**COP 5556 // Project #5 // Fall 2018**

|  |  |
| --- | --- |
| **Date Assigned:** | October 22, 2018 |
| **Date Due:** | November 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 ( **November 5** ). Save your solution as a **jar** file and name the file **p5** ( p5.jar ).

# Assignment

At the top of every solution file you submit this semester include: your name, the assignment number, and the date due. PLPTypeChecker.java, PLPTypeCheckerTest.java and PLPTypes.java has been provided. You will need to complete the implementation of PLPTypeChecker.java. Any additional JUnit testing you would like to perform can be included in PLPTypeCheckerTest.java.

You will also need to implement a data structure for your symbol table. An implementation of Leblanc-Cook symbol table (refer to our class discussion and the supplement in Files/notes/scott\_3\_4\_leblanc\_cook.pdf on Canvas). The specification below assumes that your symbol table has a method *lookup* that will return a Declaration if an identifier has been declared and is visible in the current scope. Otherwise, it will return null.

In the specification, for convenience, symbolTable is treated as a global attribute rather than being redefined everywhere as an inherited attribute. This can be directly implemented by making a reference to symbolTable as field in your ASTVisitor. You will need to design and implement an appropriate class structure for your symbol table representation.

Some of the AST nodes are already decorated with attribute values (name, value, etc) when the AST was constructed in p4. In this project, **type** and **dec** attributes need to be added for some nodes. If an attribute is a type, its declared type should be a value from the enum PLPTypes.Type. A dec attribute should be a Declaration object.

We have provided a utility class PLPTypes that contains an enum Type. Do not change the names in the enum or reorder them. You should not need to modify PLPTypes.java for this project.

If a type error is discovered, throw a SemanticException. The Token argument should be the first Token of the AST node where the error was detected. As in previous projects, the content of the error messages will not be graded but having a descriptive and helpful error message will be beneficial to you as you debug your code.

Whenever possible, fields to represent attributes should be declared in abstract classes so they will be inherited by all subclasses and can be accessed without needing a cast. For example, put the *type* attribute in Expression where it will be inherited by all the concrete expression classes. It is often convenient to return attributes from the visit methods where they were computed. Note, this is the case for the *type* of expression.

PLPTypeCheckerTest.java provides a few JUnit tests to illustrate how the pieces fit together. Currently, all three tests fail due to an UnsupportedOperationException. All tests should pass once you are finished.

In order for us to complete type checking, the AST needs to record the type of each variable. Therefore, we have updated the AST for assignment statements to keep track. You will note, there is a new field in the Abstract Syntax table. The full table is below, however here we highlight these specific additions for your reference:

|  |  |
| --- | --- |
| **Abstract Syntax [additions]** | **Semantic Rules and Conditions** |
| AssignmentStatement LHS Expression | LHS.type == Expression.type |
| LHS IDENTIFIER | LHS.name ← IDENTIFIER.name  LHS.dec ←SymbolTable.lookup(LHS.name)  LHS.dec != null  LHS.type ← LHS.dec.type |

p5.jar also the AST package with changes that accommodate the above. Please import this package before you start working on your solution.

Submission

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

Submit a jar file containing your source code for:

* PLPTypeChecker.java
* PLPTypeCheckerTest.java
* PLPParser.java
* PLPScanner.java
* all of the AST classes
* PLPTypes.java
* any classes you may have added

Make sure your symbol table class implementation is also included.

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 previous projects. We suggest trying to work incrementally and reviewing the lecture on symbol table before you begin.

|  |  |
| --- | --- |
| **Abstract Syntax** | **Semantic rules and conditions** |
| Program IDENTIFIER Block |  |
| Block ( Declaration | Statement )\* | Block enterScope  ( Declaration | Statement )\* leaveScope |
|  |  |
| Declaration VariableDeclaration | VariableListDeclaration |  |
| VariableDeclaration Type IDENTIFIER ( | Expression ) | Declaration.name ← IDENTIFIER.name  Declaration.name SymbolTable.currentScope  Expression == ε   or Expression.type == type  SymbolTable ←  SymbolTable ∪ (name, Declaration) |
| VariableListDeclaration Type IDENTIFIER IDENTIFIER + | for each IDENTIFIER do  Declaration.name ← IDENTIFIER.name  Declaration.name SymbolTable.currentScope  SymbolTable ←  SymbolTable ∪ (name, Declaration)  end |
|  |  |
| Type int | float | boolean | char | string |  |
|  |  |
|  |  |
| Statement IfStatement | AssignmentStatement | SleepStatement | PrintStatement | WhileStatement |  |
|  |  |
|  |  |
| AssignmentStatement LHS Expression | LHS.type == Expression.type |
| WhileStatement Expression Block | Expression.type == boolean |
| IfStatement Expression Block | Expression.type == boolean |
| PrintStatement Expression | Expression.type ∈ {int, boolean, float, char, string} |
| SleepStatement Expression | Expression.type == integer |
| LHS IDENTIFIER | LHS.name ← IDENTIFIER.name  LHS.dec ←SymbolTable.lookup(LHS.name)  LHS.dec != null  LHS.type ← LHS.dec.type |
| Expression ExpressionBinary | ExpressionConditional | FunctionWithArg | ExpressionUnary | ExpressionIdent | ExpressionIntegerLiteral | ExpressionBooleanLiteral | ExpressionFloatLiteral | ExpressionCharLiteral | ExpressionStringLiteral | Expression.type ← type of right-hand side expression |
| ExpressionConditional Expression0 Expression1 Expression2 | Expression0 .type == boolean  Expression1.type == Expression2 .type  ExpressionConditional.type ←  Expression1.type |
| ExpressionBinary Expression0 op Expression1 | ExpressionBinary.type ← inferredType(Expression0.type, Expression1.type, op)  (inferredType is defined below) |
| ExpressionUnary Op Expression | ExpressionUnary.type ← Expression.type  ! can be used only if Expression.type is integer or Boolean  +, - can be used only if Expression.type is integer or float. |
| ExpressionIdent | ExpressionIdent.dec ← SymbolTable.lookup(ExpressionIdent.name)  ExpressionIdent.dec != null  ExpressionIdent.type ← ExpressionIdent.dec.type |
| ExpressionIntegerLiteral | ExpressionIntegerLiteral.type ← integer |
| ExpressionBooleanLiteral | ExpressionBooleanLiteral.type ← boolean |
| ExpressionFloatLiteral | ExpressionFloatLiteral.type ← float |
| ExpressionCharLiteral | ExpressionCharLiteral.type ← char |
| ExpressionStringLiteral | ExpressionStringLiteral.type ← string |
|  |  |
| FunctionWithArg  FunctionName Expression | FunctionWithArg.type ← inferredTypeFunctionWithArg (FunctionName, Expression.type)  (see below) |
| FunctionName sin | cos | atan | abs | log | int | float |  |

The tables below provide the legal argument types for operators and functions along with the type of the result which is the inferred type. Any combination that is not in the table is not legal.

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression0.type** | **Expression1.type** | **Operator** | **inferred type for ExpressionBinary.type** |
| integer | integer | +,-,\*,/,%,\*\*, &, | | integer |
| float | float | +,-,\*,/,\*\* | float |
| float | integer | +,-,\*,/,\*\* | float |
| integer | float | +,-,\*,/,\*\* | float |
| string | string | + | string |
| boolean | boolean | &, | | boolean |
| integer | integer | &, | | integer |
| integer | integer | ==, !=, >,>=, <, <= | boolean |
| float | float | ==, !=, >,>=, <, <= | boolean |
| boolean | boolean | ==, !=, >,>=, <, <= | boolean |

|  |  |  |
| --- | --- | --- |
| **Expression.type** | **Function** | **inferred type for FunctionWithArg** |
| integer | abs | integer |
| float | abs, sin, cos, atan, log | float |
| int | float | float |
| float | float | float |
| float | int | int |
| int | int | int |