Compilation - Programming Assignment 3

Team:
Ehud Tamir 036934644
Arbel Zinger 034666610
Inon Holdengreber 026504183
Team account: ehudtami

Please note that we implemented checks for both Bonuses

Semantic Analysis

Symbol Table and Type Table

A visitor implementation constructs the Symbol table and type table. The visitor iterates through the AST and creates symbol tables for the whole program, for Classes, for methods and for statement blocks inside methods. The type table is constructed at the same time by first inserting the primitive types and then adding every declared type in the AST to the table. This is the phase where the class hierarchy is verified.

Structure checks

The basic structure checks are performed: break and continue only inside loops, the program must have only one main method with the correct signature and the ‘this’ keyword is only used inside instance methods. This phase also verifies correct overriding of methods and makes sure that there is no method overloading.

Type and scope checking

Scope and type rules are enforced together through the AST. All the rules are enforced according to the IC specification document.

Return path check

A visitor implementation is used to verify that every non-void method has a return statement in every path. This is done by verifying that every method has a return statement in its body, or in the body of both branches of one of its if-else statements (cannot be if without else).

Variable initialization check

Variable initialization is verified in a manner similar to the return path checking. A variable is considered initialized if it was initialized in a previous statement in the current statement block, or if it was initialized in both branches of a previous if-else statement in the current statement block.

Code Structure and description of major classes

Compiler.java is our program's main class. This class handles the input. It calls the parser to parse the source file. The Parser creates an AST for the source file, and in case an error occurs it throws an exception. If the user chooses to print the AST it uses the PrettyPrinter class in order to print the AST. The library file (either the default file or the one the user inputs), is parsed as well, and added into the program AST (as a child of Root).
The compiler also creates the symbols and type tables, and enforces the structure, scope and type rules.
The main method in Compiler.java accepts 4 arguments at most: A path to a source file, and optionally a path to a library file, a flag, "-print-ast" and a flag "-dump-symtab".
Lexer.java is the lexical analysis scanner, class Lexer, is generated automatically using JFlex from the specification file IC.lex. The Lexer scans the input files and generates instances of class Token, which are returned to the caller (class Compiler in this case). In case of an invalid token the Lexer throws a LexicalError.
IC.lex contains our rules for tokenization. A full list of tokens and their respective regular expressions is given below.
sym.java defines a representation of token classes by integers. For example, the EOF token is represented by the constant sym.EOF, whose value is 0. This file is created automatically by the Java CUP Library according to the Terminals defined in the IC.cup file.
Token.java holds the representation of a token in a given input file. A token has 4 characteristics: The token ID (numeric representation of the token's name), the line where the token appears, the token's value (if applicable) and the token's name. The token's name is resolved according to its numeric value.
LexicalError.java implements an exception for errors in lexical analysis. Each instance of this exception has a line number where the error appeared, the string that caused the error and a custom message, sent by the Lexer.
Parser.java is the parser for IC language. It was generated automatically by the Java CUP library, by the grammar we defined in the IC.cup file. The parser receives tokens that the Lexer reads, and generates an AST (abstract syntax tree) using a set of rules that was defined in the IC.cup file (i.e the grammar of the language).The creation of the AST is done by instantiating object from the IC.AST package, more on this package will be explained later. If during the run of the Parser an error occurs, a syntax error will be thrown.
IC.cup is the input file for the Java CUP library. It contains a set of rules and definitions for creating the IC grammar.
Notes:
1. We perform no range checking on the integers.
2. Bonus 1: We perform error recovery on statements, method declaration and class declaration.
3. Bonus 2: We implemented the grammar fix for if, else and while statements.

LibraryParser.java this file is also created automatically by the Java CUP library. This java file receives a .sig file and parses it to an AST according to the grammar that is defined in the Library.cup file. It uses

the same sym.java file that was created by the Java CUP during its run on IC.cup. The .sig file defines external methods that can be used with the IC language.
Library.cup is an input file for Java CUP library. it defines a set of rules and definition of the Library grammar. After running the Java CUP library on this file we will get LibraryParser.java as the output.
SyntaxError.java implements an exception for errors in syntax analysis. Each instance of this exception generates an error message containing the line number and token where the error occured.
SemanticError.java implements an exception for semantic errors that occurs during the semantic analysis. Each instance of this exception generates an error message containing the line number and a simple explanation about the errors.
IC.AST package includes various classes used for the creation of the AST. We used all the files that were given to us in the assignment skeleton, and added a few more: FieldMethodList.java. this class hold 2 Lists: one of Method class and the other from Field class. We needed it to define the fields and methods declared in a class. We also added 3 classes for error recovery mechanism: EmptyStatement, ErrorClass and ErrorMethod.
IC package includes the Compiler.java, BinaryOps.java, DataTypes.java, LiteralTypes.java and UnaryOps.java. All those files except Compiler.java were given to us in the skeleton.zip.
IC.Types package includes MethodType.java and TypeTable.java. For the other types (not MethodType) we used the existing Types classes that are in the AST package i.e UserType and ICClass for holding the new types that the user added, PrimitiveType for the primitive types. Also the Type abstract class holds the information about the Type dimension, so we used that fact for holding arrays data. The TypeTable is created during the Build of the symbol tables. Everytime a symbol requests his type, he asks the TypeTable to supply him with the correct type. If this type already exists in the table it returns a reference for that object, otherwise creates a new instance, and then returns the reference for that object. That case it is easier to check if 2 types are equal only by using '==' operation on them. The TypeTable also maintains a counter, which is used both to count the total types in the table and also to supply a new entrance in the table with its TypeTable ID.
There is a static "toString" function for getting the string representation of the table for later printing, as requested in PA3. notice that the primitive types get the same ID as in the example in the course website but the rest of them get different ID number, that is according to the order of their insert which is slightly different that in the example (in the course forum you mentioned that its ok).
IC.Semantic package Here we have all the visitors we implemented to check and enforce the different rules: Types, Structure and Scope. The bonuses are also implemented in this package.
TypeCheckVisitor.java This class is in charge of the type and scope checks.
This visitor iterates over the expressions and statements in the program and makes sure that all the types are correct according to IC spec.
ReturnStatementVisitor.java This class implements the second bonus and checks that a method with a non-void return type returns a value on every control path.
StructureCheckVisitor.java this class is in charge of the structure checks.
This visitor recursively go over the program and makes sure that keywords are used in the correct context (this, break, continue), that the main method of the program is correct and single and that method overriding is done correctly.

VariableInitializeVisitor.java This class implements the first bonus and checks that a every local variable is used only after it has been initialized.
IC.SymbolTable package this package holds all the classes that are used to implement the symbol table. It includes the abstract Symbol class which defines the details that are common to all symbols. All the symbols extends the Symbol class, each one with the specific fields it needs. There are 4 kinds of symbols: ClassSymbol, FieldSymbol, MethodSymbol and VarSymbol. Kind is a simple enum that defines what is the symbol kind. The Package also holds the abstract SymbolTable class which defines the details that are common in all the SymbolTables kinds.In the SymbolTable class we keep a reference to its parent and its children. There are also 4 kinds of SymbolTables: BlockSymbolTable, ClassSymbolTable, GlobalSymbolTable and MethodSymbolTable. Each one of them extends the SymbolTable class with the fields it needs. The last class is BuildSymbolTables class that implements Visitor interface. It builds the symbol tables.

Class Hierarchy

Package IC:

- class **Compiler**
- enum **BinaryOps**
- enum **DataTypes**
- enum **LiteralTypes**
- enum **UnaryOps**
- class **SemanticError** extends **Exception**

Package IC.AST

- class **ArrayLocation** extends **Location**
- class **Assignment** extends **Statement**
- abstract class **ASTNode**
- abstract class **BinaryOp** extends **Expression**
- class **Break** extends **Statement**
- abstract class **Call** extends **Expression**
- class **CallStatement** extends **Statement**
- class **Continue** extends **Statement**
- class **EmptyStatement** extends **Statement**
- class **ErrorClass** extends **ICClass**
- class **ErrorMethod** extends **Method**
- abstract class **Expression** extends **ASTNode**
- class **ExpressionBlock** extends **Expression**
- class **Field** extends **ASTNode**
- class **FieldMethodList** extends **ASTNode**
- class **Formal** extends **ASTNode**
- class **ICClass** extends **ASTNode**

class **If** extends **Statement**
class **Length** extends **Expression**
class **LibraryMethod** extends **Method**
class **Literal** extends **Expression**
class **LocalVariable** extends **Statement**
abstract class **Location** extends **Expression**
class **LogicalBinaryOp** extends **BinaryOp**
class **LogicalUnaryOp** extends **UnaryOp**
class **MathBinaryOp** extends **BinaryOp**
class **MathUnaryOp** extends **UnaryOp**
abstract class **Method** extends **ASTNode**
abstract class **New** extends **Expression**
class **NewArray** extends **New**
class **NewClass** extends **New**
class **PrettyPrinter** implements **Visitor**
class **PrimitiveType** extends **Type**
class **Program** extends **ASTNode**
class **Return** extends **Statement**
abstract class **Statement** extends **ASTNode**
class **StatementsBlock** extends **Statement**
class **StaticCall** extends **Call**
class **StaticMethod** extends **Method**
class **This** extends **Expression**
abstract class **Type** extends **ASTNode**
abstract class **UnaryOp** extends **Expression**
class **UserType** extends **Type**
class **VariableLocation** extends **Location**
class **VirtualCall** extends **Call**
class **VirtualMethod** extends **Method**
interface **Visitor**
class **While** extends **Statement**

Package IC.Parser:

class **Lexer** implements **java_cup.runtime.Scanner**
class **LexicalError** extends **Exception**
class **sym**
class **Token** extends **java_cup.runtime.Symbol**
class **SyntaxError** extends **Exception**
class **Parser** extends **java_cup.runtime.lr_parser**
class **LibraryParser** extends **java_cup.runtime.lr_parser**

Package IC.Semantic:

class **ReturnStatementVisitor** implements **Visitor**

class **StructureChecksVisitor** implements **Visitor**
class **TypeCheckVisitor** implements **Visitor**
class **VariableInitializeVisitor** implements **Visitor**

Package IC.SymbolTable:

class **BlockSymbolTable** extends **SymbolTable**
class **ClassSymbolTable** extends **SymbolTable**
class **GlobalSymbolTable** extends **SymbolTable**
class **MethodSymbolTable** extends **SymbolTable**
class **SymbolTable**
class **BuildSymbolTables**implements **Visitor**
class **ClassSymbol** extends **Symbol**
class **FieldSymbol** extends **Symbol**
class **MethodSymbol** extends **Symbol**
class **VarSymbol** extends **Symbol**
class **Symbol**
enum **Kind**

Package IC.Types:

class **MethodType**
class **TypeTable**

Testing Strategy

Our tests, consisting on both semantic correct and semantically incorrect files, were divided, as everything else, to three- Structure, scope and type checks. Here are the main things we tests:

1. Checking that there is at most one main method and that its signature is correct
2. break and continue statements are only inside loops
3. Correct overriding of methods
4. No overloading
5. Bonuses: We implemented both bonuses. We tested that a local variable is only used after it is initialized, and that there is a return value in every path of a non-void method.
6. Local variables and local method parameters have to be declared before they are used.
7. Correct usage of methods (both static and instance methods).
8. No shadowing of method parameters.
9. Return types of methods are the same as the return value type
10. correct types in every expression/statement as defined in section 15 of the scope: We have a lot of examples for these tests in the submitted project in /test/semantic/Type Checking.
11. Correct usage of **this**.
12. Correct structure of the class hierarchy.