Project 10: Compiler I - Syntax Analysis

Background

The Jack compiler, like those of Java and C#, is two-tiered: the compiler's front-end translates from the high-level language to an intermediate VM language; the compiler's back-end translates further from the VM language to the native code of the host platform. In projects 7-8 we've built the compiler's back-end (the VM translator); we now turn to building the compiler's front-end. This development will span two projects: syntax analysis (this project), and code generation (next project). Welcome to syntax analysis.

Objective

In this project we build a syntax analyzer that parses Jack programs according to the Jack grammar, producing an XML file that renders the program's structure using marked-up text. In the next project, the logic that generates the XML output will be morphed into logic that generates VM code.

Contract

Write a syntax analyzer for the Jack language. Use it to parse all the .jack class files supplied below. For each input .jack file, your analyzer should generate an .xml output file. The generated files should be identical to the supplied compare-files, up to white space.

Resources

The relevant reading for this project is book chapter 10. You will need two tools: the programming language with which you will implement your syntax analyzer, and the supplied TextComparer utility, available in your nand2tetris/tools directory. This program will help you compare the output files generated by your syntax analyzer to the compare files supplied by us. You may also want to inspect the generated and supplied output files visually, using some XML viewer. To do so, simply load these files into some web browser or text editor. Some of these tools, e.g. Chrome, are designed to display XML text nicely - give it a try. All the files necessary for this project are available in nand2tetris/projects/10 on your computer.

Proposed Implementation

Chapter 10 includes a proposed, language-independent syntax analyzer API, which can serve as your implementation's blueprint. We propose implementing the syntax analyzer in two stages. First, write and test the Tokenizer module. Next, write and test the CompilationEngine module, which implements the parser described in the chapter.

Stage I: Tokenizer

Tokenizing, a basic service of any syntax alayzer, is the act of breaking a given textual input into a stream of tokens. And while it is at it, the tokenizer can also classify the tokens into lexical categories. With that in mind, your first task it to implement, and test, the JackTokenizer module specified in chapter 10. Specifically, you have to develop (i) a Tokenizer implementation, and (ii) a test program that goes through a given input file (.jack file) and produces a stream of tokens using your Tokenizer implementation. Each token should be printed in a separate line, along with its classification: symbol, keyword, identifier, integer constant or string constant. Here is an example:

Source code (input)

Source code (input)

```
if (x < 0) {
   let state = "negative";
}
```

Tokenizer output

Tokenizer output

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

Note that in the case of string constants, the tokenizer throws away the double-quote characters. This behavior is intended, and is part of our tokenizer specification.

Also note that four of the symbols used in the Jack language (<, >, ", and &) are also used for XML markup, and thus they cannot appear verbatim as XML data. To solve the problem, and following convention, we require the tokenizer to output these tokens as &lt;, &gt;, &quot;, and &amp;, respectively. For example, in order for the symbol "less than" to be displayed properly in a web browser, it should be generated as "<symbol>&lt; </symbol>".

Finally, note that unlike the Tokeinizer module, the tokenizer test program that you are to write is not part of the syntax analyzer. This test program entails an intermediate testing stage, focusing on unit-testing the Tokenizer only. Once this test is completed successfully, the test program is no longer necessary.

Stage II: Parser (CompilationEngine)

In the context of this project, parsing is defined narrowly as the act of going over the tokenized input and rendering its grammatical structure using some agreed-upon format. The specific parser that we implement here is based on the Jack grammar, and is designed to emit XML output. Both the grammar and the agreed-upon XML tags are described in chapter 10.

The Jack parser is implemented by the CompilationEngine module, whose API is given in chapter 10. Your task is to implement this API: write each one of the specified methods, and make sure that it emits the correct XML output. For the benefit of unit-testing, we recommend to begin by first writing a compilation engine that handles any given Jack code except for expressions; next, extend the compilation engine to handle expressions as well. The test programs supplied below are designed to support this staged testing strategy.

Test Programs

A natural way to test your syntax analyzer it is to have it parse some representative Jack programs. We supply two such test programs: Square Dance and Array Test. The former includes all the syntactic features of the Jack language except for array processing, which appears in the latter. We also provide an expression-less version of the Square Dance program, as explained below.

Square Dance: a simple interactive application, described in project 9. The implementation is organized in three classes:

Source class file	Description (irrelevant to this project, which deals with syntax)	Tokenizer output	Parser output
Sourced class file Main.jack	Description (quite irrelevant to this project) Initializes and starts a new "square dance" session".	Tokenizer output MainT.xml	Parser output Main.xml
Sourced class file Square.jack	Description (quite irrelevant to this project) Implements an animated square. A Square object has a screen location and size properties, and methods for drawing, erasing,moving, and size changing.	Tokenizer output SquareT.xml	Parser output Square.xml
Sourced class file SquareGame.jack	Description (quite irrelevant to this project) Runs the show according to the game rules.	Tokenizer output SquareGameT.xml	Parser output SquareGame.xml

Note: The three source Jack files comprising the Square Dance application are identical to those stored in the projects/09/Square directory. For completeness, an identical copy of these files is also available in the projects/10/Square directory.

Expressionless Square Dance: in this version of Square Dance, each expression in the original source code has been replaced with a single identifier (some variable name in scope). This version of the program was designed in order to facilitate unit-testing of your syntax analyzer's ability to parse everything except for expressions. Note that the replacement of expressions with variables has resulted in nonsensical code which, however, is grammatically correct. For convenience, the expressionless files have the same names as those of the original files, but they are stored in a separate projects/10/ExpressionlessSquare directory.

Array Test: a single-class Jack program designed to test how the syntax analyzer handles array processing:

Source class file	Description (irrelevant to this project, which deals with syntax)	Tokenizer output	Parser output
Source class file Main.jack	Description (quite irrelevant to this project) Computes the average of a user-supplied sequence of integers using an array data	Tokenizer output MainT.xml	Parser output Main.xml

structure and array manipulation
commands.

Experimenting with the test programs: if you want, you can compile the supplied SquareDance and TestArray programs using the supplied Jack compiler, then use the supplied VM emulator to run the compiled code. This activity is irrelevant to the current project. However, it serves to show that the test programs are not just plain text; they also have semantics, or meaning, something that the syntax analyzer does not care about.

Testing

Tokenizer Testing:

- Test your tokenizer on the Square Dance and the TestArray programs.
- For each Xxx.jack source file, have your tokenizer test program give the output file the name XxxT.xml. Apply your tokenizer test to each 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 test will have the same names and extensions as those of the supplied compare files, we suggest putting them in separate directories.

Parser Testing:

- Apply your syntax analyzer 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 syntax 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 ignore white space.