Project 2: scanners and parsers

1. Wstęp.

This assignment involves writing a scanner and parser for HTML that enforce some simple grammar rules, e.g., that the $< li >$ tag for list items can occur only within lists, or that tags specifying boldface $< b >$ and italics $< i >$ should be properly nested. As with the previous assignment, the goal is to get you sufficiently acquainted with *lex* and *yacc* that you can start the main compiler project. For this reason, the grammar rules used in this assignment cover only a very small part of the complete HTML syntax.

Documentation for *lex/flex* is to be found here:
`http://www.kompilatory.agh.edu.pl/pages/tk-laboratorium/flex.html`
and for *bison* here:
`http://www.kompilatory.agh.edu.pl/pages/tk-laboratorium/bison.html`

1.1. **Functionality.** Your program should read its input from *stdin*, ensure that the input follows the grammar rules for our subset of HTML, discard all HTML tags, and write the remaining text to *stdout*. Error messages (see below) should be written to *stderr*.

The HTML grammar (see below) provided may generate conflicts when you translate it to a YACC input file. If this happens (it may or may not), for this assignment you are not required to remove these conflicts. However, if this happens you should figure out why the conflict(s) are occurring, so that you can determine whether the default action taken by your parser is appropriate.

1.2. **HTML grammar. Lexical rules:**

A *tag* is a sequence of characters of the form $< S >$, where S is a sequence of printable characters not beginning with a whitespace character and not containing any ">" characters. Our grammar recognizes the following tags:

DOC_START	:	$< html >$
DOC_END	:	$< /html >$
HEAD_START	:	$< head >$
HEAD_END	:	$< /head >$
BODY_START	:	$< body >$
BODY_END	:	$< /body >$
BF_START	:	$< b >$
BF_END	:	$< /b >$
IT_START	:	$< i >$
IT_END	:	$< /i >$
UL_START	:	$< ul >$
UL_END	:	$< /ul >$
OL_START	:	$< ol >$
OL_END	:	$< /ol >$
LI_START	:	$< li >$
LI_END	:	$< /li >$

Additionally, the token TAG will match any tag that is not one of the tags listed above, and the token TEXT will match any (single) character that is not within a tag or comment; ; and the token SPACE will match any non-empty sequence of whitespace characters.

Syntax rules:

Syntax rules are made up of *tokens* and *nonterminals.* A token denotes one or more related strings that are matched by the scanner (e.g., "identifier", "integer constant"). A nonterminal denotes a set of strings with similar syntax structure (e.g., "declaration", "while loop"). In the rules below, tokens are written in `teletype` font, like this; nonterminals are written in *italics*, like this. The symbol $\emptyset$ denotes the empty sequence.

A syntax rule consists of a left hand side and a right hand side, separated by a colon ":". The left hand side is a nonterminal whose structure is defined by the rule. A right hand side consists of a set of alternatives, separated by "|". Each alternative is a sequence (possibly empty) of tokens and nonterminals.

Doc	:	*Wspace* `DOC_START` *Wspace Head Wspace Body Wspace* `DOC_END` *Wspace*
Head	:	`HEAD_START` *Html* `HEAD_END`
Body	:	`BODY_START` *Html* `BODY_END`
Wspace	:	`SPACE`
	|	$\emptyset$
Html	:	*Item Html*
	|	$\emptyset$
Item	:	`BF_START` *Html* `BF_END`
	|	`IT_START` *Html* `IT_END`
	|	*List*
	|	*Other*
List	:	`UL_START` *Wspace ItemList Wspace* `UL_END`
	|	`OL_START` *Wspace ItemList Wspace* `OL_END`
ItemList	:	*ItemList Wspace OneItem*
	|	*OneItem*
OneItem	:	`LI_START` *Html* `LI_END`
Other	:	`TAG`
	|	`TEXT`
	|	`SPACE`

The start symbol for the grammar is *Doc.*

1.3. **Syntax errors.** Your program will be expected to deal with errors in a "reasonable" way. Error messages should be printed to *stderr.* They should be specific and should contain enough information (with at least a line number) to allow the user to locate the problems. Since we have not yet discussed error recovery, you are not required to recover from syntax errors: it is OK for your program to exit after detecting the first syntax error. However, if you choose to implement error recovery, that is OK too.

2. Invoking your program.

Your executable program will be called *myhtml2txt.* It will read input from *stdin* and write its output to *stdout.* Thus, to translate an HTML file *foo.html* to a text file *bar.txt*, invoke your program as

myhtml2txt < *foo.html* > *bar.txt*