CS 316 Fall 2017
Observe **course policies** in undertaking this project.
All programs must be written in Oracle Standard Edition compliant Java or ANSI/ISO standard compliant C++.

PROJECT 2: Top-Down Parser
Due: 11/10/17, Friday, 11 PM
Late projects will not be accepted.

This project is a continuation of Project 1. You will implement a top-down parser for the following BNF for a functional language:

⟨fun def list⟩ → ⟨fun def⟩ | ⟨fun def⟩ ⟨fun def list⟩
⟨fun def⟩ → ⟨header⟩ = ⟨exp⟩
⟨header⟩ → ⟨type⟩ ⟨fun name⟩ ⟨parameter list⟩
⟨type⟩ → "int" | "float" | "boolean"
⟨fun name⟩ → ⟨id⟩
⟨parameter list⟩ → ε | "(" ⟨parameters⟩ ")"
⟨parameters⟩ → ⟨parameter⟩ | ⟨parameter⟩ "," ⟨parameters⟩
⟨parameter⟩ → ⟨type⟩ ⟨id⟩
⟨exp⟩ → ⟨id⟩ | ⟨int⟩ | ⟨float⟩ | ⟨floatE⟩ | ⟨boolLiteral⟩ | "(" ⟨fun exp⟩ ")" | "if" ⟨exp⟩ "then" ⟨exp⟩ "else" ⟨exp⟩
⟨boolLiteral⟩ → "false" | "true"
⟨fun exp⟩ → ⟨fun op⟩ ⟨exp list⟩
⟨exp list⟩ → ε | ⟨exp⟩ ⟨exp list⟩
⟨fun op⟩ → ⟨fun name⟩ | ⟨arith op⟩ | ⟨bool op⟩ | ⟨comp op⟩
⟨arith op⟩ → + | − | * | /
⟨bool op⟩ → "and" | "or" | "not"
⟨comp op⟩ → "<" | "<=" | ">" | ">=" | "="

NOTE: ε stands for the empty string.

This grammar defines all iterations by right-recursive production rules to make the construction of explicit parse trees relatively uniform and straightforward.

A lexical analyzer for this language has been implemented in Project 1.

Your program will read any text file that contains (what is intended to be) a string in the category ⟨fun def list⟩. It will then construct an **explicit parse tree** and display it in linearly indented form: each syntactic category name in the parse tree is displayed on a separate line, prefixed with the integer *i* representing its depth and indented by *i* blanks. This is a basic form of syntax profiler.

Explicit parse trees are to be constructed by class objects using the method described in lecture. Recall that in this method, for each production rule with alternatives:

$$\langle X \rangle \rightarrow \alpha_1 \mid \cdots \mid \alpha_n, \quad n \geq 2$$

the class for ⟨X⟩ is *abstract* and the classes for α_i are subclasses of it. Class fields represent sub parse trees.

The categories ⟨fun def list⟩, ⟨parameters⟩, ⟨exp list⟩ have a linear list structure defined by right-recursive production rules. All these can be represented by the same class schema pattern. For example, ⟨fun def list⟩ can be represented by an abstract class `FunDefList` together with two subclasses `FunDef` and `MultipleFunDef`. The class `MultipleFunDef` would have two components:

```
class MultipleFunDef extends FunDefList
{
        FunDef funDef;
        FunDefList funDefList;
}
```

Alternatively, they can be represented by more straightforward recursive, non-abstract classes. For example,

```
class FunDefList
{
        FunDef funDef;
```

```
        FunDefList funDefList;
}
```

The *null* value is used to represent the end of the list.

Parse trees should be displayed by functions in syntactic-category classes.

You may build your parser based on this sample parser; in this project, you may ignore the classes *Interpreter*, *Compiler*, *Val* and the functions *M*, *Eval*, *emitInstructions*.

An appropriate error message should be issued when the first syntax error is found; in this project there is no need to recover from it and continue parsing. (Real-world compilers do some type of syntax-error recovery and attempt to find more syntax errors.)

To make grading efficient and uniform, the program is to read the input/output file names as external arguments to the main function. How to set external arguments to Java main function in Eclipse.

If your Project 1 lexical analyzer wasn't correct, you may use this sample lexical analyzer: State.java, LexAnalyzer.java.

Here's a sample set of test input files:

in1 | out1
in2 | out2
in3 | out3
in4 | out4
in5 | out5
in6 | out6
in7 | out7
in8 | out8

You should make your own additional input files to test the program. Your outputs don't have to be identical to the samples, but they should display the parse tree structure clearly.

Submission

Email the following materials to keitaro.yukawa@gmail.com with the subject header:

CS 316, Project 2, your full name

- All the classes comprising your source code, including the lexical analyzer you used. Since there will be many classes, **make sure to double check no classes are missing in your submission**.
- A list of all class names arranged like this page. This may be in text, html, PDF, or WORD file.
- Concise instructions for how to compile and run your program.

You may email the entire materials in a .zip or .rar compressed file.

The due date is 11/10/17, Friday, 11 PM. No late projects will be accepted. If you haven't been able to complete the project, you may send an incomplete program for partial credit. In this case, include a description of what is and is not working in your program along with what you believe to be the sources of the problems.