Jack Grammar and Module APIs

This document outlines the grammar for the Jack programming language and the APIs for the Jack Tokenizer and Compilation Engine modules.

Jack Grammar

Lexical Elements

- **keyword**: class | constructor | function | method | field | static | var | int | char | boolean | void | true | false | null | this | let | do | if | else | while | return
- **symbol**: { | } | (|) | [|] | . | , | ; | + | - | * | / | & | | | < | > | = | ~
- **integerConstant**: A decimal number in the range 0 to 32767.
- **stringConstant**: A sequence of Unicode characters not including a double quote or a newline.
- **identifier**: A sequence of letters, digits, and underscore (_) that does not start with a digit.

Program Structure

- **class**: 'class' className '{' classVarDec* subroutineDec* '}'
- **classVarDec**: ('static' | 'field') type varName (, varName)* ';'
- **type**: 'int' | 'char' | 'boolean' | className
- **subroutineDec**: ('constructor' | 'function' | 'method') ('void' | type) subroutineName '(' parameterList ')' subroutineBody
- **parameterList**: ((type varName) (, type varName)*)?
- **subroutineBody**: '{' varDec* statements '}'
- **varDec**: 'var' type varName (, varName)* ';'
- **className**: identifier
- **subroutineName**: identifier
- **varName**: identifier

Statements

- **statements**: statement*
- **statement**: letStatement | ifStatement | whileStatement | doStatement | returnStatement
- **letStatement**: 'let' varName ('[' expression ']')? '=' expression ';'
- **ifStatement**: 'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')?
- **whileStatement**: 'while' '(' expression ')' '{' statements '}'
- **doStatement**: 'do' subroutineCall ';'
- **returnStatement**: 'return' expression? ';'

Expressions

- **expression**: term (op term)*
- **term**: integerConstant | stringConstant | keywordConstant | varName | varName '[' expression ']' | subroutineCall | '(' expression ')' | unaryOp term

- **subroutineCall**: subroutineName '(' expressionList ')' | (className | varName) '.' subroutineName '(' expressionList ')'
- **expressionList**: (expression (, expression)*)?
- **op**: + | - | * | / | & | | | < | > | =
- **unaryOp**: - | ~
- **keywordConstant**: 'true' | 'false' | 'null' | 'this'

Module APIs

The JackTokenizer Module

This module removes all comments and white space from the input stream and breaks it into Jack-language tokens, as specified by the Jack grammar.

Routine	Arguments	Returns	Function
Constructor	input file/stream	—	Opens the input file/stream and gets ready to tokenize it.
hasMoreTokens	—	Boolean	Checks if there are more tokens in the input.
advance	—	—	Gets the next token from the input and makes it the current token.
tokenType	—	KEYWORD, SYMBOL, etc.	Returns the type of the current token.
keyword	—	CLASS, METHOD, etc.	Returns the keyword which is the current

			token.
symbol	—	`Char`	Returns the character which is the current token.
identifier	—	`String`	Returns the identifier which is the current token.
intVal	—	`Int`	Returns the integer value of the current token.
stringVal	—	`String`	Returns the string value of the current token.

The CompilationEngine Module

This module takes input from a `JackTokenizer` and emits the parsed structure into an output file/stream, effectively handling the compilation.

Routine	Arguments	Returns	Function
Constructor	`Input stream/file`, `Output stream/file`	—	Creates a new compilation engine.

CompileClass	—	—	Compiles a complete class.
CompileClassVarDec	—	—	Compiles a static or field declaration.
CompileSubroutine	—	—	Compiles a complete method, function, or constructor.
compileParameterList	—	—	Compiles a parameter list.
compileVarDec	—	—	Compiles a `var` declaration.
compileStatements	—	—	Compiles a sequence of statements.
compileDo	—	—	Compiles a do statement.
compileLet	—	—	Compiles a let statement.
compileWhile	—	—	Compiles a while statement.

compileReturn	—	—	Compiles a return statement.
compileIf	—	—	Compiles an if statement.
CompileExpression	—	—	Compiles an expression.
CompileTerm	—	—	Compiles a term.
CompileExpressionList	—	—	Compiles a comma-separated list of expressions.

The SymbolTable Module

Provides a symbol table abstraction, associating identifier names with their properties (type, kind, and index). It manages class and subroutine scopes.

Routine	Arguments	Returns	Function
Constructor	—	—	Creates a new empty symbol table.
startSubroutine	—	—	Starts a new subroutine scope, resetting the

			existing one.
Define	`name (String)`, `type (String)`, `kind (STATIC, FIELD, ARG, or VAR)`	—	Defines a new identifier with the given properties.
VarCount	`kind (STATIC, FIELD, ARG, or VAR)`	`int`	Returns the number of variables of the given kind in the current scope.
KindOf	`name (String)`	`(STATIC, FIELD, ARG, VAR, NONE)`	Returns the kind of the named identifier in the current scope.
TypeOf	`name (String)`	`String`	Returns the type of the named identifier.
IndexOf	`name (String)`	`int`	Returns the index assigned to the named identifier.

The VMWriter Module

Emits VM commands into an output file using the standard VM command syntax.

Routine	Arguments	Returns	Function
Constructor	Output file/stream	—	Creates a new file and prepares it for writing.
writePush	segment (CONST, ARG, etc.), index (int)	—	Writes a VM push command.
writePop	segment (CONST, ARG, etc.), index (int)	—	Writes a VM pop command.
WriteArithmetic	command (ADD, SUB, etc.)	—	Writes a VM arithmetic command.
WriteLabel	label (String)	—	Writes a VM label command.
WriteGoto	label (String)	—	Writes a VM goto command.
WriteIf	label (String)	—	Writes a VM if-goto command.
writeCall	name (String), nArgs (int)	—	Writes a VM call command.
writeFunction	name (String), nLocals	—	Writes a VM function

	(int)		command.
writeReturn	—	—	Writes a VM `return` command.
close	—	—	Closes the output file.