Compilation \ PA4 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 RegCounterVisitor is used to determine the number of registers used by each node in the AST, for implementing the Setti-Ullman optimization when creating the LIR translation. The translation to LIR is created by the TranslatePropagatingVisitor or in its optimized version by the OptTranslatePropagatingVisitor, all in the IC.LIR package.

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. When the code has passed all semantic checks, the compiler creates a LIR representation for the code, based on the AST (optionally optimized). 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 translation into LIR. This will be tested with a few complex tests that incorporate all language features.

Soundness: Run-time error checking – a few simple tests with run time errors.

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 interface 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 interface 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 interface 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.

Package IC.LIR:

- Class ClassLayout: implements the class layout for the LIR translation, including all fields and methods offsets. The classes' dispatch vectors are determined by this class.
- Class RegCounterVisitor: implements interface IC.AST.Visitor. Calculates and sets the number of registers used by each AST node in order to determine evaluation order later by the Setti-Ullman optimization algorithm.
- PropagatingVisitor interface: used for dynamic type handling.
- Class TranslatePropagatinVisitor: implements interface PropagatingVisitor. Creates an unoptimized LIR translation for the IC code. Optimizations are done by the following class. The visit method for each node receives the index of the next free register and returns the string representation for the LIR code for the AST (and its children) with some more information relevant for the LIR translation.
- Class OptTranslatePropagatinVisitor: implements interfacePropagatingVisitor, extends class TranslatePropagatinVisitor. Creates an optimized LIR representation for the IC code. The optimizations are detailed in the next section.
- Class LIRUpType: the type returned by the translating visitors' visit method. Hold the AST node's LIR translation, a representation for the type returned (register, memory, etc.) and the type itself.
- Enumerated types used by the visitors: LIRFlagEnum.

LIR Translation Implementation:

Translation:

The LIR translation is done after the AST is built and all semantic checks and symbol tables are done.

First the RegCounterVisitor sets the number of registers used for each AST node. In case of a VirtualCall or a StaticCall (including library method call), it is assumed the call may change values for some virtual fields, thus the original order of evaluation needs to be maintained. The register count values are used by the optimized LIR translator.

The second phase is done by one of the two LIR translating propagating visitors, according to the user's input – regular or optimized. The regular translator, TranslatePropagatingVisitor, receives the global symbol table as input and outputs the LIR translation (as String). The Program ASTNode combines all the LIR code into the final representation in the following format:

- String literals
- Class dispatch tables
- Methods (all but main, including runtime error checking methods)
- Main method

Generally the translation is done in the following way: each node translates its children recursively, finishing with the relevant LIR code based on the translation that corresponds to its action (e.g. LogicalBinaryOp – translates the operands and according to the relevant operator translates the needed action). Each recursive call passes the child a number representing the next free register index. Each child returns a LIRUpType which is a data structure that holds the LIR code, the string representation of the target holding the final result of that child and its type, e.g. an operand child for the LogicalBinaryOp may return as a REGISTER represented by "R2", a LITERAL for "true" value represented by an explicit "1", an ARR_LOCATION represented by "R1[R2]" etc.

Some special cases are handled along the translation. For instance, string literals update a list of the string literals to be set as the first piece of the LIR code, ClassLayout for each class are created in the Program AST node before each recursive call to the relevant ICClass and so on.

The optimized translator, OptTranslatePropagatingVisitor, extends TranslatePropagatingVisitor and overrides some of its methods with optimizations. The optimizations are detailed below.

Optimizations:

- Weighted register allocation and reducing the number of used registers by the Setti-Ullman algorithm: The optimized translator uses the information on the number of registers used by each AST node in order to determine order of evaluation for each expression. For instance, which operand should be evaluated first in a LogicalBinaryOp.
- Avoiding unnecessary storage of some types of values in registers: e.g. variables and constants; when possible, no storing into a new register is done.
- Use of accumulator registers: serial operations are accumulated into one register throughout the calculation.
- Reuse of "dead" registers: by passing the next free register index to each child and receiving the type of its target (may use one, two or no registers at all), a minimal use of registers is maintained and registers are reused.
- Reducing the number of instructions: as a result of the above optimizations.

Feedback:

- We spent **approximately 30 hours** on all part of the assignment.
- The assignment was in an appropriate level of difficulty. We have no special suggestions concerning the assignment.