**COP 5556 // Project #3 // Fall 2018**

|  |  |
| --- | --- |
| **Date Assigned:** | September 21, 2018 |
| **Date Due:** | October 5, 2018 |

# 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 ( October 5 ). Save your solution as a **jar** file and name the file **p3** ( p3.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 within the class PLPParser [located within the PLPParser.java file]. Any additional JUnit testing you would like to perform can be included in class PLPParserTest [see the PLPParserTest.java file]. Also, include in your p3.jar submission your implementation of the class PLPScanner [PLPScanner.java file].

Use the class frameworks provided in PLPParser.java and PLPParserTest.java as well as your P2 solution as a starting point. PLPParser will determine whether a given sentence is legal or not. If the sentence is legal, you will simply return from the parsing process [just return from the method *parse*] and if it does not parse, a SyntaxException should be thrown. *Note*: a completed parser should not throw an UnsupportedOperationException.

The code provided in the distributed p3.jar compiles, and two of the three test cases pass. All three cases will pass when the parser has been implemented correctly. It is recommended you create additional JUnit test cases to evaluate your parser.

*Turn in p3.jar* file containing your source code for *PLPParser.java*, *PLPScanner.java*, and *PLPParserTest.java*. 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 or your PLPScanner.java.

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 Project #2.

We suggest trying to work incrementally. It is possible to call the routines corresponding to a certain fragment of the grammar in JUnit tests. For example, if you have a method named *statement*, you could call that directly, parsing input that satisfies this specific grammar fragment. Descriptive error messages that output the location of an offending token might also be helpful.

# Description

Enhance your implementation of PLPScanner.java to tokenize the following additional symbols. The combined lexical analysis of P2 and P3 is provided in the next section. Symbols corresponding to these terminals have been added to the enum Kind defined in PLPScanner.java found within the p3.jar distribution. The productions updated and additional terminals are:

* Operators ? | : | | | & | !=
* Keywords while | sin | cos | atan | abs | log

Next, implement a parser for the language specified by the context-free grammar given below.

We recommend using the approach described in the lecture to build the parser [see Parsing2.pptx]. Recall, when a grammar does not satisfy LL(1), you will need to be transform the grammar or use an *ad hoc* solution.

# Lexical Analysis

InputCharacter any 7-bit ASCII character.

LineTerminator LF | CR | CR LF

*LF is the ASCII character also known as “newline”. The Java character literal is ‘\n’.*

*CR is the ASII character also known as “return”, The Java character literal is ‘\r’.  
CR immediately followed by LF counts as one line terminator, not two.*

Input (WhisteSpace | Comment | Token)\*

WhiteSpace SP | HT | FF | LineTerminator

*SP is the ASCII character also known as “space”. The Java char literal is ‘ ‘.*

*HT is the ASCII character also known as “horizontal tab”. The Java char literal is ‘\t’.*

*FF is the ASCII character also known as “form feed”. The Java char literal is ‘\f’.*

Comment %{ ( (% NOT({) ) | NOT(%) )\* %+}

Comments will be identified and discarded [you do not need to keep them as a token].

Token Identifier | Keyword | Literal | Separator | Operator

Identifier IdentifierChar but not a keyword or a Boolean literal

IdentifierChar IdentifierStart IdentifierPart\*

IdentifierStart UnderScoreStart IdentifierStart |A..Z IdentifierPart | a..z IdentifierPart

UnderScoreStart \_ UnderScoreStart | \_

IdentifierPart A..Z IdentifierPart | a..z IdentifierPart | Digit IdentifierPart | \_ IdentifierPart |

Literal IntegerLiteral | FloatingPointLiteral | BooleanLiteral | StringLiteral | CharLiteral

IntegerLiteral 0 | NonZeroDigit Digit\*

FloatingPointLiteral IntegerLiteral . Digit Digit\*

StringLiteral " ASCII\* "

CharLiteral ' ASCII '

ASCII ASCII\_CHAR |

NoneZeroDigit 1 .. 9

Digit NonZeroDigit | 0

BooleanLiteral true | false

Separators ( | ) | [ | ] | ; | , | { | }

Operators < | > | <= | >= | - | + | \* | / | % | ! | \*\* | == | = | ? | : | | | & | !=

Keywords print | int | float | boolean | char | string | sleep | if | while | sin | cos | atan | abs | log

# Context Free Grammar

The following grammar extends the analysis you performed in P1 and defines the context free grammar for P3. *Note*: the terminal symbols are in red and are given by their text values, not each token's Kind value. For example, the grammar uses if and not KW\_if. In order to work with the tokens your scanner generates, your parser will use KW\_if [the token's Kind value].

The grammar is structured using *iterative* replacement. Doing so will assist us in future projects when we build the abstract syntax tree [AST] and use the tree to generate executable code. Recall this means a flatter tree will be built and we will be able to match on the standard left-to-right order of operator precedence. Here are the productions in our grammar:

Program Identifier Block

Block { ( (Declaration | Statement) ; )\* }

Declaration Type Identifier ( = Expression | ) | Type IDENTIFIERLIST

IDENTIFIERLIST Identifier (, Identifier)\*

Type int | float | boolean | char | string

Statement IfStatement | AssignmentStatement | SleepStatement

| PrintStatement | WhileStatment

IfStatement if ( Expression ) Block

WhileStatement while ( Expression ) Block

AssignmentStatement Identifier = Expression

SleepStatement sleep Expression

PrintStatement print Expression

Expression OrExpression ? Expression : Expression | OrExpression

OrExpression AndExpression ( | AndExpression )\*

AndExpression EqExpression ( & EqExpression )\*

EqExpression RelExpression ( ( == | != ) RelExpression )\*

RelExpression AddExpression ( ( < | > | <= | >= ) AddExpression )\*

AddExpression MultExpression ( ( + | - ) MultExpression )\*

MultExpression PowerExpression ( ( \* | / | % ) PowerExpression )\*

PowerExpression UnaryExpression ( \*\* PowerExpression | )

UnaryExpression + UnaryExpression | - UnaryExpression | ! UnaryExpression | Primary

Primary INTEGER\_LITERAL | BOOLEAN\_LITERAL ​| ​FLOAT\_LITERAL | CHAR\_LITERAL

​| ​STRING\_LITERAL | ( Expression ) | IDENTIFIER | Function

Function FunctionName ( Expression )

FunctionName sin | cos | atan | abs | log | int | float