CSE 340 Principles of Programming Languages
Fall 2014

Programming Assignment 2
Due on October 16, 2014 by 11:59 PM

Abstract

Create a syntax analyzer (parser) for the grammar listed below. You will be required to use the lexical analyzer developed in Assignment #1 as part of this project.

INSTRUCTIONS

1. Download and set up the source code published on Blackboard. The code includes:

   - Gui.java. It is an updated version of the GUI from assignment #1; this new version incorporates a new panel (to show the parser results) and changes the menu "Run|Lexer" to "Run|Compile".

   - Token.java. It is the same class that you had from Assignment #1.

   - Parser.java. It is the file in which you are going to develop your parser. This is the only file that you will be allowed to modify and the only one you will be submitting for your Assignment #2.

   - Lexer.jar. It is a fully functional Lexer that you can use if yours (from Assignment #1) is not complete. NO source code is included. This file should be added as a Library (jar file) in your project. You can disregard it if you prefer to use your own Lexer.

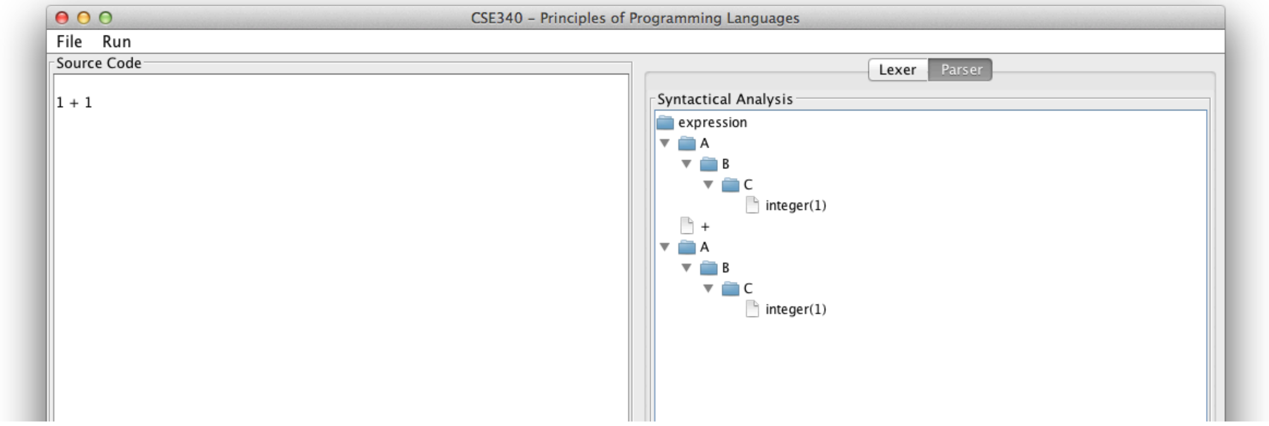
2. Compile, run, and play with the code. The code implements a parser for arithmetic expressions (one line length), such as:

   ```
   1 + 1

   (X) + (20) + (34)

   2 * (4 + Y) - 1 / (counter + i)
   ```

   Write these examples (one at a time) in the editor Panel, click "compile" and look into the "parser" tab. It will show the "parse tree" of the expression. The parser is able to create the parse trees for expressions that follow the rules that were described in lectures 8 and 9.

3. Review the code in the file Parser.java. Parser.java is the file in which you are going to be working for this assignment. Read the code there and understand how it works. A summary of what is implemented there is as follows: The method `run()` receives as a parameter the vector

of Tokens (that the Lexer created) and returns a parse tree (specifically, returns one node that is the root of the parse tree). This data structure (tree) is then visualized in the GUI inside the "Parser" panel, as shown below



```
1 + 1
```

```
expression
▼ A
    ▼ B
        ▼ C
            integer(1)
    +
▼ A
    ▼ B
        ▼ C
            integer(1)
```

4. Write a predictive descendent recursive parser for the grammar listed below. The code must be in the file "Parser.java". One method needs to be coded for each of the following rules:

```
<PROGRAM>     → '{' <BODY> '}'

<BODY>        →  {<PRINT>';'|<ASSIGNMENT>';'|<VARIABLE>';'|<WHILE>|<IF>|<RETURN>';'}

<ASSIGNMENT>  → identifier '=' <EXPRESSION>

<VARIABLE>    → ('int'|'float'|'boolean'|'char'|'string'|'void')identifier

<WHILE>       → 'while' '(' <EXPRESSION>  ')' <PROGRAM>

<IF>          → 'if' '(' <EXPRESSION>  ')' <PROGRAM> ['else' <PROGRAM>]

<RETURN>      → 'return'

<PRINT>       → 'print' '(' <EXPRESSION> ')'

<EXPRESSION>  → <X> {'|' <X>}

<X>           → <Y> {'&' <Y>}

<Y>           → ['!'] <R>

<R>           → <E> {('>'|'<'|'=='|'!=') <E>}

<E>           → <A> {('+'|'-') <A>}

<A>           → <B> {('*'|'/') <B>}

<B>           → ['-'] <C>

<C>           → integer | octal | hexadecimal | binary | true | false |
                string | char | float | identifier|'(' <EXPRESSION> ')'
```

Review lecture 12 slides for a description of the grammar as well as for a draft of the code to be implemented. Slides from Lecture 11 to 14 show several examples of inputs that can be used to test the parser.

5. The parser must report the errors and the line number in which they occur. The errors that the parser should be able to report are as follow:

   - `Line <n>: expected {`
   - `Line <n>: expected }`
   - `Line <n>: expected ;`
   - `Line <n>: expected identifier or keyword`
   - `Line <n>: expected =`
   - `Line <n>: expected identifier`
   - `Line <n>: expected )`
   - `Line <n>: expected (`
   - `Line <n>: expected value, identifier, (`

   The <n> represent the line number in which the error occurs. The process to implement error detection as well as examples of inputs / errors can be consulted in lecture 13 slides.

6. Include in your parser.java file the method `error(int err)` provided in lecture 13 slides. This method will print the error's messages. Do not change the messages.

7. Update your Gui.java and Parser.java as indicated in slide 21 of lecture 13 to allow your parser to write messages in the graphical user interface.

8. Implement error recovery as described in lecture 14.

9. Read and actively participate in the discussion board about common errors in grammar analysis and use those discussions to improve your implementation.

10. Create a zip file, using the following naming convention: **Firstname_Lastname_P2.zip**. This file should contain your source code for:

    Parser.java

    And should include a compiled version of your application:

    CSE340.jar.

GRADING

The assignment will be graded in a scale of 0-100 considering the following:

- Convention followed for java implementation: three classes (Lexer, Parser, Token); one method for each rule in Parser.

- All Arithmetic operators, and their precedence, are recognized. Rules E, A, B
- All Logic operators, and their precedence, are recognized. Rules EXPRESSION, X, Y.
- All Relation operators, and their precedence, are recognized. Rule R.
- Expressions with nested parenthesis are recognized. Rule C.
- PRINT and RETURN rules works properly.
- IF rule works properly.
- WHILE rule works properly.
- VARIABLE rule works properly.
- ASSIGNMENT rule works properly.
- BODY rule works properly.
- PROGRAM rule works properly.
- It recognizes ERROR: Line <n>: expected {
- It recognizes ERROR: Line <n>: expected }
- It recognizes ERROR: Line <n>: expected ;
- It recognizes ERROR: Line <n>: expected identifier or keyword
- It recognizes ERROR: Line <n>: expected =
- It recognizes ERROR: Line <n>: expected identifier
- It recognizes ERROR: Line <n>: expected)
- It recognizes ERROR: Line <n>: expected (
- It recognizes ERROR: Line <n>: expected value, identifier, (
- It correctly implements error recovery.

Be aware that:

- No credit will be given to programs that do not compile.
- No credit will be given if the project does not follow the format described above (input, output, number and structure of classes, attributes, and methods).

If you have any question, contact us (me or the TAs) to clarify them. And remember, actively participate in the discussion board.