# Project 3: XML Parsing

## Motivation:

Tools are the foundation of making computing useful for solving engineering problems. While we could all layout transistors by hand or program processors with assembly code, this is clearly inefficient to build larger hardware and software systems. Compilers are automation tools that translate a high-level language to specify an algorithm into lower level code, that can directly run on a processor. However, for this language to work, we need to be able to parse a grammar and report on whether the code provided follows the correct syntax (is translatable to instructions) or not. This is one of the fundamental aspects of compilers, which we use throughout 301, 302, and beyond.

XML, or the Extensible Markup Language is a portable way to represent a data structure as a text file that a parser, and perhaps even a human, would be able to read. There is a specific format that an XML file must have to be a valid XML file—recognizing a "valid" XML file is the goal of this project. Given a valid XML file, there is a second step where the computer needs to recognize that the XML file is a valid instance of the data structure expected by an application based on an *XML Schema*. In this project we will be satisfying the first condition (syntax) not the second (Schema).

### Description

We need to define a valid XML file. A valid XML file is a text file (Unicode) that follows several simple rules based on the following definitions.

*Markup and Content:* All text in an XML document is either *markup* or *content*. Markup is any text enclosed by the angle brackets '<' and '>' (the markup does not include the brackets). Content is anything that is not markup. The angle brackets that define markup may not be nested and must be in the proper order. That is, you cannot have a another set of angle brackets within the same markup:

<a>   </a> Valid

<a <b> > Invalid

(Note: in real XML one can also define markup as being enclosed by the characters '&' and ';'—we are not considering this definition of markup for this project.)

*Tag:* A *tag* is an individual markup section (i.e., enclosed by the characters '<' and '>'). For all intents and purposes, markup and tags are the same, we just use tag to refer to a specific section. Tags can be of three types (or a declaration) which are defined as follows:

1. *Start-tag*: a *start-tag* is a markup with the of the form <element>.

2. *End-tag*: an *end-tag* is markup with of the form </element>.
3. *Empty-tag*: an *empty-tag* is <element/>.
4. *Declaration*: not really a tag, but a special XML item that begins with the two characters "<?" and ends with the two characters "?>". This is usually at the beginning of an XML document—an example would be <?xml version="1.0" encoding="UTF-8"?>. Your code will need to recognize declarations.

*Tag Name*: The *tag name* is valid if it satisfies the following rules:

1. For start-tags and empty-tags, the tag name is the text immediately after the '<' up to the first white space or the ending angle bracket '>' or "/>". For example, given the start-tag <Edgar src="edgar.jpg"> the tag name is "Edgar".
2. For end-tags, the tag name is the text between the delimiters. For example, "Edgar" for the end-tag </Edgar>.
3. The name cannot contain any of the following characters (I have not put quotes around these characters): !"#$%&'()*+,/;<=>?@[\]^`{|}~. A tag name cannot contain a space character (no white space), and cannot begin with "-", ".", or a numeric digit. All other Unicode is valid.
4. For start-tags and empty-tags there may be additional text following the tag name. This additional text is known as *attributes*. As stated in (1), the tag name is the text following the initial '<' character up to the first white space, or the ending delimiter if there are no attributes. For example, the tag name is "Edgar" for the following start-tags and empty-tags: <Edgar>, <Edgar src="edgar.jpg">, <Edgar/>, <Edgar src="edgar.jpg" version="1.0"/>. The attributes for the last string are src="edgar.jpg" and version="1.0". Note that there is a specific format for the attributes; however, we are not going to verify this for this project. The only thing that matters for this project is to extract the tag name, and make sure that it is valid per the above rules.
5. Tag names are case-sensitive, and the start-tag and end-tag pairs must match exactly.
6. Start-tags and end-tags define elements that must be nested (see the definition of an element and discussion on nested tag names).

*Nested Tag Names*: As rule (6) for the tag names states, the tag names must be correctly nested. To be nested, a start-tag and the corresponding end-tag must satisfy a balanced parenthesis grammar (BPG) (that is, instead of a particular set of matching parentheses, replace the left parenthesis with the start-tag and replace the right parenthesis with the corresponding end-tag.) Empty-tags and declarations are not involved in this matching grammar.

*Element:* An *element* is a section of an XML document contained between matching start-tag and end-tag, or an empty tag. Any text between the element's start-tag and end-tag is its content, which may contain both content and other elements. Elements contained within a tag in this way are called *child elements*. The *element name* is the tag name for the matching start-tag and end-tag pair that define the element, or the name of the empty-tag. An element

is *valid* if it satisfies the above tag name rules and is properly nested with itself and other child elements.

*XML Document*: An XML document must begin and end with a tag (white space is permitted before the initial tag or after the final tag). The entire document must be contained in one element (a root element), although declarations are permitted before the start-tag for this root element. A document that satisfies these tag syntax rules and, for which all elements are valid, is said to be a valid XML document.

**Note**: the restriction of the XML document prevents content from existing **outside** of an element (outside of an enclosing pair of start- and end-tags). However, within an element, it is valid to have both content and elements in any combination. For example:

```
<?xml version="1.0" ?>
<html>
<head>
        Content by itself within an element.
</head>
<body>
        This is content within an element.
        <div>
                This is content as well.
                <div>
                        Content.
                </div>
        Hey look it's content.
        </div>
</body>
Content here also.
</html>
```

## Specification

For this project we will be implementing two classes: an XML Parser, and a Stack ADT. You will be completing the implementation of both these classes and developing tests to ensure that your implementations of these two classes is correct. The specifications of both classes are given in the header files provided with the Starter Code. You are also given a simple Bag class and Node class implementation (from the textbook) and a starter test file. Specific requirements for this project are given below:

1. The Stack ADT header and implementation files are set up for the linked-based implementation of the Stack presented in the textbook. You must complete the link-based implementation for the Stack. mainStack.cpp is just a program to test your implementation of the Stack ADT.

2. The Stack linked-based implementation uses the Node data type given in the textbook. The Node definition and implementation is identical to that in the text and is included with the Starter Code. (The Node data type is also used in the Bag implementation given to you.)
3. The XML specification we use is described above and the start-tag and end-tag names must satisfy the BPG. The Wikipedia page on XML also contains a good discussion but follow the specific XML specification given above.
4. The XMLParser class takes a string as input (for example, if you wanted to parse a file, you would read the file in as a long string, and then pass this to the instance of the XMLParser). See the mainXMLParser driver program for an example. To determine if the string is valid XML, there are two steps:
    a. **First**, we run the string through a "scanner" or "lexer" that breaks this long string into "tokens". The method that performs this process is tokenizeInputString(). This method returns true if successful, false otherwise.
    b. **Second**, we try to "parse" a successfully tokenized input string (stored internally in the XMLParse class). If this tokenized string satisfies the XML grammar, the parseTokenizedInput() method returns true.
5. Use the defined constants in the XMLParser.hpp file, as enums with type StringTokenType, for the five types of XML tokens. The five types are: START_TAG, END_TAG, EMPTY_TAG, CONTENT, and DECLARATION.
6. The comments in the declaration files are in Doxygen format. You can automatically generate web page documentation from these header files using the Doxygen program. You can also read the documentation as html and it is contained in the /html directory in the folder that you get when you unzip the Starter Code.

You will be completing both the implementations of our ADT Stack class and XMLParser class, and you will have to implement unit tests that cover your implementations. And, as before, you will have to check your implementation against the (unknown) instructor tests.