Final Project – General Description

The Task: Implement the project which is able to compile and run SPLAT language!

For your final project, you are asked to implement Basic SPLAT (Software Principles Language And Tools), which is a simple programming language and associated tools for parsing, typechecking, and interpretive execution of the language. The actual grammar and semantics of the Basic SPLAT language are provided in separate documents. The main components that you will need to implement for the project are as follows:

1. **Lexer/Tokenizer** – This takes a single SPLAT program file, and performs lexical processing to produce the List of Tokens that make up the program. Tokens should contain location information—line and column number, to indicate the character in the file where the token began.

2. **Parser** – This takes the Tokens in the Token List produced by the Lexer, and uses them to produce the abstract syntax trees (ASTs) which represent the structures of the language. The Basic SPLAT grammar is simple enough that you can use a recursive descent parsing approach to do this (more on this later).

   When developing your parser, you will have to create numerous classes to represent each of the major constructs in the language, including functions, variable declarations, assignments, if-statements, etc. You will also want to create Java interfaces for Statements and Expressions… you will understand why this is critical later.

3. **Semantic Analyzer (including Typechecking)** – During this phase, you will need to make sure your program (which should be structurally correct if you got to this point) is indeed valid, and follows all of the rules and restrictions in its definition and use of functions, variables, types, etc. It is during this phase that typechecking and other checks to make sure that referenced labels have actually been defined are performed. The semantic rules for the language can be found in the semantics.doc document.

   The semantic analyzer should take the ASTs generated by the parser in the previous step as input, and can work through the entire structure of the program recursively. Keep in mind that for any place that we are analyzing in the program, we will need a symbol table (i.e., a list of variables and/or parameters and their types) for the current scope, along with the list of function signatures for the whole program. Again, more on this later.

4. **Executor (Interpreter version, for CSCI 501 students)** – This component will run the actual program by essentially executing the individual program statements in the program body ASTs, one-by-one, and updating the state map(s) -- i.e., the values currently stored in the variables and/or parameters for the current scope. Perhaps the most difficult part of this component is in dealing with function calls, which require a new local state map to be created for each call, as well as a way to "return" the value back to the calling code.

4a. **Executor (Compiler version, for CSCI 701 students)** – An alternative to the interpreted approach for executing SPLAT programs is to translate it down into equivalent assembly code, and running it. For this, you will be targeting the MIPS assembly language, and running it on a MIPS interpreter. The challenge is to translate those language elements represented in the AST into sequences of MIPS commands, simulating function calls and returns, while keeping in mind that you have a limited number of registers to work with. The end result will be a true compiler for the SPLAT language!