

## CMPSC 461: Programming Language Concepts

Spring 2017

Programming assignment 1: Recursive Descent Parsing

**Total: 20 points. Due on Feb 3rd at 12:20pm in Canvas.**

Your assignment is to use Java to write a recursive descent parser for a restricted form of SQL, the popular database query language. The lexical syntax is specified by regular expressions:

<u>Category</u>	<u>Definition</u>
digit	[0-9]
letter	[a-zA-Z]
int	<digit>+
float	<digit>+.<digit>+
id	<letter>(<letter>   <digit>)*
keyword	SELECT   FROM   WHERE   AND
operator	=   <   >
comma	,

The concrete syntax is specified by the following E-BNF grammar, where <Query> is the start symbol:

```
<Query> -> SELECT <IDList> FROM <IDList> [WHERE <CondList>]
<IDList> -> <id> {, <id>}
<CondList> -> <Cond> {AND <Cond>}
<Cond> -> <id> <operator> <Term>
<Term> -> <id> | <int> | <float>
```

Here <id>, <float>, and <operator> are token categories defined by the regular expressions above, and the terminal symbols SELECT, FROM, WHERE, and AND are of category keyword, while “,” is a terminal symbol with category <comma>. Note arbitrary whitespace can appear between tokens, and no space is needed between an operator and its operands, or between the items in a list and the commas that separate them.

This project is broken down into the following series of tasks.

1. (3 points) Write a class Token that can store the value and category of any token.
2. (7 points) Develop a lexical analyzer. You should start by drawing on paper a finite state automaton that can recognize types of tokens and

then convert the automaton into code. Name this class `Lexer` and ensure that it has a public method `nextToken()` which returns a `Token`. `Lexer` should have a constructor with the signature `Lexer(String input)`, where `input` is the string representing the query to be parsed. As mentioned before, the code for recognizing an identifier should check to see if the terminal string is a keyword, and if so, then the category of the token should be set to keyword, instead of `ID`. If `nextToken()` is called when no input is left, then it should return a token with category `EOI` (`EndOfInput`). If the next terminal string is not considered a token by the lexical syntax, then return a token with category `Invalid`.

3. (10 points) Write a syntactic analyzer which parses the tokens given by `Lexer` using the recursive descent technique. Name this class `Parser`. Like `Lexer`, `Parser` should have a public constructor with the signature `Parser(String input)`. This input string should be used to create a `Lexer` object. There should also be a public method `run()` that will start the parse. As you parse the input, `Parser` should output (by writing to `System.out`) the non-terminal that was matched, surrounded by angle-brackets, e.g. `<Query>`. Whenever it applies a new rule, it should indent by a tab. When it has parsed all tokens that constitute a nonterminal, it should print out the non-terminal's name, preceded by a slash, and surrounded by angle-brackets, e.g. `</Query>`. When a terminal symbol (token) is matched, it should output the token category, surrounded by angle-brackets, the tokens string, and the token category preceded by a slash and surrounded by angle-brackets (e.g. `<Int>42</Int>`). Whenever the parser comes across a token that does not fit the grammar, it should output a message of the form "Syntax error: expecting expected-token-category; saw token" and immediately exit. Note, the input string is expected to contain exactly one sentence of the form `Query`.

In order to test your file, you will have to create a `Test` class with a `main()` method that creates one or more `Parser` objects using different query strings and then calls the `run()` method on each object. For example, the code:

```
Parser p = new Parser("SELECT C1,C2 FROM T1 WHERE C1=5.23");
p.run();
```

should have the output given in Figure 1.

Although it is not important for completing this assignment, you may be interested to know that this output format is well-formed XML.

```

<Query>
  <Keyword>SELECT</Keyword>
  <IdList>
    <Id>C1</Id>
    <Comma>,</Comma>
    <Id>C2</Id>
  </IdList>
  <Keyword>FROM</Keyword>
  <IdList>
    <Id>T1</Id>
  </IdList>
  <Keyword>WHERE<Keyword>
  <CondList>
    <Cond>
      <Id>C1</Id>
      <Operator>=</Operator>
      <Term>
        <Float>5.23</Float>
      </Term>
    </Cond>
  </CondList>
</Query>

```

Figure 1: Sample output.

**Submission format:** The electronic version of your program files must be submitted through Canvas.

Make sure that your code can be compiled and run on those Linux machines in the IST 218 lab (p218inst09.cse.psu.edu to p218inst33.cse.psu.edu). We will grade your submission on those machines.

Please zip all your .java and .class files into one single file and submit the zipped file. Please ensure that each of your files has a comment that includes your name, Penn State network user id, and a description of the purpose of the file. Also say which version of the Java you are using (e.g., Java 1.7) and what operating system under which you compiled your program. Please include comments for each method, as well as any complicated logic within methods.