**COP 5556 // Project #4 // Fall 2018**

|  |
| --- |
| **Date Assigned:** |
| **Date Due:** |

# Submission Format

You will submit a soft copy of your solution using e-Learning ( <http://elearning.ufl.edu> ) by the end of the day ( 23:59 / 11:59 PM ) on the assigned date ( **TBD** ). Save your solution as a **jar** file and name the file **p4** ( p4.jar ).

# Assignment

At the top of every solution file you submit this semester include: your name, the assignment number, and the date due. You will implement your solution by modifying the PLPParser class [located with PLPParser.java]. Any additional JUnit testing you would like to perform can be included in class PLPParserTest [see the PLPParserTest.java file]. Also, include in your p4.jar submission your implementation of the class PLPScanner [PLPScanner.java file] and the provided AST code.

To Do Before Starting P4

We have updated the PLPParser.java file to include a method *matchEOF*. You **must copy** this method into your PLPParser.java solution and **make a call** to it from the method parse(). You can see an implementation of the call on line #30 and the method itself on lines #86 through #91.

Now on to P4…

For this project, your PLPParser should return an instance of cop5556fa18.PLPAST.Program if the sentence is legal. If the sentence does not parse you will need throw a SyntaxException. (Your parser should throw an exception for exactly the same input as a correctly implemented p3 would.)

Code for all AST nodes and an interface called PLPASTVisitor has been provided. Do not modify these classes for this project. You will find that some of the AST nodes have synthesized attributes like name or value. These are obtained from the Scanner via the corresponding token. It is convenient for test cases to invoke some of the parser’s methods directly. As an example, one of the methods in the provided ParserTest.java directly invokes expression(). To ensure that all of our test work with your parser, **make sure that your Parser has the following methods** (with the indicated case-sensitive name and return type and that they are package visible (i.e. not private):

* Expression expression()
* Statement statement()
* Declaration declaration()

The abstract superclass of all of the abstract syntax tree nodes is PLPASTNode.java. It

contains a single field Token firstToken which must be provided in the constructor. This

is there to connect the AST with the program source for error messages later and should

be the first Token in the construct being parsed.

*Turn in p3.jar* file containing your source code for *PLPParser.java*, *PLPScanner.java*, and *PLPParserTest.java*. Also, include the source for the provided AST node classes so that your jar file is complete. [Note, this means you will have a package (directory) cop5556fa18 containing PLPParser.java, PLPScanner.java, PLPParserTest.java, as well as the package (directory) PLPAST. The package (directory) PLPAST will contain all of the AST classes: AssignmentStatement.java, Block.java, Declaration.java, and so on.] Your code must remain in package *cop5556fa18* [case sensitive]: **do not** create additional packages. Class names, method names, variables and so on in the starter code must not be changed. Unless otherwise specified, your code should not import any classes other than those from the standard Java distribution, your PLPScanner.java, and the provided cop5556fa18.PLPAST package.

All code, including the code for scanner must be your own work.

To ensure that we will be able to compile and run your submissions, test your code one of the UF CISE servers, like you did for the previous projects.

We suggest trying to work incrementally. Try testing by calling the routines corresponding to fragments of the grammar in JUnit tests. Also, it might be helpful to understand the structure of the provided code and how it relates to the syntax. You will need to look inside each class in order to see which fields it contains and what the constructor expects. If a field is optional in the syntax and is not provided in the input, you should set the corresponding field in the AST node to *null*. (The exception is the list of statements and declarations in Program. If there are no statements or declarations, the list should be empty, but not null. Note, this means there is a block, just not statements or declarations within that block.)

Here is the CFG from Project 3 and the related Abstract Syntax you will use to generate your Abstract Syntax Tree (AST).

|  |  |
| --- | --- |
| **CFG [Concrete Syntax]** | **Abstract Syntax** |
| Program Identifier Block | Program IDENTIFIER Block |
| Block ​{ ​( (Declaration | Statement) ​;​ )\* }​ | Block ( Declaration | Statement )\* |
|  |  |
| Declaration Type ​Identifier​ (= Expression | ) | Type IDENTIFIERLIST​ | Declaration VariableDeclaration | VariableListDeclaration |
|  | VariableDeclaration Type IDENTIFIER ( | Expression ) |
| IDENTIFIERLIST Identifier (, Identifier)\* | VariableListDeclaration Type IDENTIFIER IDENTIFIER + |
| Type int | float | boolean | char | string | Type int | float | boolean | char | string |
|  |  |
| Statement IfStatement | AssignmentStatement | SleepStatement | PrintStatement | WhileStatement | Statement IfStatement | AssignmentStatement | SleepStatement | PrintStatement | WhileStatement |
|  |  |
| IfStatement if ( Expression ) Block | IfStatement Expression Block |
| WhileStatement while ( Expression ) Block | WhileStatement Expression Block |
| AssignmentStatement Identifier = Expression | AssignmentStatement IDENTIFIER Expression |
| SleepStatement sleep Expression | SleepStatement Expression |
| PrintStatement print Expression | PrintStatement Expression |
| Expression OrExpression ​?​ Expression ​: Expression | OrExpression | ExpressionConditional Expression Expression Expression |
| OrExpression AndExpression ( ​​| AndExpression ​) \* | ExpressionBinary Expression op Expression |
| AndExpression EqExpression ( ​& EqExpression )\* | ExpressionBinary Expression op Expression |
| EqExpression RelExpression ( (​==​ | ​!=​ ) RelExpression )\* | ExpressionBinary Expression op Expression |
| RelExpression AddExpression ( (​<​ | ​>​ | ​<= | ​>=​ ) AddExpression)\* | ExpressionBinary Expression op Expression |
| AddExpression MultExpression ( ( ​+​ | ​-​ ) MultExpression )\* | ExpressionBinary Expression op Expression |
| MultExpression PowerExpression ( ( ​\*​ | ​/​ | %​ ) PowerExpression )\* | ExpressionBinary Expression op Expression |
| PowerExpression UnaryExpression (​\*\* PowerExpression | ε) | ExpressionBinary Expression op Expression |
| UnaryExpression ​+​ UnaryExpression | ​- UnaryExpression | ! UnaryExpression | Primary | ExpressionUnary Op Expression |
| Primary ​IDENTIFIER | ExpressionIdent |
| Primary ​INTEGER\_LITERAL | ExpressionIntegerLiteral |
| Primary ​BOOLEAN\_LITERAL | ExpressionBooleanLiteral |
| Primary ​FLOAT\_LITERAL | ExpressionFloatLiteral |
| Primary ​CHAR\_LITERAL | ExpressionCharLiteral |
| Primary ​STRING\_LITERAL | ExpressionStringLiteral |
| Primary ​(​ Expression​ ) ​| Function |  |
| Function ​FunctionName ( Expression ​) | FunctionWithArg FunctionName Expression |
| FunctionName ​sin​ | ​cos​ | ​atan​ | ​abs​ | ​log​ ​| ​int​ | ​float | FunctionName sin | cos | atan | abs | log | int | float |

In order to implement OrExpression, AndExpression, and so on, you will need to identify the first tokens in each expression. Here is an example of the analysis of OrExpression:

OrExpression first = t; leftExpression =​ AndExpression ( ​op =​ ​|​ ​rightExpression =​ AndExpression ​e0 = new ExpressionBinary(first, leftExpression,op,rightExpression)​ ) \* ​return leftExpression