Compilation \ PA3 Documentation

Group members:

- Kalev Alpernas, user: kalevalp, ID: 304385727
- Ariel Stolerman, user: arielst1, ID: 039432489
- Vadim Stotland, user: stotland, ID: 314282682

Code structure:

IC.lex is the source for the lexical analyzer which is used to scan the input IC code. IC.cup and Library.cup are the source for the syntactic parser for the input IC file and the input IC-library (optional) file respectively. The parsers are used to scan the output tokens of the lexer, derive legal sequences of tokens according to the IC syntax and build an AST for the input. All the AST nodes are dynamic types of class ASTNode, determined with respect to the syntactic type they represent (e.g. class StaticMethod for a static-method node). Semantic check for the library class name is done by the library-parser in this phase (due to implementation comfort).

The SymbolTableBuilder is used to build and extract the scoping hierarchy, represented by hierarchal symbol tables, from the AST, by using the IC.SymbolTable and IC.TypeTable packages, while checking all scope definition rules. Each symbol-table holds entries for every class, method or variable under it, represented by a symbol and its type, built and taken from the type table. Also, existence and uniqueness of main method is checked in this phase. The DefTypeSemanticChecker is used to check all other semantic rules, including illegal use of undefined variables, type checking and other semantic rules (scope rules for "break", "continue" and "this").

The compiler's main method runs over the input file using the Parser (/LibraryParser) class (and the Lexer class), for building the program's AST. After the build, the AST is scanned for semantic symbol tables build and semantic checks. Lexical exceptions are handled with the LexicalError class, Syntax exceptions are handled with the SyntaxError class and semantic exceptions are handled with the SemanticError class. Pretty-printing of the AST and symbol and types tables printing are optional upon invocation.

Testing Strategy:

Correctness:

- Correct building of symbol tables.
- Correct building of type table.

Soundness:

- Syntax error handling: correct error report for any type of syntax error.
- Parser error handling: based on error tests from PA2.
- Lexical analyzer error handling: based on error tests from PA1.Project Hierarchy (under directory src).

Project Hierarchy (under directory src):

Package IC:

- Class Compiler: contains the main method; as described above, takes an input IC program file (with an optional IC library file) and parses it, creating an AST for the input. After the AST build, the compiler builds the symbol-tables hierarchy and the type-table and performs all semantic checks. With optional pretty-printing of the AST and optional symbol-tables and type-table printing. Uses:
    - Package IC.Parser for classes Parser, Lexer, Token, Sym, LexicalError, SyntaxError and optional LibraryParser.
    - Package IC.AST for the Pretty-printing of the AST with class PrettyPrinter.
    - Package IC.Visitors for the symbol-tables and type-table builds with class SymbolTableBuilder, and for the semantic checks with class DefTypeSemanticChecker.
    - Packages IC.SymbolTable (for GlobalSymbolTable class) and IC.TypeTable for the semantic checks and the optional tables printing.

- Enumerated types used mainly in package IC.AST: BinaryOps, DataTypes, LiteralTypes, UnaryOps.

Package IC.AST:

- Class ASTNode: The extended classes for the AST build. Used in package IC.Parser.
- A group of classes extending class ASTNode, used for each syntactic type (e.g. Program, Literal etc.). Some of theses classes use the enumerated types in package IC.
- The Visitor interface: used for dynamic type handling, implemented in this package by class PrettyPrinter.
- Class PrettyPrinter: implements class Visitor, used for pretty-printing the AST generated by the parsers.

Package IC.Parser:

- Class Lexer: The lexical analyzer class generated by JFlex from IC.lex. Extends java_cup.runtime.Scanner.
- Class Parser: The syntactic parser class generated by CUP from IC.cup. Extends java_cup.runtime.lr_parser.
- Class LibraryParser: The syntactic parser class generated by CUP from Library.cup. Extends java_cup.runtime.lr_parser.
- Class LexicalError: extends Exception; used for lexical errors.
- Class SyntaxError: extends Exception; used for syntax errors.
- Class Sym: independent definitions class.
- Class Token: used by class Lexer, extends java_cup.runtime.Symbol.
- Additions:
  - IC.cup: The cup source code from which class Parser is generated by CUP. Compiled separately.
  - Library.cup: The cup source code from which class LibraryParser is generated by CUP. Compiled separately.
  - IC.lex: The lex source code from which class Lexer is generated by JFlex. Compiled separately.

Package IC.SymbolTable:

Symbol-tables: represents scope-structure and rules

- Class SymbolTable: the abstract class on which all the symbol-tables are built.
- Class GlobalSymbolTable: extends SymbolTable; instanced only once for the input program, the root of the symbol-tables tree. Holds all class symbols with no super-class defined in the program (including the Library class, if included).
- Class ClassSymbolTable: extends SymbolTable; instanced for each class in the program. In case the class has a super-class, its symbol-table's parent will be the super-class symbol-table, or the GlobalSymbolTable otherwise. Holds all field and method symbols defined under the class. May have MethodSymbolTable and ClassSymbolTable as its children symbol-tables.
- Class MethodSymbolTable: extends BlockSymbolTable (next one); instanced for each method in each class in the program. Holds parameter, local variable and return variable symbols defined in the method. May have BlockSymbolTable as its children symbol-tables.
- Class BlockSymbolTable: extends SymbolTable; instanced for each statement block in every procedural code in the program (under a method or other statement block). Holds local variable symbols defined in the block. May have BlockSymbolTable as its children symbol-tables.

Symbols: entries for all symbol-tables, holding name, kind and type referenced in the TypeTable

- Class Symbol: the abstract class on which all the symbols are built.
  - Class ClassSymbol: extends Symbol; represents a class entry in the GlobalSymbolTable.
  - Class MethodSymbol: extends Symbol; represents a method entry in a ClassSymbolTable.
  - Class VarSymbol: extends Symbol; represents a local variable entry in a MethodSymbolTable or a BlockSymbolTable.
    - Class FieldSymbol: extends VarSymbol; represents a field entry in a ClassSymbolTable.

- Class ParamSymbol: extends VarSymbol; represents a method-parameter entry in a MethodSymbolTable.
- Class ReturnVarSymbol: extends VarSymbol; represents a return entry (mainly for type) in a MethodSymbolTable.

Package IC.TypeTable:

- Class TypeTable: holds static fields for every type in the program, including primitive types, array types, method types and user-defined class types. Initialized by the SymbolTableBuilder. Has methods for adding and getting types, used by the SymbolTableBuilder when building the symbols-tables, the DefTypeSemanticChecker for semantic checking and the compiler for the type-table dumping option. All its type entries extend class Type.
- Class Type: the abstract class on which all types are built.
  - Class ClassType: extends Type; represents a user-defined class type.
  - Class MethodType: extends Type; represents a method type, defined by its returned-type and all parameter-types.
  - Class ArrayType: extends Type; represents an array type, defined by its elementary type and dimension.
  - Class IntType: extends Type; represents the primitive int type.
  - Class BoolType: extends Type; represents the primitive bool type.
  - Class StringType: extends Type; represents the primitive string type.
  - Class NullType: extends Type; represents the primitive null type.
  - Class VoidType: extends Type; represents the primitive void type.
- Class SemanticError: extends Exception; used for semantic errors (detailed in the next section).

Package IC.Visitors:

- Class SymbolTableBuilder: implements class IC.AST.Visitor, uses SymbolTable and TypeTable packages for building the symbol-tables hierarchy and type-table recursively from the AST built by the parser. Does all definition-derived on-the-fly semantic checks.
- Class DefTypeSemanticChecker: implements class IC.AST.Visitor, does all semantic resolving, type checking and other semantic checks left, from scanning the AST recursively and using the symbol-table hierarchy and type table built by SymbolTableBuilder.

Semantic Analysis Implementation:

The semantic analysis is divided into two main parts:

Scope hierarchy representation and definition-derived semantic analysis:

This part, managed by the SymbolTableBuilder class, is responsible for building the symbol-tables hierarchy and on-the-fly type-table, representing the scoping rules as defined for the IC language. The built is done top-down recursively from the program's AST root, starting with the global symbol table, through the classes symbol tables, methods symbol tables and block symbol tables.

While constructing the symbol tables, user-defined types, method types and array types are inserted into the TypeTable, which is initialized to hold all primitive-types.

The hierarchy build is done with backward compatibility to already defined symbol-tables, including rules:

- No duplicate class names.
- No extending pre-defined class.
- Scope definition rules: no illegal variables redefinitions (including fields and methods), no illegal shadowing, no illegal method overriding (overloading).
- Existence and uniqueness of "main" method.

When finishing this part, the symbol-tables hierarchy and type-table are built and no definition conflicts should occur.

Usage-derived semantic analysis:

This part, managed by the DefTypeSemanticChecker class, is responsible for all the usage-derived semantic checks. The scan is done also top-down recursively from the program's AST root, using the symbol-tables and type-table built before it. Checks:

- Correct variable usage: checks illegal use of undefined variables / fields / methods, subject to the scoping rules.
- Correct variable and virtual / static methods calls.
- Correct types usage, derived from the IC language type-rules.
- Correct usage of key-words: "break" and "continue" in loops, "this" usage.

The correctness of the library class name ("Library") is catched earlier in the syntactic parsing phase (was easier for implementation).

Feedback:

- We spent **approximately 35-40 hours** on all part of the assignment.
- The assignment was in an appropriate level of difficulty, but felt to be taking much resources, both in time and energy (after all, the due date was extended by two weeks from its original date). Although we've seen many ways to implement the main visitors (using 2 or 3 visitors, using Visitor vs. PropagatingVisitor etc.), perhaps giving a skeleton for the visitors (and maybe also for the symbol tables and type table) can achieve the main goal of this assignment while shortening its length.