Compilation - Programming Assignment 4

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

Intermediate Representation

package IC.LIR includes our implementation of the Intermediate Representation (IR). The package includes an interface with the name "PropagatingVisitor" and a class "TranslateVisitor" which implements the interface. The translate visitor traverses the AST and creates the LIR code. The translation is done by translating each AST node into a corresponding LIR code block, and then concatenating these blocks in an appropriate order.
We also have a class LirBlock which represents a basic LIR block that will later form the whole LIR code. Another important class is "ClassLayout" which will later be in charge on creating the dispatch vector.
We implemented 2 of the bonuses (bonus 2 & 3). The Weighted register allocation algorithm (Sethi-Ullman) is implemented in 2 classes: **SethiUllmanWeightVisitor**, which calculates the weight of all the expressions in the AST and checks where applying the algorithm is legal (i.e. no expressions with side effects); and **OptimizedTranslateVisitor**, which changes the calculation order where applies in order to save registers.
Bonus #2 (Reusing registers) is implemented in **TranslateVisitor** class. During translation we assign registers according to the technique in order to reuse as many registers as possible (this optimization is very effective in reusing registers across a whole program!).

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 compiler also generates the optimized LIR code by running the Sethi-Ullman weight calculating visitor and then the translating visitor.
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.

IC.LIR package as written in the beginning of this document this package handles the creation of the LIR code which will later be translated to Assembly code in the code generation phase of the compiler.

for each visitor call we pass a target register which will eventually hold the result of the call. We do this smartly, not just by defining a new target register for each call. We call the visitor with the right register number and therefore save a lot of register from going to waste.

ClassLayout.java holds data about methods in several tables. This data will be translated to a dispatch vector using the getDispatchVector function.

LirBlock.java represents a basic LIR block which will later form a full LIR code using the translate visitor.

PropagatingVisitor.java is an Interface for tree traversal .

TranslateVisitor.java implements the Propagating Visitor and is in charge on building the full Lir code. It first places a list of all the string literals in the IC program. Afterwards a definition of all dispatch vectors(taken from classLayout), one for each of the classes. the DV includes a list of all the methods that are included in this class. Then a series of runtime checks required, such as division by 0 , Array access etc. and lastly the whole LIR code for the IC program.

OptimizedTranslateVisitor.java implements the top-down phase of the Sethi-UIlman algorithm, by first generating the code for heavier expression sub-trees (where applicable).

SethiUllmanWeightVisitor.java a visitor that traverses the AST and calculates how many registers are required for each expression, and whether it is optimizable (contains no side-effects).

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**

Package IC.LIR

- class **ClassLayout**
- class **LirBlock**
- enum **LirValueType**
- interface **PropagatingVisitor**
- class **TranslateVisitor** implements **PropagatingVisitor**
- class **OptimizedTranslateVisitor** extends **TranslateVisitor**
- class **SethiUllmanWeightVisitor** implements **Visitor**

Testing Strategy

We built a set of IC programs to test the correctness of our LIR code generation phase of the compiler.The set can be found under test/lir folder.

We first tested every small portion of the code generated, meaning testing the correctness of each of the visitor methods. We compiled the IC program using our compiler and ran them in Micro LIR to match the required result with the real results. We also tested each of the runtime errors to see if we catch them. We looked at the generated LIR code to see that its correct.

Afterwards we moved to checking large IC programs that actually have a meaning, like an example

quicksort to see if they run correctly and returns the required result. During the tests we tested both correct code and incorrect code to verify that we catch all possible errors.