SOFTWARE DESIGN DOCUMENT

MICRO-PASCAL COMPILER

Version 1.0

William Mork

Augsburg University

PROJECT OVERVIEW

Introduction

This project, written in Java 8, compiles Micro-Pascal code to generate MIPS assembly code. The project was developed in six independent modules, and are more thoroughly described following the introduction. Please refer to the “Project Structure” section for clarity on the files and modules used in the project. The User Manual provides brief instruction on how to run the compiler.

Module 1: Scanner

The scanner module reads a Micro-Pascal text file and scans each line. Keywords and symbols which are recognized as valid (listed below) by the scanner are converted into “tokens”, which are handled by the parser module.

Scanner.java is a file which has been generated by JFlex, a lexical analyzer (scanner) generator. The generator uses a specified set of token types, expected patterns, and lexical rules to create a deterministic finite automata (DFA) which is used to construct the aforementioned token stream.

Token.java defines a token object containing the token lexeme and type.

TokenType.java enumerates the list of valid keywords and symbols.

Valid keywords:

AND ARRAY BEGIN DIV DO ELSE END FUNCTION IF INTEGER MOD NOT OF OR PROCEDURE PROGRAM REAL THEN VAR WHILE READ WRITE RETURN

Valid symbols (token type is listed first, followed by the symbol itself):

SEMI ; COMMA , PERIOD . COLON : LBRACE [RBRACE] LPAREN (RPAREN) PLUS + MINUS - EQUAL = NOTEQ <> LTHAN < LTHANEQ <= GTHAN > GTHANEQ >= ASTERISK * FSLASH / ASSIGN :=

Module 2: Parser

The parser module employs an instance of the scanner class to iterate through tokens of an input stream and match them with the production rules articulated in the Micro-Pascal grammar (Grammar.pdf).

To use the current version of the parser class, first create an instance within CompilerMain.java using either the constructor structure *([filename], true)* or *([input string], false)*. Then, call the top-level function *program()*; if the function returns without encountering an error, the input file or string was successfully parsed as a valid micro-Pascal program.

The Parser class can be used to read the token stream of a pascal file or a provided input String.

Module 3: Symbol Table

The symbol table module is used to store information on the identifiers used in a pascal program. As of version 1.0, the symbol table is unable to fully handle functions and procedures and will instead only store their respective lexemes to the symbol table. In a later build of this project, the symbol table will be able to manage subprogram calls through the use of multiple hashmaps, stored in a stack, with the global scope (program) being pushed onto the stack first.

SymbolTable.java contains a single hashmap, whereby each key-value pair contains the lexeme of an identifier (key) as well as information appropriate to the "kind" of identifier recognized by the parser. A full list of the symbols contained in the hashmap can be generated by calling the toString() method of this class.

Symbol.java defines a symbol object, with discrete constructors for every "kind" of identifier that the parser might encounter. The toString() method of this class can be called to generate a short descriptor of the symbol's information.

Kind.java enumerates the list of valid identifiers that can be added to the symbol table.

Module 4: Syntax Tree

The syntax tree represents the syntactic structure of the pascal program being compiled. The tree is constructed of nodes, denoting constructs that appear in the source code as defined by the Micro-Pascal grammar. The full indented syntax tree of a program can be printed to a file from CompilerMain.java using the exportSyntaxTree() method.

Module 5: Semantic Analysis

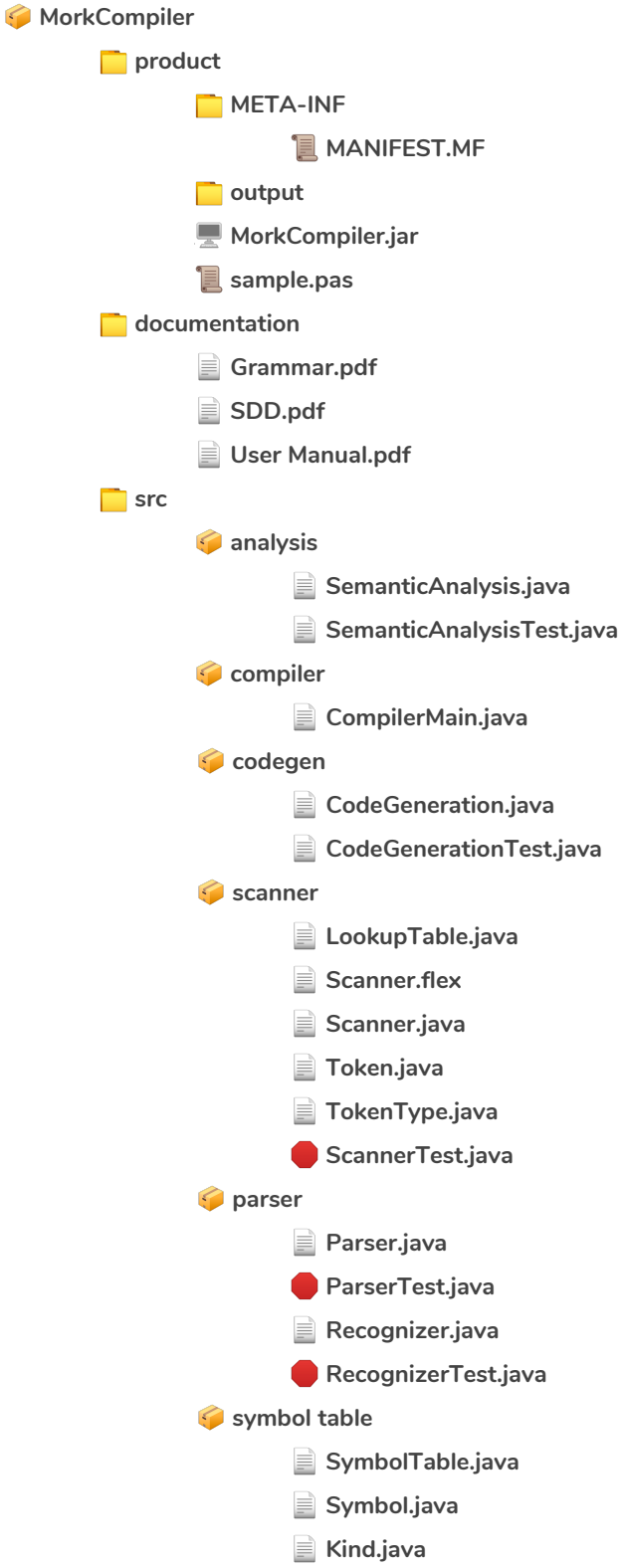
The semantic analysis module checks for errors that were parsed as having valid syntax, but are semantically invalid. As of version 1.0, the Semantic Analyzer checks that all program variables are properly declared before they are used, and that all operations and assignments are written with correct, consistent data types. Semantic analysis can be run by creating an instance of the SemanticAnalyzer class using a program node and a symbol table as parameters (both generated by the parser) of the constructor.

Module 6: Code Generation

The code generation module traverses the syntax tree, generating MIPS assembly code for each node of the tree as produced by the parser. As the syntax tree is traversed, MIPS

assembly commands are generated using syntax-specific method calls. In cases of variable declarations, a memory address is assigned to the respective variable symbol within the symbol table. To run code generation, an instance of the CodeGeneration class can be created with the root program node of the syntax tree and the symbol table as parameters of the constructor.

Project Structure



- SymbolTableTest.java
- syntaxtree
  - AssignmentStatementNode.java
  - CompoundStatementNode.java
  - DeclarationsNode.java
  - ExpressionNode.java
  - FunctionNode.java
  - IfStatementNode.java
  - OperationNode
  - ProcedureStatementNode.java
  - ProgramNode.java
  - StatementNode.java
  - SubProgramDeclarationsNode.java
  - SubProgramNode.java
  - SyntaxTreeNode.java
  - ValueNode.java
  - VariableNode.java
  - WhileStatementNode.java
  - SyntaxTreeTest.java
- pascal
  - money.pas
  - simple.pas
  - simplest.pas
- .gitignore
- README.md

Master Changelog

Commit ID	Commit Tag	Version	Description	Date
Finalized first .jar package of the compiler.				
-	JAR	1.0	Final commit for .jar submission.	4/30/2019
Finished Code Generation and related unit testing. Module 6 complete.				
4ec9fac	CodeGeneration	0.6	Final commit for Module 6.	4/29/2019
Finished Semantic Analysis and related unit testing. Module 5 complete.				
b8083d2	SemanticAnalysis	0.5	Final commit for Module 5.	4/28/2019
Finished part two of syntax tree module and related unit testing. Part two of module 4 complete.				
de2b0a9	SyntaxTreeFinal	0.4.1	Final commit for Module 4 part 2.	4/15/2019
Finished part one of syntax tree module and related unit testing. Part one of Module 4 complete.				
0a31746	SyntaxTreeStart	0.4	Final commit for Module 4 part 1.	4/8/2019
Finished Symbol table module and related unit testing. Part two of Module 3 complete.				
2c071e1	SymbolTableInt	0.3.1	Final commit for Module 3 part 2.	3/15/2019
Finished Symbol table module and related unit testing. Part one of Module 3 complete.				
838cce9	SymbolTable	0.3	Final commit for Module 3. (Updated SDD)	3/12/2019
Old Master branch deleted and recreated for ease of version control readability.				
0e35530	N/A	0.2	Working on fixing version control ambiguities.	3/3/2019
Finished Recognizer module and related unit testing. Module 2 complete.				
b75ebda	Recognizer	0.2	Final commit for Module 2.	3/3/2019
Finished writing the recognizer module.				
310e480	N/A	0.1	Finished Recognizer functions.	3/3/2019
Imported more old files to begin development on the recognizer.				
56892b8	N/A	0.1	Imported example files.	3/3/2019
Finished Scanner module and related unit testing. Module 1 complete.				

fd20cc7	**Scanner**	**0.1**	**Final commit for Module 1.**	**3/3/2019**
Finished writing the scanner module.				
e492bd8	**N/A**	**0.0**	**Finished Scanner edits for first module.**	**3/3/2019**
Imported old files to begin working on the scanner.				
46d4e1d	**Import**	**0.0**	**Imported old files.**	**3/3/2019**