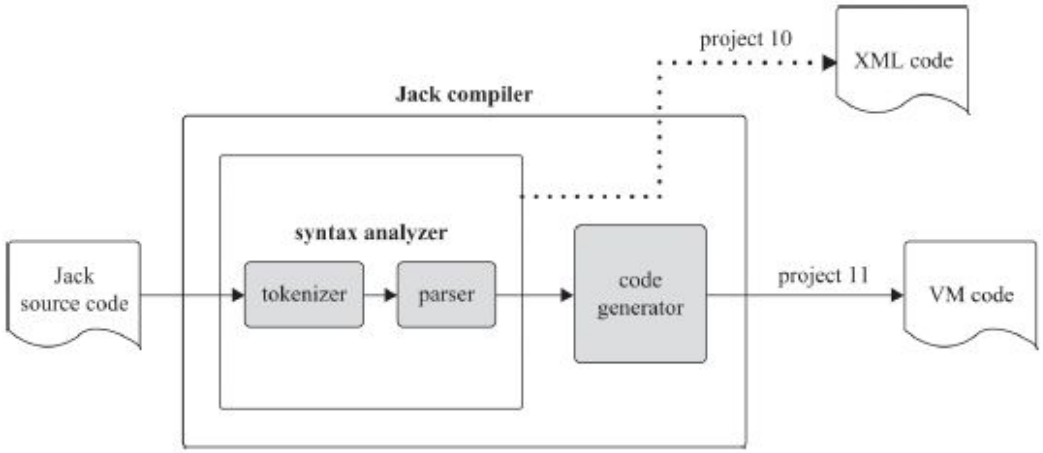

Staged development plan of the Jack compiler.

Jack grammar (subset)

statements: *statement**

statement: *letStatement* | *ifStatement* | *whileStatement*

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

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

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

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

term: *varName* | *constant*

varName: a string not beginning with a digit

constant: a non-negative integer

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

Input examples

```
let x = 100;
```
✓

```
let x = x + 1;
```
✓

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

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

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

Prog.jack (character stream)

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

Tokenizing

Output (token stream)

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

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

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

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

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

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

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

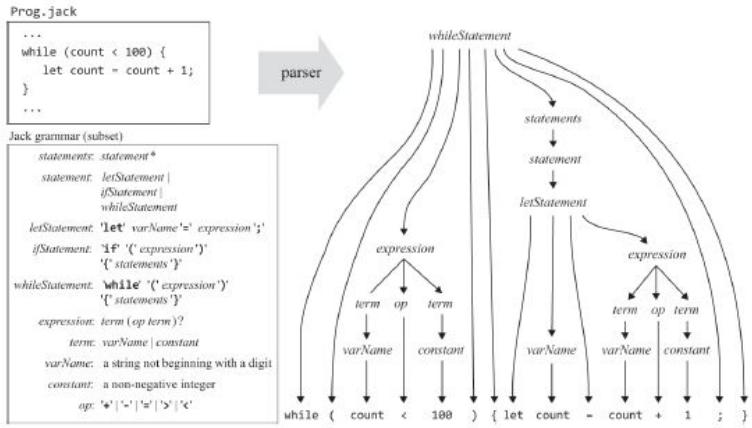

Figure 10.4a Parse tree of a typical code segment. The parsing process is driven by the grammar rules.

10.3 Implementation

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

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

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

The JackTokenizer

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

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

Tokenizer

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

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

The Jack grammar.

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

Jack analyzer: handling inputs that correspond to non-terminal rules

If we encounter a *nonTerminal* element of type *class declaration*, *class variable declaration*, *subroutine declaration*, *parameter list*, *subroutine body*, *variable declaration*, *statements*, `let` *statement*, `if` *statement*, `while` *statement*, `do` *statement*, `return` *statement*, *an expression*, *a term*, or an *expression list*,

the parser generates the output:

```
<nonTerminal>
    Recursive output for the non-terminal body
</nonTerminal>
```

where *nonTerminal* is:
`class`, `classVarDec`, `subroutineDec`, `parameterList`, `subroutineBody`, `varDec`; `statements`, `LetStatement`, `ifStatement`, `whileStatement`, `doStatement`, `returnStatement`; `expression`, `term`, `expressionList`

Example: if the input is `return x;`

```
<returnStatement>
   <keyword>
      return
   </keyword>
   <expression>
      <term>
         <identifier> x </identifier>
      </term>
   </expression>
   <symbol> ; </symbol>
</returnStatement>
```