Last updated: Thu 30 Apr 2020 11:29:17 AEST.

PL0 Compiler Java Source Code (Java-CUP generated version)

The main changes from the assignment 1 compiler are in the package **parse** which makes use of the parser and lexical analyser generators.

The Java files that make up the compiler are listed below, along with a brief description of their purpose. The source code for the compiler is divided into the following packages:

- **pl0** contains the Java files for the main program for the compiler
- **parse** contains the Java files to handle scanning and parsing as well as defining the lexical tokens (major changes from assignment 1 approach)
- **tree** contains the abstract syntax tree, the static semantic checker and the code generator
- **syms** contains the symbol table, its entries, and type descriptors
- **machine** contains the Java files for the stack machine interpreter, instruction set operation codes, and instructions.
- **source** contains the Java files for processing the source input and handling error messages

Package pl0

You shouldn't need to look in here too much. The main program defines some command line arguments that may be useful.

- **PL0_LALR.java** (class) The main program for the compiler ("LALR" stands for the LALR parser generator Java-CUP).

Package machine

You may want to look inside the stack machine at some stage to work out exactly what an instruction does.

- **Instruction.java** (class) Defines the different formats of instruction.
- **Operation.java** (enumeration) Define the operation codes for the stack machine.
- **StackMachine.java** (class) Implements an interpreter for the Stack Machine.

Package parse

You'll need to understand how the Parser works and what the lexical Tokens are, but you shouldn't need to look at the Scanner. The whole approach to parsing is completely different to assignment 1.

- **PL0.cup** (Java-CUP specification for PL0 parser) The parser recognises the input program and creates and abstract syntax tree.
- **PL0.flex** (JFlex specification for PL0 lexical analyser) The lexical analyser recognises the input character stream and splits it up into a stream of lexical tokens.
- **CUPScanner.java** (class) A tiny bit of plumbing between the lexical analyser (Lexer) and the parser generated by CUP.
- **CUPParser.java** (class) The parser (written in Java) generated from PL0.cup. You should not edit this directly; edit PL0.cup and re-generate this. If you get Java errors in this file, they usually correspond to something invalid in one of the semantic actions within your PL0.cup specification; try to determine what is wrong in the Java but then fix the corresponding action in PL0.cup.
- **CUPToken.java** (interface) A Java interface defining all the lexical tokens. This file is generated by Java-CUP from the terminal symbol specifications within PL0.cup. If it is incorrect do not modify this file, but modify PL0.cup and re-generate this file using Java-CUP.
- **Lexer.java** (class) The lexical analyser or scanner (written in Java) generated from PL0.flex.

Package tree

You'll need to know the structure of the Abstract Syntax Tree and the definitions of the Operators. Then you'll need to understand how the StaticChecker and CodeGenerator work. These both implement the Visitor interfaces to traverse the abstract syntax tree.

- **Code.java** (class) Data structure for a sequence of instructions used in the code generation.
- **CodeGenerator.java** (class, implements TreeTransform, ExpTransform, and StatementTransform) Implements the code generation for the compiler via a tree traversal. You'll need to modify this to generate code for the extra statements.
- **ConstExp.java** (class) Symbolic constant expressions are evaluated at compile time. This class provides tree nodes to represent constants and evaluate them.
- **DeclNode.java** (class) This class provides the abstract syntax tree nodes representing declarations lists and procedure declarations.
- **DeclVisitor.java** (interface) Visitor interface for declarations and procedures (including main program).
- **ExpNode.java** (class) Defines the nodes in the abstract syntax tree for expressions as well as methods for allowing tree traversals using the visitor pattern.
- **ExpTransform.java** (interface) Visitor interface for expressions returning a transformed expression.
- **Operator.java** (enumeration) Enumeration for the binary and unary operators for the abstract syntax tree.
- **Procedures.java** (class) Provides data structure to keep track of the start and finish addresses of procedures. Also used to provide a run-time stack trace (on a run-time error).

- **StatementNode.java** (class) Defines the nodes in the abstract syntax tree for statements as well as methods for allowing tree traversals using the visitor pattern. You'll need to modify this to generate the appropriate abstract syntax tree structure.
- **StatementTransform.java** (interface) Visitor interface for statement transformation (used in code generation).
- **StatementVisitor.java** (interface) Visitor interface for statements (used by static checker).
- **StaticChecker.java** (class, implements TreeVisitor, ExpTransform, and StatementVisitor) Implements the static (type) checking for the compiler via a tree traversal. You'll need to modify this to static check the new constructs.

Package syms

This package defines the symbol table and its entries as well as type descriptors.

- **Predefined.java** (class) Handles all the predefined constants, types and operators.
- **Scope.java** (class) Provides a single scope within the symbol table.
- **SymEntry.java** (class) Defines the symbol table entries for CONST, TYPE, VAR and PROCEDURE identifiers, and operators.
- **SymbolTable.java** (class) The main symbol table that puts together symbol tables for each Scope (procedure, main program or predefined scope) that contain entries (SymEntry) for each identifier. Handles constant, variable, type, procedure identifiers, as well as the type structures.
- **SymbolTableTest.java** (JUnit test class) JUnit test for the symbol table.
- **Type.java** (class) Defines the symbol table structures that represent types, including basic types like int and boolean, as well as subrange types.
- **TypeTest.java** (JUnit test class) JUnit test for types.

Package source

This is low level boring stuff. You shouldn't have to look in here unless you just want to find out what it does.

- **CompileError.java** (class) Stores a single error message.
- **ErrorHandler.java** (class, implements Errors) Handles the saving and reporting of error messages.
- **Errors.java** (interface) Errors interface defines error methods.
- **LineLocations.java** (class) Used for finding the source line in the input file when reporting errors.
- **Severity.java** (enumeration) Enumeration containing the error message severities.
- **Source.java** (class) Handles reading the source input, keeping track of the location within the source input and output of error messages and a listing of the input.

Other files

Test programs

- **test-pgm/** A directory containing a set of test PL0 programs (each with suffix ".pl0").

Java-CUP and JFlex archives

- **java-cup-11a.jar** The Java archive containing the parser generator Java-CUP.
- **JFlex.jar** The Java archive containing the lexical analyser generator JFlex.

For Eclipse

- **build-cup.xml** An ant script for running the parser generator Java-CUP on PL0.cup to generate CUPParser.java and CUPTokens.java. Can be used within Eclipse.
- **build-jflex.xml** An ant script for running the lexical analyser generator JFlex on PL0.flex to generate Lexer.java. Can be used within Eclipse.