005-XML Document Parser

Difficulty: Hard Expect Time: 120 min Author: Aga, Su

Extensible Markup Language (XML) is a markup language and file format for storing, transmitting, and reconstructing arbitrary data. It defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

In this problem, only simplified XML representation called **Limited-XML** is used.

Limited-XML is composed of three parts:

- **Attribute**

  An ***Attribute*** consists of a key–value pair, expressed using `key="value"`, note that the double quotes `"` are required.

  - ***Key*** consists of alphanumeric characters and hyphens `-`.
  - ***Value*** consists of **ASCII printable** characters, excluding `<`, `>`, and `"`, `&` characters.

    Notice: Characters that need to be escaped, i.e. `\n`, `\t`, `\r` etc., are **NOT** printable characters.

  Example:

  i. `item="30"`,

  In this case, the ***name*** of the attribute is "`item`"; the ***value*** is "`30`".

  ii. `value="abc  456  789"`

  In this case, the ***name*** of the attribute is "`item`"; the ***value*** is "`abc  456  789`".

  iii. `value="1 > 2"`

  This attribute is **invalid**, the ***value*** contains `>`.

- **Tag**

  A ***Tag*** is a markup construct that begins with `<` and ends with `>`.

  There are two types of tags: ***start tags*** and ***end tags***.

  - ***Start-tag*** is represented as follows:

    `<TagName [...Attributes]>`

    A ***start-tag*** begins with the `<` character, followed immediately by the ***TagName***, then optionally by ***attributes***, ends with the `>` character.

    TagName consist of alphanumeric characters and hyphens `-`.

    Attributes can be composed of many individual ***attributes***, separated by spaces ` `.

    Example:

    i. `<width>`

    In this case, ***TagName*** is "`width`", ***attributes*** not included.

    ii. `<point x="10" y="20">`

    In this case, ***TagName*** is "`point`", including two ***attributes*** (`x` and `y`).

    iii. `<point x="10" y="20" >`

    In this case, ***TagName*** is "`point`", and there can **zero or more spaces** between ***attributes*** and `>`.

    iv. `<point x="10"y="20">`

    This case is **invalid**, attributes are **NOT** separated by spaces.

  - ***End-tag*** is represented as follows:

    `</TagName>`

    Similar to a ***start-tag***, an ***end-tag*** inserts a slash (`/`) before the ***tag name*** and doesn't include attributes.

    Example:

    i. `</width>`: In this case, ***TagName*** is "`width`".

    ii. `</point>`: In this case, ***TagName*** is "`point`".

    iii. `</point x="10" y="20">`: This case is **Invalid**, ***end-tag*** are **NOT** allowed to contain ***attributes***.

- **Element**

  An ***Element*** consists of paired ***start-tag*** and ***end-tag*** with ***content*** in between, using the ***TagName*** in ***start-tag*** and ***end-tag*** as its **name**.

  Elements are categorized into ***Children element*** and ***Parent element***.

  - **Child element**

    An ***element*** is a ***child element*** when its ***content*** is text consists by **ASCII characters** excluding `<`, `>`, or **nothing**.

    The ***text content*** should to be **trimmed**, the meaning of trimming is given below:

    Trimming a string value means removing whitespaces from both ends of the string. whitespace is a character that is returned as a non-zero value after using std::isspace.

    Example:

    i. `<a>test</a>`

    In this case, ***name*** of the ***child element*** is "`a`", the ***content*** is "`test`".

    ii. `<b>  test  </b>`

    In this case, ***name*** of the ***child element*** is "`b`", the ***content*** is "`test`".

    iii. `<c>\n test \n</c>`

    In this case, ***name*** of the ***child element*** is "`c`", the ***content*** is "`test`".

    iv. `<d></d>`

    In this case, ***name*** of the ***child element*** is "`d`", the ***content*** is "".

    v. `<e>   </e>`

    In this case, ***name*** of the ***child element*** is "`e`", the ***content*** is "".

    vi. `<abc></xyz>`

    This case is **invalid**, ***TagName*** **DOESN'T** appear in pairs.

  - **Parent element**

    The ***content*** consists only by **ordered sequence** of ***child elements***.

    Example:

    i.
    ```
    <parent attr="1" attr2="2">
        <child attr="1" attr2="2"></child>
    </parent>
    ```

    In this case, ***parent element*** named "`parent`" has a ***child element***, named "`child`".

    ii.
    ```
    <parent attr="1" attr2="2">
        <children attr="1" attr2="2">text of children 1</children>
        <children attr="1" attr2="2">text of children 2</children>
    </parent>
    ```

    In this case, ***parent element*** named "`parent`" has two ***child elements***, both named "`children`".

    First one has ***content*** "`text of children 1`"and the second one has "`text of children 2`".

    iii.
    ```
    <parent>
        string
        <child></child>
    </parent>
    ```

    This case is **invalid**, its ***content*** contains ***text content*** and a ***child element***.

All whitespaces outside the text content of a ***child element*** and ***value*** **of an** ***attribute*** are **ignored.**

An example Tokenizer is provided to convert text to a programmatically representation without ambiguity. Please declare and implement the class in solution.cpp. This class should implement IXMLParser defined in IXMLParser.hpp, which also includes other interfaces you should implement.

The description of methods in the interfaces are shown below:

1. **IXMLParser**
   a) **Parse(string)**: Parses the given XML string and returns the root element.
2. **IAtrributeEnumerable**
   a) **Count()**: Returns count of attributes.
   b) **Get(string)**: Returns the attribute ***value*** as an ***IValue*** with the given name **exactly the same**, if the attribute does not exist, return ***nullptr***.
3. **IElement**
   a) **Name()**: Returns the ***name*** of ***element***
   b) **Attributes()**: Returns the ***attributes***, should always return a **not-null pointer**, if there's no ***attributes*** exists, **Attributes()->Count() == 0**.
   c) **Elements()**: Returns all ***child elements***, should always return a **not-null pointer**, if the ***element*** is not a ***parent element***, **Elements()->Count() == 0**.
   d) **Content()**: Returns ***text content*** as an ***IValue***, if the ***element*** is not a ***child element***, return a ***nullptr***.
4. **IElementEnumerable**
   a) **Count()**: Returns the **count** of ***elements***.
   b) **At(size_t)**: Returns the ***element*** at the given index, if the index is out of range, return ***nullptr***.
   c) **Filter(string)**: Filters the ***elements*** with the given name exactly the same.
5. **IValue**
   a) **AsString()**: Returns the value as a ***std::string***.
   b) **IsInt()**: Returns **true** only if value is represented as an ***integer***, as shown in the examples below:
      i. **"123", "+123", "-123" → true**
      ii. **"123.4", "+123.4", "-123.4" → false**
      iii. **"123.0", "+123.0", "-123.0" → false**
      iv. **".123", "+.123", "-.123" → false**
      v. **"123.", "+123.", "-123." → false**
      vi. **"" → false**
      vii. **"a" → false**
      viii. **"1a" → false**
      ix. **"1 " → false**
   c) **AsInt()**: If the value **not** represented as an ***integer*** (i.e., IsInt()==false), return "**0**".
   d) **IsDouble()**: Returns **true** only if represented as a ***number***, as shown in the examples below:
      i. **"123", "+123", "-123" → true**
      ii. **"123.4", "+123.4", "-123.4" → true**
      iii. **"123.0", "+123.0", "-123.0" → true**
      iv. **".123", "+.123", "-.123" → true**
      v. **"123.", "+123.", "-123." → true**
      vi. **"" → false**
      vii. **"a" → false**
      viii. **"1a" → false**
      ix. **"1 " → false**
   e) **AsDouble()**: If the value **not** represented as a ***number***, (i.e., IsDouble()==false), return "**0.0**".

Once you done the implementation, ensure your class that implements IXMLParser can be constructed in **CreateParser()** function provided in solution.cpp

Input

1. Please implement the required classes in solution.cpp.
2. The input for the program will be handled by the provided code.
3. The Online Judge will replace the following files:
   a) IXMLParser.h
   b) XMLTokenizer.h
   c) main.cpp
   d) TestUtils.hpp
   e) Case1.hpp
   f) Case2.hpp
   g) Case3.hpp
   h) Case4.hpp
   i) Case5.hpp
4. The following files are part of the test cases of Online Judge. Please include the following file and run the test functions in main.cpp for testing.
   a) Case1.hpp
   b) Case2.hpp
   c) Case3.hpp
   d) Case4.hpp

   There're **3** Limited-XML strings will be provided in **5** test cases.
5. Input bound:

   All Limited-XMLs used in this problem are **valid**.

   There is and can only be one root **element** for an XML (and also Limited-XML).

   Any XML functionality not mentioned by definition of Limited-XML **will NOT appears and not needed to consider**.

Output

1. Please **DON'T** print any data to **STDOUT**.
   If all tests **passed**, the program won't print any data and it's the **correct** result.
2. The output for the problem will be verified by the provided code.
   If any test **failed**, the program will terminate immediately then print the following description of the test as following format:
   Expression <call hierarchy>, Expected <value>, Got <value>.
   Example:
   Assertion failed: expression root->Elements()->Filter("food")->At(0)->Name(), expected food, got food123.
   Any test failure will print to "**your output**", only one per case.
3. The sample behaviors and the corresponding input XMLs are given in following files:
   a) Case1.hpp
   b) Case2.hpp
   c) Case3.hpp
   d) Case4.hpp

Hint

1. ALL file containing comments: **Will Replace by OJ, DO NOT EDIT!**, will be replaced by OJ system.
2. STR token in given **tokenizer** was trimmed.
   May have some whitespaces between **key**, **=**, and **"** for an **attribute**. Given **tokenizer** will ignore them.
   There may have some **back-slash (\)** in value of attribute.
3. TL; DR, Using the given Tokenizer will properly handle all formatting issues.