CS 310 | Fall 2020 | Units 1 + 2 | Lexer & Parser

This document covers two cumulative units. Complete each unit by its respective due date listed on the course calendar.

Follow the instructions using Java code and documentation. Submit the indicated Java source files and PDF documents (📎) via the assignment in the eCampus lecture section by the due date on the course calendar. For each source file, include JavaDoc with your full name as `@author` and a brief statement acknowledging that your work complies with the academic integrity policy. The grading rubric is in this document (percentages are scaled as in the syllabus table).

Unit 1 | Lexer

Topics: *Finite state automata (FSA), lexical analysis (lexer), tokens, lexemes*

20% | **Finite State Automaton**

- A Diagram a finite state automaton which tokenizes the language defined by the grammar in this document (see the appendix).
- B Illustrate the diagram using professional software and publish it as a 📎 PDF document.
- C Label each terminal state with the corresponding token.
- D Do not include pseudocode, regular expressions, or more than one diagram.

80% | **Lexer**

- E Implement a lexer equivalent to your FSA (revising each as necessary to maintain that equivalence).
- F Code the class 📎 `unit.Lexer` as Java source.
- G Extend the provided abstract class `model.AbstractLexer` and comply with its JavaDoc comments.
- H Pass the provided unit tests in `grade.LexerTests` (each test is weighted equally).

Unit 2 | Parser

Topics: *Railroad diagrams (syntax diagrams), LL(k) parsing, recursive descent parsing, precedence, associativity*

20% | **Railroad Diagram**

- I Diagram a set of railroad diagrams equivalent to the grammar in this document (see the appendix).
- J Illustrate the diagrams using professional software and publish it as a 📎 PDF document.
- K Each production is a separate diagram.

80% | **Parser**

- L Implement an LL(1) recursive descent parser equivalent to your railroad diagram (revising each as necessary).
- M Code the class 📎 `unit.Parser` using Java source.
- N Extend the provided abstract class `model.AbstractParser` and comply with its JavaDoc comments.
- O Left associativity must be implemented with iteration. Right associativity must be implemented with recursion.
- P Utilize your previously implemented 📎 `unit.Lexer` for lexical analysis (revised if necessary).
- Q Pass the provided unit tests in `grade.ParserTests` (each test is weighted equally).

Code Restrictions

The following code restrictions apply to both units.

- A Do not modify any code in the `grade` or `model` packages.
- B The following are the only data types, classes, methods, and constructors permitted to be used in your code.
  1. Primitive data types, enumeration types, and arrays thereof.
  2. Exception types (but only those required by the Javadoc comments in the abstract classes).
  3. Any which are in the `grade` and `model` packages.
  4. Any which you implement in the `unit` package.
- C By exclusion above, the following data types, classes, methods, and constructors are forbidden in your code.
  1. Strings (except as parameters to exception constructors or to tracing framework methods).
  2. String builders, string buffers, string readers, scanners, regular expressions, pattern matching, user or file input or output, equality and inequality, case sensitivity and insensitivity, splitting, joining, trimming, slicing, hashing, and copying.
  3. Lists, sets, maps, and any other abstract data types.

Appendix 1 | Grammar

The following EBNF grammar with informal semantics defines a language for propositional logic with variable assignments.

A Terminal symbols are in `blue` text and are tokenized by the lexer.
B Metasymbols are in black text.
C Whitespace (space, tab, new line, and carriage return) is not a lexeme. It can only appear between lexemes, not within them.
D Keywords are case insensitive.
E Variable names are case sensitive.

Syntax			Semantics
`<program>`	→	`{ <assignment> }* <evaluation>`	Initializes a global data structure called the lookup table whose data associates variable names with boolean values Returns the result of `<evaluation>` as a boolean, or throws an exception if the `<program>` is invalid for any reason
`<assignment>`	→	`bool <variable> = <equivalence> ,`	Associates the variable name `<variable>` with the boolean value `<equivalence>` in the lookup table
`<evaluation>`	→	`test <equivalence> ?`	Returns the result of `<equivalence>` as a boolean
`<equivalence>`	→	`<implication> { <-> <implication> }*`	Returns the left-associative logical equivalence of `<implication>`$_1$ through `<implication>`$_n$ as a boolean
`<implication>`	→	`<disjunction> { -> <disjunction> }*`	Returns the right-associative logical implication of `<disjunction>`$_1$ through `<disjunction>`$_n$ as a boolean
`<disjunction>`	→	`<conjunction> { v <conjunction> }*`	Returns the left-associative logical disjunction of `<conjunction>`$_1$ through `<conjunction>`$_n$ as a boolean
`<conjunction>`	→	`<negation> { ^ <negation> }*`	Returns the left-associative logical conjunction of `<negation>`$_1$ through `<negation>`$_n$ as a boolean
`<negation>`	→	`<expression> [ ' ]`	Returns the value of `<expression>` or, if the optional terminal is given, its logical negation as a boolean
`<expression>`	→	`( <equivalence> ) | <boolean>`	Returns the value of the given `<equivalence>` or of the given `<boolean>` as a boolean
`<boolean>`	→	`1 | 0 | <variable>`	Returns the value *true* if the literal 1 is given or *false* if the literal 0 is given as a boolean Otherwise, returns the value *true* or *false* associated with the given `<variable>` name in the lookup table as a boolean, or throws an exception if there is none
`<variable>`	→	`A | B | C | … | Z`	Returns the given variable name

Appendix 2 | Example Sentences

See the provided unit tests in `grade.LexerTests` and `grade.ParserTests` for example sentences in the language.