Lexer-Parser Project

Create a lexer and parser for the programming language SIMPLE. The language has only assignment statements with expressions containing only integers, variable identifiers and the (+) operator. The following program is valid for the language:

```
x23 = 18
y = x23+ 45  +2
```

The parser will consist of two major components:

- Lexer: identifies the words (tokens) in a file of text.
- Parser: Determines if a statement or statements is valid based on the expected structure of those statements, and displays an error if not.

Part 1: Lexer

A Lexical Analyzer (or Lexer for short) performs the task of finding the tokens in a large set of text.

A token is a sequence of characters that form a single entity in a language. For an English sentence, each word is a token. For the SIMPLE coding language, there are a number of token types, including Identifier, AssignOp, Plus, and Integer. Your program will have a Token class that identifies two parts of a token, its type and its value.

For instance, given the text “x23=32”, a Lexer would identify three tokens: an Identifier with value “x23”, an AssignOp with value “=” and an Integer with value “32”.

Your Lexer will take a text file as input and create a list of the tokens contained in the file. The Lexer will identify the following token types:

Identifier: starts with a letter followed by 0 or more letters and digits.

Integer: a sequence of digits

AssignOp: a single '='

PlusOp: a single '+'

UnknownOp: e.g., '$'

EndOfFile: for every file, the last token generated by the Lexer will be this one.

If your Lexer was provided the file:

```
aa=34
bb=45+6
```

It will generate the following Tokens:

Token Type	**Token Value**
Identifier	aa
AssignOp	=
Integer	34
Identifier	bb
AssignOp	=
Integer	45
PlusOp	+
Integer	6
EndOfFile	-

Lexer: Java Specifications

- Your program will have a Lexer class and a Token class. The Lexer provided in the starter code is incomplete. You can use the Token class as is.
- The class Lexer in the starter code provides the following methods:
  - public constructor which takes a file name as a parameter
  - private *getInput* which reads the data from the file into a String.
- You'll need to add the following methods to Lexer:
  - public *getNextToken* which returns a Token object, the next one in the input stream.
  - private functions for *getIdentifier* and *getInteger*. These helper functions should handle extracting the rest of an identifier/integer once one is identified.
  - getAllTokens is in the starter code but just a stub. Code it so that it calls *getNextToken* within a loop and returns a list of Tokens.
- The class Token has two data members, type and value, both of type String, along with a constructor which takes two parameters, and a toString method which prints a token's type and value.

Testing your code

The starter code has several input files for testing. You want your code to work for all of those. There is Junit test code provided which will be used to grade your project. You can run Lexer as normal, but run LexerTest to test all the input files as once. Once you pass those tests, your app is working.

Submission

You’ll submit your project by committing/pushing your code from the starter repo you get from GitHub classroom.

Part 2: Parser

The Parser’s job is to determine if a given code file is syntactically correct and report either “valid program” or a particular error message. For instance, the code:

```
x12=17+43
a = x12+35+1
```

is syntactically correct for the SIMPLE language so the parser should report, “valid program”. The code:

```
17=x+3
```

is not valid and the parser should report, “Expecting Identifier on line 1”.

The Parser should find and report the following errors:

- Expecting identifier
- Expecting assignment operator
- Expecting identifier or integer
- Expecting identifier or add operator
- Identifier not defined

The Parser should not access the input stream directly, but should instead create a Lexer and call getAllTokens, then loop through those Tokens.

Use a *recursive descent parsing* scheme: define a method *ParseX* method for each particular "part of speech". For example, parseAssignOp will be called when you are expecting a token of type "AssignOp".

IdTable

The IdTable is a class that tracks the identifiers within a program. When an identifier appears on the left-hand-side of an assignment statement, add it to the IdTable. When an identifier appears on the right-hand-side of an assignment statement, check the IdTable to see if the identifier has been defined (it is an error if not).

Parser Java Specifications

Define the class Parser with the following data members and methods:

- A data member for the tokens and the IdTable.
- A constructor which creates a Lexer and places the results of "getAllTokens" into a data member
- *parseProgram* -- this method drives the process and parses an entire program. This method should call parseAssignment within a loop.
- *parseAssignment* -- this method should parse a single assignment statement. It should call parseId, parseAssignmentOp, and parseExpression.
- *parseId*-- this method parses a single identifier
- *parseAssignOp*-- this method parses a single assignment operator

- *parseExpression*-- this method parses an expression, i.e., the right-hand-side of an assignment. Note that expressions can include an unlimited number of "+" signs, e.g., "Y+3+4"
- *nextToken*-- this method gets the next token in the list and increments the index.
- *putToken*-- this method decrements the index so that the token just looked at will be looked at again.
- *toString*

Define the class IdTable with the following data members and methods:

- a hashmap data member with String key and Integer value. The keys are the identifiers and the values represent the address in memory in which the identifier will be stored (this is used in the Code Generator/Interpreter project). Assign the index for each entry such that the first id will have address 0, the second id will have address 1, and so on.
- *add*-- this method adds an entry to the map, you need only send the id.
- *getAddress*-- this method returns the address associated with an id, or -1 if not found.
- *toString*

Testing your code

The starter code has several input files for testing. You want your code to work for all of those. There is Junit test code provided which will be used to grade your project.

Submission

You’ll submit your project by committing/pushing your code from the starter repo you get from GitHub classroom.