

Project #3
CS 3510 – Spring 2015
Nathan Kallman & Daniel Rees

I. Requirements: Develop a codegen package that will take the AST created by the parser and generates low level code.

II. Design: Each class of the AST has a codegen method that will create the operations and basic blocks necessary that will fit inside the functions. The low level code created by the codegen is fed through another translator/optimizer that generates x86/x64 assembly that may be assembled by an assembler.

III. Implementation: We wrote a series of codegen functions that each could parse a single node of the AST and would call the other codegen functions as needed.

IV. Testing: We used the same codebase for testing as well as three files provided by D.r. G.

V. Summary/Conclusion: Our codegen will correctly generate the low level code as specified, to the best of our knowledge.

VI. Lessons Learned: It has been very interesting to see how something as simple as if (a == b) foo(); can turn into a very large set of assembly instructions. Compilers give a huge amount of power. They are way cool!

```
package compiler;

import cminus_compiler.grammar.Program;
import cminus_compiler.interfaces.ParserInterface;
import cminus_compiler.tool.Parser;
import x64codegen.X64AssemblyGenerator;
import lowlevel.*;
import java.util.*;
import java.io.*;
import optimizer.*;
import x86codegen.*;
import x64codegen.*;
import dataflow.*;

public class CMinusCompiler implements Compiler {

    public static HashMap globalHash = new HashMap();
    private static boolean genX64Code = false;

    public CMinusCompiler() {
    }

    public static void setGenX64Code(boolean useX64) {
        genX64Code = useX64;
    }

    public static boolean getGenX64Code() {
        return genX64Code;
    }

    public void compile(String filePrefix) {

        String fileName = filePrefix + ".c";
        try {
            ParserInterface myParser = new Parser(fileName);

            Program parseTree = myParser.parse();
            String tree = parseTree.printTree();
            System.out.println(tree);
            myParser.printAST(parseTree);

            CodeItem lowLevelCode = parseTree.genLLCode();

            fileName = filePrefix + ".ll";
            PrintWriter outFile = new PrintWriter(new BufferedWriter(new FileWriter(fileName)));
            lowLevelCode.printLLCode(outFile);
            outFile.close();

            int optiLevel = 2;
            LowLevelCodeOptimizer lowLevelOpti =
                new LowLevelCodeOptimizer(lowLevelCode, optiLevel);
            lowLevelOpti.optimize();

            fileName = filePrefix + ".opti";
            outFile =
```

```
        new PrintWriter(new BufferedWriter(new FileWriter(fileName)));
        lowLevelCode.printLLCode(outFile);
        outFile.close();

        if (genX64Code) {
            X64CodeGenerator x64gen = new X64CodeGenerator(lowLevelCode);
            x64gen.convertToX64();
        }
        else {
            X86CodeGenerator x86gen = new X86CodeGenerator(lowLevelCode);
            x86gen.convertToX86();
        }
        fileName = filePrefix + ".x86";
        outFile =
            new PrintWriter(new BufferedWriter(new FileWriter(fileName)));
        lowLevelCode.printLLCode(outFile);
        outFile.close();

//    lowLevelCode.printLLCode(null);

        // simply walks functions and finds in and out edges for each BasicBlock
        ControlFlowAnalysis cf = new ControlFlowAnalysis(lowLevelCode);
        cf.performAnalysis();
//    cf.printAnalysis(null);

        // performs DU analysis, annotating the function with the live range of
        // the value defined by each oper (some merging of opers which define
        // same virtual register is done)
//    DefUseAnalysis du = new DefUseAnalysis(lowLevelCode);
//    du.performAnalysis();
//    du.printAnalysis();

        LivenessAnalysis liveness = new LivenessAnalysis(lowLevelCode);
        liveness.performAnalysis();
        liveness.printAnalysis();

        if (genX64Code) {
            int numRegs = 15;
            X64RegisterAllocator regAlloc = new X64RegisterAllocator(lowLevelCode,
                numRegs);
            regAlloc.performAllocation();

            lowLevelCode.printLLCode(null);

            fileName = filePrefix + ".s";
            outFile =
                new PrintWriter(new BufferedWriter(new FileWriter(fileName)));
            X64AssemblyGenerator assembler =
                new X64AssemblyGenerator(lowLevelCode, outFile);
            assembler.generateX64Assembly();
            outFile.close();
        }
        else {
            int numRegs = 7;
```

```
        X86RegisterAllocator regAlloc = new X86RegisterAllocator(lowLevelCode,
            numRegs);
        regAlloc.performAllocation();

        lowLevelCode.printLLCode(null);

        fileName = filePrefix + ".s";
        outFile =
            new PrintWriter(new BufferedWriter(new FileWriter(fileName)));
        X86AssemblyGenerator assembler =
            new X86AssemblyGenerator(lowLevelCode, outFile);
        assembler.generateAssembly();
        outFile.close();
    }

    } catch (IOException ioe) {
    }

}

public static void main(String[] args) {
    String filePrefix = "testcode";
    CMinusCompiler myCompiler = new CMinusCompiler();
    myCompiler.setGenX64Code(true);
    myCompiler.compile(filePrefix);
}
}
```

```
package cminus_compiler.grammar;

import lowlevel.CodeItem;
import lowlevel.Function;
import lowlevel.Operand;
import lowlevel.Operation;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: AssignmentOperation.java
 * Created: March 2015
 *
 * Description:
 */
public class AssignmentOperation extends Expression {

    private Var variable;
    private Expression rightHandExpression;
    private String operation = "=";

    // Constructors
    public AssignmentOperation() {

    }

    public AssignmentOperation(Var var, Expression rhs) {
        this.variable = var;
        this.rightHandExpression = rhs;
    }

    // Getters
    public Var getVariable() {
        return variable;
    }

    public Expression getRightHandExpression() {
        return rightHandExpression;
    }

    public String getOperation() {
        return operation;
    }

    // Setters
    public void setVariable(Var variable) {
        this.variable = variable;
    }

    public void setRightHandExpression(Expression rightHandExpression) {

        this.rightHandExpression = rightHandExpression;
    }

    // Public Methods
    @Override
```

```

public String printTree(int indent) {
    StringBuilder builder = new StringBuilder();

    builder.append(indent(indent));

    builder.append(operation);
    builder.append(variable.printTree(indent+1));
    builder.append(rightHandExpression.printTree(indent+1));

    return builder.toString();
}

@Override
public CodeItem gencode(Function function) {

    variable.gencode(function);
    rightHandExpression.gencode(function);
    this.setRegNum(variable.getRegNum());

    Operation assignOperation = new Operation(Operation.OperationType.ASSIGN, function.getCurrBlock());

    //
    Operand dest = new Operand(Operand.OperandType.REGISTER, variable.getRegNum());
    Operand src = new Operand(Operand.OperandType.REGISTER, rightHandExpression.getRegNum());

    assignOperation.setDestOperand(0, dest);
    assignOperation.setSrcOperand(0, src);

    function.getCurrBlock().appendOper(assignOperation);

    // Store global variables if they changed
    if(variable.isGlobal(function)) {
        this.storeGlobalVariable(function);
    }

    return function;
}

private void storeGlobalVariable(Function function) {
    Operation storeOperation = new Operation(Operation.OperationType.STORE_I, function.getCurrBlock());

    Operand srcZero = new Operand(Operand.OperandType.REGISTER, rightHandExpression.getRegNum());
    Operand srcOne = new Operand(Operand.OperandType.STRING, variable.getVariableName());

    storeOperation.setSrcOperand(0, srcZero);
    storeOperation.setSrcOperand(1, srcOne);

    function.getCurrBlock().appendOper(storeOperation);
}
}

```

```
package cminus_compiler.grammar;

import lowlevel.CodeItem;
import lowlevel.Function;
import lowlevel.Operand;
import lowlevel.Operation;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: BinaryOperation.java
 * Created: March 2015
 *
 * Description:
 */
public class BinaryOperation extends Expression {

    private Expression leftHandExpression;
    private Expression rightHandExpression;
    private String operation;

    // Constructors
    public BinaryOperation() {
        this(null, null, null);
    }

    public BinaryOperation(Expression leftHandExpression, Expression rightHandExpression, String op) {
        this.leftHandExpression = leftHandExpression;
        this.rightHandExpression = rightHandExpression;
        this.operation = op;
    }

    // Getters
    public Expression getLeftHandExpression() {
        return leftHandExpression;
    }

    public Expression getRightHandExpression() {
        return rightHandExpression;
    }

    public String getOperation() {
        return operation;
    }

    // Setters
    public void setLeftHandExpression(Expression leftHandExpression) {
        this.leftHandExpression = leftHandExpression;
    }

    public void setRightHandExpression(Expression rightHandExpression) {
        this.rightHandExpression = rightHandExpression;
    }
}
```

```

public void setOperation(String operation) {
    this.operation = operation;
}

// Public Methods
@Override
public String printTree(int indent) {
    StringBuilder builder = new StringBuilder();
    builder.append(indent(indent));

    builder.append(operation);
    builder.append(leftHandExpression.printTree(indent+1));
    builder.append(rightHandExpression.printTree(indent+1));

    return builder.toString();
}

@Override
public CodeItem gencode(Function function) {
    this.setRegNum(function.getNewRegNum());
    leftHandExpression.gencode(function);
    rightHandExpression.gencode(function);

    // Generate Operand sources and destination for the binary operation
    Operand srcLeft = new Operand(Operand.OperandType.REGISTER, leftHandExpression.getRegNum());
    Operand srcRight = new Operand(Operand.OperandType.REGISTER, rightHandExpression.getRegNum());
    Operand dest = new Operand(Operand.OperandType.REGISTER, this.getRegNum());

    // Create the binary operation of specified type and set sources/destination
    Operation.OperationType operationType = convertToOperationType();
    Operation binaryOperation = new Operation(operationType, function.getCurrBlock());
    binaryOperation.setDestOperand(0, dest);
    binaryOperation.setSrcOperand(0, srcLeft);
    binaryOperation.setSrcOperand(1, srcRight);

    function.getCurrBlock().appendOper(binaryOperation);

    return null;
}

private Operation.OperationType convertToOperationType() {
    Operation.OperationType type = Operation.OperationType.UNKNOWN;

    switch(operation) {
        case "+":
            type = Operation.OperationType.ADD_I;
            break;
        case "-":
            type = Operation.OperationType.SUB_I;

            break;
        case "*":
            type = Operation.OperationType.MUL_I;
            break;
        case "/":
            type = Operation.OperationType.DIV_I;
            break;
    }
}

```



```
        case "<":
            type = Operation.OperationType.LT;
            break;
        case "<=":
            type = Operation.OperationType.LTE;
            break;
        case ">":
            type = Operation.OperationType.GT;
            break;
        case ">=":
            type = Operation.OperationType.GTE;
            break;
        case "==":
            type = Operation.OperationType.EQUAL;
            break;
        case "!=":
            type = Operation.OperationType.NOT_EQUAL;
            break;
        default:
            break;
    }
    return type;
}
```

```
package cminus_compiler.grammar;

import java.util.ArrayList;
import lowlevel.Attribute;
import lowlevel.CodeItem;
import lowlevel.Function;
import lowlevel.Operand;
import lowlevel.Operation;

/**
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: Call.java
 * Created: March 2015
 *
 * Description:
 */
public class Call extends Expression {

    private String callName;
    private ArrayList<Expression> args;

    public Call() {
        this(null, new ArrayList<>());
    }

    public Call(String callName, ArrayList<Expression> args) {
        this.callName = callName;
        this.args = args;
    }

    public String getCallName() {
        return callName;
    }

    public ArrayList<Expression> getArgs() {
        return args;
    }

    public void setCallName(String callName) {
        this.callName = callName;
    }

    public void setArgs(ArrayList<Expression> args) {
        this.args = args;
    }

    @Override
    public String printTree(int indent) {
        StringBuilder builder = new StringBuilder();
        builder.append(indent(indent));

        builder.append(callName);
        for(Expression arg : args) {

            builder.append(arg.printTree(indent+1));
        }
        return builder.toString();
    }

    @Override
```

```
public CodeItem gencode(Function function) {
    // Pass Operations
    ArrayList<Operation> passOperations = new ArrayList<>();
    int count = 0;
    for(Expression arg : args) {
        arg.gencode(function);

        Operand src = new Operand(Operand.OperandType.REGISTER, arg.getRegNum());
        Operation passOperation = new Operation(Operation.OperationType.PASS, function.getCurrBlock());
        passOperation.setSrcOperand(0, src);

        String size = Integer.toString(count);
        Attribute attribute = new Attribute("PARAM_NUM", size);
        passOperation.addAttribute(attribute);
        count++;

        passOperations.add(passOperation);
    }

    for(Operation passOp : passOperations) {
        function.getCurrBlock().appendOper(passOp);
    }

    // Call Operation

    String size = Integer.toString(args.size());
    Attribute attribute = new Attribute("numParams", size);
    Operand callSrc = new Operand(Operand.OperandType.STRING, this.callName);
    Operation callOperation = new Operation(Operation.OperationType.CALL, function.getCurrBlock());
    callOperation.setSrcOperand(0, callSrc);
    callOperation.addAttribute(attribute);

    function.getCurrBlock().appendOper(callOperation);

    // RetReg Operation
    Operand src = new Operand(Operand.OperandType.MACRO, "RetReg");
    Operand dest = new Operand(Operand.OperandType.REGISTER, function.getNewRegNum());
    Operation assignOperation = new Operation(Operation.OperationType.ASSIGN, function.getCurrBlock());
    assignOperation.setDestOperand(0, dest);
    assignOperation.setSrcOperand(0, src);

    function.getCurrBlock().appendOper(assignOperation);

    return null;
}
```

```
package cminus_compiler.grammar;

import java.util.ArrayList;
import lowlevel.CodeItem;
import lowlevel.Function;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: CompoundStatement.java
 * Created: Feb 2015
 *
 * Description:
 */
public class CompoundStatement extends Statement {

    private ArrayList<VarDeclaration> variableDeclarations;
    private ArrayList<Statement> statements;

    // Constructors
    public CompoundStatement() {
        this(new ArrayList<>(), new ArrayList<>());
    }

    public CompoundStatement(ArrayList<VarDeclaration> varDecls, ArrayList<Statement> statements) {
        this.variableDeclarations = varDecls;
        this.statements = statements;
    }

    // Getters
    public ArrayList<VarDeclaration> getVariableDeclarations() {
        return variableDeclarations;
    }

    public ArrayList<Statement> getStatements() {
        return statements;
    }

    // Setters
    public void setVariableDeclarations(ArrayList<VarDeclaration> variableDeclarations) {
        this.variableDeclarations = variableDeclarations;
    }

    public void setStatements(ArrayList<Statement> statements) {
        this.statements = statements;
    }

    // Public Methods
    @Override

    public String printTree(int indent) {
        StringBuilder builder = new StringBuilder();
```

```
        for(VarDeclaration decl : variableDeclarations) {
            builder.append(decl.printTree(indent+1));
        }

        for(Statement stmt : statements) {
            builder.append(stmt.printTree(indent+1));
        }

        return builder.toString();
    }

    @Override
    public CodeItem gencode(Function function) {

        for(VarDeclaration decl : variableDeclarations) {
            decl.gencode(function);
        }

        for(Statement stmt : statements) {
            stmt.gencode(function);
        }

        return function;
    }
}
```

```
package cminus_compiler.grammar;

import cminus_compiler.interfaces.CodeGen;
import cminus_compiler.tool.IndentTool;
import lowlevel.CodeItem;
import lowlevel.Function;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: Declaration.java
 * Created: Feb 2015
 *
 * Description:
 */
public abstract class Declaration implements CodeGen {

    // Attributes
    protected String declarationName;

    // Getters
    public String getDeclarationName() {
        return declarationName;
    }

    // Setters
    public void setDeclarationName(String declarationName) {
        this.declarationName = declarationName;
    }

    public String indent(int indent) {
        return IndentTool.indent(indent);
    }

    // Abstract Methods
    public abstract String printTree(int indent);

    //    public abstract CodeItem gencode(Function function);

    @Override
    public abstract CodeItem gencode(Function function);
}
```

```
package cminus_compiler.grammar;

import cminus_compiler.interfaces.CodeGen;
import cminus_compiler.tool.IndentTool;
import lowlevel.CodeItem;
import lowlevel.Function;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: Expression.java
 * Created: Feb 2015
 *
 * Description:
 */
public abstract class Expression implements CodeGen {

    public int regNum;

    public String indent(int indent) {
        return IndentTool.indent(indent);
    }

    public void setRegNum(int regNum) {
        this.regNum = regNum;
    }

    public int getRegNum() {
        return this.regNum;
    }

    @Override
    public abstract CodeItem gencode(Function function);
    public abstract String printTree(int indent);
}
```

```
package cminus_compiler.grammar;

import lowlevel.CodeItem;
import lowlevel.Function;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: ExpressionStatement.java
 * Created: Feb 2015
 *
 * Description:
 */
public class ExpressionStatement extends Statement {

    private Expression expression;

    public ExpressionStatement() {
        this(null);
    }

    public ExpressionStatement(Expression expression) {
        this.expression = expression;
    }

    public Expression getExpression() {
        return expression;
    }

    public void setExpression(Expression expression) {
        this.expression = expression;
    }

    // Public Methods
    @Override
    public String printTree(int indent) {
        String output = "";
        if(expression != null) {
            output = expression.printTree(indent+1);
        }
        return output;
    }

    @Override
    public CodeItem gencode(Function function) {
        if(expression != null) {
            expression.gencode(function);
        }
        return function;
    }
}
```



```
package cminus_compiler.grammar;

import java.util.ArrayList;
import lowlevel.BasicBlock;
import lowlevel.CodeItem;
import lowlevel.Data;
import lowlevel.FuncParam;
import lowlevel.Function;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: FunDeclaration.java
 * Created: Feb 2015
 *
 * Description:
 */
public class FunDeclaration extends Declaration {

    private String returnType; // TODO: Can we avoid having returnType as a String?
    private ArrayList<Param> params;
    private CompoundStatement compoundStatement;

    // Constructors
    public FunDeclaration() {
        this(null, new ArrayList<>(), null, null);
    }

    public FunDeclaration(String type, ArrayList<Param> params, CompoundStatement stmt, String name) {
        this.returnType = type;
        this.params = params;
        this.compoundStatement = stmt;
        this.declarationName = name;
    }

    // Getters
    public String getReturnType() {
        return returnType;
    }

    public ArrayList<Param> getParams() {
        return params;
    }

    public CompoundStatement getCompoundStatement() {
        return compoundStatement;
    }

    // Setters
    public void setReturnType(String returnType) {
        this.returnType = returnType;
    }

    public void setParams(ArrayList<Param> params) {
```

```

        this.params = params;
    }

    public void setCompoundStatement(CompoundStatement compoundStatement) {
        this.compoundStatement = compoundStatement;
    }

    // Public Methods
    @Override
    public String printTree(int indent) {
        StringBuilder builder = new StringBuilder();
        builder.append(indent(indent));

        builder.append(returnType);
        builder.append(" ");
        builder.append(declarationName);
        for(Param param : params) {
            builder.append(param.printTree(indent+1));
        }

        builder.append(compoundStatement.printTree(indent+1));

        return builder.toString();
    }

    @Override
    public CodeItem gencode(Function f) {

        int type = convertReturnType();
        Function function = new Function(type, declarationName);

        // Convert all of the function parameters into FuncParams to pass to the Function object. Need
        // to maintain a pointer to the front of the linked list, while still adding to the linked list
        FuncParam firstParam;
        if(!params.isEmpty()) {
            firstParam = generateFuncParams(function);
        } else {
            firstParam = new FuncParam(Data.TYPE_VOID, "void");
            function.getTable().put(firstParam.getName(), function.getNewRegNum());
        }

        // Generating the function and compound statement of the function
        function.setFirstParam(firstParam);

        // Create BB0
        function.createBlock0();

        // Make BB
        BasicBlock bbOne = new BasicBlock(function);

        // Append BB
        function.appendBlock(bbOne);

        // Set CB = BB
        function.setCurrBlock(bbOne);

        // gencode { }

```

```

this.compoundStatement.gencode(function);

// append return block
BasicBlock returnBlock = function.getReturnBlock();
function.appendBlock(returnBlock);

// append Unconnected chain
BasicBlock unconnectedChainBlock = function.getFirstUnconnectedBlock();
if(unconnectedChainBlock != null) {
    function.appendBlock(unconnectedChainBlock);
}

// Return CodeItem
return function;
}

private FuncParam generateFuncParams(Function f) {
    FuncParam firstParam = new FuncParam();
    FuncParam tempParam = new FuncParam();
    int i = 0;
    for(Param param : params) {
        if(i == 0) {
            tempParam = param.gencode(f);
            firstParam = tempParam;
        } else {
            tempParam.setNextParam(param.gencode(f));
            tempParam = tempParam.getNextParam();
        }
        i++;
    }

    return firstParam;
}

private int convertReturnType() {
    if(this.returnType.equalsIgnoreCase("void")) {
        return Data.TYPE_VOID;
    } else {
        return Data.TYPE_INT;
    }
}
}

```

```
package cminus_compiler.grammar;

import lowlevel.BasicBlock;
import lowlevel.CodeItem;
import lowlevel.Function;
import lowlevel.Operand;
import lowlevel.Operation;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: IterationStatement.java
 * Created: Feb 2015
 *
 * Description:
 */
public class IterationStatement extends Statement {

    private Expression expression;
    private Statement statement;

    public IterationStatement() {
        this(null, null);
    }

    public IterationStatement(Expression expression, Statement statement) {
        this.expression = expression;
        this.statement = statement;
    }

    public Expression getExpression() {
        return expression;
    }

    public Statement getStatement() {
        return statement;
    }

    public void setExpression(Expression expression) {
        this.expression = expression;
    }

    public void setStatement(Statement statement) {
        this.statement = statement;
    }

    @Override
    public String printTree(int indent) {
        StringBuilder builder = new StringBuilder();
        builder.append(indent(indent));

        builder.append("while");
        builder.append(expression.printTree(indent+1));

        builder.append(statement.printTree(indent+1));

        return builder.toString();
    }

    @Override
```

```
public CodeItem gencode(Function function) {
    // 1. Gencode expression
    expression.gencode(function);

    // 2. Make 2 blocks
    BasicBlock thenBlock = new BasicBlock(function);
    BasicBlock postBlock = new BasicBlock(function);

    // 3. create the branch operation to based on the condition given in while expression
    Operation branchOperation =
        getBranchOperation(Operation.OperationType.BEQ, postBlock, function.getCurrBlock());
    function.getCurrBlock().appendOper(branchOperation);
    function.appendToCurrentBlock(thenBlock);
    function.setCurrBlock(thenBlock);

    // 6. gencode statement
    statement.gencode(function);

    // Recheck condition
    expression.gencode(function);

    // Loop Condition
    Operation bneBranchOperation =
        getBranchOperation(Operation.OperationType.BNE, thenBlock, function.getCurrBlock());
    function.getCurrBlock().appendOper(bneBranchOperation);
    function.appendToCurrentBlock(postBlock);
    function.setCurrBlock(postBlock);

    return function;
}

private Operation getBranchOperation(Operation.OperationType type, BasicBlock block, BasicBlock cur) {
    Operation branchOperation = new Operation(type, cur);
    Operand srcOne = new Operand(Operand.OperandType.REGISTER, expression.getRegNum());
    Operand srcConst = new Operand(Operand.OperandType.INTEGER, 0);
    Operand srcTarget = new Operand(Operand.OperandType.BLOCK, block.getBlockNum());

    branchOperation.setSrcOperand(0, srcOne);
    branchOperation.setSrcOperand(1, srcConst);
    branchOperation.setSrcOperand(2, srcTarget);

    return branchOperation;
}
```

```
package cminus_compiler.grammar;

import cminus_compiler.model.Token;
import cminus_compiler.tool.IndentTool;
import lowlevel.CodeItem;
import lowlevel.Function;
import lowlevel.Operand;
import lowlevel.Operation;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: Num.java
 * Created: March 2015
 *
 * Description:
 */
public class Num extends Expression {

    private int value;

    // Constructors
    public Num() {
        this(0);
    }

    public Num(Token token) {
        this(Integer.parseInt(token.data()));
    }

    public Num(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    @Override
    public String printTree(int indent) {
        StringBuilder builder = new StringBuilder();
        builder.append(IndentTool.indent(indent));
        builder.append(toString());
        return builder.toString();
    }

    @Override
    public String toString() {
        return Integer.toString(value);
    }

    @Override
```

```
public CodeItem gencode(Function function) {  
  
    int regNum = function.getNewRegNum();  
    this.setRegNum(regNum);  
  
    Operand src = new Operand(Operand.OperandType.INTEGER, this.value);  
    Operand dest = new Operand(Operand.OperandType.REGISTER, regNum);  
  
    Operation operation = new Operation(Operation.OperationType.ASSIGN, function.getCurrBlock());  
    operation.setDestOperand(0, dest);  
    operation.setSrcOperand(0, src);  
  
    function.getCurrBlock().appendOper(operation);  
  
    return function;  
}
```

```
package cminus_compiler.grammar;

import cminus_compiler.model.Token;
import cminus_compiler.tool.IndentTool;
import lowlevel.CodeItem;
import lowlevel.Data;
import lowlevel.FuncParam;
import lowlevel.Function;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: Param.java
 * Created: March 2015
 *
 * Description:
 */
public class Param {

    private String paramName;
    private boolean isArray;

    public Param() {
        this("", false);
    }

    public Param(Token ID, boolean isArray) {
        this((String)ID.getTokenData(), isArray);
    }

    public Param(String paramName, boolean isArray) {
        this.paramName = paramName;
        this.isArray = isArray;
    }

    // Getters

    public String getParamName() {
        return paramName;
    }

    public boolean isArray() {
        return isArray;
    }

    // Setters
    public void setParamName(String paramName) {
        this.paramName = paramName;
    }

    public void setIsArray(boolean isArray) {
```



```
        this.isArray = isArray;
    }

    // Public Methods
    public String printTree(int indent) {

        StringBuilder builder = new StringBuilder();
        builder.append(IndentTool.indent(indent));

        builder.append(paramName);
        builder.append(" is array: ");
        builder.append(isArray);

        return builder.toString();
    }

    public FuncParam gencode(Function function) {
        FuncParam param = new FuncParam(Data.TYPE_INT, paramName);
        function.getTable().put(paramName, function.getNewRegNum());
        return param;
    }
}
```

```
package cminus_compiler.grammar;

import java.util.ArrayList;
import lowlevel.CodeItem;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: Program.java
 * Created: Feb 2015
 *
 * Description:
 */
public class Program {

    // Program variables
    private ArrayList<Declaration> declarations;

    // Program constructor
    public Program() {
        declarations = new ArrayList<>();
    }

    // Program Methods
    public void addDeclaration(Declaration declaration) {
        declarations.add(declaration);
    }

    public CodeItem genLLCode() {
        CodeItem nextItem = declarations.get(0).gencode(null);
        CodeItem firstItem = nextItem;
        for(int i = 1; i < declarations.size(); i++) {
            nextItem.setNextItem(declarations.get(i).gencode(null));
            nextItem = nextItem.getNextItem();
        }

        return firstItem;
    }

    public String printTree() {
        StringBuilder builder = new StringBuilder();
        builder.append("\n*** Begin Tree *** \nProgram");
        for(Declaration declaration : declarations) {
            builder.append(declaration.printTree(1));
        }

        return builder.toString();
    }
}
```

```
package cminus_compiler.grammar;

import lowlevel.CodeItem;
import lowlevel.Function;
import lowlevel.Operand;
import lowlevel.Operation;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: ReturnStatement.java
 * Created: Feb 2015
 *
 * Description:
 */
public class ReturnStatement extends Statement {

    private Expression expression;

    // Constructors
    public ReturnStatement() {
        this(null);
    }

    public ReturnStatement(Expression expression) {
        this.expression = expression;
    }

    // Getters
    public Expression getExpression() {
        return expression;
    }

    // Setters
    public void setExpression(Expression expression) {
        this.expression = expression;
    }

    // Public Methods
    @Override
    public String printTree(int indent) {
        StringBuilder builder = new StringBuilder();
        builder.append(indent(indent));

        builder.append("return");
        if(expression != null) {
            builder.append(expression.printTree(indent+1));
        }

        return builder.toString();
    }

    @Override
    public CodeItem gencode(Function function) {
        int returnRegNum;
```

```
    if(expression != null) {
        expression.gencode(function);
        returnRegNum = expression.getRegNum();
    } else {
        returnRegNum = function.getNewRegNum();
    }

    // Source to retReg operation
    Operand src = new Operand(Operand.OperandType.REGISTER, returnRegNum);
    Operand dest = new Operand(Operand.OperandType.MACRO, "RetReg");

    Operation op = new Operation(Operation.OperationType.ASSIGN, function.getCurrBlock());
    op.setDestOperand(0, dest);
    op.setSrcOperand(0, src);

    // Jump operation to return block
    Operand jmpSrc = new Operand(Operand.OperandType.BLOCK, function.getReturnBlock().getBlockNum());

    Operation jmp = new Operation(Operation.OperationType.JMP, function.getCurrBlock());
    jmp.setSrcOperand(0, jmpSrc);

    // Append blocks
    function.getCurrBlock().appendOper(op);
    function.getCurrBlock().appendOper(jmp);

    return function;
}
```

```
package cminus_compiler.grammar;

import lowlevel.BasicBlock;
import lowlevel.CodeItem;
import lowlevel.Function;
import lowlevel.Operand;
import lowlevel.Operation;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: SelectionStatement.java
 * Created: Feb 2015
 *
 * Description:
 */
public class SelectionStatement extends Statement {

    private Expression expression;
    private Statement primaryStatement;
    private Statement optionalStatement;

    // Constructors
    public SelectionStatement() {
        this(null, null, null);
    }

    public SelectionStatement(Expression expression, Statement primary, Statement optional) {
        this.expression = expression;
        this.primaryStatement = primary;
        this.optionalStatement = optional;
    }

    // Getters
    public Expression getExpression() {
        return expression;
    }

    public Statement getPrimaryStatement() {
        return primaryStatement;
    }

    public Statement getOptionalStatement() {
        return optionalStatement;
    }

    // Setters
    public void setExpression(Expression expression) {
        this.expression = expression;
    }

    public void setPrimaryStatement(Statement primaryStatement) {
        this.primaryStatement = primaryStatement;
    }
}
```

```
public void setOptionalStatement(Statement optionalStatement) {
    this.optionalStatement = optionalStatement;
}

// Public Methods
@Override
public String printTree(int indent) {
    StringBuilder builder = new StringBuilder();
    builder.append(indent(indent));

    builder.append("if");
    builder.append(expression.printTree(indent+1));
    builder.append(primaryStatement.printTree(indent+1));

    if(optionalStatement != null) {
        builder.append(indent(indent));
        builder.append("else");
        builder.append(optionalStatement.printTree(indent+1));
    }

    return builder.toString();
}

@Override
public CodeItem gencode(Function function) {

    // 1. Gencode expression
    expression.gencode(function);

    // 2. Make 2/3 blocks
    BasicBlock thenBlock = new BasicBlock(function);
    BasicBlock postBlock = new BasicBlock(function);
    BasicBlock elseBlock = null;

    // 3. Branch to else/post
    int blockNum = -1;
    if (optionalStatement != null) {
        // Branch to elseBlock
        elseBlock = new BasicBlock(function);
        blockNum = elseBlock.getBlockNum();
    } else {
        // Branch to postBlock
        blockNum = postBlock.getBlockNum();
    }
    Operation branchOperation = new Operation(Operation.OperationType.BEQ, function.getCurrBlock());
    Operand srcOne = new Operand(Operand.OperandType.REGISTER, expression.getRegNum());
    Operand srcConst = new Operand(Operand.OperandType.INTEGER, 0);
    Operand srcTarget = new Operand(Operand.OperandType.BLOCK, blockNum);

    branchOperation.setSrcOperand(0, srcOne);
    branchOperation.setSrcOperand(1, srcConst);
    branchOperation.setSrcOperand(2, srcTarget);

    function.getCurrBlock().appendOper(branchOperation);

    // 4. Append 'then' block
    function.appendToCurrentBlock(thenBlock);
}
```

```
// 5. CB = THEN
function.setCurrBlock(thenBlock);

// 6. gencode then
primaryStatement.gencode(function);

// 7. append post
function.appendToCurrentBlock(postBlock);

if(optionalStatement != null) {
    // 8. CB = Else
    function.setCurrBlock(elseBlock);

    // 9. gencode else
    optionalStatement.gencode(function);

    // 10. JMP to POST
    Operation jmp = new Operation(Operation.OperationType.JMP, function.getCurrBlock());
    Operand jmpSrc = new Operand(Operand.OperandType.BLOCK, postBlock.getBlockNum());
    jmp.setSrcOperand(0, jmpSrc);

    function.getCurrBlock().appendOper(jmp);

    // 11. Else to unconnected Chain
    function.appendUnconnectedBlock(elseBlock);
}

// 12.
function.setCurrBlock(postBlock);

return function;
}
```

```
package cminus_compiler.grammar;

import cminus_compiler.interfaces.CodeGen;
import cminus_compiler.tool.IndentTool;
import lowlevel.CodeItem;
import lowlevel.Function;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: Statement.java
 * Created: Feb 2015
 *
 * Description:
 */
public abstract class Statement implements CodeGen {

    public String indent(int indent) {
        return IndentTool.indent(indent);
    }

    public abstract String printTree(int indent);
    @Override
    public abstract CodeItem gencode(Function function);
}
```



```
package cminus_compiler.grammar;

import cminus_compiler.model.Token;
import lowlevel.CodeItem;
import lowlevel.Function;

/**
 *]
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: Var.java
 * Created: March 2015
 *
 * Description:
 */
public class Var extends Expression {

    private String variableName;
    private Expression expression;

    // Constructors
    public Var() {
    }

    public Var(Token ID, Expression expression) {
        this((String)ID.getTokenData(), expression);
    }

    public Var(String variableName, Expression expression) {
        this.variableName = variableName;
        this.expression = expression;
    }

    // Getters
    public String getVariableName() {
        return variableName;
    }

    public Expression getExpression() {
        return expression;
    }

    // Setters
    public void setVariableName(String variableName) {
        this.variableName = variableName;
    }

    public void setExpression(Expression expression) {
        this.expression = expression;
    }
}
```

```
// Public Methods
@Override
public String printTree(int indent) {
    StringBuilder builder = new StringBuilder();
    builder.append(indent(indent));

    builder.append(variableName);
    if(expression != null) {
        builder.append("(");
        builder.append(expression.printTree(indent+1));
        builder.append(")");
    }

    return builder.toString();
}

@Override
public CodeItem gencode(Function function) {
    Integer obj = (Integer) function.getTable().get(variableName);

    // Load from global table
    if(obj == null) {
        this.setRegNum(function.getNewRegNum());
    } else {
        this.setRegNum(obj);
    }

    return function;
}

public boolean isGlobal(Function function) {
    Integer obj = (Integer) function.getTable().get(variableName);
    return obj == null;
}
}
```

```
package cminus_compiler.grammar;

import cminus_compiler.model.Token;
import lowlevel.CodeItem;
import lowlevel.Data;
import lowlevel.Function;

/**
 *
 * @authors Daniel Rees, Nathan Kallman
 * @version 1.0
 * File: VarDeclaration.java
 * Created: Feb 2015
 *
 * Description:
 */
public class VarDeclaration extends Declaration {

    int size;

    // Constructors
    public VarDeclaration() {
        this(0, null);
    }

    public VarDeclaration (Num number, Token ID) {
        this(number.getValue(), (String)ID.getTokenData());
    }

    public VarDeclaration(int size, String name) {
        this.size = size;
        this.declarationName = name;
    }

    // Getters
    public int getSize() {
        return size;
    }

    // Setters
    public void setSize(int size) {
        this.size = size;
    }

    // Public Methods
    @Override
    public String printTree(int indent) {
        StringBuilder builder = new StringBuilder();
        builder.append(indent(indent));
        builder.append("int ");
        builder.append(declarationName);
    }
}
```

```
        if(size > 0) {
            builder.append("[");
            builder.append(size);
            builder.append("]");
        }

        return builder.toString();
    }

    @Override
    public CodeItem gencode(Function function) {
        Data data = new Data(Data.TYPE_INT, declarationName);

        if(function != null) {
            function.getTable().put(declarationName, function.getNewRegNum());
        }

        return data;
    }
}
```