Lab 2 Tutorial
(An Informative Guide)

Jon Eyolfson

University of Waterloo

October 18 - October 22, 2010

Outline

- Introduction
- Good News
- Lexer and Parser
  - Infrastructure
  - Your Task
  - Example
- Conclusion

Introduction

- In this lab you will be implementing a lexer and parser without ANTLR
- The grammar you will be using is a subset of SQL which you can find in the hand out
- You will have to write a recursive descent parser, for further explanation refer to Wikipedia (just kidding!)

The Good News

- A skeleton of the Java code is provided
- The time commitment is far less than lab 1 (you only need to write two classes)
- All the information should be provided in the lab hand out and should provide additional insight

Lexer Infrastructure (1)

- As expected the lexer takes reads the input and seperates it into tokens
- The TokenStream class contains two methods **hasMoreElements** which returns a boolean and **nextElement** which returns a Token (this is because it implements the Enumeration<Token> interface)
- Don't mind the reader object, the details are not important
- **hasMoreElements** returns true if the file has not been completly read, false otherwise

Lexer Infrastructure (2)

- **nextElement** creates a LexerToken object, which is a String with line and column information, from the previously unread input
- Using the LexerToken object it creates a Token object which will determine and store its type (the Token class has a constructor which accepts a LexerToken)
- The Token object holds a reference to the LexerToken so it can print out the String read from the file with the line and column information; it also uses the String to determine its type

Lexer Infrastructure (3)

- Token contains an enum which will keep track of its type, this enum contains a constructor which accepts a String which will represent a regular expression (in the skeleton this is initially set to "<BLANK>")
- The constructor creates a regular expression pattern from the String which you provide
- To determine the type the Token constructor loops over every Type in order and attempts to match the regular expression using a case insenstive match
- On the first match it stops and sets the type to the regular expression it matched (note the order is important!)

Task 1

- Write the regular expressions for each token, most of them will simply be the keyword
- **Note:** If you want to use a \ character to escape a regular expression character you must first escape it in the Java string (you must also escape ")
- **Example:** If you want to match the + character you will need to escape it so the regular expression doesn't use it as an operator, which is \+ the corresponding Java String would be "\\+"
- You can test your regular expressions using the TokenDriver as the main class which expects a filename
- **java ca.uwaterloo.ece251.TokenDriver <filename>** in a terminal or in Eclipse to set the main class and argument

Lexer Example

Input

```
CREATE TABLE foo (name);
```

Output

```
t: [CREATE: ('CREATE'), line 1, col 0-6]
t: [TABLE: ('TABLE'), line 1, col 7-12]
t: [ID: ('foo'), line 1, col 13-16]
t: [LPAREN: ('('), line 1, col 17-17]
t: [ID: ('name'), line 1, col 18-21]
t: [RPAREN: (')'), line 1, col 22-22]
t: [SEMICOLON: (';'), line 1, col 23-23]
```

Parser Infrastructure

- The Parser class uses a TokenStream and parses the tokens using the **stmt()** method corresponding to the beginning production (like in Lab 1)
- This returns a Stmt which will be an instance of a class which extents Stmt (you'll see a clearer picture later)
- The XMLVisitor is responsable for printing the parse tree which is done after a successful parse
- **Note:** The visitor also starts printing using the starting production

AST Class Reference (1)

Legend: Abstract Class, Concrete Class

- ASTNode
  void accept(Visitor)
  - Stmt
    boolean explain
    boolean queryPlan
  - Expr
  - SingleSource
  - ResultColumn
  - IdExprPair
    IdExprPair(String id, Expr)
  - ColumnDef
    ColumnDef(String id, Type)
- Type (enum)
  NULL, INTEGER, REAL, TEXT, BLOB, UNKNOWN
- Literal
  - BooleanLiteral
    BooleanLiteral(String)
  - NumericLiteral
    NumericLiteral(String)
  - StringLiteral
    StringLiteral(String)
- QualifiedTableName
  QualifiedTableName(String db, String table)

AST Class Reference (2)

- Stmt
  boolean explain
  boolean queryPlan
  - CreateTableStmt
    CreateTableStmt(boolean temporary, boolean ifNotExists, QualifiedTableName, List<ColumnDef>, SelectStmt)
  - DropTableStmt
    DropTableStmt(boolean ifExists, QualifiedTableName)
  - InsertStmt
    InsertStmt(boolean insert, boolean replace, QualifiedTableName, List<String> columns, List<Expr> values, SelectStmt, boolean defaultValues)
  - SelectStmt
    SelectStmt(boolean distinct, boolean all, List<ResultColumn> cols, List<SingleSource> js, Expr where, Expr limit)
  - UpdateStmt
    UpdateStmt(QualifiedTableName, List<IdExprPair> sets, Expr where)
  - VacuumStmt
    VacuumStmt()

AST Class Reference (3)

- Expr
  - BinaryOpExpr
    static String PLUS, MINUS, TIMES, DIV, MOD, LT, LE, GT, GE, EQ
    BinaryOpExpr(String op, Expr left, Expr right)
  - LiteralExpr
    LiteralExpr(Literal value)
  - QualifiedTableNameExpr
    QualifiedTableNameExpr(QualifiedTableName tn)
  - UnaryOpExpr
    static String PLUS, MINUS, ISNULL, NOTNULL
    UnaryOpExpr(String op, Expr)

AST Class Reference (4)

- SingleSource
  - JoinSingleSource
    JoinSingleSource(List<SingleSource>)
  - QualifiedTableNameSingleSource
    QualifiedTableNameSingleSource(QualifiedTableName tn, String as)
  - SelectSingleSource
    SelectSingleSource(SelectStmt select, String as)
- ResultColumn
  - ExprResultColumn
    ExprResultColumn(Expr e)
  - StarResultColumn
    StarResultColumn(String table)

Task 2 (1)

- You will implement your recursive descent parser in *Parser.java*
- To test your parser use **ParserDriver** as your main class
- Similar to ANTLR each production should have its own method associated with it which returns an object representing the rule
- The grammar is in LL(1) form (left to right, left most derivation looking ahead 1 token) for the most part, you will have to eliminate left recursion and use precedence
- For methods which return a List<ClassName>, you can create a LinkedList<ClassName> and add objects using its **add(ClassName object)** method.

Task 2 (2)

- There are 3 helper functions in *Parser.java* for you to use: **accept(Token.Type)**, **consume(Token.Type)** and **syntax_error()**
- **accept** will return true if the next token matches the type you use an argument and false otherwise
- **consume** will check that the type you use an argument matches, if it does not it will generate a syntax error and quit the program, otherwise it will advance to the next token in the parser and return a reference to the original token matched
- **syntax_error** prints a message indicating a syntax error has occured along with the current token at the point of failure, it then exits the program
- **Remember:** Your Token class has a **getText()** method which returns the String of the actual token

Parser Example

Input

```
CREATE TABLE foo (name);
```

Output

```
<create_table tn='foo'>
 <column_def id=name type=UNKNOWN />
</create_table>
```

Flow of events

QualifiedTableName tn = new QualifiedTableName("foo");
List<ColumnDef> cdl = new LinkedList<ColumnDef>();
ColumnDef cd = new ColumnDef("name", Type.UNKNOWN);
cdl.add(cd);
Stmt s = new CreateTableStmt(false, false, tn, cdl, null);

Conclusion

- The overall program structure is almost identical to lab 1, minus the interpreter, just that you are writing the lexer and parser
- To complete the lexer you just have to fill in the token types with their corresponding regular expression
- For the parser you will implement your own recursive descent parser using the given helper functions
- Use the reference section of this tutorial to speed up the coding process