2016 Haskell January Test
XML and XSL Transformations

This test comprises three parts and the maximum mark is 30. The **2016 Haskell Programming Prize** will be awarded for the best overall solution.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You should therefore define your own tests to complement the ones provided.

1 Introduction

The E**X**tensible **M**arkup **L**anguage, or 'XML', is a ubiquitous standard for the storage and transportation of data[1]. The syntax of XML is very simple, which makes it easy for a human to understand and also for a computer to parse and analyse. Furthermore, and of particular significance to this exercise, it is possible to define languages for processing XML "documents" (represented as text strings) that are themselves specified in XML. One such language is the e**X**tensible **S**tylesheet **L**anguage, or 'XSL', which provides a templating language (XSLT) for transforming one XML document into another. This exercise aims to develop an implementation of a very simple subset of XSLT.

2 XML

At the topmost level, an XML document comprises exactly one XML *element*, which has an associated *name* and an arbitrary combination of:

- Attributes
- Text fields
- Other 'child' elements

A new element is introduced by placing its name, often referred to as its *start tag*, within angle brackets, as in `<name>`. For the XML document to be well formed there must be a matching *end tag* later in the document which is identical to the start tag, but with a '`/`' preceding the tag name, as in `</name>`. An optional set of *attributes*, each of which is simply a name/value pair separated by an '`=`', can be associated with the element, as in `a="value"`, for example. Attribute values are strings delimited by double quotes. Element and attribute names must begin with a letter[2] but can thereafter comprise any combination of letters, digits, full stops (`.`) and hyphens (`-`); they are not allowed to contain spaces.

As an example, the XML document shown in Figure 1 represents a *mark-up* of some data relating to films. When we come to process such documents they will be in the form of a Haskell `String` containing explicit 'escape' characters such as '`\n`' and '`\"`', but the document in the figure is shown as it would appear in a text file.

Note that at the topmost level the document comprises *exactly* one element called `filmlist`. This contains three child elements, each named (tagged) `film` and each with a single attribute containing its title, e.g. `"Rear Window"`. Notice that any whitespace around the components of an attribute definition (name, '`=`' and value) is not significant, i.e. is not part of the data specified by the XML. To illustrate the point, Figure 1 contains slightly different whitespace in and around the three `title` attributes.

Each film element contains a number of child elements that define properties of the film, such as its `director`, music `composer`(s) and `year` of release. The values of these elements are strings that are delimited by '`>`' and '`<`' characters rather than double quotes (`"`), e.g. `Franz Waxman` representing the composer of Rear Window. Importantly, *all* characters between a '`>`' character and the next '`<`' character, including whitespace, collectively constitute a text field of the enclosing element. For example, the whitespace between the '`>`' of `<filmlist>` and the '`<`' of `<film title`

[1] JSON (JavaScript Object Notation) is another data interchange standard which has a number of advantages over XML. However, XML is still widely used and has a number of advantages of its own, one of which is the ability to define template-based transformations using XSLT.

[2] The standard also allows names to begin with a '_', but we will banish underscores from this test for obvious reasons!

```
<filmlist>
  <film title = "Rear Window">
    <director>Alfred Hitchcock</director>
    <composer>Franz Waxman</composer>
    <year>1954</year>
  </film>
  <film   title =  "2001: A Space Odyssey">
    <director>Stanley Kubrick</director>
    <composer>Richard Strauss</composer>
    <composer>Gyorgy Ligeti</composer>
    <composer>Johann Strauss</composer>
    <year>1968</year>
  </film>
  <film title="Lawrence of Arabia"  >
    <duration>228</duration>
    <director>David Lean</director>
    <composer>Maurice Jarre</composer>
  </film>
</filmlist>
```

Figure 1: A list of films marked up in XML

`= "Rear Window">` constitutes a text field (indeed, the first text field) of the `filmlist` element. In this case the whitespace comprises a newline character and two space characters: `"\n  "`.

Remark: Whitespace between elements is sometimes removed from XML documents and element/attribute values are sometimes *normalised* to reduce the whitespace. However, in this exercise we will preserve all whitespace during parsing; it can easily be removed/normalised later, if required.

Elements have no prescribed format with respect to the elements they contain, or the order in which they occur. All that matters is that the XML is syntactically well formed. For example, the element for the film 'Lawrence of Arabia' contains a child element named `duration` which is not included in the other two. It also does not contain a `year` element whereas the other two do. Furthermore, three composers are listed for "2001: A Space Odyssey", whereas the others have only one. The XML is perfectly valid; it's just that different elements with the same name are allowed to contain different data depending on need.

2.1 Haskell representation

We can represent the type of XML data just described with the following Haskell types:

```
type Name = String

type Attributes = [(String, String)]

data XML = Text String | Element Name Attributes [XML]
         deriving (Eq, Show)
```

Note that an element can contain an arbitrary number of child elements, so the corresponding constructor (`Element`) has a *list* of `XML` objects representing its children[3]. As an example, consider the following mark-up for the film Casablanca:

```
<film title="Casablanca">
  <director>Michael Curtiz</director>
  <year>1942</year>
</film>
```

Written out as a string the input might look like this[4]:

```
"<film title=\"Casablanca\">\n  <director>Michael Curtiz</director>\n  \
\<year>1942</year>\n</film>\n\n\n"
```

You can see whitespace in the text, in between the various elements and after the last '`>`' character. The internal representation of the document, its *parsed* version, will be the following object of type `XML`, suitably formatted here to aid readability:

```
Element "film"
        [("title","Casablanca")]
        [Text "\n  ",
         Element "director" [] [Text "Michael Curtiz"],
         Text "\n  ",
         Element "year" [] [Text "1942"],
         Text "\n"]
```

Notice how the whitespace has been preserved during parsing except for the trailing whitespace (`"\n\n\n"`) which is not part of the mark-up and which has been removed. Notice also that the only attribute is a film's `title`, so the `Attributes` list associated with the `director` and `year` elements is `[]`. The text string above is predefined in the template as `casablanca` and its parsed version as `casablancaParsed`.

You are now in a position to answer the questions in Part I, but you might like to read the next section on parsing before attempting them.

3 XML parsing

An XML parser takes the contents of an XML document as a `String`, and generates an internal representation of the document as an object of type `XML`. In what follows, and throughout the rest of the exercise, all XML input is assumed to be well formed in the sense that:

- All elements are properly nested and each start tag is matched by a corresponding end tag
- Element and attribute names are correctly formed and contain no spaces
- Attribute names and values are separated by a '`=`' character and attribute values are enclosed in matching double quotes

[3] A tree structure whose nodes have a variable number of children is called a *Rose Tree*.

[4] Recall that Haskell literal strings can be spread over several lines to aid readability by using '\' characters to mark the end of one line and its resumption on the next. Recall also that a double quote character in a Haskell string must be prefixed with a backslash, as in '`\"`'.

3.1 Parsing rules

The parsing process works by maintaining a *stack* of partially-processed elements, which is needed in order to match up the start and end tags of an element: an element is pushed onto the stack when its start tag is detected and it is popped off when its end tag is detected. The stack will be modelled in Haskell by a list of `XML` elements:

```
type Stack = [XML]
```

Notionally, the stack is initially empty, but in practice it will be primed with a single 'sentinel' element, `Element "" [] []`, whose role will soon become apparent. Assuming that we first skip any whitespace preceding the first '`<`' character (this is insignificant whitespace and should be ignored), the parsing rules, which should be read from top to bottom, are as follows:

1. If the input string is empty (`""`) then the result is the *first child* of the element at the top of the stack (see below).

2. Otherwise, apply the following rules *in the order written*:

   (a) If the first character is a '`<`' and the second is a '`/`' then we are at the beginning of an end tag, so we read the characters up to and including the next '`>`' and recurse on the remaining text[5]:

   $$\underbrace{\texttt{</element-name>}}_{\text{skip this}}\underbrace{\texttt{rest of text...}}_{\text{recurse on this}}$$

   The element on the top of the stack is now complete and this must be popped off and added to the children of the next outermost element, i.e. the element which is in the next position on the stack. Note that we should strictly check that the name inside the end tag matches that of the element at the top of the stack, but if the XML is well formed then this will always be the case: a start tag `name` is always matched by an end tag `/name`, for example, so there is no need to do the check.

   (b) If the first character is a '`<`' (i.e. no following '`/`') then we are at the beginning of a start tag, so we read the element name and any attributes which are present, including the next '`>`' character, and recurse on the remaining text:

   $$\underbrace{\texttt{<element-name}}_{\text{read the name}}\ \underbrace{\texttt{attributes>}}_{\text{read attributes}}\underbrace{\texttt{rest of text...}}_{\text{recurse on this}}$$

   Because we have discovered a new element, at this point we must push a new, partially-completed element, `Element element-name attributes []`, onto the stack before making the recursive call.

   (c) Otherwise, the first character must be the start of a new text field, so this needs to be read up to, but not including, the next '`<`' character. The text field is added to the children of the element on top of the stack, i.e. the one we are currently assembling, and we recurse on the remaining text:

   $$\underbrace{\texttt{some text characters}}_{\text{read the text}}\underbrace{\texttt{<rest of text...}}_{\text{recurse on this}}$$

   Note that the next character on the input for the recursive call in this case will be a '`<`' by construction, unless there is trailing whitespace in the document (see Section 3.3 below).

[5]Note that if the first character is a '`<`' then there is guaranteed to be a character following it if the XML is well formed.

3.2 Parsing attributes

Assuming the XML document is well formed, the structure of an attribute list is very precisely defined and is easy to parse:

1. First skip over any whitespace, giving the string `s`, say, and inspect its first character.
2. If this character is a '`>`' we're done and the result is the pair comprising the empty list of attributes and the characters *after* the '`>`'.
3. Otherwise there is at least one attribute, which can be read by:
   (a) Reading its name
   (b) Reading, and skipping over, the next '`=`' character
   (c) Reading, and skipping over, the next (opening) '`"`' character
   (d) Reading the attribute's value, which is (all) the text up to the next '`"`' character
   (e) Reading, and skipping over the next (closing) '`"`' character

   The attribute name and value form a pair, which is the first attribute in the list returned. Any remaining attributes are obtained by recursing on the characters that are left after the second '`"`' has been read and discarded.

Note that the "skipping over" process above includes reading and discarding any whitespace either side of the '`=`' and after the closing '`"`', as such whitespace is not part of the data specified by the XML.

At this point you are in a position to answer the questions in Parts I and II, although you may wish to go through the parsing example below before attempting Part II. You are advised to complete Parts I and II before attempting Part III.

3.3 Example

Note: You may wish to skip over the details in this section on first reading, and refer back to it later.

To illustrate the parsing process the following shows the state of the input string and the stack when parsing the XML mark-up for Casablanca shown above. In each case the stack is shown after applying the named parsing rule from Section 3.1. The characters of the input text that are consumed in each step are shown underlined. Initially the stack contains just the sentinel element, `Element "" [] []`.

Input: `"<film title=\"Casablanca\">\n  <director>Michael Curtiz</director>..."`
Rule: 2b
New stack:
```
[Element "film"
         [("title","Casablanca")]
         [],
 Element "" [] []]
```

Input: "\n <director>Michael Curtiz</director>\n <year>1942</year>..."
Rule: 2c
New stack:
```
[Element "film"
         [("title","Casablanca")]
         [Text "\n  "],
 Element "" [] []]
```

Input: "<director>Michael Curtiz</director>\n <year>1942</year>..."
Rule: 2b
New stack:
```
[Element "director" [] [],
 Element "film"
         [("title","Casablanca")]
         [Text "\n  "],
 Element "" [] []]
```

Input: "Michael Curtiz</director>\n <year>1942</year>\n</film>\n\n\n""
Rule: 2c
New stack:
```
[Element "director" [] [Text "Michael Curtiz"],
 Element "film"
         [("title","Casablanca")]
         [Text "\n  "],
 Element "" [] []]
```

Input: "</director>\n <year>1942</year>\n</film>\n\n\n"
Rule: 2a
New stack:
```
[Element "film"
         [("title","Casablanca")]
         [Text "\n  ",
          Element "director" [] [Text "Michael Curtiz"]],
 Element "" [] []]
```

Input: "\n <year>1942</year>\n</film>\n\n\n"
Rule: 2c
New stack:
```
[Element "film"
         [("title","Casablanca")]
         [Text "\n  ",
          Element "director" [] [Text "Michael Curtiz"],
          Text "\n  "],
 Element "" [] []]
```

Input: "<year>1942</year>\n</film>\n\n\n"
Rule: 2b
New stack:
```
[Element "year" [] [],
 Element "film"
         [("title","Casablanca")]
         [Text "\n  ",
          Element "director" [] [Text "Michael Curtiz"],
          Text "\n  "],
 Element "" [] []]
```

Input: "1942</year>\n</film>\n\n\n"
Rule: 2c
New stack:
```
[Element "year" [] [Text "1942"],
 Element "film"
         [("title","Casablanca")]
         [Text "\n  ",
          Element "director" [] [Text "Michael Curtiz"],
          Text "\n  "],
 Element "" [] []]
```

Input: "</year>\n</film>\n\n\n"
Rule: 2a
New stack:
```
[Element "film"
         [("title","Casablanca")]
         [Text "\n  ",
          Element "director" [] [Text "Michael Curtiz"],
          Text "\n  ",
          Element "year" [] [Text "1942"]],
 Element "" [] []]
```

Input: "\n</film>\n\n\n"
Rule: 2c
New stack:
```
[Element "film"
         [("title","Casablanca")]
         [Text "\n  ",
          Element "director" [] [Text "Michael Curtiz"],
          Text "\n  ",
          Element "year" [] [Text "1942"],
          Text "\n"],
 Element "" [] []]
```

Input: "</film>\n\n\n"
Rule: 2a
New stack:
```
[Element ""
         []
         [Element "film"
                  [("title","Casablanca")]
                  [Text "\n  ",
                   Element "director" [] [Text "Michael Curtiz"],
                   Text "\n  ",
                   Element "year" [] [Text "1942"],
                   Text "\n"]]]
```

Input: "\n\n\n"
Rule: 2c
New stack:
```
[Element ""
                    []
                    [Element "film"
                             [("title","Casablanca")]
                             [Text "\n  ",
                              Element "director" [] [Text "Michael Curtiz"],
                              Text "\n  ",
                              Element "year" [] [Text "1942"],
                              Text "\n"],
                     Text "\n\n\n"]]
```

The input string is now empty (""), so rule 1 now applies and we return the first child of the element at the top of the stack, which is the correctly parsed XML, but without the trailing whitespace.

4 Extensible Stylesheet Language, XSL(T)

XML imposes no constraints on the names of the elements and attributes in a document. However, by giving a defined meaning to specific names and attributes it becomes possible to use the XML syntax to define a *language* for manipulating other documents. One such example is XSLT (XSL Transformations), which is an XML-based language for generating an output XML document from a given *source* XML document. For brevity we will refer to XSLT as 'XSL', although, strictly, XSL denotes a family of recommendations for defining XML document transformation and presentation. The beauty is that an XSL(T) document is itself a valid XML document and so can be parsed by an XML parser.

An XSL document is essentially a template file that contains a mixture of output elements and XSL transformation operator elements (rewrite rules). When an input file is applied to it, each transformation is applied and the result, which comprises zero or more new elements, replaces the transformation element in the output.

Figure 2 shows an example of an XSL document[6] that will generate HTML (**H**yper**T**ext **M**arkup **L**anguage) – an XML-like syntax for specifying how information should be displayed by a web browser. You don't need to know HTML for the exercise but if you're curious, `h2` denotes a level-2 heading, `th` denotes a table header, `tr` denotes a table row and `td` denotes table data[7]. The rest is fairly self-explanatory.

Figure 3 shows the output that will be generated by the XSL of Figure 2 using Figure 1 as the source XML document. Notice how the output is a copy of the XSL template but with one table row (`tr`) for each combination of `filmlist` and `film` in the source. These `tr` elements are generated by the `for-each` transformation operator, as described below, and replace the `for-each` element in the output document.

4.1 XSL operators and XPaths

XSL supports a number of transformation operators and in this exercise we will consider just two of them: `for-each` and `value-of`. Both have a single attribute called `select` whose value is an

[6]The syntax in Figure 2 isn't precisely that of XSLT: various headers have been omitted and we do not support so-called *namespaces*, which XSL would normally assume.

[7]The HTML embedded within the XSL document is slightly archaic as it uses "presentational" markup, rather than the preferred Cascading Style Sheets (CSS) language. No matter – it works fine and simplifies the exercise.

```
<html>
<body>
  <h2>Film List</h2>
  <table border="1">
    <tr>
      <th align="left">Title</th>
      <th align="left">Director</th>
      <th align="left">Principal composer</th>
    </tr>
    <for-each select="filmlist/film">
      <tr>
        <td><value-of select="@title"></value-of></td>
        <td><value-of select="director"></value-of></td>
        <td><value-of select="composer"></value-of></td>
      </tr>
    </for-each>
  </table>
</body>
</html>
```

Figure 2: An XSL document

XPath, which defines a set of paths through the element hierarchy of the source document. These paths are defined relative to a given starting element/attribute, referred to as the *context*, and lead to zero or more elements/attributes in the source.

In this exercise we will use a subset of the XPath language wherein an XPath is a sequence of *steps* separated by '`/`' characters. A step can be:

- The current context, written '`.`'.
- An element name, in which case the step refers to all children of the current context that have the specified name. These children are 'visited' in the same order as they appear in the source XML, i.e. elements are processed from top to bottom.
- An attribute name of the form `@name`, in which case the step denotes the value of the named attribute of the current context.

For example, with reference to the film list (source) of Figure 1, if the current context is the top-level ('root') element then the XPath '`filmlist/film`' specifies every `film` element contained within every `filmlist` element, in the order that they appear in the source. If the context is a particular `film` element, e.g. set by some outer XPath '`filmlist/film`', then the XPath '`director`' specifies the `director` element of that film and '`@title`' specifies the film title (an attribute). If the context is a particular `composer` element then the XPath '`.`' refers to the current context, i.e. to the same composer element.

A `for-each` element has zero or more child elements and for each element/attribute specified by its associated XPath these child elements are recursively transformed; thus, a single `for-each` element may generate many copies of its child elements. In Figure 2, for example, there is one `for-each` element whose XPath is '`filmlist/film`' and it has a single child element named `"tr"` and this is duplicated for each `film` in the `filmlist`.

A `value-of` element denotes a single value, i.e. a text string, which is the value of the element/attribute specified by its associated XPath. However, an XPath leads to a *set* of ele-

```
<html>
<body>
  <h2>Film List</h2>
  <table border="1">
    <tr>
      <th align="left">Title</th>
      <th align="left">Director</th>
      <th align="left">Principal composer</th>
    </tr>

      <tr>
        <td>Rear Window</td>
        <td>Alfred Hitchcock</td>
        <td>Franz Waxman</td>
      </tr>

      <tr>
        <td>2001: A Space Odyssey</td>
        <td>Stanley Kubrick</td>
        <td>Richard Strauss</td>
      </tr>

      <tr>
        <td>Lawrence of Arabia</td>
        <td>David Lean</td>
        <td>Maurice Jarre</td>
      </tr>

  </table>
</body>
</html>
```

Figure 3: Output of XSL transformation

ments/attributes so the value returned is defined to be that of the *first* element/attribute that is found. For example, if the source XML is that of Figure 1 then the XSL operation `<value-of select="filmlist/film/year">` denotes the value `"1954"` (represented by `Text "1954"`), corresponding to the first film in the list that has a `year` element defined. Similarly, the XSL operation `<value-of select="filmlist/film/duration">` denotes the value `"228"` (represented by `Text "228"`), corresponding to the first (and only) film in the list that has a `duration` element defined. Of course, it may be that no elements/attributes are found by the XPath; in this case the result is the empty string, `""`. A `value-of` element is not expected to have any text fields or child elements. However, if it does then these are ignored when the XSL transformation is applied.

Referring back to Figure 3, note that the XSL transformation naturally introduces some additional whitespace. This is because the whitespace within the child elements of the `for-each` element is faithfully reproduced when the transformation is applied. This does not affect the meaning of the HTML. The output in Figure 3 can be found in a pre-prepared file called `filmTable.html` in your Lexis directory. If you wish, you can render the result in a browser by typing `firefox filmTable.html &` at the Linux prompt. The result is shown in Figure 4 (left).

Figure 4: Sample rendered HTML output

5 What to do

There are three parts to this exercise. You are strongly advised to complete Parts I and II before attempting Part III. The template file contains various test data including the input strings `s1`, `s2` and `s3` and their parsed equivalents, `x1`, `x2` and `x3`. The following questions will variously refer to these. Also, in addition to the `casablanca` example referred to earlier, the text string for the XML of Figure 1, including the whitespace, is defined in the template as `films` and its parsed version as `filmsParsed`.

A function `showXML :: XML -> String` for producing a printable representation of an `XML` object is provided in the template and this may prove useful for testing. A generalisation of this, `showXMLs`, does the same for a list of `XML` objects. Another function called `printXML` will print out an `XML` object as it would appear in a text file. The function `printXMLs` does the same for a list of `XML` objects. Some examples are given later on.

5.1 Part I: Utilities

1. Using the built-in function `isSpace`, define a function `skipSpace :: String -> String` that will discard any *leading* whitespace in a given string. For example,

```
*Exam> skipSpace "\n \n\nsome \n \n text"
"some \n \n text"
```

[1 Mark]

2. Define a function `getAttribute :: String -> XML -> String` that will look up the value of a given attribute in a given `XML` element. If the element has no attribute with that name then the result should be `""`. For example,

```
*Exam> getAttribute "x" x2
"1"
*Exam> getAttribute "x" (Text "t")
""
```

[2 Marks]

3. Define a function `getChildren :: String -> XML -> [XML]` that will return those child elements of a given `XML` element that have the specified name (`String`). For example,

```
*Exam> getChildren "b" x2
[Element "b" [] [Text "A"],Element "b" [] [Text "B"]]
*Exam>  getChildren "c" x2
[]
```

[2 Marks]

4. Using `getChildren`, or otherwise, define a function `getChild :: String -> XML -> XML` that will return the *first* child of a given `XML` element that has the specified name (`String`). If no such child exists the result should be `Text ""`. For example,

```
*Exam> getChild "b" x2
Element "b" [] [Text "A"]
*Exam> getChild "c" x2
Text ""
```

[2 Marks]

5. Define a function `addChild :: XML -> XML -> XML` that will add a new child to the existing children of a given `XML` element. The new child should be added at the *rightmost* end of the existing list. For example,

```
*Exam> addChild (Text "B") (Element "a" [] [Text "A"])
Element "a" [] [Text "A",Text "B"]
```

A precondition is that the second argument will always be an `Element`, i.e. no rule is required for the `Text` case.

[1 Mark]

6. Define a function `getValue :: XML -> XML` that will return the value of a given element. This is defined to be the concatenation of *all* text fields within the given element, i.e. including (recursively) the values of the text fields of its children. The resulting `String` should be wrapped in a `Text` constructor. For example,

```
*Exam> getValue x1
Text "A"
*Exam> getValue x2
Text "AB"
```

[4 Marks]

5.2 Part II: Parsing functions

The following questions refer to the 'sentinel' element referred to in Section 3 which is defined in the skeleton file thus:

```
sentinel :: XML
sentinel
  = Element "" [] []
```

By construction, all `XML` objects on a stack are `Elements`, i.e. there will be no `Text` objects to consider.

Importantly, all XML input is assumed to be well formed (see Section 3), so *you do not need to check for, or report, any parsing errors.* You may assume that an element value is, verbatim, the text appearing between consecutive '`>`' and '`<`' characters and that an attribute value is the text appearing between matching double quotes ('`"`').

1. Using `addChild`, or otherwise, define a function `addText :: String -> Stack -> Stack` that will add a new text field, whose value is the given `String`, to the children of the element at the top of a given stack of elements. A precondition is that there is at least one element on the stack.

   [1 Mark]

2. Again using `addChild`, or otherwise, define a function `popAndAdd :: Stack -> Stack` that will add the element at the top of a given stack to the children of the element that sits below it on the same stack. A precondition is that there are at least two elements on the stack. For example,

   ```
   *Exam> popAndAdd [x1, Element "ab" [("a","1")] [], sentinel]
   [Element "ab" [("a","1")] [Element "a" [] [Text "A"]],Element "" [] []]
   ```

   [1 Mark]

3. Define a function `parseAttributes :: String -> (Attributes, String)` that will parse a list of attributes, as described in Section 3.2. Because the XML input is assumed to be well formed, the separating '`=`' character and the double quote ('`"`') value delimiters will all be present and in the right place, although there may be redundant whitespace around the '`=`' and after the closing '`"`'. You should use your `skipSpace` function to remove this.

   A function `parseName :: String -> (String, String)` has been defined in Part II of the template file and you should use this to parse the name of an attribute. This returns the name and the text that follows it as a pair of `String`s.

   Once you've skipped over the opening value delimiter ('`"`') the attribute's value is all the text up to, but not including, the closing '`"`'.

   For example, noting that attribute lists *always* end with a '`>`',

   ```
   *Exam> parseAttributes "x=\"7\">rest of text"
   ([("x","7")],"rest of text")
   *Exam> parseAttributes "a = \"0\"  b = \"1\"  >rest of text"
   ([("a","0"),("b","1")],"rest of text")
   ```

   [3 Marks]

4. The parsing function, `parse`, has been defined for you in the template file in terms of a helper function, `parse'`:

   ```
   parse :: String -> XML
   -- Pre: The XML string is well-formed
   parse s
     = parse' (skipSpace s) [sentinel]
   ```

This uses your `skipSpace` function to remove any leading whitespace before the first '`<`' character and primes the stack with the sentinel value. Your job is to define the helper function `parse' :: String -> Stack -> XML` using the parsing rules described in Section 3.1 and the `addText`, `popAndAdd` and `parseAttributes` functions defined above. The `parseName` function that you used in the previous question should be used to parse the name of an element. Because element values are always terminated by a '`<`' in well-formed XML, the first character after the value read will always be a '`<`'[8].

Because all XML input is assumed to be well formed, the first character in the given string is guaranteed to be either a '`<`' or the next character after a '`>`'.

For example, using the `showXML` and `printXML` functions defined in the template,

```
*Exam> parse s1
Element "a" [] [Text "A"]
*Exam> showXML (parse s2)
"<a x=\"1\"><b>A</b><b>B</b></a>"
*Exam> printXML (parse casablanca)
<film title="Casablanca">
  <director>Michael Curtiz</director>
  <year>1942</year>
</film>
*Exam> parse films == filmsParsed
True
```

[7 Marks]

5.3 Part III: XSL Transformations

Assuming the following type synonyms:

```
type Context = XML

type XSL = XML
```

the following function applies the transformations defined by a given XSL document to a given source XML document by invoking a helper function, `expandXSL'`:

```
expandXSL :: XSL -> XML -> [XML]
expandXSL xsl source
  = expandXSL' root xsl
  where
    root = Element "/" [] [source]
```

Your job is to define the function `expandXSL' :: Context -> XSL -> [XML]` that takes the current context element and a parsed XSL document and returns a *list* of `XML` items that arise from expanding the XSL with respect to the source document, as described in Section 4. You can use your XML parser to parse an XSL document or use the pre-parsed test data provided (see below). Initially the context refers to a 'root' node which is a supplementary element named `"/"` whose only child is the original source element.

[8]The exception is if the text appears as whitespace *after* the last '`>`' character in the input. The remaining text in this case will be `""`.

The template file contains a number of XSL test strings `xsl1`–`xsl9` and their parsed equivalents, `xsl1Parsed`–`xsl9Parsed`. The first six can be used in part to test your implementation of `value-of`:

```
*Exam> expandXSL xsl1Parsed x1
[Text ""]
*Exam> expandXSL xsl1Parsed x3
[Text "text1"]
*Exam> expandXSL xsl2Parsed x3
[Text "text1text2"]
*Exam> expandXSL xsl4Parsed x3
[Text "att1"]
*Exam> expandXSL xsl5Parsed x3
[Text "text1"]
*Exam> showXMLs (expandXSL xsl6Parsed x3)
"<t1><t2>Preamble</t2><t3>text1</t3></t1>"
```

The last three can be used in part to test your implementation of `for-each`:

```
*Exam> expandXSL xsl7Parsed x3
[Text "att1",Text "att2",Text "att3"]
*Exam> printXMLs (expandXSL xsl8Parsed x3)
<t>text1</t>
<t>text3</t>
*Exam> printXMLs (expandXSL xsl9Parsed x3)
<t1></t1>
<t1></t1>
```

A `String` corresponding to the XSL of Figure 2 is defined in the template as `filmsXSL`. An additional XSL object called `composersXSL` is an XSL template for building an HTML unordered list (`<ul>`) of the composers of each film. These will also prove useful for testing.

You can write the output of your XSL transformation to a file using the `output` function defined in Part III of the template. This invokes `expandXSL` on given parsed XSL and source XML files and then uses `showXMLs` to generate a `String` that is written to a specified file. For example:

```
*Exam> output "films.html" (parse filmsXSL) filmsParsed
*Exam> output "composers.html" (parse composersXSL) filmsParsed
```

The two files produced should be the same as the files `filmTable.html` and `composerList.html` provided in your Lexis directory. When rendered using Firefox, they should look like those in Figure 4. Note that these files are missing some HTML headers and tags that are normally expected, e.g. `<title>`, `<html>` and `<body>`, but Firefox still renders them correctly.

Good luck!

[6 Marks]