10. The Compiler I: Syntax Analysis[1]

"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 BC - 43 BC)

This chapter is work in progress. In this chapter we start the process of building a compiler for the Jack high-level language. The process of compilation is usually partitioned into two conceptual parts: syntactic understanding of the program structure, and semantic generation the compiled code. This chapter deals with the first issue, that of *parsing* a program written in the Jack language as to "understand its structure". The second part, code generation, is the subject of chapter 11.

The concept of "understanding the structure" of a program needs some explanation. When humans reads a program, they immediately see the "structure" of the program: where classes and methods begin and end, what are declarations, what are statements, what are expressions and how they are built, and so on. Notice that this is a complex nested structure: classes contain methods that contain statements that contain expressions, etc. The allowable structure of programs may be formalized, and programming languages today have formal syntax rules, usually given as a "context free language".

Parsing a program that was written according to these rules means determining the exact correspondence between the program and the syntax rules. This correspondence is usually hierarchal, and may be specified by a "derivation tree" for the program. Compilers often keep an explicit data structure that corresponds to this tree and use this data structure for code generation. Alternatively, they may generate this information implicitly and use it on the fly for code generation. Since in this chapter we do not generate any code yet, we have chosen to explicitly output the parsed structure in a particular format, specifically in XML. This will demonstrate the correct parsing of the program, in a way that is easily displayed in any web browser. In the next chapter we will simply replace the parts of the current program that output the parsing in XML with parts that do actual code generation.

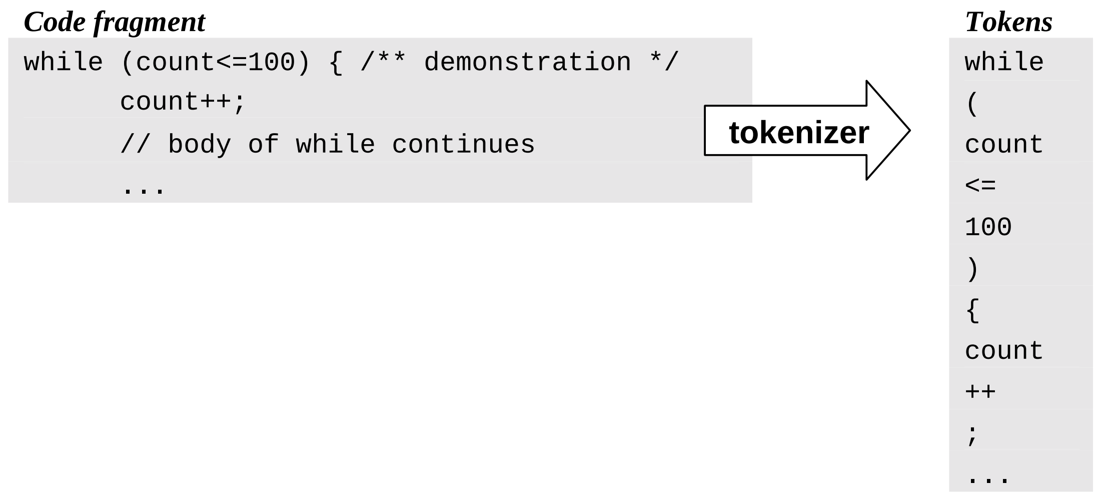
It is worthwhile to note that the same types of syntax rules used for specifying programming languages are also used for specifying the syntax of many other types of files. While most programmers will never need to write a real compiler, it is very likely that they will often need to parse files of some other type with a complex syntax. This parsing will be done in the same way that the parsing of a programming language is done.

[1] From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2003, www.idc.ac.il/csd

1. Background

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 the program is to group the characters into *words*, also called *tokens*, 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. Thus the tokens stream becomes the main input of the compiler. Program 1 illustrates the tokenizing of a typical code fragment, taken from a Java or C program.



Code fragment

```
while (count<=100) { /** demonstration */
      count++;
      // body of while continues
      ...
```

Tokens

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

PROGRAM 1: Lexical Analysis, also called *tokenizing*, converts the input text into a list of tokens. These tokens are then taken to be the elementary atoms from which the program is made.

As Program 1 illustrates, there are several distinct types of tokens: `while` is a keyword; `count` is an identifier; <= is an operator, `100` is a constant, and so on. Also notice that white space (blanks and newline characters) is eliminated in the tokenizing process, and so are comments.

In general, each programming language specifies the types of tokens it allows, as well as the exact syntax rules for combining them into programmatic structures. For example, some languages may specify that "++" is an operator, while other languages may not. In the latter case, an expression containing two consecutive + operators will be considered invalid.

Context Free Languages

Once we have lexically analyzed a program into a stream of tokens, we are now faced with the main challenge of parsing it into its formal structure. We first need to consider how the formal syntax of languages is specified. There is a rich theory called "formal languages" that discusses many types of languages, including the formalisms used to specify them. Almost all programming languages, as well as most other formal languages used for describing the syntax of complex files types, use a formalism known as "context free grammars".

A *context free grammar* is a specification of allowable syntactic elements, and rules for composing them from other syntactic elements. For example, the grammar of the Java language allow us to combine the atoms `100`,`count`, and `<=` into the pattern `count<=100`. In a similar fashion, we can observe that according to the Java grammar, the input pattern `count<=100` is valid, i.e. it is consistent with the language's rules. Indeed, every language has a dual perspective. From a constructive standpoint, the grammar specifies allowable ways to combine *words*, also called *terminals*, into higher-level syntactic elements, called *non-terminals*. From an analytic standpoint, the grammar is also a prescription for doing the reverse: decomposing a given input pattern into non-terminals, lower-level non-terminals, and eventually into terminals that cannot be decomposed further. These terminals correspond to the tokens of the lexical analysis phase.

The syntactic structure of the language -- the context free grammar -- is a set of rules that specify how non-terminals can be derived from other non-terminals and terminals. The grammar may be recursive. There may be more than one possible rule for deriving any particular non-terminal, and the different alternatives are usually indicated using the "|" notation. Grammar 2 gives an example of a small part of the context-free grammar of the C-language.

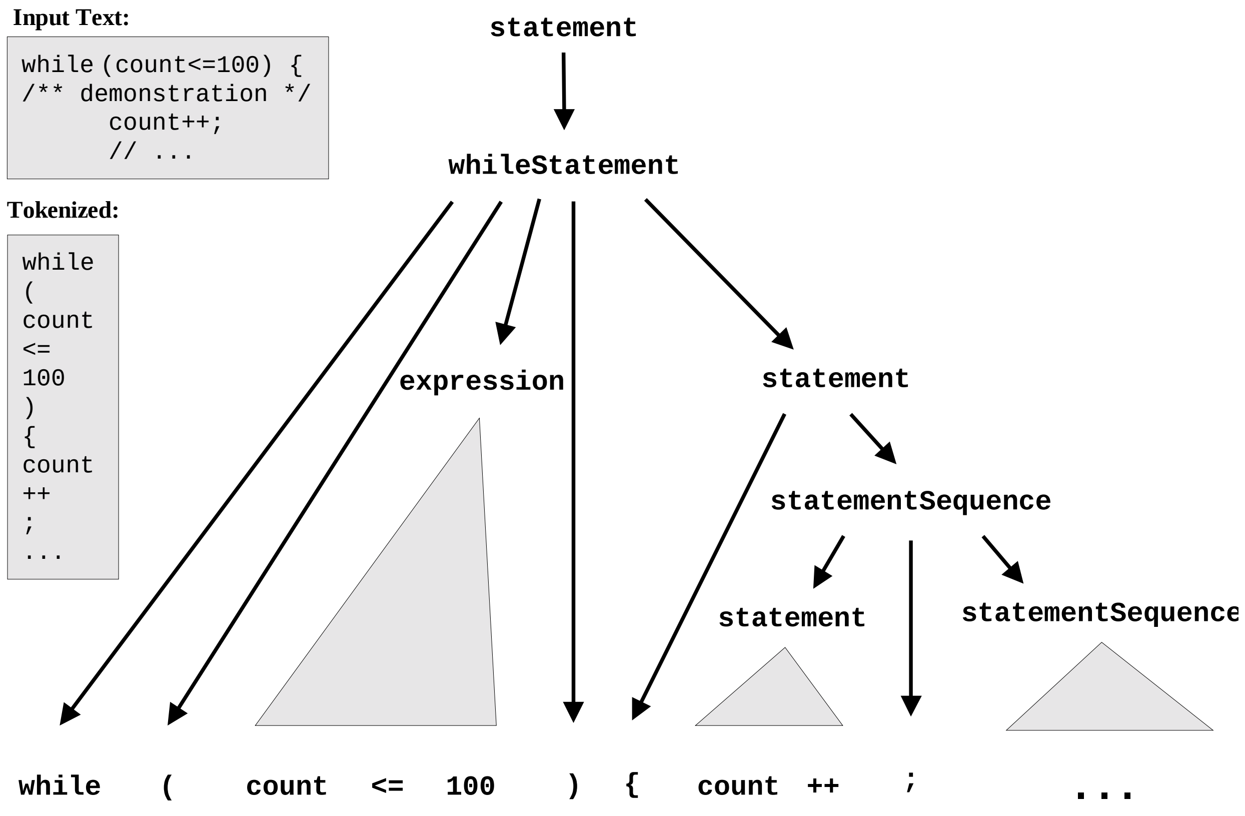
```
...
statement:   whileStatement
           | ifStatement
           | ...          // other statement possibilities follow
           | '{' statementSequence '}'

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

ifStatement: ...          // if definition comes here

statementSequence:  ''    // null, i.e. the empty sequence
                  | statement ';' statementSequence

expression: ...   // definition of an expression comes here

...               // more definitions follow
```

GRAMMAR 2: A Context Free Grammar is a set of rules that describes the syntactic structure of a language. Here we see part of the C language grammar.

What does Grammar 2 mean? First, the grammar implies that *statement*, *whileStatement*, *ifStatement*, *expression*, and *statementSequence* are non-terminals, whereas "`while`", '`{`', '`}`', and '`;`' are terminals. Further, the grammar implies that a *statement* in the C language may be one of several forms, including a *while* statement, an *if* statement, other possibilities not shown here for lack of space, and any sequence of statements enclosed in curly brackets. The resulting grammar is highly recursive, allowing nested structures like the following example:

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

Parsing: The act of checking whether a grammar "accepts" an input text as valid (according to the grammar rules) is called *parsing*. As a side effect of the parsing process, the entire syntactic structure of the input text is uncovered. Since the grammar rules are hierarchical, the result is a tree-oriented data structure, called *parse tree* or *derivation tree* (weather the tree is stored in memory or recognized on-line is a different issue that will be addressed later). For example, if we apply Grammar 2 to the tokenized version of Program 1, we will obtain the parse tree depicted in Figure 3.


Input Text:

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

Tokenized:

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

FIGURE 3: Parse tree of Program 1 according to Grammar 2. Solid triangles represent lower-level parse trees. The input of the parsing process is the tokenized version of the program.

Recursive Descent Parsing

The last section ended with a description of a *parse tree*. We now turn to describe the algorithms that can be used to construct such trees from given input programs, according to the syntax rules of a given language. There are general algorithms that can do that for any context free language. The general algorithms are not efficient enough for practical use on very long programs, and there are more efficient parsing algorithms that apply to certain restricted classes of context free languages, classes that contain the syntax rules of essentially all program languages. These more efficient algorithms usually run "online" – they parse the input as they read it, and do not have to keep the entire input program in memory. There are essentially two types of strategies for this parsing. The simple strategy works top-down, and this is the one we present here. The more advanced algorithms work bottom-up, and are not described here since they require a non-trivial elaboration of theory.

The top-down approach to parsing, also called *recursive descent parsing*, parses the input stream recursively, using the nested structure prescribed by the language grammar. Let us consider how a *parser* program that implements this strategy can be constructed. For every non-terminal building block of the language, we can equip the parser with a recursive procedure designed to parse that non-terminal. If the non-terminal consists of terminal atoms only, the procedure will simply read them. Otherwise, for every non-terminal building block, the procedure will recursively call the procedure designed to parse the non-terminal. The process will continue recursively, until all the terminal atoms have been reached and read.

For example, suppose we have to write a recursive descent parser that implements Grammar 2. Since the grammar has five derivation rules, the parser implementation can consist of five major procedures: `parseStatement()`, `parseWhileStatement()`, `parseIfStatement()`, `parseStatementSequence()`, and `parseExpression()`. The parsing logic of these procedures should follow the syntactic patterns found in the corresponding grammar rules. Thus `parseStatement()` should probably start its processing by determining what is the first token. Having established the token's identity, the procedure could determine which statement we are in, and then call the parsing procedure associated with this statement type.

For example, if the input stream were Program 1, the procedure will establish that the first token is `while`, and then call the procedure `parseWhileStatement()`. According to the corresponding grammar rule, this procedure should next attempt to read the terminals "`while`" and "`(`", and then call `parseExpression()` to parse the non-terminal *expression*. After `parseExpression()` would return (having read and parsed the "`count<=100`" sequence in our example), the grammar dictates that `parseWhileStatement()`should continue parsing the remainder of the while statement. In particular, the grammar states that it should attempt to read the terminal "`)`" and then recursively call `parseStatement()` to parse the non-terminal statement. This call would continue recursively, until at some point only terminal atoms are read.

LL(0) grammars: Recursive parsing algorithms are simple and elegant. If you will think about them, you will realize that the only thing that complicates matters is the existence of 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, a curly-bracket enclosed statement list, and so on. The span of possibilities is determined by the underlying grammar, and in some cases it is easy to tell which alternative we are in. For example, consider Grammar 2. If the first token is "`while`", it is clear that we are faced with a while statement, since this is the only alternative 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 specifies without ambiguity which rule to use. Grammars that have this property are called *LL(0)* grammars, and they 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 algorithm. The only exception is the parsing of expressions, where just a little look ahead is necessary.

2. Specification

In this chapter we will write a syntax analyzer for the Jack programming language. In the next chapter we will add the functionality of code generation to the syntax analyzer, and obtain a full compiler. The main purpose of the syntax analyzer is to read a Jack program and "understand" its structure according to the Jack language syntax specification. The meaning of "understanding" is that the program "knows" at each point the meaning of what it is reading: an expression, a statement, a variable name, etc. It has to have this knowledge in a complete recursive sense. This is what will be needed for later enabling the code generation.

One way to demonstrate that the analyzer has "understood" the programmatic structure of the input is to have it print the text in a way that provides a visual image of the program structure. Therefore, while syntax analyzers are normally not stand-alone programs, we define our syntax analyzer as having a specific output: an XML description of the program. Thus the syntax analyzer you build here will output an XML file whose structure reflects the structure of the underlying program. In the next chapter you will replace the parts of the program that output the XML code with software that generates executable VM code instead.

Usage: The Jack syntax analyzer accepts a single command line argument that specifies either a file name or a directory name:

```
prompt> JackCompiler source
```

If *source* is a file name of the form `Xxx.jack`, the analyzer compiles it into a file named `Xxx.xml`, created in the same folder in which the input `Xxx.jack` is located. If *source* is a directory name, all the `.jack` files located in this directory are compiled. For each `Xxx.jack` file in the directory, a corresponding `Xxx.xml` file is created in the same directory.

Jack Language Syntax

The functional specification of the Jack language was given in chapter 9. We now turn to give a formal specification of the Jack language *syntax*, using a context free grammar. The grammar is based on the following conventions:

'**xxx**'	quoted boldface is used for characters 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	means that either x or y can appear
x?	means that x appears 0 or 1 times
x*	means that x appears 0 or more times

Input: The input to the Jack syntax analyzer is simply a stream of characters. This stream should be tokenized into a stream of tokens according to the rules specifying the lexical elements in the table. These tokens may be separated by an arbitrary amount of white space (space and newline characters) and comments, which are ignored. Comments are of the standard formats `/* comment until closing */` , `/** API comment */`, and `// comment to end of line`.

The complete language grammar is given in Grammar 4.

Lexical elements	There are five types of lexical elements in the Jack language:
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	'**"**' sequence of ASCII characters not including double quote or newline '**"**'
identifier:	sequence of letters, digits, and underscore ('_') not starting with a digit
Program structure:	A program is a collection of classes, each appearing in a separate file. The compilation unit is a class, and is given by 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
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**'

GRAMMAR 4: Complete grammar of the Jack language

XML Output Format

The output of the syntax analyzer should be an XML description of the program. Figure 5 gives a detailed example. Basically, the analyzer has to recognize two major types of language constructs: terminal elements, and non-terminal elements. These constructs are handled as follows.

Non-terminals: Whenever a non-terminal element of type xxx of the language is encountered, the analyzer should generate the 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 element of type xxx of the grammar is encountered, the following output should be generated:

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

Where xxx is one of the five terminals specified in the "lexical elements" part of the Jack grammar: keyword, symbol, integerConstant, stringConstant, identifier.

```
Class Bar {
  method Fraction foo(int y) {
      var int temp; // a variable
      let temp = (xxx+12)*-6 ;
      ...
```

```
<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>
            <term>
              <symbol> ( </symbol>
              <expression>
                <term>
                  <identifier> xxx </identifier>
                </term>
                <symbol> + </symbol>
                <term>
                  <integerConstant> 12 </integerConstant>
                </term>
              </expression>
              <symbol> ) </symbol>
            </term>
            <symbol> * </symbol>
            <term>
              <symbol> - </symbol>
              <term>
                <integerConstant> 6 </integerConstant>
              </term>
            </term>
          </expression>
         <symbol> ; </symbol>
       </letStatement>
         ...
```

FIGURE 5: Analyzer input (top) and output (bottom)

3. Implementation

We suggest to arrange the implementation of the syntax analyzer in three modules:

- `JackAnalyzer:` a main driver that organizes and invokes everything;
- `JackTokenizer:` a tokenizer;
- `CompilationEngine:` a recursive top-down syntax analyzer.

These modules handle the syntax of the language. In the next chapter we will extend this implementation 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 compiler for the Jack language.

JackAnalyzer

The analyzer program operates on a given *source*. If *source* is a file name of the form `Xxx.jack`, the analyzer compiles it into a file named `Xxx.xml`, created in the same folder in which the input `Xxx.jack` is located. If *source* is a directory name, all the `.jack` files located in this directory are compiled. For each `Xxx.jack` file in the source directory, the analyzer creates a corresponding `Xxx.xml` file in the same directory. The logic is as follows:

For each source `Xxx.jack` file:

1. Create a *tokenizer* from the `Xxx.jack` file
2. Open an `Xxx.xml` file and prepare it for writing
3. Compile(`INPUT`: *tokenizer*, `OUTPUT`: *output file*)

Where *output file* refers to the Xxx.xml file.

JackTokenizer

The tokenizer removes all comments and white space from the input stream and breaks it into Jack-language tokens, as specified in 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`
`stringVal`		string	returns the string value of the current token, without the double quotes. Should be called only when *tokenType()* is `STRING_CONST`.

CompilationEngine

This module effects the actual compilation into XML form. It gets its input from a `JackTokenizer` and writes its parsed XML structure into an output file/stream. This is done by a series of `compilexxx()` methods, where `xxx` is a corresponding syntactic element of the Jack grammar. The contract between these methods is that each `compilexxx()` method should read the syntactic construct `xxx` from the input, `advance()` the tokenizer exactly beyond `xxx`, and output the XML parsing of `xxx`. Thus, `compilexxx()`may only be called if indeed `xxx` is the next syntactic element of the input.

In the next chapter, this module will be modified to output the compiled code rather than XML.

Routine	**Arguments**	**Returns**	**Function**
Constructor	Input stream/file Output stream/file	--	creates a new compilation engine with the given input and output. The next method 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 “`()`”.
`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 method is faced with a slight difficulty when trying to decide between some of the alternative rules. Specifically, if the current token is an identifier, it must still distinguish between a variable, an array entry, and a subroutine call. The distinction can be made by looking ahead one extra token. A single look-ahead token, which may be one of "`[`", "`(`", "`.`", 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.

4. Perspective

In this chapter we have side-stepped almost all of the formal language theory studied in a typical compilation course. We were able to do this by choosing a very simple syntax for Jack that could be easily compiled using recursive descent techniques. In particular, our grammar for expressions did not mandate the usual operator precedence (e.g. of multiplication over addition). This avoided the need for bottom-up parsing of "LR" languages, usually used in other programming languages.

In reality, programmers rarely write syntax analyzers by hand. Instead, they use, so called, "compiler-compilers" (such as "yacc") utilities. These programs receive as input a context free grammar, and produce as output a syntax analysis code for this grammar. Following the "show me" spirit of this book, we have chosen not to use such black boxes in the implementation of our compiler.

5. Build it

In this project you will build a syntactic analyzer for the Jack language. In the next chapter, we will extend this analyzer into a full-scale Jack compiler.

Objective: Develop a syntactic analyzer that parses Jack programs according to the Jack grammar. The output of the analyzer should be written in XML format, following the example given in Figure 5.

Resources: The main tool that you need is the programming language in which you will implement the 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. If you want to inspect the XML code generated by the analyzer, you will also need an XML viewer (any standard Web browser should do the job).

Contract: Write the Jack analyzer program in two stages, as described below. Use it to parse all the `.jack` files mentioned below. For each source `.jack` file, your analyzer should generate an `.xml` output file. The generated files should be identical to the supplied `.xml` compare-files.

Test Programs

We supply three test programs, as follows.

Square Dance A simple interactive game that will be used to test your compiler in both projects 10 and 11. Although the details of the game are irrelevant to the compilation process, we describe it briefly. *Square Dance* is a trivial "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. The game implementation is organized in three classes:

- `Class Main`: Initializes a new game and starts it;
- `Class Square`: Implements an animated square. A square object has a screen location and size properties, and methods for drawing, erasing, moving, and size changing;
- `Class SquareGame`: Runs the game according to the game rules.

We provide three sets of test files for this program:

- Input source code: `Main.jack`, `Square.jack`, `SquareGame.jack`
- Tokenizer output (compare files): `MainT.xml`, `SquareT.xml`, `SquareGameT.xml`
- Analyzer output (compare files): `Main.xml`, `Square.xml`, `SquareGame.xml`

Expressionless Square Dance: An identical copy of the *Square Dance* game, except that each expression in the latter is replaced with a single identifier (a variable name in scope). This version of the program is especially useful in the project's second stage, in which it is advised to first implement a *Parser* that handles everything except expressions. The replacement of the expressions with variables required the introduction of some illegal variable castings into the source code, and so this version of the game cannot be compiled using the Jack Compiler. Still, it follows all the Jack grammar rules. The provided test files have the same names as those of the *Square Dance* program.

Array test: A single-class Jack program that computes the average of a user-supplied sequence of integers. This program uses some array notation not used in the *Square Dance* program, and therefore it is recommended to test it only after successful testing of the latter. The provided test files include the source code (`Main.jack`), the compare file for the *Tokenizer* output (`MainT.xml`) and the compare file for the *Parser* output (`Main.xml`).

Implementation Tips

- Since the output files that your tokenizer and analyzer will generate will have the same names and extensions as those of the supplied compare files, we suggest putting them in separate directories.

- Since each one of the test programs focuses on different aspects of the Jack language, it is recommended to perform the tests in the following order: *Expressionless Square Dance*, then *Square Dance*, then *Array test*.
- All the source test files are written in Jack. If you want, you can compile the *Square* and *Array* programs using the supplied Jack compiler, and then run them on the supplied VM emulator. These activities are completely irrelevant to the analyzer implementation, 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 a Jack tokenizer. In order to test this stage, have your tokenizer output an XML file describing the list of the parsed tokens. 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. For example, consider the text:

```
let x=5+yy; let city=”Paris”;
```

This input should generate the following output:

```
<keyword> let </keyword>
<identifier> x </identifier>
<symbol> = </symbol>
<integerConstant> 5 </integerConstant>
<symbol> + </symbol>
<identifier> yy </identifier>
<symbol> ; </symbol>
<keyword> let </keyword>
<identifier> city </identifier>
<symbol> = </symbol>
<stringConstant> Paris </stringConstant>
<symbol> ; </symbol>
```

Note that the tokenizer throws away the double quote characters. That's OK.

A slight difficulty, and a solution: 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, have your tokenizer output these tokens as `&lt;`, `&gt;`, `&quot;`, and `&amp;`, respectively. For example, in order for the text "`<symbol> & </symbol>`" to be displayed in an XML viewer, the source XML should be written as "`<symbol> &amp; </symbol>`".

Testing: For each source file `Xxx.jack`, have your tokenizer give the output file the name `XxxT.xml`. For each one of the three test programs, apply your tokenizer to every class file in the test program. This should generate an `.xml` output file for each one of the source `.jack` files. Next, use the supplied `TextComparer` utility to compare the generated output to the supplied `.xml` compare files.

Stage 2: Parser

Next, implement the *Compilation Engine* (also referred to as *Parser*). Write each method of the engine, as specified in the API, and make sure that it emits the correct XML output.

Implementation tips:

- The indentation of XML code is only for readability. XML viewers and the supplied *TextComparer* utility ignore white space.
- Note that conceptually speaking, the output of the tokenizer is embedded within the XML output. In other words, the parser builds the language "super structure" on top of the terminal tokens.
- You may want to start by writing a parser that can handle everything except expressions. For example, assume that the only expressions that the input source code can contain may be single identifiers, and handle everything else. Next, extend the parser to handle expressions as well.

Testing: For each source file `Xxx.jack`, have your parser give the output file the name `Xxx.xml`. For each one of the three test programs, apply your parser to every class file in the test program. This should generate an `.xml` output file for each one of the source `.jack` files. Next, use the supplied `TextComparer` utility to compare the generated output to the supplied `.xml` compare files.