Part 1: The Lexical Analyzer

Write a lexical analyzer for processing HTML tables. Eventually, you will be asked to convert the HTML table to some other form (e.g., a CSV file with some conversions carried out). The input will be a text file containing source code for an HTML table. The output will be tokens and their corresponding lexemes, and possibly some error messages (there aren't too many of these, for this phase). Later on, these tokens will be input to a parser (Part 2 of the project) which completes the translation process.

The goal of lexical analysis is to scan source code, filter out white spaces and comments, identify lexical errors, and most importantly, break up the code (which is a stream of characters) into lexical tokens, the most basic elements of a source program. Given the following HTML source, for instance:

```
<!-- a comment -->
<table>
<tr><td> ballpen </td> <td> 10.25 </td>   </tr>
<tr> <td> pencil </td> <td> 5.55 </td>    </tr>
</table>
```

The lexical analyzer should produce the following tokens and lexemes:

```
TAGIDENT    <table
GTHAN       >
TAGIDENT    <tr
GTHAN       >
TAGIDENT    <td
GTHAN       >
IDENT       ballpen
ENDTAGHEAD  </
IDENT       td
GTHAN       >
TAGIDENT    <td
GTHAN       >
NUMBER      10.25
ENDTAGHEAD  </
IDENT       td
GTHAN       >
ENDTAGHEAD  </
IDENT       tr
GTHAN       >
TAGIDENT    <tr
GTHAN       >
TAGIDENT    <td
GTHAN       >
.
.
.
```

(*Note:* The output above is incomplete; please refer to the sample files on Moodle for the complete output.)

There are only three lexical errors possible:

- badly formed number — occurs when text like `5.=` exist
- illegal character
- unexpected end of file

The lexical analyzer should recognize the following tokens:

token	informal description	sample lexemes
PLUS	the character +	+
MINUS	the character -	-
MULT	the character *	*
DIVIDE	the character /	/
MODULO	the character %	%
EXP	characters *, *	**
LPAREN	the character (	(
RPAREN	the character)	)
EQUALS	the character =	=
LTHAN	the character <	<
GTHAN	the character >	>
COLON	the character :	:
SCOLON	the character ;	;
COMMA	the character ,	,
PERIOD	the character .	.
QUOTE	the character '	'
DQUOTE	the character "	"
NUMBER	any numeric constant	`0`, `123`, `-5.5`, `2.35`, `0.88888`, `1e20`, `2.2E-5`
IDENT	any sequence of letters	`table`, `description`, `value`, `th`
TAGIDENT	< followed by any sequence of letters	`<table`, `<th`
ENDTAGHEAD	characters <, /	</
EOF	the end-of-file marker	

You are required to use a lexical analyzer generator or library (such as `flex`) so that you can focus on designing the regular expressions.

Your program should specify the input and output file names through the command line arguments. For example, if the program is named `lexer`, then it should be executed as:

```
./lexer tab.html tab.txt
```

Part 2: The Parser

Using the lexical analyzer you built for Part 1 of this project, write a parser that converts the HTML table to some a corresponding CSV (comma-separated values) file with some conversions carried out. The input is still a text file containing source code for an HTML table. The output, assuming the source has no lexical or syntax errors, is a CSV file that contains equivalent entries.

The goal of parsing is to process the tokens produced during lexical analysis and check if it follows the prescribed grammar of the source programming language, in this case, HTML for tables.

Given the following HTML source, for instance:

```
<!-- a comment -->
<table>
<tr><th> particulars </th> <th> amount </th>   </tr>
<tr><td> ballpen </td> <td> 10.25 </td>   </tr>
<tr> <td> pencil </td> <td> 5.55 </td>    </tr>
<tr> <td> total  amount </td> <td> =10.25+2*5.55 </td>  </tr>
<tr> <td> formula used :</td> <td> [10.25 + 2*5.55] </td>    </tr>
</table>
```

The parser should produce the following CSV file:

```
particulars,amount
ballpen,10.25
pencil,5.55
total amount,21.35
formula used:,10.25+2*5.55
```

Note that if a table entry begins with an equal sign (=), an arithmetic expression follows; in this case, the expression should be evaluated, and the result is placed instead of the formula. A table entry may also begin with an opening square bracket ([), in which case, an expression will also follow and will be terminated by a closing square bracket (]); the expression should be a valid expression but will simply be echoed back to the CSV file as is.

For non-expression entries, the tokens should be written verbatim, but separated by exactly one space each, except for punctuation symbols. Whenever a punctuation symbol (non-identifier or non-number) occurs, the symbol should be appended to the previous token followed by a space to separate it from the next token. For example, the text stream

```
Conversion rule:12 eggs ,as a  group,  make 1dozen;thanks  .
```

if it is a table entry, should be converted to:

```
Conversion rule: 12 eggs, as a group, make 1 dozen; thanks.
```

You are required to use a parser generator tool or library (such as `bison`) so that you can focus on designing the grammar.

Your program should specify the input and output file names through the command line arguments. For example, if the program is named `parser`, then it should be executed as:

```
./parser tab.html tab.csv
```

Other Admin Matters

- You may work individually or in groups of at most three members, but if working on groups, please choose your groupmates wisely. Don't just give in to peer pressure.

- You may use programming languages other than C/C++ (such as Java or Python) when implementing your project, but please keep in mind it will be at your own risk to do so, i.e., no "official" support will be given for tools/libraries other than `flex` and `bison`.

  You are allowed to use methods and functions from the built-in standard libraries.

- You are *encouraged* to use a Git repository (e.g., GitHub, Gitlab, BitBucket) to manage your project's source code.

  (Pro-tip: DISCS hosts its own instance of Gitlab at `https://gitlab.discs.ateneo.edu/`)

- You are *required* to document your project. For this purpose, create a `readme` file that describes/explains the following:

  - Listing of the complete set of files included in the submission
  - Description of the minimum machine specs required (e.g., which operating systems are supported)
  - Complete instructions on how to recompile/rebuild your programs from the source code
  - Complete instructions on how to run and use your program

  You are encouraged to include documentation within your source files. No need to use a special format (e.g., Javadoc, Doxygen); comments are fine.

- Submit all pertinent and relevant project files via Moodle. Your submission should include the following:

  - All the source files (`*.l`, `*.y`, `*.c`, `*.h`, `*.cpp`, `*.hpp`, `*.java`, `*.py`), whichever is applicable.
    Do not include the resulting binary files (`*.o`, `*.out`, `*.exe`, `*.class`, `*.pyc`, `*.pyo`).
  - A fully-accomplished and signed Certificate of Authorship.
    There is a COA for individual work and a COA for groups, so use whichever is appropriate.
  - The `readme` file, as decribed above.
  - A short PDF document outlining the contributions of each member (for groups).

- Project submission deadline is on **May 2, 2019 (Saturday).** Consult the course syllabus for policies regarding late submissions.

- Please direct all your questions and clarifications about the final project in the `#project-help` channel on the Discord server.

Grading System

Your project will be graded using the following criteria:

- 60 points — correct handling of the sample test cases
- 30 points — correct handling of my secret test cases
- 10 points — documentation

For those who will work in groups, a peer rating system will be used to determine your individual grades. Details to follow.

A bonus of 1 point per day early (with a maximum of 5 points) will be awarded to you if you submit at least one day before the prescribed deadline.