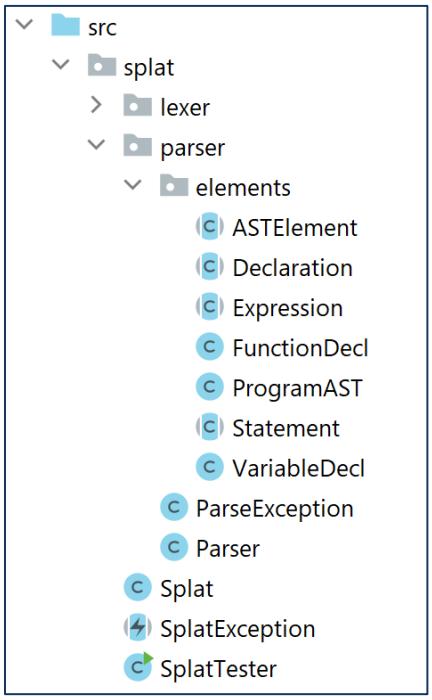
Phase 2 – SPLAT Parser

Target Date: Wednesday, November 5

PHASE 2 SETUP

Use the same Java project from the previous task, but you will be adding in two additional packages under **src**, specifically **splat.parser**, and its subpackage **splat.parser.elements**. These packages, along with several starter classes, are provided in the **src** folder of the parse_start.zip archive file. Make sure that when you copy things over, the parser package folder is in the splat package folder to preserve the proper package structure. After this step, things should look something like what is shown below:



TASK DESCRIPTION

For this task, you need to create the parser component for SPLAT, which takes a list of Tokens (generated by your lexer), and structures them into an Abstract Syntax Tree (AST). Specifically, the primary parse() method of the Parser class should return a ProgramAST object if parsing is successful.

Aside from uncommenting the two lines of code under "Step 2. Parse" in Splat.java, you do not need to make any other changes to the classes in the base package **splat,** and you shouldn't have to make any more changes in your **splat.lexer** classes either if your Lexer works properly from the previous phase.

Here, we will briefly go over the newly provided .java files that are involved in the parsing task.

The **splat.parser** package contains the main classes that are involved in parsing.

- **Parser.java** – Much of your work for Phase 2 will focus on implementing parsing methods in this class. The constructor takes a list of tokens (which should have previously been produced by a lexer), and then will proceed to build up AST structures representing the language elements as they are processed.

  The approach that we will use here is called recursive descent parsing, which essentially uses a single method for each grammar rule to generate a particular language element AST. Each parse method will read and remove tokens from the front of the token list that should represent *terminal symbols* given in the grammar rule (e.g., keywords, literals, labels, symbols, etc.), and then recursively call other parse methods to handle the *non-terminal symbols* in the rule. The top-level parse() method that returns a ProgramAST, which has been implemented for you, is a good example of how this works.

  In several cases, we may not automatically know which grammar rule, and thus what parse method, to call. This requires that we "look ahead" into the token list to see what we need to process next. At most, our grammar requires "two-symbol look ahead" to determine which grammar rule (and thus what parser method) needs to be applied. To help with this, the methods peekNext and peekTwoAhead are provided. (See the provided implementation for parseDecl() to see how this works.) In cases where we absolutely know what the next token should be, we call checkNext which "eats" the front-most token from the token list, and throws a ParseException if it isn't what it should be.

  To complete this task, you will have to add numerous parsing methods to Parse.java to take care of each of the different grammar rules of the language, and build up appropriate AST structures along the way. As part of this, you will also have to add many new AST classes to represent the elements of the language.

- **ParseException.java** – The specific type of exception that should be thrown by the Parser when something goes wrong there. As with LexException.java, nothing needs to be added or changed with this class.

The **splat.parser.elements** package should eventually contain all of the AST classes that represent elements of SPLAT programs. For this task, you will be adding many more AST classes to this package. Here are some starter classes that are provided:

- **ASTElement.java** – All AST elements that are used to build up your Abstract Syntax Trees will be a subclass of this abstract class. The only functionality that really provides is the ability to store the line and column number of any program element – these fields and getters should be inherited by all of the AST classes in this package.

- **ProgramAST.java** – This class is used to represent an entire SPLAT program, and is what is returned by the main parse method in Parser.java. You shouldn't have to change this class for Phase 2.

- **Statement.java** – This abstract class should be the parent class of all statement ASTs in your program (e.g., assignments, if-then-else's, while loops, print statements, etc.). When you create a new statement class, such as maybe IfThenElse, be sure that it extends Statement – without this, you won't be able to use polymorphism to properly use a List<Statement> for storing function or program bodies. Furthermore, there are two methods there that are currently commented out that you will need for future tasks… just be aware that they are there for now.

- **Expression.java** – This abstract class should be the parent class of all expression ASTs in your program (e.g., binary operation expressions, non-void function calls, etc.) When you create a new expression class, such as maybe NonVoidFunctionCall, be sure that it extends Expression. As with statements, this allows you to leverage polymorphism so you can do things like use any sort of expression as arguments to a binary operation (itself an expression). As with Statement.java, there are also two methods that are currently commented out that you will need to implement for all expression classes in Phases 3 and 4.

- **Declaration.java** – This abstract class acts as a super-type for FunctionDecl and VariableDecl. Without this class, we wouldn't be able to bundle together all of our program declarations into a single List<Declaration> item. You shouldn't have to change anything in this class.

- **FunctionDecl.java** and **VariableDecl.java** – These two subclass stubs of Declaration are used to represent function and variable declarations. You will need to add several things to each of these classes, as they contain nothing at this point.

Here is a UML class diagram that should give you an idea about the interrelationships between these classes, as well as additional AST classes that you might want to create for this task. There is no concrete correct number of classes you must create – this is totally up to you - your way of thinking, your imagination and creativity. The given classes that extend "Statement" and "Expression" just give you a hint of what should be implemented.