Lab: Lexical Analysis - Tokenizer

Objective

In this project we build a Lexical analyzer that parses Jack programs according to the Jack grammar, producing an XML file that renders the program's structure using marked-up text. Write a Lexical 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 9 and chapter 10. You will need two tools: the programming language with which you will implement your Lexical analyzer, and ANY Text-Comparer utility. 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.

Proposed Implementation - 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.

Below is an example. 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.

Source code (input)	Tokenizer output

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

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 <, >, ", and &, respectively. For example, in order for the symbol "less than" to be displayed properly in a web browser, it should be generated as "<symbol><</symbol>".

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.

What to submit

- Your code. You can write your program in **ANY language** if your choice.
- Single Word/PDF Report including
  - screenshots of output,
  - screenshots of Text Compare,
  - short description of your approach,
  - challenges you faced and key takeaways