Contents:

I: Original Goal
II: Developed Goal
III: Technical Description
IV: Results
V: Error Handling
VI: Conclusion and Future Plans

I: Original Goal

Original (condensed) description from proposal:

"…I wanted to create a practical implementation of a data structure. After learning about the abstract concepts behind data structures, I felt it was necessary to do a project where a data structure could be used as a tool to do something else.
…
I decided to write a simple XML parser. XML is a type of markup language that is used to formally describe and store documents by using tags to denote elements and tag attributes to describe elements.
…
The parser will be written in Java and will read in an XML document. Then, in memory, it will generate a tree where the XML's root element corresponds to the root of the tree and the XML children elements will be children in the tree (with their relationships being consistent as they are in the document)…After, as both a test and a show of functionality, the parser will reconstruct the XML file using only the stored tree and give output as an XML text file."

II: Developed Goal

During my research of XML and XML parsers, I developed a clearer and more succinct goal about what I wanted to accomplish with my project. Upon closer inspection, I realized that XML has a greater variety of uses, formatting styles, and formatting preferences than I originally thought. I thought that building a complete parser would be sort of like reinventing the wheel; it would also be too generic and too similar to parsers that already exist, along with being too complex. With this reasoning, I researched some real world applications and uses of XML.

For my project, I decided to build a parser that would deal with a subset of XML that is used for formatting configuration files for various systems. This is a useful subset of XML that follows all standard XML rules with a few necessary constraints, as listed here:

1. the XML prolog is not needed
2. elements are described only by attributes — descriptive text is not needed (meaning <foo>bar</foo> would not be valid in this implementation; instead <foo id="bar"/> or <foo id="bar"></foo> is required)
3. comments are not allowed or needed

Otherwise, all syntax is exactly the same. Likewise, the project description is consistent with the original proposal — the main data structure focus would exist with the DOM-type tree that the program would construct in memory, along with a few minor data structures used to build the actual tree, and after constructing the tree, the program would then recreate the XML file to show that the original information was stored correctly in the tree.

III: Technical Description

To approach this problem, I first determined all of the different pieces I would need for my implementation. I broke this up into several components:

1. a finite state machine to parse my document
2. a parser class with a parseXml function to build the tree
3. the tree itself, consisting of elements corresponding to the tree elements
4. a map to map each attribute to its value, if any exist
5. an ArrayList to store the children of each element
6. a way to convert the tree back to XML

Now I will describe each component in more detail.

1. State machine: I determined that a state machine is the best way to parse XML — switching states is intuitive since tags and names and attributes have easily identifiable beginning and ending points. I broke down an XML document into the following states, which I then used to write my parseXml function. (see next page)

Description of state machine states based on single character iterative input from XML file:

```
CONTENT
   whitespace =
   <          -> TAG_START
   else       error

TAG_START
   /          -> TAG_CLOSE
   a-zA-Z     -> TAG_NAME
   else       error

TAG_CLOSE
   a-zA-Z     -> TAG_CLOSE_NAME
   else       error

TAG_CLOSE_NAME
   a-zA-Z0-9  =
   >          -> CONTENT
   else       error

TAG_NAME
   a-zA-Z0-9  =
   whitespace -> TAG_BODY
   /          -> TAG_BODY_END
   >          -> CONTENT
   else       error

TAG_BODY
   whitespace =
   a-zA-Z     -> ATTR_NAME
   /          -> TAG_BODY_END
   >          -> CONTENT
   else       error

ATTR_NAME
   a-zA-Z0-9  =
   =          -> ATTR_VALUE_START
   else       error

ATTR_VALUE_START
   "          -> ATTR_VALUE
   else       error

ATTR_VALUE
   "          -> ATTR_VALUE_END
   else       =

ATTR_VALUE_END
   whitespace -> TAG_BODY
   /          -> TAG_BODY_END
   >          -> CONTENT
   else       error

TAG_BODY_END
   >          -> CONTENT
   else       error
```

In this chart, the current character is given in the column on the left and the parser's next state is on the right. The arrow -> signifies a state change and = signifies that the parser will remain in the same state for the next iteration.

2. Parse function: As mentioned before, I wrote my parseXml function to reflect the state machine described above, in addition to recursively building the tree. The addition of children and attributes is determined by different combinations of current and next states, as seen in the code.
3. Tree: Originally, I was planning on using a basic tree implementation that would use nodes and pointers. However, I determined that it would be easier to store attributes and elements in different structures, since they have a different meaning in the XML document. I wrote an element class that would serve as a general class to store elements.
4. Map: With this, I decided to use a map to match each attribute with its value (instead of making more nodes and pointers). That way, attributes and their values could easily be accessed by their element.
5. Children: I used an ArrayList to store the children of each element because it was easier to keep track of the nesting style of XML. I chose an ArrayList because it resizes dynamically, allowing for any number of children.
6. Conversion of tree to XML: In my element class, I wrote a toXml function where the tree is recursively traversed by looking at each elements attributes and children.

IV: Results

I was able to implement everything described above successfully. My program is run by TestParser.java. The program can either be hardcoded with a file to take in (easier for an IDE, such as Eclipse) or the file name or text can be read from standard input (more on this in the README). I have included test files with my source code whose purposes are described in detail in the README, but the parser will work on any XML document that follows the modified syntax rules listed in section II. The parser parses the file and constructs a tree in memory that corresponds to the exact structure of the XML document. Then, the program reconstructs the XML document from the tree and prints the result to the standard output (or console). Also, the parser can properly handle whitespace and line breaks, and can properly format the output in a properly spaced and indented way.

V: Error handling

The parser also serves as a validator for the particular modified syntax rules described in section II. The parse function has been equipped with necessary error handling — if a file is read in that does not follow the modified syntax rules, the program throws an error and prints an informative message about what went wrong. I have included a way to exhaustively test all possible errors that the parser can encounter, which is described in detail in the README.

VI: Conclusion and Future Plans

My final program is complete and has all functionality described in this report. While working on this project, I had to implement many different pieces and then use data structures to connect them into a final product. I believe that my knowledge of tree structures was useful and enabled me to connect several different concepts effectively. I believe that my parser can be used realistically with a system that uses this particular subset of XML.

There are several things that I could do to build upon my implementation. For example, I could choose to add in the capability of having text within tags, making the parser more general. I could also analyze runtimes on documents of different sizes or perhaps write an analogous parser in a different language and compare their efficiencies. I could also write or find a program that would convert classic XML to the attribute-style XML that I work with in this project.