an algorithm can be implemented as follows. For every nontrivial rule in the grammar, we equip the parser program with a routine designed to parse the input according to that rule. For example, the grammar listed in figure 10.3 can be implemented using a set of routines named compileStatement, compileStatements, compileLet, compileIf, …, compileExpression, and so on. We use the action verb *compile* rather than *parse*, since in the next chapter we will extend this logic into a full-scale compilation engine.

The parsing logic of each compile*xxx* routine should follow the syntactic pattern specified by the right side of the *xxx* rule. For example, let us focus on the rule *whileStatement*: '**while**' '(' *expression* ')' '{' *statements* '}'. According to our scheme, this rule will be implemented by a parsing routine named compileWhile. This routine should realize the left-to-right derivation logic specified by the pattern '**while**' '(' *expression* ')' '{' *statements* '}'. Here is one way to implement this logic, using pseudocode:

```
// This routine implements the rule whileStatement:
// 'while' '('expression')' '{' statements '}'
// Should be called if the current token is 'while'.
compileWhile():
   print("<whileStatement>")
   process("while")
   process("(")
   compileExpression()
   process(")")
   process("{")
   compileStatements()
   process("}")
   print("</whileStatement>")
```

```
// A helper routine that handles
// the current token, and advances
// to get the next token.
process(str):
   if (currentToken == str)
      printXMLToken(str)
   else
      print("syntax error")
   // Gets the next token
   currentToken =
      tokenizer.advance()
```

This parsing process will continue until the *expression* and *statements* parts of the while statement have been fully parsed. Of course the *statements* part may well contain a lower-level while statement, in which case the parsing will continue recursively.

The example just shown illustrates the implementation of a relatively simple rule, whose derivation logic entails a simple case of straight-line parsing. In general, grammar rules can be more complex. For example, consider the following rule, which specifies the definition of class-level static and instance-level field variables in the Jack language:

classVarDec: ('**static**' | '**field**') *type varName* ('**,**' *varName*)* '**;**'
(where *type* and *varName* are defined elsewhere in the full Jack grammar)

Examples:
```
static int count;
static char a, b, c;
field boolean sign;
field int up, down, left, right;
```

This rule presents two parsing challenges that go beyond straight-line parsing. First, the rule admits either static or field as its first token. Second, the rule admits multiple variable declarations. To address both issues, the implementation of the corresponding compileClassVarDec routine can (i) handle the processing of the first token (static or field) directly, without calling a helper routine, and (ii) use a loop for handling all the variable declarations that the input contains. Generally speaking, different grammar rules entail slightly different parsing implementations. At the same time, they all follow the same contract: each compile*xxx* routine should get from the input, and handle, all the tokens that make up *xxx*, advance the tokenizer exactly beyond these tokens, and output the parse tree of *xxx*.

Recursive parsing algorithms are simple and elegant. If the language is simple, a single token lookahead is all that it takes to know which parsing rule to invoke next. For example, if the current token is let, we know that we have a *letStatement*; if the current token is while, we know that we have a *whileStatement*, and so on. Indeed, in the simple grammar shown in figure 10.3, looking ahead one token suffices to resolve, without ambiguity, which rule to use next. Grammars that have this lingual property are called *LL* (1). These grammars can be handled simply and elegantly by recursive descent algorithms, without backtracking.

The term *LL* comes from the observation that the grammar parses the input from *left* to right, performing *leftmost* derivation of the input. The (1)

parameter informs that looking ahead 1 token is all that it takes to know which parsing rule to invoke next. If that token does not suffice to resolve which rule to use, we can look ahead one more token. If this lookahead settles the ambiguity, the parser is said to be *LL* (2). And if not, we can look ahead yet another token, and so on. Clearly, as we need to look ahead further and further down the token stream, things become complicated, requiring a sophisticated parser.

The complete Jack language grammar, which we now turn to present, is *LL* (1), barring one exception that can be easily handled. Thus, Jack lends itself nicely to a recursive descent parser, which is the centerpiece of project 10.

10.2 Specification

This section consists of two parts. First, we specify the Jack language's grammar. Next, we specify a syntax analyzer designed to parse programs according to this grammar.

10.2.1 The Jack Language Grammar

The functional specification of the Jack language presented in chapter 9 was aimed at Jack programmers; we now give a formal specification of the Jack language, aimed at developers of Jack compilers. The language specification, or *grammar*, uses the following notation:

'xxx'	:	Represents language tokens that appear verbatim
xxx	:	Represents names of terminal and nonterminal elements
( )	:	Used for grouping
x | *y*	:	Either *x* or *y*
x y	:	*x* is followed by *y*
x ?	:	*x* appears 0 or 1 times
x *	:	*x* appears 0 or more times

With this notation in mind, the complete Jack grammar is specified in figure 10.5.

Lexical elements:	The Jack language includes five types of terminal elements (tokens):
keyword:	**'class'** | **'constructor'** | **'function'** | **'method'** | **'field'** | **'static'** | **'var'** | **'int'** | **'char'** | **'boolean'** **'void'** | **'true'** | **'false'** | **'null'** | **'this'** | **'let'** | **'do'** | **'if'** | **'else'** | **'while'** | **'return'**
symbol:	'{' '}' '(' ')' '[' | ']' | '.' | ',' | ';' | '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~'
integerConstant:	A decimal integer in the range 0…32767
StringConstant:	'"' A sequence of characters not including double quote or newline '"'
identifier:	A sequence of letters, digits, and underscore ('_'), not starting with a digit
Program structure:	A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A *class* is a sequence of tokens, as follows:
class:	**'class'** *className* **'{'** *classVarDec** *subroutineDec** **'}'**
classVarDec:	(**'static'** | **'field'**) *type varName* (**','** *varName*)* **';'**
type:	**'int'** | **'char'** | **'boolean'** | *className*
subroutineDec:	(**'constructor'** | **'function'** | **'method'**) (**'void'** | *type*) *subroutineName* **'('** *parameterList* **')'** *subroutineBody*
parameterList:	((*type varName*) (**','** *type varName*)*)?
subroutineBody:	**'{'** *varDec* statements* **'}'**
varDec:	**'var'** *type varName* (**','** *varName*)* **';'**
className:	*identifier*
subroutineName:	*identifier*
varName:	*identifier*
Statements:	
statements:	*statement**
statement:	*letStatement ifStatement* | *whileStatement* | *doStatement* | *returnStatement*
letStatement:	**'let'** *varName* (**'['** *expression* **']'**)? **'='** *expression* **';'**
ifStatement:	**'if'** **'('** expression **')'** **'{'** statements **'}'** (**'else'** **'{'** statements **'}'**)?
whileStatement:	**'while'** **'('** *expression* **')'** **'{'** *statements* **'}'**
doStatement:	**'do'** *subroutineCall* **';'**
returnStatement	**'return'** *expression*? **';'**
Expressions:	
expression:	*term* (*op term*)*
term:	*integerConstant* | *stringConstant* | *keywordConstant* | *varName* | *varName* **'['** *expression* **']'** | **'('** *expression* **')'** | (*unaryOp term*) | *subroutineCall*
subroutineCall:	*subroutineName* **'('** *expressionList* **')'** | (*className* | *varName*) **'.'** *subroutineName* **'('** *expressionList* **')'**
expressionList:	(*expression* (**','** *expression*) *)?
op:	**'+'** **'-'** **'*'** **'/'** **'&'** | **'|'** | **'<'** **'>'** **'='**
unaryOp:	**'-'** **'~'**
keywordConstant:	**'true'** | **'false'** | **'null'** | **'this'**

Figure 10.5 The Jack grammar.

10.2.2 A Syntax Analyzer for the Jack Language

A syntax analyzer is a program that performs both tokenizing and parsing. In Nand to Tetris, the main purpose of the syntax analyzer is to process a Jack program and understand its syntactic structure according to the Jack grammar. By *understanding* we mean that the syntax analyzer must know, at each point in the parsing process, the structural identity of the program element that it is currently handling, that is, whether it is an expression, a statement, a variable name, and so on. The syntax analyzer must possess this syntactic knowledge in a complete recursive sense. Without it, it will be

impossible to move on to code generation—the ultimate goal of the compilation process.

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

prompt> JackAnalyzer *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 analyzer operates on the current folder. For each *Xxx*.jack file, the parser creates an output file *Xxx*.xml and writes the parsed output into it. The output file is created in the same folder as that of the input. If there is a file by this name in the folder, it will be overwritten.

Input: An *Xxx*.jack file is a stream of characters. If the file represents a valid program, it can be tokenized into a stream of valid tokens, as specified by the Jack lexicon. The tokens may be separated by an arbitrary number of space characters, newline characters, and comments, which are ignored. There are three possible comment formats: /* comment until closing */, /** API comment until closing */, and // comment until the line's end.

Output: The syntax analyzer emits an XML description of the input file, as follows. For each terminal element (*token*) of type *xxx* appearing in the input, the syntax analyzer prints the marked-up output <*xxx*> *token* </*xxx*>, where *xxx* is one of the tags keyword, symbol, integerConstant, stringConstant, or identifier, representing one of the five token types recognized by the Jack language. Whenever a nonterminal language element *xxx* is detected, the syntax analyzer handles it using the following pseudocode:

```
print("<xxx>")
    Recursive code for handling the body of the xxx element
print("</xxx>")
```

where *xxx* is one of the following (and only the following) tags: class, classVarDec, subroutineDec, parameterList, subroutineBody, varDec, statements, letStatement, ifStatement, whileStatement, doStatement, returnStatement, expression, term, expressionList.

To simplify things, the following Jack grammar rules are not accounted for explicitly in the XML output: *type*, *className*, *subroutineName*, *varName*, *statement*, *subroutineCall*. We will explain this further in the next section, when we discuss the architecture of our compilation engine.

10.3 Implementation

The previous section specified *what* a syntax analyzer should do, with few implementation insights. This section describes *how* to build such an analyzer. Our proposed implementation is based on three modules:

- JackAnalyzer: main program that sets up and invokes the other modules
- JackTokenizer: tokenizer
- CompilationEngine: recursive top-down parser

In the next chapter we will extend this software architecture with two additional modules that handle the language's semantics: a *symbol table* and a *VM code writer*. This will complete the construction of a full-scale compiler for the Jack language. Since the main module that drives the parsing process in this project will end up driving the overall compilation process as well, we name it CompilationEngine.

The JackTokenizer

This module ignores all comments and white space in the input stream and enables accessing the input one token at a time. Also, it parses and provides the *type* of each token, as defined by the Jack grammar.

Routine	*Arguments*	*Returns*	*Function*
Constructor / initializer	Input file / stream	–	Opens the input `.jack` file / stream and gets ready to tokenize it.
`hasMoreTokens`	–	boolean	Are there more tokens in the input?
`advance`	–	–	Gets the next token from the input, and makes it the current token. This method should be called only if `hasMoreTokens` is true. Initially there is no current token.
`tokenType`	–	`KEYWORD`, `SYMBOL`, `IDENTIFIER`, `INT_CONST`, `STRING_CONST`	Returns the type of the current token, as a constant.
`keyWord`	–	`CLASS`, `METHOD`, `FUNCTION`, `CONSTRUCTOR`, `INT`, `BOOLEAN`, `CHAR`, `VOID`, `VAR`, `STATIC`, `FIELD`, `LET`, `DO`, `IF`, `ELSE`, `WHILE`, `RETURN`, `TRUE`, `FALSE`, `NULL`, `THIS`	Returns the keyword which is the current token, as a constant. This method should be called only if `tokenType` is `KEYWORD`.
`symbol`	–	char	Returns the character which is the current token. Should be called only if `tokenType` is `SYMBOL`.
`identifier`	–	string	Returns the string which is the current token. Should be called only if `tokenType` is `IDENTIFIER`.
`intVal`	–	int	Returns the integer value of the current token. Should be called only if `tokenType` is `INT_CONST`.
`stringVal`	–	string	Returns the string value of the current token, without the opening and closing double quotes. Should be called only if `tokenType` is `STRING_CONST`.

The CompilationEngine

The CompilationEngine is the backbone module of both the syntax analyzer described in this chapter and the full-scale compiler described in the next chapter. In the syntax analyzer, the compilation engine emits a structured

representation of the input source code wrapped in XML tags. In the compiler, the compilation engine will instead emit executable VM code. In both versions, the parsing logic and API presented below are exactly the same.

The compilation engine gets its input from a JackTokenizer and emits its output to an output file. The output is generated by a series of compile*xxx* routines, each designed to handle the compilation of a specific Jack language construct *xxx*. The contract between these routines is that each compile*xxx* routine should get from the input, and handle, all the tokens that make up *xxx*, advance the tokenizer exactly beyond these tokens, and output the parsing of *xxx*. As a rule, each compile*xxx* routine is called only if the current token is *xxx*.

Grammar rules that have no corresponding compile*xxx* routines: *type*, *className*, *subroutineName*, *varName*, *statement*, *subroutineCall*. We introduced these rules to make the Jack grammar more structured. As it turns out, the parsing logic of these rules is better handled by the routines that implement the rules that refer to them. For example, instead of writing a compileType routine, whenever *type* is mentioned in some rule *xxx*, the parsing of the possible types should be done directly by the compile *xxx* routine.

Token lookahead: Jack is almost an *LL*(1) language: the current token is sufficient for determining which CompilationEngine routine to call next. The only exception occurs when parsing a *term*, which occurs only when parsing an *expression*. To illustrate, consider the contrived yet valid expression `y+arr[5]-p.get(row)*count()-Math.sqrt(dist)/2`.This expression is made up of six terms: the variable y, the array element arr[5], the method call on the p object p.get (row), the method call on the this object count (), the call to the function (static method) Math.sqrt(dist), and the constant 2.

Suppose that we are parsing this expression and the current token is one of the identifiers y, arr, p, count, or Math. In each one of these cases, we know that we have a *term* that begins with an *identifier*, but we don't know which parsing possibility to follow next. That's the bad news; the good news is that a single lookahead to the next token is all that we need to settle the dilemma.

The need for this irregular lookahead operation occurs in the CompilationEngine twice: when parsing a *term*, which happens only when parsing an *expression*, and when parsing a *subroutineCall*. Now, an inspection of the Jack grammar shows that *subroutineCall* appears in two places only: either in a do *subroutineCall* statement or in a *term*.

With that in mind, we propose parsing do *subroutineCall* statements as if their syntax were do *expression*. This pragmatic recommendation obviates the need to write the irregular lookahead code twice. It also implies that the parsing of *subroutineCall* can now be handled directly by the compileTerm routine. In short, we've localized the need to write the irregular token lookahead code to one routine only, compileTerm, and we've eliminated the need for a compileSubroutineCall routine.

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 (by the `JackAnalyzer` module) 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 parentheses 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.

The compileExpressionList routine: returns the number of expressions in the list. The return value is necessary for generating VM code, as we'll see when we'll complete the compiler's development in project 11. In this project we generate no VM code; therefore the returned value is not used and can be ignored by routines that call compileExpressionList.

The JackAnalyzer

This is the main program that drives the overall syntax analysis process, using the services of a JackTokenizer and a CompilationEngine. For each source *Xxx*.jack file, the analyzer

1. creates a JackTokenizer from the *Xxx*.jack input file;
2. creates an output file named *Xxx*.xml; and
3. uses the JackTokenizer and the CompilationEngine to parse the input file and write the parsed code to 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.

10.4 Project

Objective: Build a syntax analyzer that parses Jack programs according to the Jack grammar. The analyzer's output should be written in XML, as specified in section 10.2.2.

This version of the syntax analyzer assumes that the source Jack code is error-free. Error checking, reporting, and handling can be added to later versions of the analyzer but are not part of project 10.

Resources: The main tool in this project is the programming language that you will use for implementing the syntax analyzer. You will also need the supplied TextComparer utility. This program allows comparing files while ignoring white space. This will help you compare the output files generated

by your analyzer with the supplied compare files. You may also want to inspect these files using an XML viewer. Any standard web browser should do the job—just use your browser's "open file" option to open the XML file that you wish to inspect.

Contract: Write a syntax analyzer for the Jack language, and test it on the supplied test files. The XML files produced by your analyzer should be identical to the supplied compare files, ignoring white space.

Test files: We provide several .jack files for testing purposes. The projects/10/Square program is a three-class app that enables moving a black square around the screen using the keyboard's arrow keys. The projects/10/ArrayTest program is a single-class app that computes the average of a user-supplied sequence of integers using array processing. Both programs were discussed in chapter 9, so they should be familiar. Note, though, that we made some harmless changes to the original code to make sure that the syntax analyzer will be fully tested on all aspects of the Jack language. For example, we've added a static variable to projects/10/Square/Main.jack, as well as a function named more, which are never used or called. These changes allow testing how the analyzer handles language elements that don't appear in the original Square and ArrayTest files, like static variables, else, and unary operators.

Development plan: We suggest developing and unit-testing the analyzer in four stages:

- First, write and test a Jack tokenizer.
- Next, write and test a basic compilation engine that handles all the features of the Jack language, except for expressions and array-oriented statements.
- Next, extend your compilation engine to handle expressions.
- Finally, extend your compilation engine to handle array-oriented statements.

We provide input .jack files and compare .xml files for unit-testing each one of the four stages, as we now turn to describe.

10.4.1 Tokenizer

Implement the JackTokenizer module specified in section 10.3. Test your implementation by writing a basic version of the JackAnalyzer, defined as follows. The analyzer, which is the main program, is invoked using the command JackAnalyzer *source*, where *source* is either a file name of the form *Xxx*.jack (the extension is mandatory) or a folder name (in which case there is no extension). In the latter case, the folder contains one or more .jack files and, possibly, other files as well. The file/folder name may include a file path. If no path is specified, the analyzer operates on the current folder.

The analyzer handles each file separately. In particular, for each *Xxx*.jack file, the analyzer constructs a JackTokenizer for handling the input and an output file for writing the output. In this first version of the analyzer, the output file is named *Xxx*T.xml (where T stands for *tokenized output*). The analyzer then enters a loop to advance and handle all the tokens in the input file, one token at a time, using the JackTokenizer methods. Each token should be printed in a separate line, as *<tokenType> token </tokenType>*, where *tokenType* is one of five possible XML tags coding the token's type. Here is an example:

Input (e.g. Prog.jack)

```
...
// Comments and white space
// are ignored.
if (x < 0) {
   let quit = "yes";
}
...
```

JackAnalyzer *Output* (e.g. ProgT.xml)

```
<tokens>
  ...
  <keyword> if </keyword>
  <symbol> ( </symbol>
  <identifier> x </identifier>
  <symbol> &lt; </symbol>
  <integerConstant> 0 </integerConstant>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <keyword> let </keyword>
  <identifier> quit </identifier>
  <symbol> = </symbol>
  <stringConstant> yes </stringConstant>
  <symbol> ; </symbol>
  <symbol> } </symbol>
  ...
</tokens>
```

Note that in the case of *string constants*, the program ignores the double quotation marks. This requirement is by design.

The generated output has two trivial technicalities dictated by XML conventions. First, an XML file must be enclosed within some begin and end tags; this convention is satisfied by the <tokens> and </tokens> tags. Second, four of the symbols used in the Jack language (<, >, ", &) are also used for XML markup; thus they cannot appear as data in XML files. Following convention, the analyzer represents these symbols as <, >, ", and &, respectively. For example, when the parser encounters the < symbol in the input file, it outputs the line <symbol> < </symbol>. This so-called *escape sequence* is rendered by XML viewers as <symbol> < </symbol>, which is what we want.

Testing Guidelines

- Start by applying your JackAnalyzer to one of the supplied .jack files, and verify that it operates correctly on a single input file.
- Next, apply your JackAnalyzer to the Square folder, containing the files Main.jack, Square.jack, and SquareGame.jack, and to the TestArray folder, containing the file Main.jack.
- Use the supplied TextComparer utility to compare the output files generated by your JackAnalyzer to the supplied .xml compare files. For example, compare the generated file SquareT.xml to the supplied compare file SquareT.xml.
- Since the generated and compare files have the same names, we suggest putting them in separate folders.

10.4.2 Compilation Engine

The next version of your syntax analyzer should be capable of parsing every element of the Jack language, except for expressions and array-oriented commands. To that end, implement the CompilationEngine module specified in section 10.3, except for the routines that handle expressions and arrays. Test the implementation by using your Jack analyzer, as follows.

For each *Xxx*.jack file, the analyzer constructs a JackTokenizer for handling the input and an output file for writing the output, named *Xxx*.xml. The

analyzer then calls the compileClass routine of the CompilationEngine. From this point onward, the CompilationEngine routines should call each other recursively, emitting XML output similar to the one shown in figure 10.4b.

Unit-test this version of your JackAnalyzer by applying it to the folder ExpressionlessSquare. This folder contains versions of the files Square.jack, SquareGame.jack, and Main.jack, in which each expression in the original code has been replaced with a single identifier (a variable name in scope). For example:

Square folder:

```
// Square.jack
...
method void incSize() {
   if (((y + size) < 254) & ((x + size) <510)
      do erase();
      let size = size + 2;
      do draw();
   }
   return;
}
...
```

ExpressionlessSquare folder:

```
// Square.jack
...
method void incSize() {
    if (x) {
        do erase();
        let size = size;
        do draw();
   }
   return;
}
...
```

Note that the replacement of expressions with variables results in nonsensical code. This is fine, since the program semantics is irrelevant to project 10. The nonsensical code is syntactically correct, and that's all that matters for testing the parser. Note also that the original and expressionless files have the same names but are located in separate folders.

Use the supplied TextComparer utility to compare the output files generated by your JackAnalyzer with the supplied .xml compare files.

Next, complete the CompilationEngine routines that handle expressions, and test them by applying your JackAnalyzer to the Square folder. Finally, complete the routines that handle arrays, and test them by applying your JackAnalyzer to the ArrayTest folder.

A web-based version of project 10 is available at www.nand2tetris.org.

10.5 Perspective

Although it is convenient to describe the structure of computer programs using parse trees and XML files, it's important to understand that compilers don't necessarily have to maintain such data structures explicitly. For example, the parsing algorithm described in this chapter parses the input as it reads it and does not keep the entire input program in memory. There are essentially two types of strategies for doing such parsing. The simpler strategy works top-down, and that is the one presented in this chapter. The more advanced parsing algorithms, which work bottom-up, were not described here since they require elaboration of more compilation theory.

Indeed, in this chapter we have sidestepped the formal language theory studied in typical compilation courses. Also, we have chosen a simple syntax for the Jack language—a syntax that can be easily compiled using recursive descent techniques. For example, the Jack grammar does not mandate the usual operator precedence in algebraic expressions evaluation, like multiplication before addition. This enabled us to avoid parsing algorithms that are more powerful, but also more intricate, than the elegant top-down parsing techniques presented in this chapter.

Every programmer experiences the disgraceful handling of compilation errors, which is typical of many compilers. As it turns out, error diagnostics and reporting are a challenging problem. In many cases, the impact of an error is detected several or many lines of code after the error was made. Therefore, error reporting is sometimes cryptic and unfriendly. Indeed, one aspect in which compilers vary greatly is their ability to diagnose, and help debug, errors. To do so, compilers persist parts of the parse tree in memory and extend the tree with annotations that help pinpoint the source of errors and backtrack the diagnostic process, as needed. In Nand to Tetris we bypass all these extensions, assuming that the source files that the compiler handles are error-free.

Another topic that we hardly mentioned is how the syntax and semantics of programming languages are studied in computer and cognitive science. There is a rich theory of formal and natural languages that discusses properties of classes of languages, as well as meta-languages and formalisms for specifying them. This is also the place where computer science meets the study of human languages, leading to the vibrant areas of research and practice known as computational linguistics and natural language processing.

Finally, it is worth mentioning that syntax analyzers are typically not standalone programs and are rarely written from scratch. Instead, programmers usually build tokenizers and parsers using a variety of *compiler generator* tools like LEX (for *LEXical analysis*) and YACC (for *Yet Another Compiler Compiler*). These utilities receive as input a context-free grammar and produce as output syntax analysis code capable of tokenizing and parsing programs written in that grammar. The generated code can then be customized to fit the specific needs of the compiler writer. Following the "show me" spirit of Nand to Tetris, though, we have chosen not to use such black boxes in the implementation of our compiler, but rather build everything from the ground up.

---

1. And here lies a crucial difference between programming languages and natural languages. In natural languages, we can say things like "Whoever saves one life, saves the world entire." In the English language, putting the adjective after the noun is grammatically incorrect. Yet, in this particular case, it sounds perfectly acceptable. Unlike programming languages, natural languages mandate a poetic license to break grammar rules, so long as the writer knows what he or she is doing. This freedom of expression makes natural languages infinitely rich.

11 Compiler II: Code Generation

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

—R. Buckminster Fuller (1895–1993)

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

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

the context of this chapter, the source and the target are Jack and the VM language, respectively.

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

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

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

So what's in it for you? Although many professionals are eager to understand how compilers work, few end up getting their hands dirty and building a compiler from the ground up. That's because the cost of this experience—at least in academia—is typically a daunting, full-semester elective course. Nand to Tetris packs the key elements of this experience into four chapters and projects, culminating in the present chapter. In the

process, we discuss and illustrate the key algorithms, data structures, and programming tricks underlying the construction of typical compilers. Seeing these clever ideas and techniques in action leads one to marvel, once again, at how human ingenuity can dress up a primitive switching machine to look like something approaching magic.

11.1 Code Generation

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

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

Throughout the chapter, we present compilation examples of various parts of the Point class presented previously in the book. We repeat the class declaration in figure 11.1, which illustrates most of the features of the Jack language. We advise taking a quick look at this Jack code now to refresh your memory about the Point class functionality. You will then be ready to delve into the illuminating journey of systematically reducing this high-level functionality—and any other similar object-based program—into VM code.

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

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

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

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

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

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

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

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

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

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

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

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

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

11.1.1 Handling Variables

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

The second challenge faced by compilers is that variables of different *kinds* have different life cycles. Class-level static variables are shared globally by all the subroutines in the class. Therefore, a single copy of each static variable should be kept alive during the complete duration of the program's execution. Instance-level field variables are treated differently: each object (instance of the class) must have a private set of its field

variables, and, when the object is no longer needed, this memory must be freed. Subroutine-level local and argument variables are created each time a subroutine starts running and must be freed when the subroutine terminates.

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

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

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

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

The variable properties can be managed conveniently using a *symbol table*. When a static, field, local, or argument variable is declared in the

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

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

High-level (Jack) code

```
class Point {
   field int x, y;
   static int pointCount;
   ...
   method int distance(Point other) {
      var int dx, dy;
      let dx = x - other.getx();
      let dy = y - other.gety();
      return Math.sqrt((dx*dx) + (dy*dy));
   }
   ...
}
```

name	type	kind	#
x	int	field	0
y	int	field	1
pointCount	int	static	0

Class-level symbol table

name	type	kind	#
this	Point	arg	0
other	Point	arg	1
dx	int	var	0
dy	int	var	1

Subroutine-level symbol table

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

The scopes are nested, with inner scopes hiding outer ones. For example, when the Jack compiler encounters the expression `x + 17`, it first checks whether `x` is a subroutine-level variable (local or argument). Failing that, the compiler checks whether `x` is a static variable or a field. Some languages feature nested scoping of unlimited depth, allowing variables to be local in any block of code in which they are declared. To support unlimited nesting, the compiler can use a linked list of symbol tables, each reflecting a single scope nested within the next one in the list. When the compiler fails to find the variable in the table associated with the current scope, it looks it up in

the next table in the list, from inner scopes outward. If the variable is not found in the list, the compiler can throw an "undeclared variable" error.

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

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

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

Handling variables in statements: When the compiler encounters a variable in a statement, it looks up the variable name in the subroutine-level symbol table. If the variable is not found, the compiler looks it up in the class-level symbol table. Once the variable has been found, the compiler can complete the statement's translation. For example, consider the symbol tables shown in figure 11.2, and suppose we are compiling the high-level statement `let y = y + dy`. The compiler will translate this statement into the VM commands `push this 1`, `push local 1`, `add`, `pop this 1`. Here we assume that the compiler knows how to handle expressions and `let` statements, subjects which are taken up in the next two sections.

11.1.2 Compiling Expressions

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

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

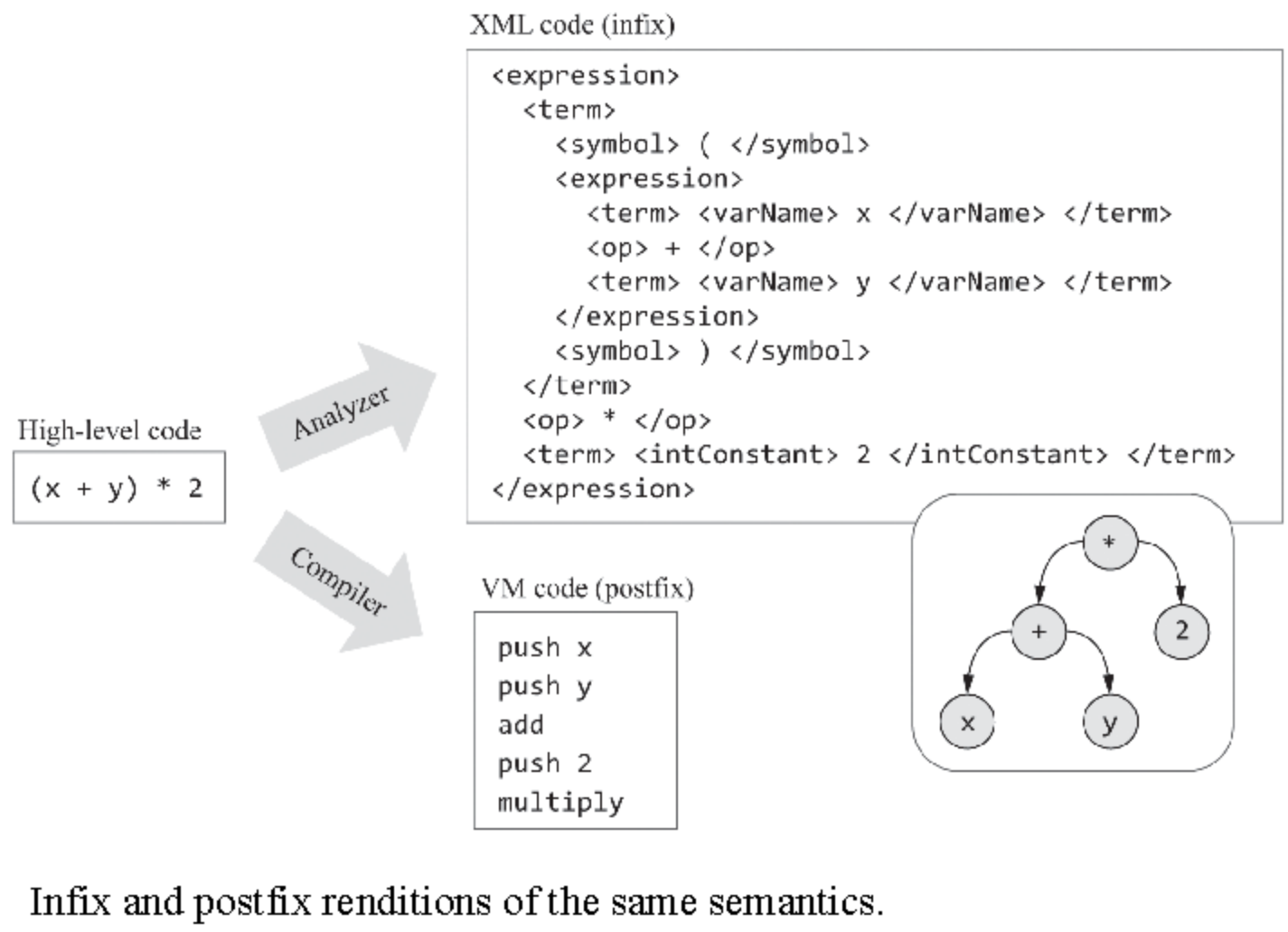

Figure 11.3 Infix and postfix renditions of the same semantics.

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

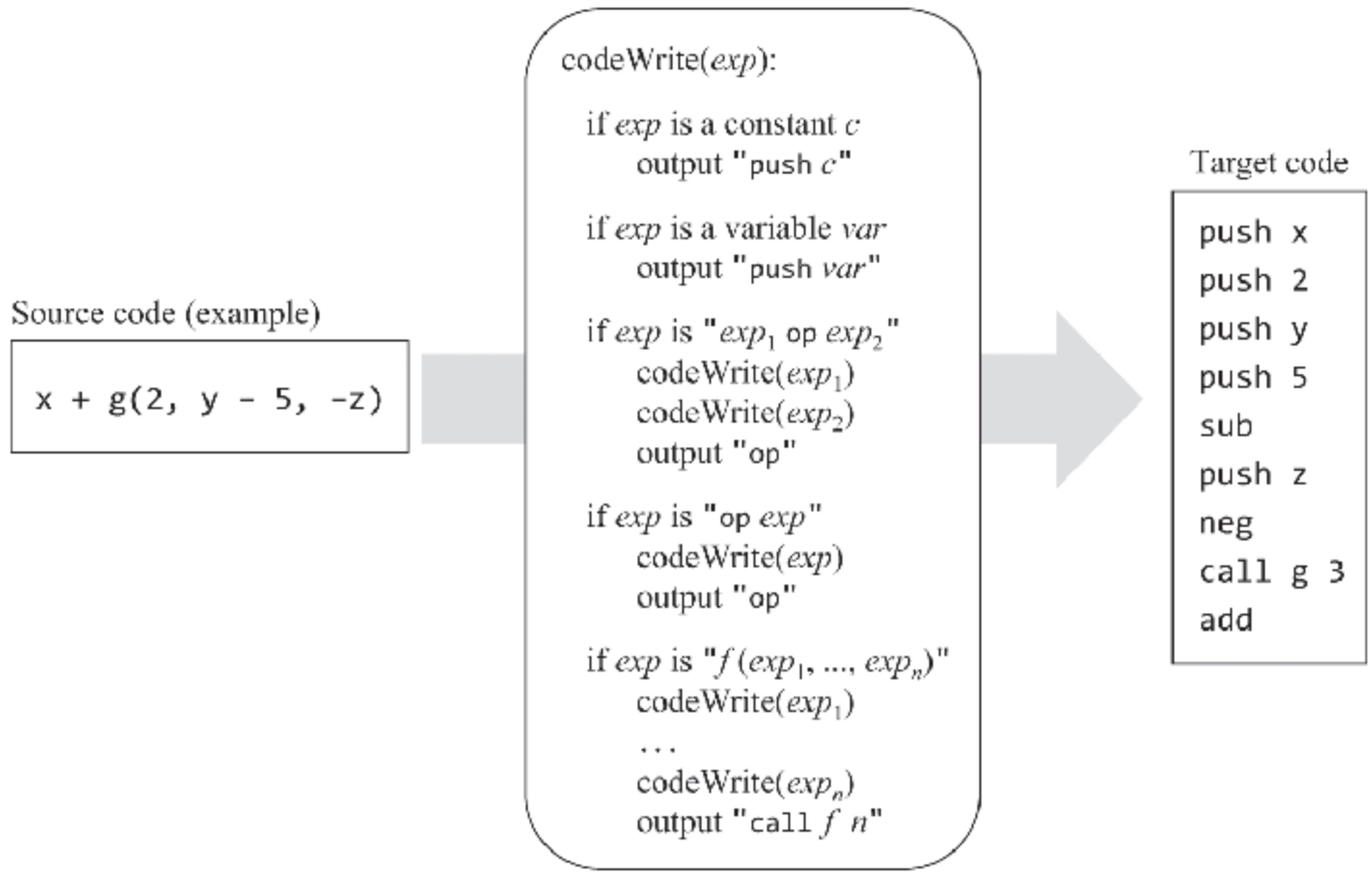

Figure 11.4 A VM code generation algorithm for expressions, and a compilation example. The algorithm assumes that the input expression is valid. The final implementation of this algorithm should replace the emitted symbolic variables with their corresponding symbol table mappings.

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

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

Definition (from the Jack grammar):

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

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

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

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

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

unaryOp: '-' | '~'

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

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

Examples:

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

Figure 11.5 Expressions in the Jack language.

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

11.1.3 Compiling Strings

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

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

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

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

11.1.4 Compiling Statements

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

Compiling return statements: Now that we know how to compile expressions, the compilation of return *expression* is simple. First, we call the compileExpression routine, which generates VM code designed to evaluate and put the expression's value on the stack. Next, we generate the VM command return.

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

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

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

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

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

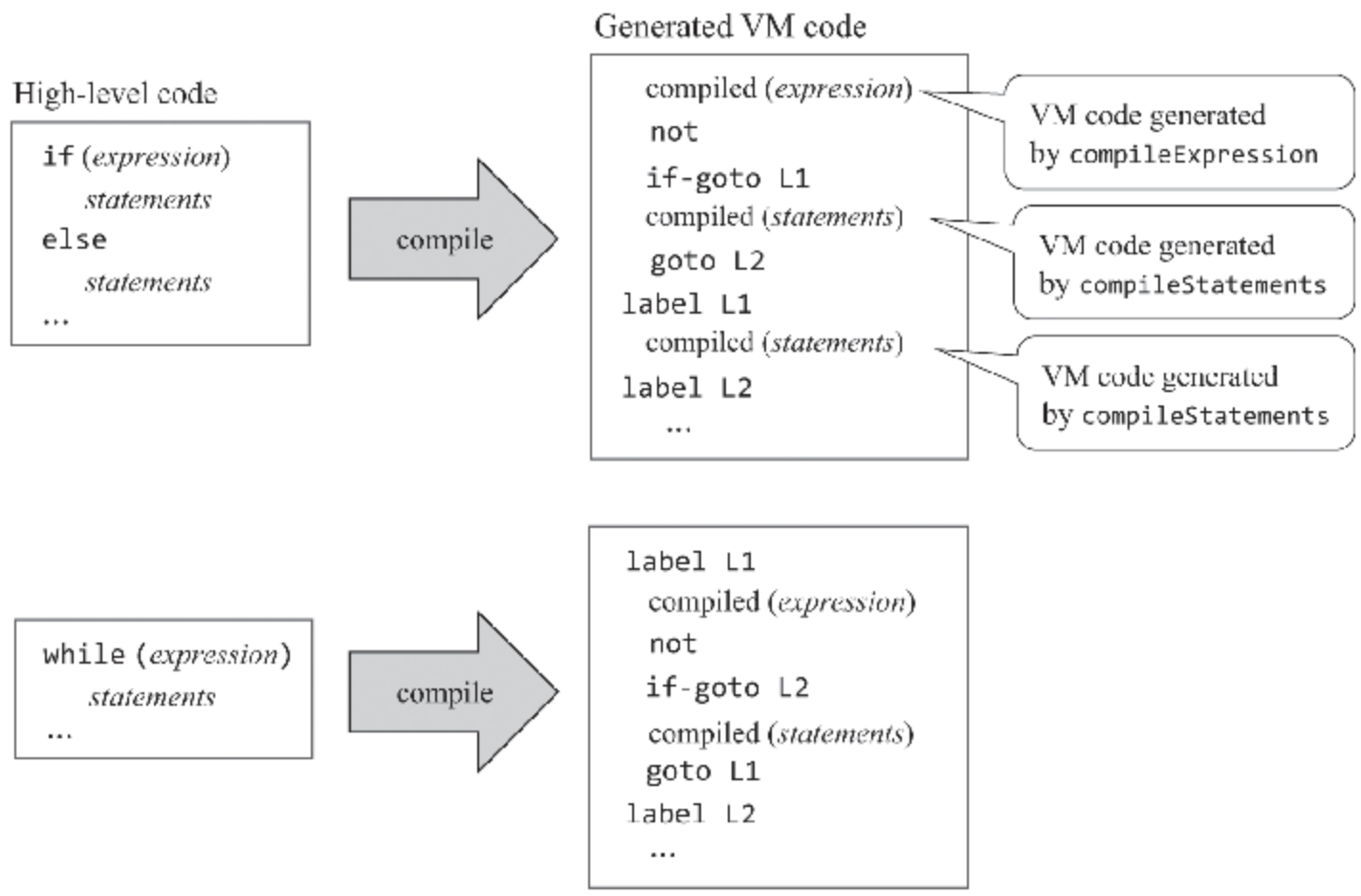

Figure 11.6 Compiling if and while statements. The L1 and L2 labels are generated by the compiler.

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

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

11.1.5 Handling Objects

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

Object-oriented languages feature means for declaring and manipulating aggregate abstractions known as *objects*. Each object is implemented physically as a memory block which can be referenced by a static, field, local, or argument variable. The reference variable, also known as an *object variable*, or *pointer*, contains the memory block's base address. The operating system realizes this model by managing a logical area in the RAM named *heap*. The *heap* is used as a memory pool from which memory blocks are carved, as needed, for representing new objects. When an object is no longer needed, its memory block can be freed and recycled back to the heap. The compiler stages these memory management actions by calling OS functions, as we'll see later.

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

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

11.1.5.1 Compiling Constructors

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

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

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

High-level code

```
// Can appear in any class:
...
// Declares three local variables:
var Point p1, p2;
var int d;
...
// Constructs two objects:
let p1 = Point.new(2,3);
let p2 = Point.new(5,7);
...
```

compile

Generated VM code

```
...
// var Point p1, p2;
// var int d;
// The compilation engine adds p1, p2, and d to the
// subroutine's symbol table; no code is generated.
...
// let p1 = Point.new(2,3);
// To handle the right-hand side, the compilation engine calls
// compileExpression. This results in putting 2 and 3 at the
// stack's top and generating the command call Point.new 2
// The compilation engine generates the command:
pop local 0
// let p2 = Point.new(5,7);
// Similar.
...
```

name	type	kind	#
p1	Point	local	0
p2	Point	local	1
d	int	local	2

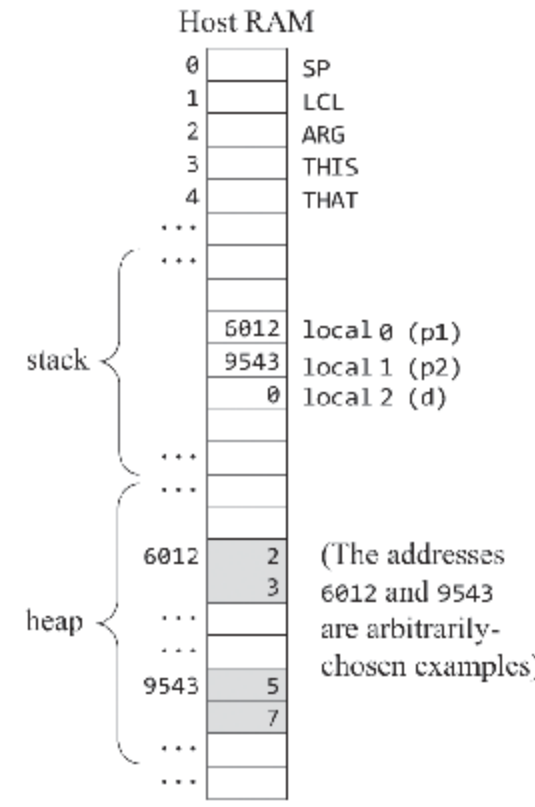

Figure 11.7 Object construction from the caller's perspective. In this example, the caller declares two object variables and then calls a class constructor for constructing the two objects. The constructor works its magic, allocating memory blocks for representing the two objects. The calling code then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

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