Homework #14
due Monday May 14, 10:00 PM

In this assignment you will extend our previous assignments to crawl web pages looking for links and XML problems.

1 Concerning the XMLTokenizer

Web pages are written in HTML, and most of these are written in largely XML-compliant form (such as XHTML) and so we can read them as if they were XML. Please consult the “XML Primer” handout about XML syntax.

We will use a home-grown XML tokenizer to read XML files.

The XML tokenizer breaks up the input into tokens of one of the following forms:

OPEN <name

ATTR name="text"

CLOSE >

ECLOSE />

ETAG </name>

TEXT text

ERROR Something went wrong, message is text. The error is fatal in that no further tokenizing should be done.

END End of input,

Not all these tokens can occur at any place. Ignoring errors (which can occur anywhere), an OPEN is always followed by zero or more ATTR tokens and then either a CLOSE or ECLOSE. The latter three tokens (ATTR, CLOSE and ECLOSE) only occur after OPEN in that way. If an element is not closed with ECLOSE, it needs to be closed with ETAG, in which case the names should match, but the tokenizer doesn’t check this property.

The following diagram shows in what order tokens (other than ERROR or END) can appear:

---

1 This tokenizer is not perfect and in particular doesn’t allow newlines in attribute values. It will find errors in certain web pages that are actually legal. Just report whatever errors it does.
This order is implemented (correctly!) by the tokenizer—you don’t need to check it. In other words, if you’re checking if an ATTR tag might appear at the start of an element, you’re checking something that will never happen. Don’t do that. Your job is not to find errors in the tokenizer, but in the XML!

To get the next token, use the method `next()`. Some of the tokens come with extra information (name or text) above. These can be queried by using the `getCurrentName` and/or `getCurrentText` methods of the XML tokenizer. The most recently returned token can be retrieved using `current()`. Alternatively, one can call `saveToken` and then call `next`; the former method ensures that `next()` reuses the current token. The `saveToken` method is very useful if you want to start reading something and then back up (call `saveToken`) and let a different method do the work. You can only back up by a single token.

If you want to handle a large number of different token types, you will find your code simpler if you use Java’s `switch` keyword.

2 Concerning the XML Reader

One of the two tasks for this homework assignment is to implement an XML reader that reads a single XML element from a character stream (otherwise known as a “Reader”). You should create a tokenizer and then read XML elements and recursively read their contents. Since the contents of an XML element may be a string as well as other elements, we recommend that you use a recursive helper method to handle reading, rather than have `readElement()` be recursive.

Along the way, you should throw an XML parse exception if you notice any of the following problems:

- The stream ends abruptly (before the end tag of the outer element is read).
- The stream has an end tag that doesn’t match the most recently unmatched open tag. (Tag matching uses LIFO order.)
- An element has the same attribute more than once.
- The tokenizer returns an ERROR.

Additionally if there are any more tokens after the single element is read, this is also an XML parse exception.

We want to avoid generating errors for things that aren’t XML at all, so if the XML tokenizer starts with an ERROR or TEXT (or ends immediately using END), the XML reader should simply return null. Only if we see an OPEN tag should we start reading XML for real and throw XML parse exceptions.

We also want to be able to read HTML as well, but HTML is not a subset of XML. We make a messy compromise and handle many (but not all) legal HTML features as long as the “root” element is html. (This means the XML reader must special case the top element, probably at the same point in the code where we decide whether we have XML at all.)

One HTML feature that you will need to handle (only inside top-level html elements) is that the end tag is always optional. That is, one can legally write:

```html
<html>
<body>
<p> I have a <i>large</i> dog at home. 
</body>
```
This example shows that an end tag may be present, and if so, implicitly ends all the intervening elements. In this case, the `<p>` element is ended implicitly at the `</body>` tag. Furthermore the `html` element is ended implicitly when the `END` comes. End tags aren’t ignored; you still must check that they are legal.

The other HTML feature, we want you to add is that when reading HTML documents (and only HTML documents), the "<script>" and "<style>" tags should be treated specially: The contents of these tags will be read as pure text (so-called CDATA) without nested tags. So the "<" character will not be treated specially, except as part of the end tag `</script>` (or `</style>` respectively). Unlike all the other tags, the end-tags for “script” and “style” elements may not be omitted. Fortunately, the XML tokenizer class provides a method called `addCDATA` that provides exactly the right semantics for these tags.

### 3 Finding URLs in HTML elements

The second half of this homework assignment involves performing a pre-order traversal over an HTML element to find URLs nested within. If a malformed URL is found, it should be recorded in the error list of the URL object from which the element was read.

HTML uses the concept of relative URLs. If the web page at [http://www.uwm.edu/schedule/](http://www.uwm.edu/schedule/) includes a URL `spring/index.html`, then this relative URL is understood in full as the following:

```
http://www.uwm.edu/schedule/spring/index.html
```

It is not a simple matter of concatenation. Instead you should use the URL constructor that takes the base URL plus the string for a relative URL to create a new URL. Normally, the base URL is the URL used to fetch the web page, but the web page may include a “base” element (mentioned below) to change how relative URLs in the document are handled.

There are four XML/HTML attributes that hold URLs: `cite`, `data`, `href` and `src`. We have declared a static array (don’t modify it!) of these names, which should make your code simpler. You should consider any such attribute of any element to hold a URL. There is one special case: a `base` element with an `href` attribute doesn’t represent a nested URL, rather it changes how all relative URLs in the document are treated, as described above.

The iterator will always keep track of the “next” URL to be returned, and so `hasNext` is very simple. Furthermore, the iterator will not support removal, so we don’t need to keep the current URL. (And how would we remove it away? We can’t edit someone’s web page!)

You should implement the iterator similarly to how we implemented the iterator in Homework #10: using a stack. In this case, the stack is of all the remaining contents of all the elements above us in the document “tree.” (In Homework #10 we only needed to handle binary trees which meant it was OK to just keep the parent node.)

We found it easiest to use `peek()` to access the top iterator rather than popping it off, only to push it right back on.

### 4 What you need to do

You need to finish the implementation of two classes.

- **edu.uwm.cs351.util.XMLReader** Use a recursive helper method.
- **edu.uwm.cs351.HTMLURLIterator** Use a non-recursive design with a stack.
These classes are used for an updated duplicate resource finder that also reports errors found. Once your code is working, it will find errors on Prof. Boyland’s pages as well as on the department and university web pages. After this semester is over, we will report back to the various webmasters so that people can fix their errors, if they are so inclined.

5 Files

The repository includes the following files:

- **src/TestConcepts.java**  Locked tests to check your knowledge of XML and the tokenizer.
- **src/Test{XMLReader,URLIterator}.java**  JUnit test cases for the engine classes.
- **src/edu/uwm/cs351/URLObject.java**  Yet another version of the URL object class, now with the ability to save error messages for later reporting.
- **src/edu/uwm/cs351/BreadthFirstWebCrawler.java**  Updated web crawler using updated classes.
- **DuplicateResourceFinder.java**  Updated duplicate resource finder that also reports XML/HTML errors found.
- **lib/testX.xml**  Sample files to test the URL iterator.

There are no random tests for this assignment.