LPL Compiler Explanation

This document explains the structure and functionality of a simple compiler written in Java for a hypothetical programming language. The compiler is organized into several classes, each responsible for a specific part of the compilation process.

# Token and TokenType

The 'Token' class represents the smallest unit of meaning within the source code, such as keywords, identifiers, literals, operators, and punctuation. Each 'Token' has a 'TokenType' (an enum) indicating its category, and a 'value' representing the actual text of the token.  
  
The 'TokenType' enum defines the possible categories of tokens recognized by the compiler, such as ID for identifiers, INTLIT for integer literals, OPERATOR for operators, KEYWORD for language keywords, and PUNCTUATION for characters like parentheses and semicolons.

# Node Classes

The abstract 'Node' class is the base for all nodes in the abstract syntax tree (AST), representing different constructs in the source code. Specific node classes inherit from 'Node', including 'ProgramNode' for the entire program, 'FunctionDefinitionNode' for function definitions, various 'StatementNode' subclasses for statements like 'IfStatementNode', and 'ExpressionNode' subclasses for expressions.  
  
Each 'StatementNode' and 'ExpressionNode' subclass implements a 'compile' method, which generates the corresponding Java code for the construct.

# Lexer (LPLLexer)

The 'LPLLexer' class is responsible for lexical analysis, converting the input source code into a list of 'Token' objects. It scans the input string, identifying tokens based on patterns like digits for integers, letters for identifiers or keywords, and specific characters for operators and punctuation.  
  
It uses regular expressions to match keywords and operators, and it carefully handles edge cases, such as distinguishing between an operator and punctuation.

# Parser (LPLParser)

The 'LPLParser' class performs syntactic analysis, transforming the list of 'Token' objects produced by the lexer into an AST. It implements a recursive descent parser that checks the sequence of tokens against the grammar of the language, constructing nodes for different constructs and adding them to the AST.  
  
It includes methods for parsing different constructs, like 'parseProgram' for the entire program, 'parseFunctionDefinition' for function definitions, and 'parseStatement' and 'parseExpression' for statements and expressions, respectively.

# Compiler (LPLCompiler)

The 'LPLCompiler' class contains the 'main' method, serving as the entry point for the compilation process. It instantiates the lexer and parser with the input source code, generates the AST, and then traverses this tree to produce the final Java code.  
  
The 'compileProgram' method and other related methods recursively visit each node in the AST, generating Java code that mirrors the functionality of the original LPL program. This approach allows the compiler to support various constructs of the LPL language by directly mapping them to their Java equivalents.