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 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.

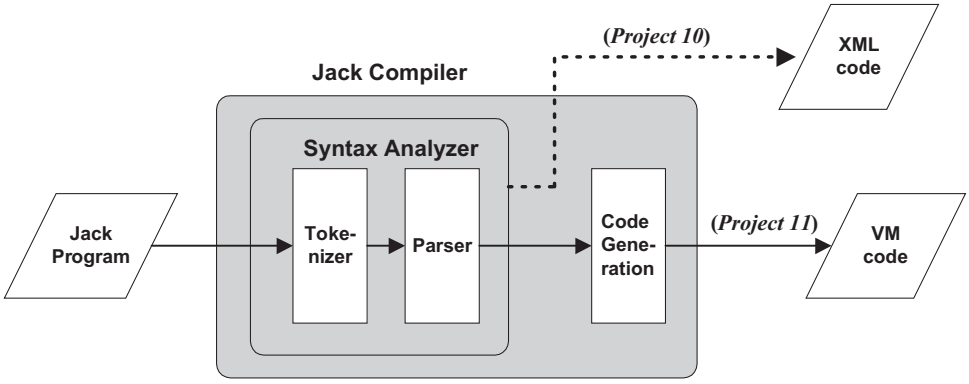


Figure 10.1 The Jack Compiler. The project in chapter 10 is an intermediate step, designed to localize the development and unit-testing of the *syntax analyzer* module.

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).

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.

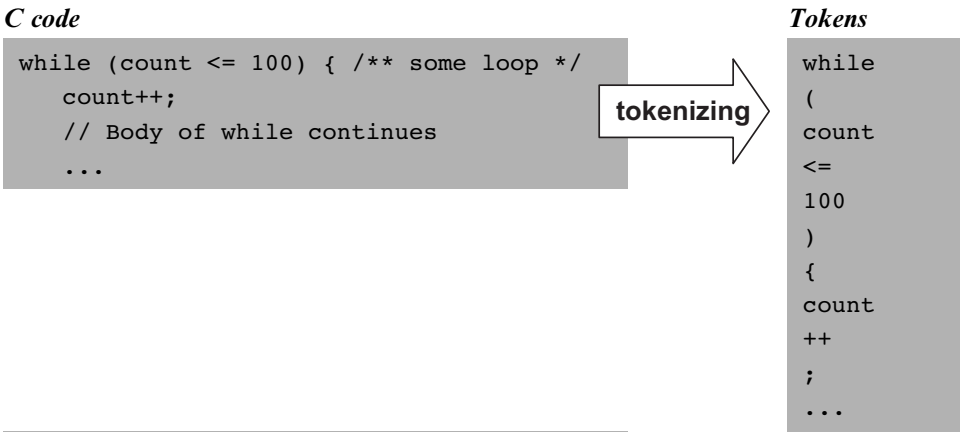


Figure 10.2 Lexical analysis.

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

```
...
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).

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.

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.

C code

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

C language grammar (partial)

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

Tokenized (parser's input):

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

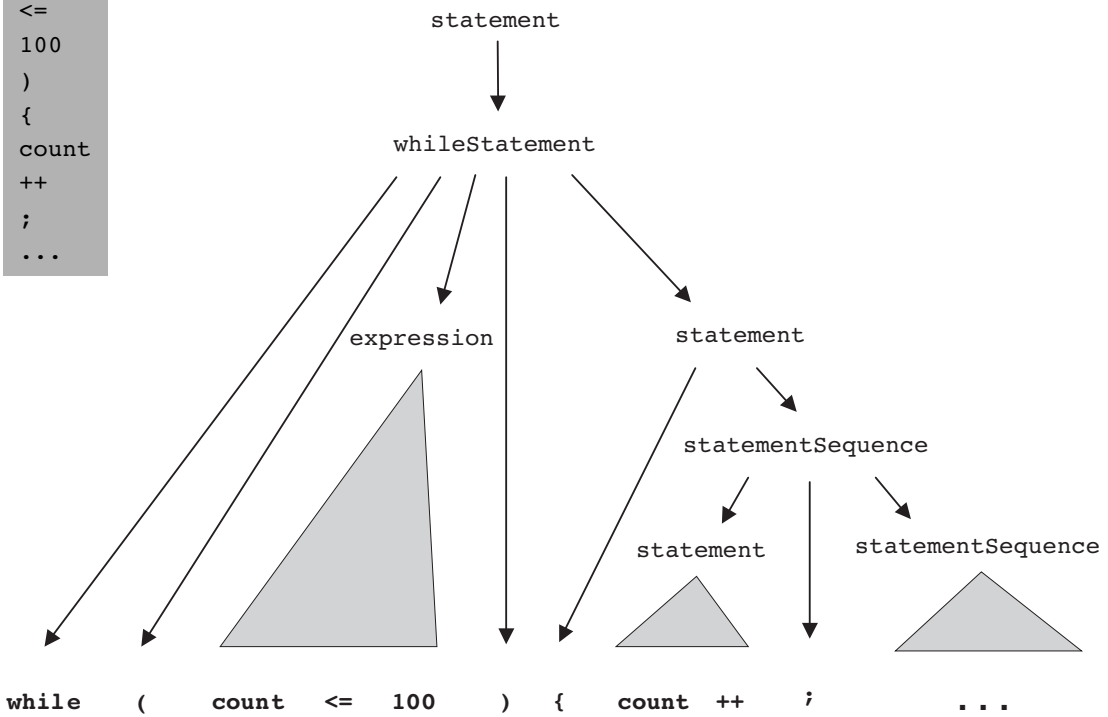


Figure 10.4 Parse tree of a program segment according to a grammar segment. Solid triangles represent lower-level parse trees.

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.

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()`, `parseIfStatement()`, `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

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'**

Figure 10.5 (continued)

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.

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, whileSatement, 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:

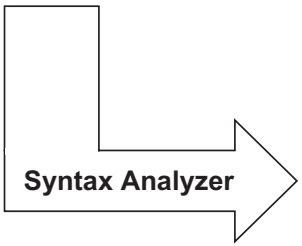
⟨*xxx*⟩ terminal ⟨/*xxx*⟩

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>
          ...
```

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 *hasMoreTokens( )* 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 `tokenType()` is `KEYWORD`.
`symbol`	—	Char	Returns the character which is the current token. Should be called only when `tokenType()` is `SYMBOL`.
`identifier`	—	String	Returns the identifier which is the current token. Should be called only when `tokenType()` is `IDENTIFIER`.
`intVal`		Int	Returns the integer value of the current token. Should be called only when `tokenType()` 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 `tokenType()` 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 `compileClass()`.
`CompileClass`	—	—	Compiles a complete class.
`CompileClassVarDec`	—	—	Compiles a static declaration or a field declaration.
`CompileSubroutine`	—	—	Compiles a complete method, function, or constructor.
`compileParameterList`	—	—	Compiles a (possibly empty) parameter list, not including the enclosing "`()`".

Routine	Arguments	Returns	Function
`compileVarDec`	—	—	Compiles a `var` declaration.
`compileStatements`	—	—	Compiles a sequence of statements, not including the enclosing "{}".
`compileDo`	—	—	Compiles a `do` statement.
`compileLet`	—	—	Compiles a `let` statement.
`compileWhile`	—	—	Compiles a `while` statement.
`compileReturn`	—	—	Compiles a `return` statement.
`compileIf`	—	—	Compiles an `if` statement, possibly with a trailing `else` clause.
`CompileExpression`	—	—	Compiles an expression.
`CompileTerm`	—	—	Compiles a *term.* 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.
`CompileExpressionList`	—	—	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 *ArrayTest* 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 `&lt;`, `&gt;`, `&quot;`, and `&amp;`, 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> &lt; </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.