Project 10

In projects 10 and 11 you'll develop a compiler for Jack, a simple, Java-like, object-based language. You will write this compiler in two main stages. In this project you'll develop a compilation engine that performs *syntax analysis* (also called *parsing*). In project 11 you'll extend the compilation engine to generate executable VM code.

Objective

Build a syntax analyzer that parses Jack programs according to the Jack grammar. To enable correctness testing, the analyzer will output XML code. This version of the syntax analyzer assumes that the source Jack code is error-free. Error checking, reporting and handling can be added to later versions of the analyzer, but are not part of this project.

Contract

Write a syntax analyzer for the Jack language, and test it on the supplied test files. The XML files produced by your analyzer should be identical to the supplied compare files, up to white space.

Resources

The main tool in this project is the programming language that you will use for implementing the syntax analyzer. You will also need the supplied TextComparer tool, or a similar tool. This program allows comparing files while ignoring white space. This will enable comparing the output files generated by your analyzer with the supplied compare files. You may also want to inspect these files using an XML viewer. Any standard web browser will do – just use the browser's file>open menu to open the XML file that you wish to inspect.

Test files

We provide several .jack files, for testing purposes. Projects/10/Square is a 3-class program that enables moving a black square around the screen using the keyboard's arrow keys. Projects/10/ArrayTest is a single-class program that computes the average of a user-supplied sequence of integers, using array processing. Both programs were discussed in project 9, so they should be familiar. Note though that we have made some harmless changes to the original code, to make sure that the syntax analyzer will be fully tested on all aspects of the Jack language. For example, we've added a static variable to Square/Main.jack, as well as a function named test, which are neither used nor called. These neutral changes allow testing how the analyzer handles language elements that don't appear in the original Square and ArrayTest files, like static variables, else, and unary operators.

Analyzer

The analyzer, which is the main program in this project, is invoked using the command "JackAnalyzer source", where source is either a file name of the form Xxx.jack (the extension is mandatory), or a folder name (in which case there is no extension). In the latter case, the folder

contains one or more .jack files, and, possibly, other files as well. The file/folder name may include a file path. If no path is specified, the analyzer operates on the current folder.

The analyzer handles each file separately. For each Xxx.jack input file, the analyzer constructs a JackTokenizer for handling the input, and an output file for writing the output (XML code).

Tokenizer

Implement the JackTokenizer module described in the chapter / lecture. Test your implementation by writing a basic JackAnalyzer. In this first version of the JackAnalyzer, the output file corresponding to each input Xxx.jack file is named XxxT.xml (where "T" stands for tokenized output). After constructing a JackTokenizer, the analyzer enters a loop that advances and handles all the tokens in the input file, one token at a time, using the JackTokenizer methods. Each token should be printed in a separate line, as <*tokenType*> token </*tokenType*>, where *tokenType* is an XML tag coding one of the five possible token types in the Jack language. Here is an example:

input (e.g. `Prog.jack`)

```
...
// Comments and white space
// are ignored.
if (x < 0) {
   let quit = "yes";
}
...
```

output (e.g. `ProgT.xml`)

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

Note that in the case of string constants, the program ignores the double quote characters. This requirement is by design.

The generated output has two trivial technicalities dictated by XML conventions. First, an XML file must be enclosed within some begin and end tags; this convention is satisfied by the <tokens> and </tokens> tags. 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. Following convention, the analyzer represents these symbols as <, >, ", and &, respectively. For example, when the parser encounters the symbol < in the input file, it outputs the line "<symbol> < </symbol>". This so-called "escape sequence" is rendered by XML viewers as <symbol> < </symbol>, which is what we want.

Testing

Start by applying your JackAnalyzer to one of the supplied .jack files, and verify that it operates correctly on a single input file.

Next, apply your JackAnalyzer to the Square folder, containing the files Main.jack, Square.jack, and SquareGame.jack, and to the TestArray folder, containing the file Main.jack.

Use the supplied TextComparer tool to compare the output files generated by your JackAnalyzer to the supplied .xml compare files. For example, compare the generated file SquareT.xml to the supplied compare file SquareT.xml.

Since the generated and compare files have the same names, we suggest putting them in separate folders.

Parser

The next version of your syntax analyzer should be capable of parsing every element of the Jack language, except for expressions and array-oriented commands. To that end, implement the CompilationEngine module specified in the book/lecture, except for the compilation routines that handle expressions and arrays.

For each Xxx.jack file, the analyzer constructs a JackTokenizer for handling the input, and an output file for writing the output, named Xxx.xml. The analyzer then calls the compileClass routine of the CompilationEngine. From this point onward, the CompilationEngine routines should call each other recursively, emitting XML output.

Unit-test this version of your JackAnalyzer by applying it to the folder ExpressionlessSquare. This folder contains versions of the files Square.jack, SquareGame.jack, and Main.jack, in which each expression in the original code has been replaced with a single identifier (some variable name in scope). For example:

Square folder:

```
// Square.jack
...
method void incSize() {
   if (((y + size) < 254) & ((x + size) < 510)
      do erase();
      let size = size + 2;
      do draw();
   }
   return;
}
...
```

ExpressionlessSquare folder:

```
   // Square.jack
   ...
   method void incSize() {
       if (x) {
            do erase();
            let size = size;
            do draw();
      }
      return;
   }
   ...
```

Note that the replacement of expressions with variables results in nonsensical code. This is just fine: The nonsensical code is syntactically correct, and that's all that matters for testing the parser. Note also that the original and expression-less files have the same names, but are located in separate folders.

Use the supplied TextComparer utility to compare the output files generated by your JackAnalyzer with the supplied .xml compare files.

Next, complete the CompilationEngine routines that handle expressions, and test them by applying your JackAnalyzer to the Square folder. Finally, complete the routines that handle arrays, and test them by applying your JackAnalyzer to the ArrayTest folder.