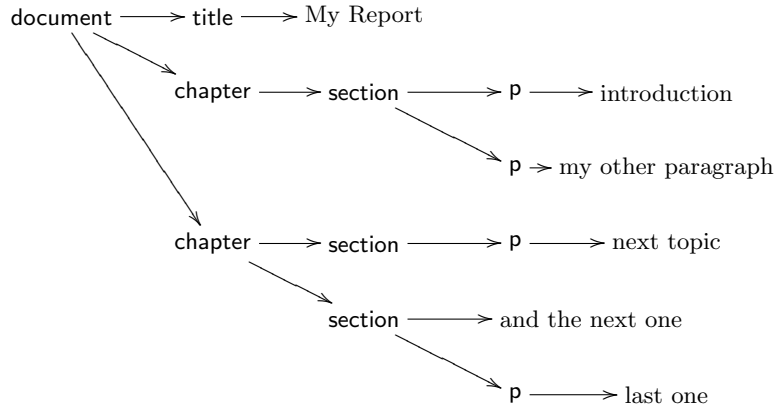
CEng 242 Homework 3

Due: 18^{th} April 2004

In this homework, you will implement an abstract data type for representing a `Document` with hierarchical markup representation. Although people mostly consider a document as a simple linear sequence of characters called text file, most of the documents with have a hierarchical structure. In XML (Extensible Markup Language) format, a document consists of nested sequence of markups and text. Such a sequence can be represented as a tree like:



The tree above can be written in an XML like file as:

```
<document>
    <title>
    My Report
    </title>
    <chapter>
        <section>
            <p>
            introduction
            </p>
            <p>
            my other paragraph
            </p>
        </section>
    </chapter>
    <chapter>
        <section>
            <p>
            next topic
            </p>
        </section>
        <section>
        and the next one
            <p>
            last one
            </p>
        </section>
    </chapter>
</document>
```

Note that all markups are in triangle brackets, and all markups should be terminated by the same tag preceded by the '/' symbol. For example `<section>` is terminated by the corresponding `</section>`. A markup can enclose 0 or more of strings and/or other markups. The strings are the actual text of the document and markups are the structure labels. When you strip down only the text portion of the above document, you will get:

```
My Report
Introduction
my other paragraph
next topic
and the next one
last one
```

The actual XML representation contains other features likes attributes in the tags, Document Type Definitions, style sheets etc. However, for simplicity we have simple tags represented by arbitrary strings. The root element in the document is always a `document` markup.

In this homework, you are asked to write a data type `Document` for processing such documents. In order to provide integrity of the document, one should start with an empty document and incrementally create markups and fill in the texts. In order to address a specific point in the tree structure, we need to have a sequence of links to follow in the document. Since these links are labeled by tags (markup names), a sequence of tags could have been enough. Starting from the root element one can address a node. For example `["document","title"]` path positions the markup containing the string "My Report". However since a markup can contain multiple occurrences of another markup, this is not enough. `["document","chapter", "section","p"]` can be the "introduction" or "my other paragraph", or "next topic" or "last one" in the example. So, such a path is defined as a list of tuples instead. First element of the tuple is the tag and second element is the integer denoting the order of that markup in the list among the same tags. So that `[("document",1), ("chapter",2), ("section",2),("p",1)]` will address the string "last one".

Implement and abstract data type (`abstype`) in ML with following interface definitions:

`PathNotFound : exception PathNotFound`
Exception to raise when an insert position is not found.

`TxtNotFound : exception TxtNotFound`
Exception to raise when a searched string is not found in the document.

`emptyDocument : Document`
A constructor value with the basic document only having a `document` markup which is empty.

`addText : Document -> (string * int) list -> int -> string -> Document`
`addText` *`Document Path Position Text`* will get a document and a path in that document. If such a path exists in the document, inserts the string *`Text`* under that path. The insert position is given by the third parameter *`Position`*. If the path is not found, `PathNotFound` exception is raised. If the insert position is greater than the last element, text is inserted as the last element.

`addTag : Document -> (string * int) list -> int -> string -> Document`
`addTag` *`Document Path Position Tag`* will get a document and a path in that document. If such a path exists in the document, inserts the markup *`Tag`* under that path. Initially markup will contain no elements. In later calls, this markup should be accessible as *`Path`*`@(`*`Tag`*`,n+1)` where `n` is the number of preceding occurrences of the *`Tag`* in the

Path, and `@` is the append operation in ML. The insert position is given by the third parameter *Position*. If the path is not found, `PathNotFound` exception is raised. If the insert position is greater than the last element, markup is inserted as the last element.

`addDocument : Document -> (string * int) list -> int -> Document -> Document`
`addDocument` *Document Path Position NewDoc* will get a document and a path in that document. If such a path exists in the document, inserts another document *NewDoc* under that path. Since it is of `Document` type, *NewDoc* has a root `document` markup. During the insertion, this root element is stripped out and all enclosed elements are inserted under *Path*. The insert position is given by the third parameter *Position*. Note that in this call more than one member can be added in a position. If the path is not found, `PathNotFound` exception is raised. If the insert position is greater than the last element, document members are inserted as the last elements.

`addTexttoLast : Document -> (string * int) list -> string -> Document`
Similar to `addText`. *Position* parameter is not given so text is added as the last element of the list.

`addTagtoLast : Document -> (string * int) list -> string -> Document`
Similar to `addTag`. *Position* parameter is not given so markup is added as the last element of the list.

`addDocumenttoLast : Document -> (string * int) list -> Document -> Document`
Similar to `addDocument`. *Position* parameter is not given so document is added as the last element of the list.

`clearPath : Document -> (string * int) list -> Document`
`clearPath` *Document Path* will get a document and a path and deletes that path, including the subtree under, from the document. If the path is not found, `PathNotFound` exception is raised. If one tries to clear the root element, the empty document is returned.

`getPath : Document -> (string * int) list -> Document`
`getPath` *Document Path* will get a document and a path and returns that markup together with all of its members enclosed in a document. Resulting document will contain a root `document` markup and that markup will only contain the markup given as the last element of the *Path* with its subtree. If the path is not found, `PathNotFound` exception is raised.

`dumpXML : Document -> string`
`dumpXML` *Document* will export the document as an XML file string as defined in the example. Carriage return is put after each markup and text element and proper indentation with 4 spaces per nesting level is put into resulting string.

`dumpTxt : Document -> string`
`dumpTxt` *Document* `will export the document as an` text file string as all markups are stripped. Carriage return is put after each text element and proper indentation with 4 spaces per nesting level is put into resulting string.

`findTxt : Document -> string -> (string * int) list`
`findTxt` *Document SearchString* will search the *SearchString* in the text elements of the document and return the path if it is found. Search is case sensitive and can not span adjacent text elements in the same level. The path is returned as the tuple list notation. If the search string is not found, `TxtNotFound` exception is raised. Only first occurrence is returned if there are multiple occurrences exist in the document.

`findReplace : Document -> string -> string -> Document`
`findReplace` *Document SearchString ReplaceString* will search the *SearchString*

in the text elements of the document and return the document with *all occurrences* is replaced by the *ReplaceString*. Search is case sensitive and can not span adjacent text elements in the same level. All other elements and document structure is kept as it is. If the search string is not found the same document is returned.

In the implementation you are free to use any internal representation and make auxiliary definitions. However you should hide the other definitions in local declaration blocks so that you will not define any other identifiers but the ones mentioned above in the global environment. Note that all document modifying functions do not update the existing document but return a new document with modifications applied. You should not use any mutable variable like references or arrays.

The required functions for text search and replacement will be posted in the newsgroup together with the sample executions.

Follow the newsgroup for submission details. Cheaters will get 0 from all of the 6 homeworks.