

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 BC)

The previous chapter introduced *Jack*—a simple object-based programming language whose syntax resembles that of Java and C#. 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. This information enables us to reconstruct the program’s logic 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 chapter 11.

How can we tell that a compiler is capable of “understanding” the language’s syntax? Well, as long as the code generated by the compiler is doing what it is supposed to do, 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 some passive way to demonstrate that it “understands” the source program. Our solution is to have the syntax analyzer output an XML file whose format reflects the syntactic structure of the input program. By inspecting the generated XML output, we should be able to ascertain that the analyzer is parsing input programs correctly.

The chapter starts 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 algorithms for building them. This sets the stage for a Specification section that presents the formal grammar of the Jack language and the format of 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 final Project section gives step-by-step instructions and test programs for actually building and testing the syntax analyzer. In the next chapter, this analyzer will be extended into a full-scale compiler.

Writing a compiler from scratch is a task that brings to bear several fundamental topics in computer science. It requires an understanding of language translation and parsing techniques, use of classical data structures like trees and hash tables, and application of sophisticated recursive compilation algorithms. For all these reasons, writing a compiler is also a challenging task. However, by splitting the compiler’s construction into two separate projects (or actually four, counting the VM projects as well), and by allowing the modular development and unit-testing of each part in isolation, we have turned the

compiler's development into a surprisingly manageable and self-contained activity.

Why should you go through the trouble of building a compiler? First, a hands-on grasp of compilation internals will turn you into a significantly better high-level programmer. Second, the same types of rules and grammars used for describing programming languages are also used for specifying the syntax of data sets in diverse applications ranging from computer graphics to database management to communications protocols to bioinformatics. Thus, while most programmers will not have to develop compilers in their careers, it is very likely that they will be required to parse and manipulate files of some complex syntax. These tasks will employ the same concepts and techniques used in the parsing of programming languages, as described in this chapter.

10.1 Background

A typical compiler consists of two main modules: syntax analysis and code generation. The syntax analysis task is usually divided further into two modules: tokenizing, or grouping of input characters into language atoms, and parsing, or attempting to match the resulting atoms stream to the syntax rules of the underlying language. Note that these activities are completely independent of the target language into which we seek to translate the source program. 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 compiled program as an XML file. This decision has two benefits. First, the XML file can be easily viewed in any Web browser, 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 a software architecture that can be later morphed into a full-scale compiler. In particular, in the next chapter we will simply replace the routines that generate the passive XML code with routines that generate executable VM code, leaving the rest of the compiler's architecture intact (see figure 10.1).

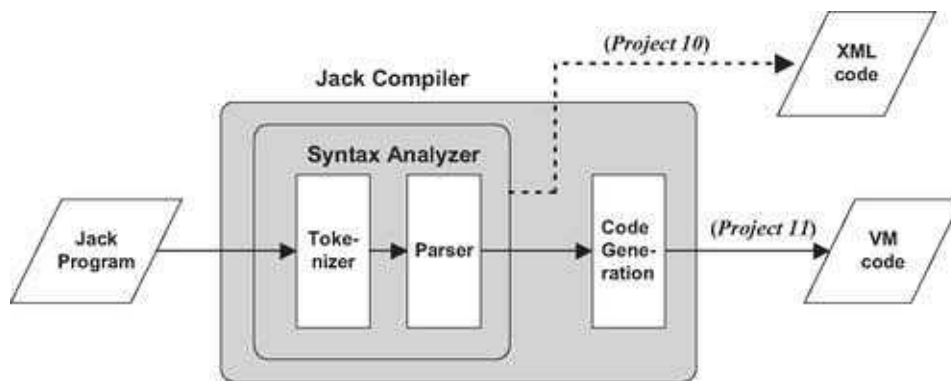


Figure 10.1 The Jack Compiler. The project in chapter 10 is an intermediate step, designed to localize the development and unit-testing of the *syntax analyzer* module.

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 some explanation. When humans read a computer program, they immediately recognize the program’s structure. They 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. This understanding is not trivial, since it requires an ability to identify and classify nested patterns: In a typical program, classes contain methods that contain statements that contain other statements that contain expressions, and so on. In order to recognize these language constructs correctly, human cognition must recursively map them on the range of textual patterns permitted by the language syntax.

When it comes to understanding a natural language like English, the question of how syntax rules are represented in the human brain and whether they are innate or acquired is a subject of intense debate. However, if we limit our attention to formal languages—artifacts whose simplicity hardly justifies the title “language”—we know precisely how to formalize their syntactic structure. In particular,

programming languages are usually described using a set of rules called context-free grammar. To understand—parse—a given program means to determine the exact correspondence between the program's text and the grammar's rules. In order to do so, we first have to transform the program's text into a list of tokens, as we now describe.

10.1.1 Lexical Analysis

In its plainest syntactic form, a program is simply a sequence of characters, stored in a text file. The first step in the syntax analysis of a program is to group the characters into tokens (as defined by the language syntax), while ignoring white space and comments. This step is usually called lexical analysis, scanning, or tokenizing. Once a program has been tokenized, the tokens (rather than the characters) are viewed as its basic atoms, and the tokens stream becomes the main input of the compiler. Figure 10.2 illustrates the tokenizing of a typical code fragment, taken from a C or Java program.

As seen in figure 10.2, tokens fall into distinct categories, or types: `while` is a keyword, `count` is an identifier, `<=` is an operator, and so on. In general, each programming language specifies the types of tokens it allows, as well as the exact syntax rules for combining them into valid programmatic structures. For example, some languages may specify that “++” is a valid operator token, while other languages may not. In the latter case, an expression containing two consecutive “+” characters will be rendered invalid by the compiler.

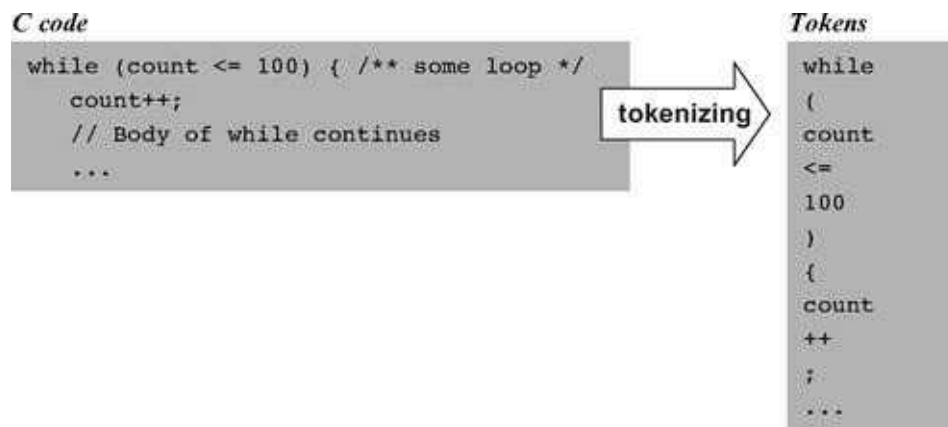


Figure 10.2 Lexical analysis.

10.1.2 Grammars

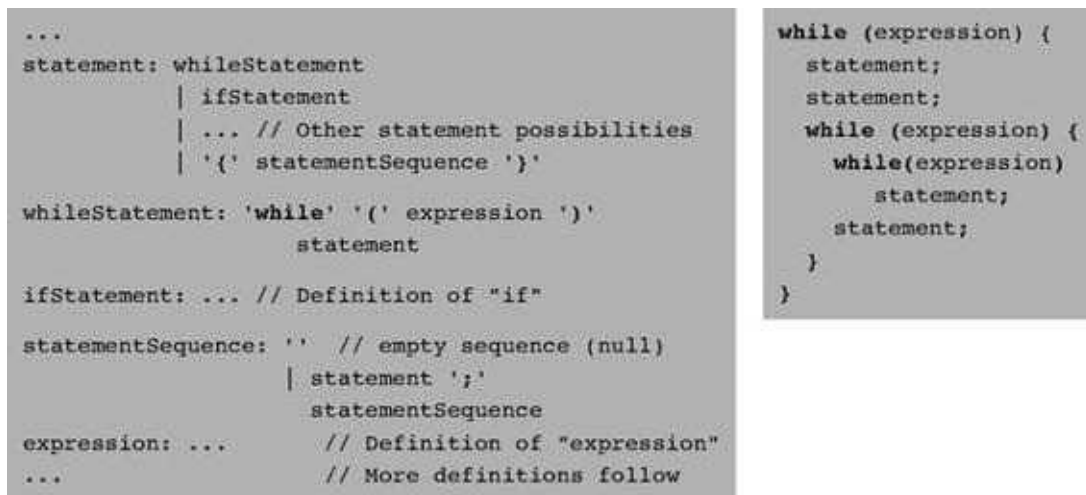
Once we have lexically analyzed a program into a stream of tokens, we now face the more challenging task of parsing the tokens stream into a formal structure. In other words, we have to figure out how to group the tokens into language constructs like variable declarations, statements, expressions, and so on. These grouping and classification tasks can be done by attempting to match the tokens stream on some predefined set of rules known as a grammar.

Almost all programming languages, as well as most other formal languages used for describing the syntax of complex file types, can be specified using formalisms known as context-free grammars. A context-free grammar is a set of rules specifying how syntactic elements in some language can be formed from simpler ones. For example, the Java grammar allows us to combine the atoms 100, count, and <= into the expression count<=100. In a similar fashion, the Java grammar allows us to ascertain that the text count<=100 is a valid Java expression. Indeed, each grammar has a dual perspective. From a declarative standpoint, the grammar specifies allowable ways to combine tokens, also called terminals, into higher-level syntactic elements, also called non-terminals. From an analytic standpoint, the grammar is a prescription for doing the reverse: parsing a given input (set of tokens resulting from the tokenizing phase) into non-terminals, lower-level non-terminals, and eventually terminals that cannot be decomposed any further. Figure 10.3 gives an example of a typical grammar.

In this chapter we specify grammars using the following notation: Terminal elements appear in bold text enclosed in single quotes, and non-terminal elements in regular font. When there is more than one way to parse a non-terminal, the “|” notation is used to list the alternative possibilities. Thus, figure 10.3 specifies that a statement can be either a whileStatement, or an ifStatement, and so on. Typically, grammar rules are highly recursive, and figure 10.3 is no exception. For example, statementSequence is either null, or a single statement followed by a semicolon and a statementSequence. This recursive definition can accommodate a sequence of 0, 1, 2, or any other positive number of semicolon-separated statements. As an exercise, the reader may use figure 10.3 to ascertain that the text appearing in the right side of the figure constitutes a valid C code. You may start by trying to match the entire text with statement, and work your way from there.

10.1.3 Parsing

The act of checking whether a grammar “accepts” an input text as valid is called parsing. As we noted earlier, parsing a given text means determining the exact correspondence between the text and the rules of a given grammar. Since the grammar rules are hierarchical, the output generated by the parser can be described in a tree-oriented data structure called a parse tree or a derivation tree. Figure 10.4 gives a typical example.



```
...
statement: whileStatement
        | ifStatement
        | ... // Other statement possibilities
        | '{' statementSequence '}'

whileStatement: 'while' '(' expression ')'
                statement

ifStatement: ... // Definition of "if"

statementSequence: '' // empty sequence (null)
                  | statement ';'
                  | statementSequence

expression: ... // Definition of "expression"
...          // More definitions follow
```

```
while (expression) {
    statement;
    statement;
    while (expression) {
        while(expression)
            statement;
        statement;
    }
}
```

Figure 10.3 A subset of the C language grammar (left) and a sample code segment accepted by this grammar (right).

Note that as a side effect of the parsing process, the entire syntactic structure of the input text is uncovered. Some compilers represent this tree by an explicit data structure that is further used for code generation and error reporting. Other compilers (including the one that we will build) represent the program’s structure implicitly, generating code and reporting errors on the fly. Such compilers don’t have to hold the entire program structure in memory, but only the subtree associated with the presently parsed element. More about this later.

Recursive Descent Parsing There are several algorithms for constructing parse trees. The top-down approach, also called recursive descent parsing, attempts to parse the tokens stream recursively, using the nested structure prescribed by the language grammar. Let us consider how a parser program that implements this strategy can be written. For every rule in the grammar describing a non-terminal, we can equip the parser program with a recursive routine designed to parse that non-terminal. If the non-terminal consists of terminal atoms only, the routine can simply process them. Otherwise, for every non-terminal building block in the rule’s right-hand side, the routine can recursively call the routine designed to parse this non-terminal. The process will continue recursively, until all the terminal atoms have been reached and processed.

```
while (count<=100) {
    count++;
    // ...
}
```

```
statement: whileStatement | ifStatement
        | ... | '{' statementSequence '}'
whileStatement: 'while' '(' expression ')'
                statement
ifStatement: ... // Definition of "if"
statementSequence: '' // Null
                  | statement ';' statementSequence
expression: ... // Definition of "expression"
```

```
while
(
count
<=
100
)
{
count
++
;
...
}
```

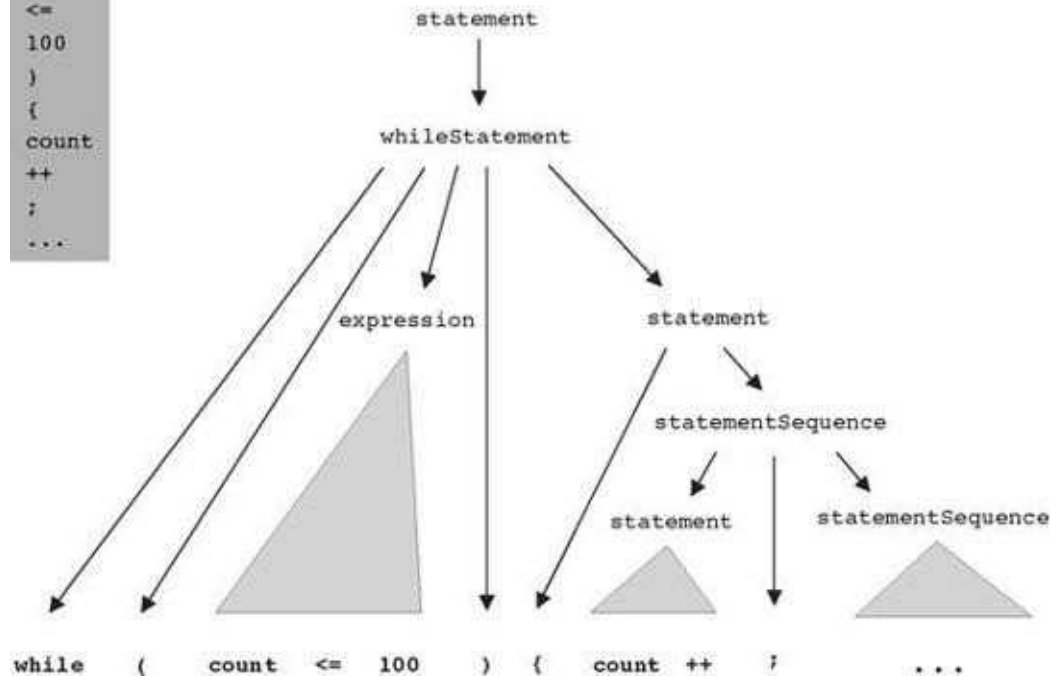


Figure 10.4 Parse tree of a program segment according to a grammar segment. Solid triangles represent lower-level parse trees.

To illustrate, suppose we have to write a recursive descent parser that follows the grammar from figure 10.3. Since the grammar has five derivation rules, the parser implementation can consist of five major routines: `parseStatement()`, `parseWhileStatement()`, `parseSelfStatement()`, `parseStatementSequence()`, and `parseExpression()`. The parsing logic of these routines should follow the syntactic patterns appearing in the right-hand sides of the corresponding grammar rules. Thus `parseStatement()` should probably start its processing by determining what is the first token in the input. Having established the token's identity, the routine could determine which statement we are in, and then call the parsing routine associated with this statement type.

For example, if the input stream were that depicted in figure 10.4, the routine will establish that the first token is `while`, then call the `parseWhileStatement()` routine. According to the corresponding grammar rule, this routine should next attempt to read the terminals “`while`” and “`(`”, and then call `parseExpression()` to parse the non-terminal expression. After `parseExpression()` would return (having parsed the “`count<=100`” sequence in our example), the grammar dictates that `parseWhileStatement()` should attempt to read the terminal “`)`” and then recursively call `parseStatement()`. This call would continue recursively, until at some point only terminal atoms are read. Clearly, the same logic can also be used for detecting syntax errors in the source program. The better the compiler, the better will be its error diagnostics.

LL(0) Grammars Recursive parsing algorithms are simple and elegant. The only possible complication arises when there are several alternatives for parsing non-terminals. For example, when `parseStatement()` attempts to parse a statement, it does not know in advance whether this statement is a while-statement, an if-statement, or a bunch of statements enclosed in curly brackets. The span of possibilities is determined by the grammar, and in some cases it is easy to tell which alternative we are in. For example, consider figure 10.3. If the first token is “while,” it is clear that we are faced with a while statement, since this is the only alternative in the grammar that starts with a “while” token. This observation can be generalized as follows: whenever a non-terminal has several alternative derivation rules, the first token suffices to resolve without ambiguity which rule to use. Grammars that have this property are called *LL(0)*. These grammars can be handled simply and neatly by recursive descent algorithms.

When the first token does not suffice to resolve the element’s type, it is possible that a “look ahead” to the next token will settle the dilemma. Such parsing can obviously be done, but as we need to look ahead at more and more tokens down the stream, things start getting complicated. The Jack language grammar, which we now turn to present, is almost *LL(0)*, and thus it can be handled rather simply by a recursive descent parser. The only exception is the parsing of expressions, where just a little look ahead is necessary.

10.2 Specification

This section has two distinct 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 given in chapter 9 was aimed at Jack programmers. We now turn to giving a formal specification of the language, aimed at Jack compiler developers. Our grammar specification is based on the following conventions:

‘xxx’: quoted boldface is used for tokens that appear verbatim (“terminals”);

xxx: regular typeface is used for names of language constructs (“non-terminals”);

(): parentheses are used for grouping of language constructs;

x|y: indicates that either x or y can appear;

x?: indicates that x appears 0 or 1 times;

x*: indicates that x appears 0 or more times.

The Jack language syntax is given in figure 10.5, using the preceding conventions.

10.2.2 A Syntax Analyzer for the Jack Language

The main purpose of the syntax analyzer is to read 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 reading, namely, 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 overall compiler.

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 number in the range 0 .. 32767.
StringConstant	''' A sequence of Unicode 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 structured according to the following context free syntax:
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

Figure 10.5 Complete grammar of the Jack language.

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 ']' subroutineCall '(' expression ')' unaryOp term
subroutineCall:	subroutineName '(' expressionList ')' (className varName) '.' subroutineName '(' expressionList ')'
expressionList:	(expression (',' expression)*)?
op:	'+' '-' '*' '/' '&' ' ' '<' '>' '='
unaryOp:	'-' '~'
KeywordConstant:	'true' 'false' 'null' 'this'

The fact that the syntax analyzer “understands” the programmatic structure of the input can be demonstrated by having it print the processed text in some well-structured and easy-to-read format. One can think of several ways to cook up such a demonstration. In this book, we decided to have the syntax analyzer output an XML file whose marked-up format reflects the syntactic structure of the underlying program. By viewing this XML output file—a task that can be conveniently done with any Web browser—one should be able to tell right away if the syntax analyzer is doing the job or not.

10.2.3 The Syntax Analyzer's Input

The Jack syntax analyzer accepts a single command line parameter, as follows:

```
prompt> JackAnalyzer source
```

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

Each Xxx.jack file is a stream of characters. This stream should be tokenized into a stream of tokens according to the rules specified by the lexical elements of the Jack language (see figure 10.5, top). The tokens may be separated by an arbitrary number of space characters, newline characters, and comments, which are ignored. Comments are of the standard formats /* comment until closing */, /** API comment */, and // comment to end of line.

10.2.4 The Syntax Analyzer's Output

Recall that the development of the Jack compiler is split into two stages (see figure 10.1), starting with the syntax analyzer. In this chapter, we want the syntax analyzer to emit an XML description of the input program, as illustrated in figure 10.6. In order to do so, the syntax analyzer has to recognize two major types of language constructs: terminal and non-terminal elements. These constructs are handled as follows.

Non-Terminals Whenever a non-terminal language element of type *xxx* is encountered, the syntax analyzer should generate the marked-up output:

```
<xxx>  
    Recursive code for the body of the xxx element.  
</xxx>
```

Where *xxx* is one of the following (and only the following) non-terminals of the Jack grammar:

- *class*, *classVarDec*, *subroutineDec*, *parameterList*, *subroutineBody*, *varDec*;
- *statements*, *whileStatement*, *ifStatement*, *returnStatement*, *letStatement*, *doStatement*;
- *expression*, *term*, *expressionList*.

Terminals Whenever a terminal language element of type *xxx* is encountered, the syntax analyzer should generate the marked-up output:

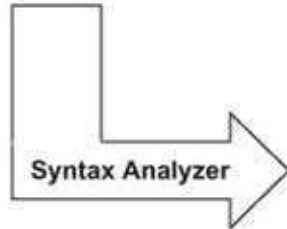
```
<xxx> terminal </xxx>
```

Where *xxx* is one of the five token types recognized by the Jack language (as specified in the Jack grammar's "lexical elements" section), namely, *keyword*, *symbol*, *integerConstant*, *stringConstant*, or *identifier*.

Figure 10.6, which shows the analyzer's output, should evoke some sense of *déjà vu*. Earlier in the chapter we noted that the structure of a program can be analyzed into a parse tree. And indeed, XML output is simply a textual description of a tree. In particular, note that in a parse tree, the non-terminal nodes form a "super structure" that describes how the tree's terminal nodes (the tokens) are grouped into language constructs. This pattern is mirrored in the XML output, where non-terminal XML elements describe how terminal XML items are arranged. In a similar fashion, the tokens generated by the tokenizer form the lowest level of the XML output, just as they form the terminal leaves of the program's parse tree.

Analyzer's input (Jack code)

```
Class Bar {  
  method Fraction foo(int y) {  
    var int temp; // a variable  
    let temp = (xxx+12)*-63;  
    ...  
  }  
}
```



Analyzer's output (XML code)

```
<class>  
  <keyword> class </keyword>  
  <identifier> Bar </identifier>  
  <symbol> { </symbol>  
  <subroutineDec>  
    <keyword> method </keyword>  
    <identifier> Fraction </identifier>  
    <identifier> foo </identifier>  
    <symbol> ( </symbol>  
    <parameterList>  
      <keyword> int </keyword>  
      <identifier> y </identifier>  
    </parameterList>  
    <symbol> ) </symbol>  
    <subroutineBody>  
      <symbol> { </symbol>  
      <varDec>  
        <keyword> var </keyword>  
        <keyword> int </keyword>  
        <identifier> temp </identifier>  
        <symbol> ; </symbol>  
      </varDec>  
      <statements>  
        <letStatement>  
          <keyword> let </keyword>  
          <identifier> temp </identifier>  
          <symbol> = </symbol>  
          <expression>  
            ...  
          </expression>  
          <symbol> ; </symbol>  
        ...  
      </statements>  
    </subroutineBody>  
  </subroutineDec>  
  </class>
```

Figure 10.6 Jack Analyzer in action.

Code Generation We have just finished specifying the analyzer's XML output. In the next chapter we replace the software that generates this output with software that generates executable VM code, leading to a full-scale Jack compiler.

10.3 Implementation

Section 10.2 gave all the information necessary to build a syntax analyzer for the Jack language, without any implementation details. This section describes a proposed software architecture for the syntax analyzer. We suggest arranging the implementation in three modules:

- JackAnalyzer: top-level driver that sets up and invokes the other modules;
- JackTokenizer: tokenizer;
- CompilationEngine: recursive top-down parser.

These modules are designed to handle the language's syntax. In the next chapter we extend this 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 module that drives the parsing process in this project will also drive the overall compilation in the next project, we call it CompilationEngine.

10.3.1 The *JackAnalyzer* Module

The analyzer program operates on a given source, where source is either a file name of the form Xxx.jack or a directory name containing one or more such files. For each source Xxx.jack file, the analyzer goes through the following logic:

1. Create a *JackTokenizer* from the Xxx.jack input file.
2. Create an *output file* called Xxx.xml and prepare it for writing.
3. Use the *CompilationEngine* to compile the input *JackTokenizer* into the *output file*.

10.3.2 The *JackTokenizer* Module

JackTokenizer: Removes all comments and white space from the input stream and breaks it into Jack-language tokens, as specified by the Jack grammar.

Routine	Arguments	Returns	Function
Constructor	input file/ stream	—	Opens the input file/stream and gets ready to tokenize it.
hasMoreTokens	—	Boolean	Do we have more tokens in the input?
advance	—	—	Gets the next token from the input and makes it the current token. This method should only be called if <i>hasMoreTokens()</i> is true. Initially there is no current token.
tokenType	—	KEYWORD, SYMBOL, IDENTIFIER, INT_CONST, STRING_CONST	Returns the type of the current token.
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. Should be called only when <i>tokenType()</i> is KEYWORD.
symbol	—	Char	Returns the character which is the current token. Should be called only when <i>tokenType()</i> is SYMBOL.
identifier	—	String	Returns the identifier which is the current token. Should be called only when <i>tokenType()</i> is IDENTIFIER.
intVal		Int	Returns the integer value of the current token. Should be called only when <i>tokenType()</i> is INT_CONST.

Routine	Arguments	Returns	Function
stringVal		String	Returns the string value of the current token, without the double quotes. Should be called only when <i>tokenType()</i> is STRING_CONST.

10.3.3 The CompilationEngine Module

CompilationEngine: Effects the actual compilation output. Gets its input from a JackTokenizer and emits its parsed structure into an output file/stream. The output is generated by a series of `compilexxx ()` routines, one for every syntactic element `xxx` of the Jack grammar. The contract between these routines is that each `compilexxx ()` routine should read the syntactic construct `xxx` from the input, `advance ()` the tokenizer exactly beyond `xxx`, and output the parsing of `xxx`. Thus, `compilexxx ()` may only be called if indeed `xxx` is the next syntactic element of the input.

In the first version of the compiler, described in chapter 10, this module emits a structured printout of the code, wrapped in XML tags. In the final version of the compiler, described in chapter 11, this module generates executable VM code. In both cases, the parsing logic and module API are exactly the same.

Routine	Arguments	Returns	Function
Constructor	Input stream/file Output stream/file	—	Creates a new compilation engine with the given input and output. The next routine called must be <code>compileClass ()</code> .
<code>CompileClass</code>	—	—	Compiles a complete class.
<code>CompileClassVarDec</code>	—	—	Compiles a static declaration or a field declaration.
<code>CompileSubroutine</code>	—	—	Compiles a complete method, function, or constructor.
<code>compileParameterList</code>	—	—	Compiles a (possibly empty) parameter list, not including the enclosing “ <code>()</code> ”.

Routine	Arguments	Returns	Function
<code>compileVarDec</code>	—	—	Compiles a <code>var</code> declaration.
<code>compileStatements</code>	—	—	Compiles a sequence of statements, not including the enclosing “{}”.
<code>compileDo</code>	—	—	Compiles a <code>do</code> statement.
<code>compileLet</code>	—	—	Compiles a <code>let</code> statement.
<code>compileWhile</code>	—	—	Compiles a <code>while</code> statement.
<code>compileReturn</code>	—	—	Compiles a <code>return</code> statement.
<code>compileIf</code>	—	—	Compiles an <code>if</code> statement, possibly with a trailing <code>else</code> clause.
<code>CompileExpression</code>	—	—	Compiles an expression.
<code>CompileTerm</code>	—	—	Compiles a <i>term</i> . This routine is faced with a slight difficulty when trying to decide between some of the alternative parsing rules. Specifically, if the current token is an identifier, the routine must distinguish between a variable, an array entry, and a subroutine call. A single look-ahead token, which may be one of “[”, “(”, or “.” suffices to distinguish between the three possibilities. Any other token is not part of this term and should not be advanced over.
<code>CompileExpressionList</code>	—	—	Compiles a (possibly empty) comma-separated list of expressions.

10.4 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 runs “on-line,” meaning that it 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 this is the one presented in this chapter. The more advanced algorithms, which work bottom-up, are not described here since they require some elaboration of theory.

Indeed, in this chapter we have sidestepped almost all the formal language theory studied in typical compilation courses. We were able to do so by choosing a very 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 expressions evaluation (multiplication before addition, and so on). This has enabled us to avoid parsing algorithms that are more powerful yet much more technical than the elegant top-down parsing techniques presented in the chapter.

Another topic that was hardly mentioned in the chapter is how the syntax of languages is specified in general. There is a rich theory called formal languages that discusses properties of classes of languages, as well as metalanguages and formalisms for specifying them. This is also the point where computer science meets the study of human languages, leading to the vibrant area of research known as computational linguistics.

Finally, it is worth mentioning that syntax analyzers are not stand-alone 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 compilation needs of the application at hand. Following the “show me” spirit of this book, we have chosen not to use such black boxes in the implementation of our compiler, but rather to build everything from the ground up.

10.5 Project

The compiler construction spans two projects: 10 and 11. This section describes how to build the syntax analyzer described in this chapter. In the next chapter we extend this analyzer into a full-scale Jack compiler.

Objective Build a syntax analyzer that parses Jack programs according to the Jack grammar. The analyzer’s output should be written in XML, as defined in the specification section.

Resources The main tool in this project is the programming language in which you will implement the syntax analyzer. You will also need the supplied TextComparer utility, which allows comparing the output files generated by your analyzer to the compare files supplied by us. You may also want to inspect the generated and supplied output files using an XML viewer (any standard Web browser should do the job).

Contract Write the syntax analyzer program in two stages: tokenizing and parsing. Use it to parse all the .jack files mentioned here. For each source .jack file, your analyzer should generate an .xml output file. The generated files should be identical to the .xml compare-files supplied by us.

Test Programs

The syntax analyzer’s job is to parse programs written in the Jack language. Thus, a reasonable way to test your analyzer it is to have it parse several representative Jack programs. We supply two such test programs, called Square Dance and Array Test. The former includes all the features of the Jack language except for array processing, which appears in the latter. We also provide a simpler version of the Square Dance program, as explained in what follows.

For each one of the three programs, we supply all the Jack source files comprising the program. For each such Xxx.jack file, we supply two compare files named XxxT.xml and Xxx.xml. These files contain, respectively, the output that should be produced by a tokenizer and by a parser applied to Xxx .jack.

- *Square Dance* (projects/10/Square): A trivial interactive “game” that enables moving a black square around the screen using the keyboard’s four arrow keys.
- *Expressionless Square Dance* (projects/10/ExpressionlessSquare): An identical copy of Square Dance, except that each expression in the original program is replaced with a single identifier (some variable name in scope). For example, the Square class has a method that increases the size of the graphical square object by 2 pixels, as long as the new size does not cause the square image to spill over the screen’s boundaries. The code of this method is as follows.

Square Class Code

```

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

```

ExpressionlessSquare Class Code

```

method void incSize() {
  if (x) {
    do erase();
    let size=size;
    do draw();
  }
  return;
}

```

Note that the replacement of expressions with variables has resulted in a nonsensical program that cannot be compiled by the supplied Jack compiler. Still, it follows all the Jack grammar rules. The expressionless class files have the same names as those of the original files, but they are located in a separate directory.

■ *Array test* (projects/10/ArrayTest): A single-class Jack program that computes the average of a user-supplied sequence of integers using array notation and array manipulation.

Experimenting with the Test Programs If you want, you can compile the Square Dance and Array Test programs using the supplied Jack compiler, then use the supplied VM emulator to run the compiled code. These activities are completely irrelevant to this project, but they serve to highlight the fact that the test programs are not just plain text (although this is perhaps the best way to think about them in the context of this project).

Stage 1: Tokenizer

First, implement the JackTokenizer module specified in section 10.3. When applied to a text file containing Jack code, the tokenizer should produce a list of tokens, each printed in a separate line along with its classification: symbol, keyword, identifier, integer constant, or string constant. The classification should be recorded using XML tags. Here is an example:

Source Code

```
if (x < 153)
    {let city="Paris";}
```

Tokenizer Output

```
<tokens>
  <keyword> if </keyword>
  <symbol> ( </symbol>
  <identifier> x </identifier>
  <symbol> &lt; </symbol>
  <integerConstant> 153
</integerConstant>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <keyword> let </keyword>
  <identifier> city </identifier>
  <symbol> = </symbol>
  <stringConstant> Paris
</stringConstant>
  <symbol> ; </symbol>
  <symbol> } </symbol>
</tokens>
```

Note that in the case of string constants, the tokenizer throws away the double quote characters. That's intentional.

The tokenizer's output has two "peculiarities" dictated by XML conventions. First, an XML file must be enclosed in some begin and end tags, and that's why the `<tokens>` and `</tokens>` tags were added to the output. Second, four of the symbols used in the Jack language (`<`, `>`, `"`, `&`) are also used for XML markup, and thus they cannot appear as data in XML files. To solve the problem, we require the tokenizer to output these tokens as `<`, `>`, `"`, and `&`, respectively. For example, in order for the text `"<symbol> < </symbol>"` to be displayed properly in a Web browser, the source XML should be written as `"<symbol> < </symbol>."`

Testing Your Tokenizer

- Test your tokenizer on the Square Dance and Test Array programs. There is no need to test it on the expressionless version of the former.
- For each source file `Xxx.jack`, have your tokenizer give the output file the name `XxxT.xml`. Apply your tokenizer to every class file in the test programs, then use the supplied `TextComparer` utility to compare the generated output to the supplied `.xml` compare files.
- Since the output files generated by your tokenizer will have the same names and extensions as those of the supplied compare files, we suggest putting them in separate directories.

Stage 2: Parser

Next, implement the `CompilationEngine` module specified in section 10.3. Write each method of the engine, as specified in the API, and make sure that it emits the correct XML output. We recommend to start by writing a compilation engine that handles everything except expressions, and test it on the expressionless Square Dance program only. Next, extend the parser to handle expressions as well, and

proceed to test it on the *Square Dance* and *Array Test* programs.

Testing Your Parser

- Apply your `CompilationEngine` to the supplied test programs, then use the supplied `TextComparer` utility to compare the generated output to the supplied `.xml` compare files.
- Since the output files generated by your analyzer will have the same names and extensions as those of the supplied compare files, we suggest putting them in separate directories.
- Note that the indentation of the XML output is only for readability. Web browsers and the supplied `TextComparer` utility ignore white space.

Compiler II: Code Generation

The syntactic component of a grammar must specify, for each sentence, a deep structure that determines its semantic interpretation.

—Noam Chomsky (b. 1928), mathematical linguist

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

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

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

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

So what's in it for you? Typically, students who don't take a formal compilation course don't have an opportunity to develop a full-scale compiler. Thus readers who follow our instructions and build the Jack compiler from scratch will gain an important lesson for a relatively small effort (of course, their

knowledge of compilation theory will remain limited unless they take a course on the subject). Further, some of the tricks and techniques used in the code generation part of the compiler are rather clever. Seeing these tricks in action leads one to marvel, once again, at how human ingenuity can dress up a primitive switching machine to look like something approaching magic.

11.1 Background

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

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

11.1.1 Data Translation

Programs manipulate many types of variables, including simple types like integers and booleans and complex types like arrays and objects. Another dimension of interest is the variables' kind of life cycle and scope—namely, whether it is local, global, an argument, an object field, and so forth.

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

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

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

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

Symbol table (of some hypothetical subroutine)

The symbol table is the “Rosetta stone” that the compiler uses when translating high-level code involving identifiers. For example, consider the statement `balance = balance + sum`. Using the symbol table, the compiler can translate this statement into code reflecting the facts that `balance` is field number 2 of the current object, while `sum` is argument number 0 of the running subroutine. Other details of this translation will depend on the target language.

The basic symbol table abstraction is complicated slightly due to the fact that most languages permit

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

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

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

That's the bad news. The good news is that we have already handled all these difficulties. In our two-tier compiler architecture, memory allocation of variables was delegated to the VM back-end. In particular, the virtual machine that we built in chapters 7-8 includes built-in mechanisms for accommodating the standard kinds of variables needed by most high-level languages: static, local, and argument variables, as well as fields of objects. All the allocation and de-allocation details of these variables were already handled at the VM level, using the global stack and the virtual memory segments.

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

Handling Arrays *Arrays* are almost always stored as sequences of consecutive memory locations (multi-dimensional arrays are flattened into one-dimensional ones). The array name is usually treated as a pointer to the base address of the RAM block allocated to store the array in memory. In some languages like Pascal, the entire memory space necessary to represent the array is allocated when the array is declared. In other languages like Java, the array declaration results in the allocation of a single pointer only, which, eventually, may point to the array's base address. The array proper is created in memory

later, if and when the array is actually constructed at run-time. This type of dynamic memory allocation is done from the heap, using the memory management services of the operating system. Typically, the OS has an `alloc(size)` function that knows how to find an available memory block of size `size` and return its base address to the caller. Thus, when compiling a high-level statement like `bar=new int [10]`, the compiler generates low-level code that effects the operation `bar=alloc(10)`. This results in assigning the base-address of the array's memory block to `bar`, which is exactly what we want. Figure 11.1 offers a snapshot of this practice.

Let us consider how the compiler translates the statement `bar[k]=19`. Since the symbol `bar` points to the array's base-address, this statement can be also expressed using the C-language notation `*(bar+k)=19`, that is, "store 19 in the memory cell whose address is `bar+k`." In order to implement this operation, the target language must be equipped with some sort of an indirect addressing mechanism. Specifically, instead of storing a value in some memory location `y`, we need to be able to store the value in the memory location whose address is the current contents of `y`. Different languages have different means to carry out this pointer arithmetic, and figure 11.2 shows two possibilities.

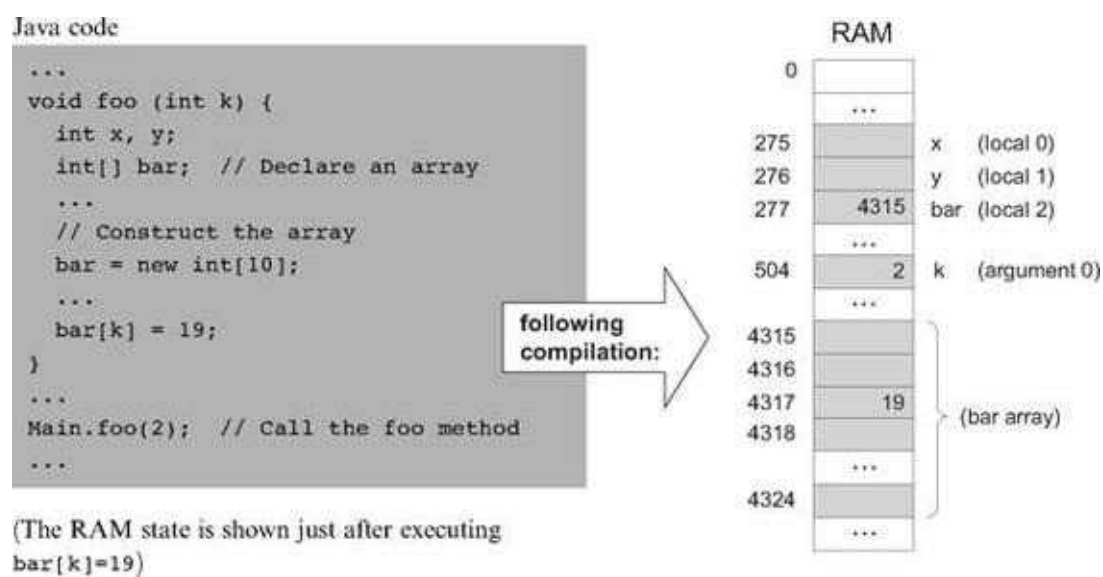


Figure 11.1 Array handling. Since memory allocations are run-time dependent, all the shown addresses are arbitrary examples.

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

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

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

<i>Pseudo VM code</i>	<i>Final VM code</i>
<pre>// bar[k]=19, or *(bar+k)=19 push bar push k add // Use a pointer to access x[k] pop addr // addr points to bar[k] push 19 pop *addr // Set bar[k] to 19</pre>	<pre>// bar[k]=19, or *(bar+k)=19 push local 2 push argument 0 add // Use the that segment to access x[k] pop pointer 1 push constant 19 pop that 0</pre>

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

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

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

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

Suppose now that we wish to apply the `mult` method to the `b` object, using a method call like `b . mult(5)`. How should the compiler handle this method call? Unlike the fields data (e.g., `re` and `im`), of which different copies are kept for each object instance, only one copy of each method (e.g., `mult`) is actually kept at the target code level for all the object instances derived from this class. In order to make it look as if each object encapsulates its own code, the compiler must force this single method to always operate on the desired object. The standard compilation trick that accomplishes this abstraction is to pass a reference to the manipulated object as a hidden argument of the called method, compiling `b . mult(5)` as if it were written as `mult(b, 5)`. In general then, each object-based method call `foo . bar(v1, v2, ...)` is translated into the VM code `push foo, push v1, push v2, ... , call bar`. This way, the compiler can force the same method to operate on any desired object for instance, creating the high-level perception that each object encapsulates its own code.

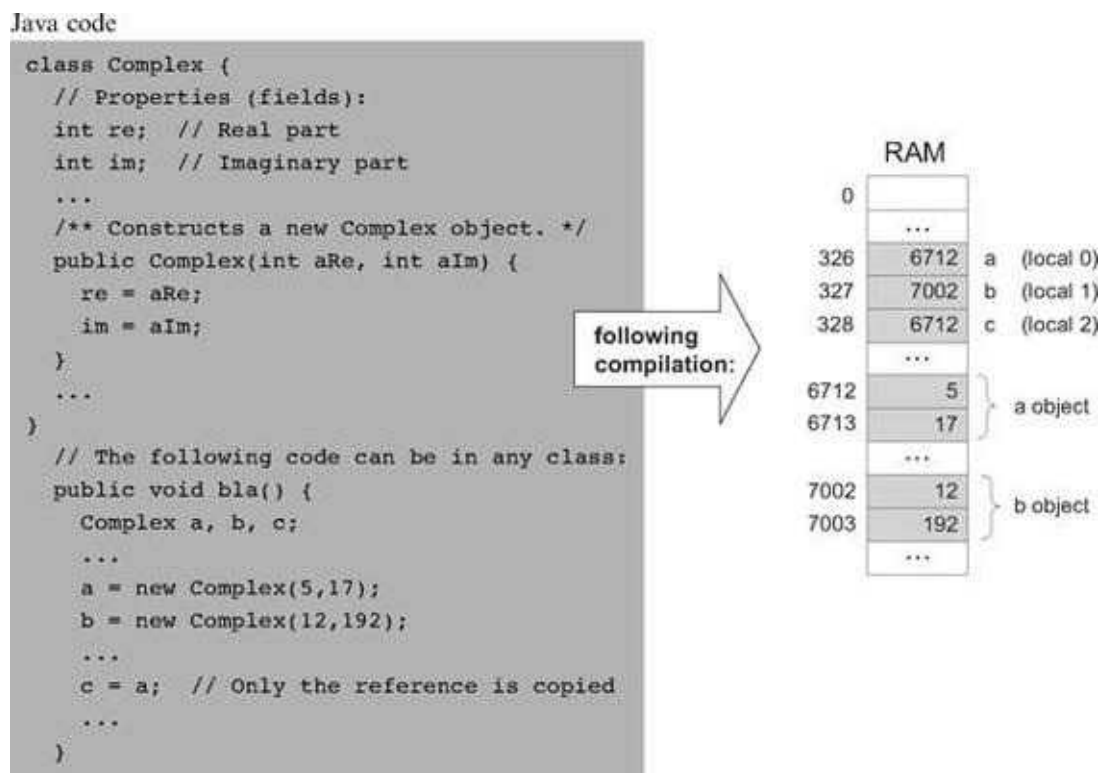


Figure 11.3 Objects handling. Since memory allocations are run-time dependent, all the shown addresses are arbitrary examples.

However, the compiler's job is not done yet. Since the language allows different methods in different classes to have the same name, the compiler must ensure that the right method is applied to the right object. Further, due to the possibility of method overriding in a subclass, compilers of object-oriented languages must do this determination at run-time. When run-time typing is out of the picture, for example, in languages like Jack, this determination can be done at compile-time. Specifically, in each method call like $x.m(y)$, the compiler must ensure that the called method $m()$ belongs to the class from which the x object was derived.

11.1.2 Commands Translation

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

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

The choice of the code generation algorithm depends on the target language into which we are translating. For a stack-based target platform, we simply need to print the tree in postfix notation, also known as Right Polish Notation (RPN). In RPN syntax, an operation like $f(x, y)$ is expressed as x, y, f (or, in the VM language syntax, `push x, push y, call f`). Likewise, an operation like $x + y$, which is $+(x, y)$ in prefix notation, is stated as $x, y, +$ (i.e., `push x, push y, add`). The strategy for translating expressions into stack-based VM code is straightforward and is based on recursive post-order traversal of the underlying parse tree, as follows:

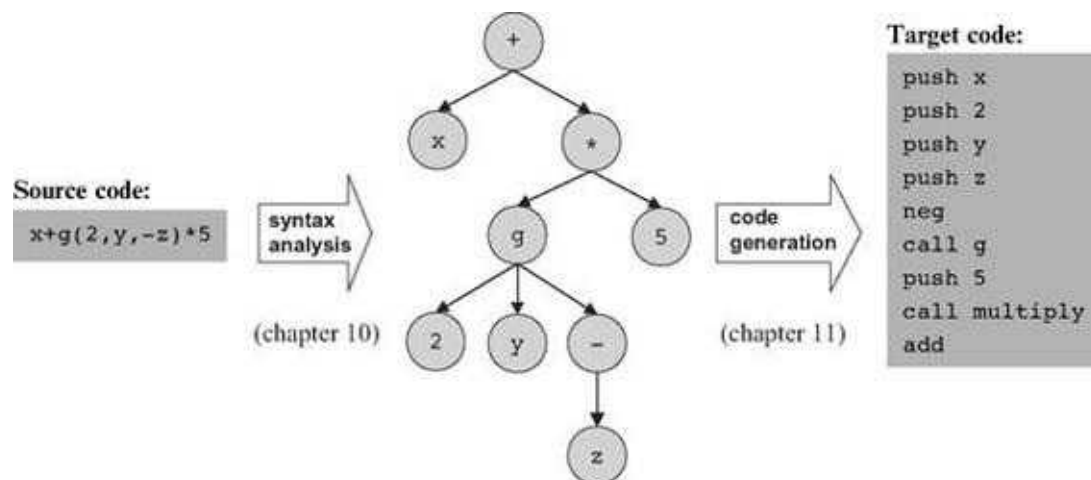


Figure 11.4 Code generation.

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

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

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

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

11.2 Specification

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

```
prompt> JackCompiler source
```

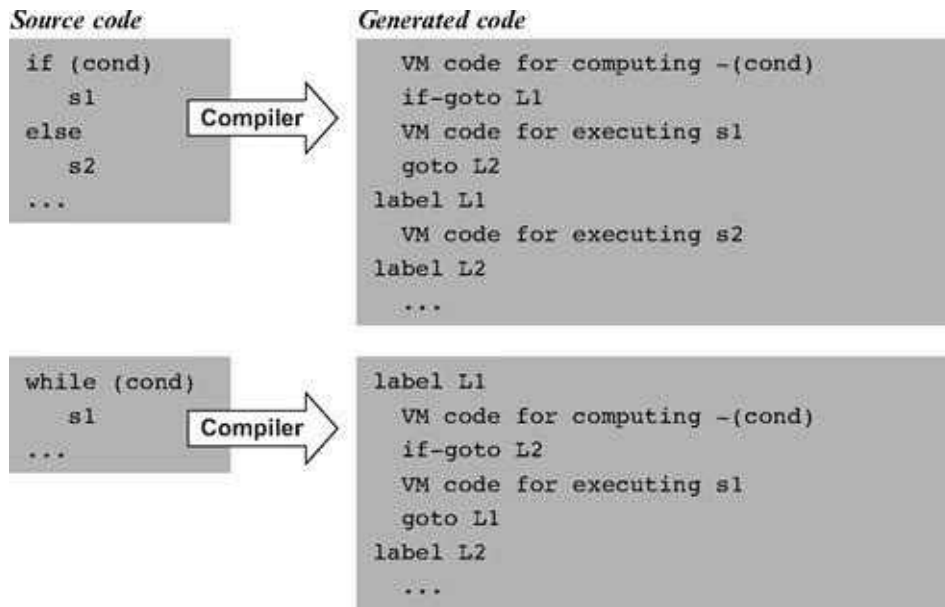


Figure 11.5 Compilation of control structures.

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

11.2.1 Standard Mapping over the Virtual Machine

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

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

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

Memory Allocation and Access

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

Subroutine Calling

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

its base.

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

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

Constants

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

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

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

11.2.2 Compilation Example

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

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

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

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

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

Class-scope symbol table

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

Method-scope (transfer) symbol table

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

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

Pseudo VM code

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

Final VM code

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

11.3 Implementation

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

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

11.3.1 The *JackCompiler* Module

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

11.3.2 The *JackTokenizer* Module

The tokenizer API was given in section 10.3.2.

11.3.3 The *SymbolTable* Module

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

Static: Scope: class.

Field: Scope: class.

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

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

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

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

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

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

11.3.4 The *VMWriter* Module

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

Routine	Arguments	Returns	Function
Constructor	Output file/stream	—	Creates a new file and prepares it for writing.
writePush	Segment (CONST, ARG, LOCAL, STATIC, THIS, THAT, POINTER, TEMP) Index (int)	—	Writes a VM push command.
writePop	Segment (CONST, ARG, LOCAL, STATIC, THIS, THAT, POINTER, TEMP) Index (int)	—	Writes a VM pop command.
WriteArithmetic	command (ADD, SUB, NEG, EQ, GT, LT, AND, OR, NOT)	—	Writes a VM arithmetic command.
WriteLabel	label (String)	—	Writes a VM label command.
WriteGoto	label (String)	—	Writes a VM goto command.
WriteIf	label (String)	—	Writes a VM If-goto command.
writeCall	name (String) nArgs (int)	—	Writes a VM call command.
writeFunction	name (String) nLocals (int)	—	Writes a VM function command.
writeReturn	—	—	Writes a VM return command.
close	—	—	Closes the output file.

11.3.5 The *CompilationEngine* Module

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

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

11.4 Perspective

The fact that Jack is a relatively simple language permitted us to sidestep several thorny compilation issues. For example, while Jack looks like a typed language, this is hardly the case. All of Jack's data types are 16-bits long, and the language semantics allows Jack compilers to ignore almost all type information. As a result, when compiling and evaluating expressions, Jack compilers need not determine their types (with the single exception that compiling a method call `x.m()` requires determining the class type of `x`). Likewise, array entries in Jack are not typed. In contrast, most programming languages feature rich type systems that have significant implications on their compilers: Different amounts of memory must be allocated for different types of variables; conversion from one type into another requires specific language operations; the compilation of a simple expression like `x+y` depends strongly on the types of `x` and `y`; and so on.

Another significant simplification is that the Jack language does not support inheritance. This implies that all method calls can be handled statically, at compile-time. In contrast, compilers of languages with inheritance must treat methods as virtual, and determine their locations according to the run-time type of the underlying object. For example, consider the method call `x.m()`. If the language supports inheritance, `x` can be derived from more than one class, and we cannot know which until run-time. Thus, if the definition of the method `m` is not found in the class from which `x` was derived, it may still be found in a class that supersedes it, and so on.

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

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

Many other simplifications of the Jack language are not significant and can be relaxed with little effort. For example, one may easily extend the language with `for` and `switch` statements. Likewise, one can add the capability to assign constants like `'c'` to `char` type variables, which is presently not supported by the language. (To assign the constant `'c'` to a Jack `char` variable `x`, one must first assign `"c"` to a `String` variable, say `s`, and then use `let x=s.charAt(0)`. Clearly, it would be nicer to simply say `let x='c'`, as in Java).

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

11.5 Project

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

Resources The main tool that you need is the programming language in which you will implement the compiler. You will also need an executable copy of the Jack operating system, as explained below. Finally, you will need the supplied VM Emulator, to test the code generated by your compiler on a set of test programs supplied by us.

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

Stage 1: Symbol Table

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

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

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

Stage 2: Code Generation

We don't provide specific guidelines on how to develop the code generation features of the compiler, though the examples spread throughout the chapter are quite instructive. Instead, we provide a set of six application programs designed to unit-test these features incrementally. We strongly suggest to test your compiler on these programs in the given order. This way, you will be implicitly guided to build the compiler's code generation capabilities in stages, according to the demands of each test program.

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

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

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

Test Programs

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

Seven This program computes the value of $(3*2)+1$ and prints the result at the top left of the screen. To

test whether your compiler has translated the program correctly, run the translated code in the VM emulator and make sure that it displays 7 correctly. Purpose: Tests how your compiler handles a simple program containing an arithmetic expression with integer constants (without variables), a do statement, and a return statement.

Decimal-to-Binary Conversion This program converts a 16-bit decimal number into its binary representation. The program takes a decimal number from RAM [8000], converts it to binary, and stores the individual bits in RAM [8001..8016] (each location will contain 0 or 1). Before the conversion starts, the program initializes RAM [8001..8016] to -1. To test whether your compiler has translated the program correctly, load the translated code into the VM emulator and go through the following routine:

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

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

Square Dance This program is a trivial interactive “game” that enables moving a black square around the screen using the keyboard’s four arrow keys. While moving, the size of the square can be increased and decreased by pressing the “z” and “x” keys, respectively. To quit the game, press the “q” key. To test if your compiler has translated the program correctly, run the translated code in the VM emulator and make sure that it works according to this description. Purpose: Tests how your compiler handles the object-oriented constructs of the Jack language: constructors, methods, fields and expressions that include method calls. It does not test the handling of static variables.

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

Pong A ball is moving randomly on the screen, bouncing off the screen “walls.” The user can move a small bat horizontally by pressing the keyboard’s left and right arrow keys. Each time the bat hits the ball, the user scores a point and the bat shrinks a little, to make the game harder. If the user misses and the ball hits the bottom horizontal line, the game is over. To test whether your compiler has translated this program correctly, run the translated code in the VM emulator and play the game (make sure to score some points, to test the part of the program that displays the score on the screen). Purpose: Provides a complete test of how your compiler handles objects, including the handling of static variables.

Complex Arrays Performs five complex calculations using arrays. For each such calculation, the program prints on the screen the expected result versus the actual result (as performed by the compiled program). To test whether your compiler has translated the program correctly, run the translated code in the VM emulator and make sure that the actual results are identical to the expected results. Purpose: Tests how your compiler handles complex array references and expressions.