SOFTWARE SYSTEMS (COP-701)

ASSIGNMENT 1-REPORT

LEXER

I've created four token types for the lexical analysis:

1. **TAG_START:** Represents the start of an html tag. This token is valid only for those tags that have both the **start** and **end** tags (i.e. not tags like <!Doctype ..>,
, <img>, etc.) and is represented by the following regular expression:

   \<[^/!]([^>])*\>

2. **TAG_END:** Represents the end of an html tag. This token is also valid only for those tags that have both the **start** and **end** tags. Represented by the following regex:

   \<\/([^>])+\>

3. **TAG_START_END:** Represents those tags that have just a single tag that represents both the start and end of the tag (eg.
 and <img>). Represented by the following regex:

   \<[^/!]([^>])*\/\>

4. **STRING:** This token represents any general string (typically found as the content within a tag), made up of all possible characters except **' '** (space), **'\t'** and **'\n'**. Represented by the following regex:

   [,'."?!\-_/\\:\;\[\]\(\)\{\}\+\=*\@\#\$\%\^\&\~\`a-zA-Z0-9]+

The lexer checks for a match with any of the above tokens as it reads through the input, and once a match occurs, it returns the corresponding token type to the parser.

PARSER

The parser receives tokens from the lexer, and builds a syntax tree according to the grammar that has been defined for it. I've defined the following grammar for HTML:

---

file -> **tag_content**

tag_content -> tag_content **tag_start** tag_content **tag_end**

| **tag_start** tag_content **tag_end**

| tag_content **tag_start_end**

| **tag_start_end**

| tag_content **tag_start tag_end**

| **tag_start tag_end**

| tag_content **string**

| **string**

tag_start -> **TAG_START**

tag_start_end -> **TAG_START_END**

tag_end -> **TAG_END**

string -> **STRING**

Where TAG_START, TAG_START_END, TAG_END and STRING are the terminals (also the tokens returned by the lexer) for the grammar and the rest are non-terminals.

As the parser parses the input, it implicitly creates a **syntax tree** for the input, corresponding to and generated by the given grammar. I have generated the **Abstract Syntax Tree** (AST) corresponding to HTML with the help of this property, and also with the help of an additional **stack**.

The stack works by maintaining the most recent HTML construct encountered at its top. By "construct", I mean HTML tags or HTML string. In other words a new element is pushed onto the stack whenever a TAG_START token is received by the parser, and the stack is popped when the TAG_END token is received. TAG_START_END is handled similarly, with some modifications.

Since HTML files follow the **LIFO** structure, the last added tag/string will also be the first to be removed, so the parent of the removed tag/string in the **syntax tree** will be the tag which is *now* on top of the stack. This bit of knowledge is all we need to create the syntax tree and the Abstract Syntax Tree of the HTML file.

Note: An HTML string will always be a leaf in the syntax tree

WHAT'S STORED IN THE ABSTRACT SYNTAX TREES

HTML ABSTRACT SYNTAX TREE

A node in the HTML AST stores the following data: -

1. **Label:**

Keeps track of whether the current token is an HTML tag or a string (to be contained within a tag) and if it is a tag, then what, it's called (i.e. whether it is an **html** tag, or a **body** tag, and so on. HTML comments have been treated like tags, with "<!--" representing the start of the tag and "-->" representing the end.

2. **Attributes:**

The attributes of the tag (stored as a vector of strings), if any. If the node is a string, then the attributes consist of the individual string tokens that make up the whole string.

3. **Children:**

The children of the node, if any. In this context, “children” refers to the the tags/strings that are initialized/exist *within* the scope of the tag in question.

The structure in C++ which implements the above node is given below:

```
struct HTML_AST_node
{
     string label;
     vector<string> attributes;
     vector<HTML_AST_node> children;
}
```

LATEX ABSTRACT SYNTAX TREE

A node in the LATEX AST stores the following data: -

1. **Type:**

The type of construct denoted by the node i.e. whether the node represents an environment, a command or a string. Most of the HTML tags/strings can be mapped onto either one of those three constructs, but some might map to neither, and such tags are assigned the empty string as their type/

2. **Label:**

In case of an environment, the label of the node represents the name of the environment and in case of a command, it represents the name of the command. The string has an empty string (“”) as it’s label.

3. **String Sequence:**

If the node is of type “string”, then what is the string sequence (or simply, a collection of string tokens) it contains.

4. **Parameterized?**

If the node is of type “command”, then the command it represents will either take a parameter (a string, for example, in `/small{“Hello”}` , small is a command that takes a string as a “parameter” and outputs the same string in a small sized font), or it won’t (for example, the command `/newline`). This information is also maintained by a node in this AST. This helps create general cases for commands during generating the LATEX code from the LATEX AST, rather than considering each indivual command and then converting separately.

5. **Children:**

The children of the node, if any. In this context, if a node has a child in the HTML AST, it will also have the same child in the LATEX AST (but now converted into a LATEX construct), with one or two exceptions.

The structure in C++ which implements the above node is given below:

```
struct LATEX_AST_node
{
    string type;
    string label;
    string string_sequence;
    vector<LATEX_AST_node> children;
    bool parameterized;
};
```

THE TRANSLATION OF THE HTML AST INTO THE LATEX AST

For the translation of the HTML AST to the LATEX AST, you need to convert the nodes of the HTML AST to the nodes of the LATEX AST. This has been done on a case-by-case basis and is listed out in the following steps:

1. Traverse the HTML AST in a **postorder/recursive** manner i.e. generate all the children of an HTML node in their LATEX form before generating the LATEX equivalent of the current node. This helps in creating the LATEX AST as you just have to link the parent to it's children that you've already generated.

2. Check the **label** of the HTML AST node to figure out which tag it represents or whether it represents a string. The **label** of the node helps us determine whether the LATEX equivalent will be an **environment**, a **command** or a **string** i.e. the **type information** of the node, stored in the **type** attribute. This is done by performing a case-by-case check as to which HTML tag the current **label** corresponds to.

3. Once we know the type of the LATEX node and the label of the HTML node, we can map the label of the HTML node to the corresponding label of the LATEX construct. If the LATEX node is an environment, then its label contains the string corresponding to that particular environment name in LATEX. Similarly, if it's a command, then its label corresponds that particular command name. This is done to make the generation of the code from the LATEX AST easier.

4. Also, if the node represents a command in LATEX and once we've mapped it to the corresponding command, we also know whether that command is **parameterized** or not.

5. Some of the nodes in the HTML AST may not correspond to a type in LATEX, and that has to be taken care of by assigning the corresponding LATEX node the empty string("") as its type. If the current node represents a **string**, then it's type will be "string", but it's label will be the empty string.

6. The s**tring sequence** attribute is valid only if the node represents a string. It is calculated by creating one single string from all the strings stored in the attribute vector of the HTML node. For all other nodes, **s**tring sequence is given the empty string("") as its value**.**

7. In other words, the **type, label** and **parameterized** values default to preset values, in case those attributes are irrelevant for the current node.

8. Once we've figured out the **type, label, string sequence** and if it's a command, whether it is **parameterized** or not, we have all the information we need about the LATEX AST node, except for it's children. But since we've been solving recursively, we already have all the children of the node generated, and we just have to link the current node to its children. When the root node is linked to its children, the procedure is complete.

THE PROGRAMMING LANGUAGE

I’ve done the programming for this assignment in **C++**, and have utilized the functionalities of tools such **Flex** for the lexical analysis, and **Bison** for the parsing.