ECE 251 Assignment #2:
Be the ANTLR (v2)

Patrick Lam

Due: November 5

1 Problem Description

One of the themes I've been repeating this term is that compiler technology is useful for many tasks besides building compilers for general-purpose programming languages.

SQL, or the Structured Query Language, is a ubiquitous domain-specific language for talking to databases. Basic SQL is not difficult to pick up, but it is beyond the scope of this course. However, the parsing of SQL is very much on-topic for this course, and it is actually fairly simple.

In this lab, you will build a lexer and parser for a small SQL subset by hand, using the recursive-descent parser construction techniques we saw in class. Please do not use a parser generator for this assignment; I would like you to build at least one parser by hand in this course.

You may wish to consult the SQLite documentation's syntax diagrams for information on SQL:

`http://www.sqlite.org/syntaxdiagrams.html`

We will only be implementing a subset of this language.

2 Task 1: Lexical Analysis

We've seen that the two first tasks in creating a compiler are lexical analysis and parsing. The first task will be to create a lexer for your language, which will account for 20% of the marks for this lab. Specifically, I provide a class which splits the stream of characters into a stream of words. Your task is to create `Token`s for these words, by plugging in the appropriate regular expressions into the `Token` class.

I've put up a simple test suite for lexical analysis. You should also create a couple of test cases (but, this time, you don't need to hand them in). I'll only run the test cases that I post.

Token specifications. SQL is case-insensitive. Your lexer must differentiate between keywords, identifiers, and literals (boolean, numeric and string). The enum type `Token.Type` contains all of the tokens that you need to recognize.

- Keywords are obvious.
- Identifiers start with a letter (a-z) or an underscore, and continue with letters, underscores, and digits, or contain arbitrary characters between two double-quote marks (`"`). (To include a double quote, write two double quotes.)

- Boolean literals may be the strings 'TRUE' or 'FALSE'.
- Numeric literals contain at least one digit and possibly a decimal point, and may finish with 'e', an optional sign, and at least one digit.
- String literals contain arbitrary characters between single-quote marks (′). Two single-quotes in a row indicate a single single-quote in a string literal.

Fill in the blanks in the `Token` class with the regular expressions for each token type. Your regular expressions should capture the definitions I provided above; in particular, they should not accept any strings that don't match these definitions.

Infrastructure. I've provided all of the infrastructure for Part 1. You just need to provide the regular expressions as the parameters to the `Type` enumeration declarations. Test your program with the included `ca.uwaterloo.ece251.LexerDriver` class, which takes a file as input and outputs the token stream.

3 Task 2: Parsing

The main part of this assignment is writing a recursive-descent parser from scratch for our subset of SQL. Given a `TokenStream` from task 1, which provides a stream of `Token` objects, you need to create an Abstract Syntax Tree containing these `Tokens`[1]. (Some tokens from task 1 won't appear in valid programs. You only need to handle the provided grammar.) Figure 1 presents the grammar for our SQL subset.

We've provided a `ParserDriver` which calls your classes and prints out a representation of the parse tree. I may also provide some graphical output if I can. We will grade your work by `diff`ing your output with the expected output.

Infrastructure. I've provided the `ParserDriver` class, which parses a single SQL `stmt`, as well as all the classes in the sub-package `ca.uwaterloo.ece251.ast.*`, which implement an AST. You provide the `Parser` class, which provides a `stmt()` method returning a `Stmt` instance representing the parsed SQL statement.

4 Submission guidelines

Submit a `tar.gz` file containing your src directory, as in Lab 1. This file should include all of the files that I've provided, modified as necessary.

[1] In a real AST, you also need to keep the `Token` objects around, so that you can report line numbers along with errors. You might do that by keeping a `Map` from `ASTNode` objects to `Token` objects. We won't do that for this lab.

Figure 1 Grammar for SQL subset.

```
stmt : (create_table_stmt
     | drop_table_stmt
     | insert_stmt
     | select_stmt
     | update_stmt
     | vacuum_stmt) SEMICOLON;

create_table_stmt : 'CREATE' ('TEMP' | 'TEMPORARY')? 'TABLE'
        ('IF' 'NOT' 'EXISTS')?
        qualified_table_name
        (('(' column_def (',' column_def)* ')') | 'AS' select_stmt);

qualified_table_name : (ID '.')? ID;

column_def : ID (TYPE_NAME)?;

TYPE_NAME : 'NULL' | 'INTEGER' | 'REAL' | 'TEXT' | 'BLOB';

drop_table_stmt : 'DROP' 'TABLE' ('IF' 'EXISTS')?
        qualified_table_name;

insert_stmt : ('INSERT' | 'REPLACE') 'INTO' qualified_table_name
              (('(' ID (',' ID)* ')')?
                   (('VALUES' '(' expr (',' expr)* ')') | select_stmt))
              | 'DEFAULT' 'VALUES');

select_stmt : 'SELECT' ('DISTINCT' | 'ALL')?
                (result_column (',' result_column)*)
                ('FROM' join_source)?
                ('WHERE' expr)?
                ('LIMIT' expr)?;

// you'll have to refactor common prefixes here
result_column : ((ID '.')? '*')
              | expr;

join_source : single_source (',' single_source)*;

single_source : (qualified_table_name 'AS' ID)
              | ('(' select_stmt ')' 'AS' ID)
              | ('(' join_source ')');

update_stmt : 'UPDATE' qualified_table_name
              'SET' ID '=' expr (',' ID '=' expr)*
              ('WHERE' expr)?;

vacuum_stmt : 'VACUUM';

expr : literal_value
     | qualified_table_name
     | unary_operator expr
     | expr binary_operator expr
     | expr ('ISNULL' | 'NOTNULL')
     | '(' expr ')';

unary_operator: '+' | '-';

binary_operator: '+' | '-' | '*' | '/' | '%' | '<' | '<=' | '>' | '>=' | '=';
```

5 Extra explanations

Let's first start with some sample input and output.

Lexer input and output. If you run the lexer driver, it'll print out the tokens that it finds.

```
plam@noether:~/courses/plt/a2-solns$ cat tests/drop.sql
DROP TABLE foo."bar";
plam@noether:~/courses/plt/a2-solns$ java -cp classes ca.uwaterloo.ece251.LexerDriver tests/drop.sql
t: [DROP: ('DROP'), line 1, col 0-4]
t: [TABLE: ('TABLE'), line 1, col 5-10]
t: [ID: ('foo'), line 1, col 11-13]
t: [DOT: ('.'), line 1, col 14-14]
t: [ID: ('"bar"'), line 1, col 15-19]
t: [SEMICOLON: (';'), line 1, col 20-20]
```

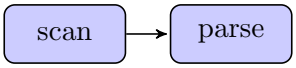
Parser input and output. The parser driver prints out a tree in XML form. (I may also provide a graphical viewer later, but I just want to get the lab out right away first.)

```
plam@noether:~/courses/plt/a2-solns$ cat tests/insert4.sql
INSERT INTO a.b (bar, baz, bat) values (2, 4, 8);
plam@noether:~/courses/plt/a2-solns$ java -cp classes ca.uwaterloo.ece251.ParserDriver tests/insert4.sql
<insert tn=a.b>
 <literal value=2/>
 <literal value=4/>
 <literal value=8/>
</insert>
```

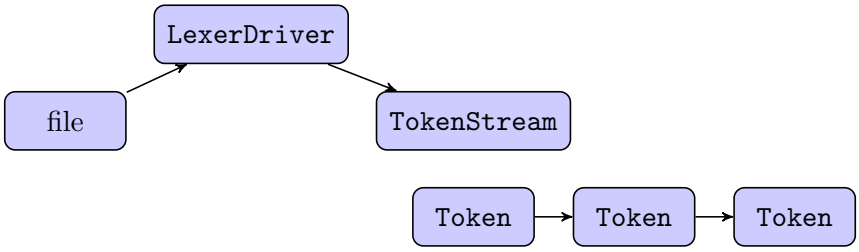
I will eventually provide test harnesses for you as I did for Lab 1.

6 Big-picture view

Here are some pictures which may help you put things in context. You are implementing these boxes:

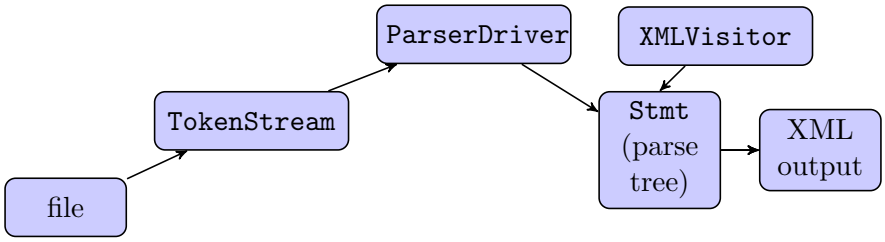


Let's look at the lexer driver:



So, your `LexerDriver` takes a file and produces/prints out a `TokenStream` consisting of `Token` objects. Every `Token` object has a `LexerToken` and a `Type` as well. You need to make sure that the `Type` is correct.

The parser driver works like this:



That is, the `ParserDriver` gets a `TokenStream` based on the file. It produces a `Stmt` parse tree. The `XMLVisitor` takes the `Stmt` and produces XML output.

Constructing the parser. So you have a `TokenStream` available to you in the `Parser` class. What do you do with it? The interface is really simple: you request a `Token` from the `TokenStream` with "nextElement()" and check whether or not there are any more tokens with "hasMoreElements()"[2]. The important bit for the `Parser` is the `Token`'s type, which is stored in its type field. You can also print error messages by looking at the included `LexerToken`, stored in the `rawToken` field of the `Token`.

To recap, you need to implement the `Parser` class, which makes calls like `new CreateStmt(...)` to produce ASTs. Look at the provided AST code to figure out what the constructors are. My code will print out the ASTs for you, using either the `XMLVisitor` or something I'll provide later.

[2] `TokenStream` implements the `Enumeration` interface from the Java library.