Individual Exercises - Lecture 16

1. Read the articles under additional references.
   Can be found here:
   http://www.techrepublic.com/article/examine-class-files-with-the-javap-command/5815354
   http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javap.html

2. Do Fischer et al exercise 3, 9, 11 on pages 441-443 (exercise 3, 10, 11 on pages 473-476 in GE)

   3. Design an AST node that implements the CondTesting interface to accommodate an if statement with no else clause.

   (a) How do you satisfy the CondTesting interface while indicating that the if statement has no else clause?
   The simplest solution is to create a node with an empty else clause that generates no instructions for the else branch. A downside of this design is that it creates an unnecessary endLabel, since it should always skip the else clause.

   (b) How is the code-generation strategy presented in Section 11.3.9 affected by your design?
   A benefit of the above design is that the existing code-generation strategy can be used without any changes.

   9. Some languages offer iteration constructs like C's for statement, which aggregates the loop's initialization, termination, and repetition constructs into a single construct. Investigate the syntax and semantics of C's for statement. Develop an AST node and suitable interface to represent its components. Design and implement its code-generation visitor.

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

   AST Node:
   ForLoopNode:
   children: initStatement, predicate, updateStatement, loopBody

Code Generation Procedure:

```
procedure visit(forLoopNode n) {
    doneLabel ← GenLabel()
    loopLabel ← GenLabel()

    n.getInitStatement().accept(this)
    call emitLabelDef(looplabel)

    n.predicate().accept(this)
    predicate ← n.predicate.getResultLocal()
    call emitBranchIfFalse(testExpressionResult, doneLabel)

    n.getLoopBody.accept(this)
    n.getUpdateStatement.accept(this)
    call emitBranch(loopLabel)

    call EmitLabelDef(doneLabel)
}
```

11. Section 11.3.10 generates code that emits the predicate test as instructions that occur before the loop body. Write a visitor method that generates the loop body's instructions first, but preserves the semantics of the WhileTesting node (the predicate must execute first).

```
procedure visit(While n){
      loopBodyLabel ← GenLabel()
      predicateLabel ← GenLabel()

      call emitBranch(predicateLabel) // Jump to predicate
      call EmitLabelDef(loopBodyLabel)
      n.getLoopBody.accept(this)

      call EmitLabelDef(predicateLabel)
      n.getPredicate().accept(this)
      predicateResult ← n.getPredicate().getResultLocal()
      // If true, jump back to body
      emitBranchIfTrue(predicateResult, loopBodyLabel)
}
```

3. Investigate how if-then-else, while, switch and other constructs in Java are implemented by the Java compiler javac using javap

   Compiling a .java file using “javac filename.java” and then running “javap -v filename.class” gives you some information about how the java compiler handles your code.

   The table below shows the result from a java file with a simple while loop.

(Part of) Compiled Code	Original Java Code

...
0: iconst_0
1: istore_0
2: iconst_0
3: istore_1
4: iload_0
5: iconst_5
6: if_icmpge 19
9: iload_1
10: iload_0
11: iadd
12: istore_1
13: iinc 0, 1
16: goto 4
19: iinc 1, 2
22: return
...
	public class Program{
	public static void main(){
	int i = 0;
	int a = 0;
	while(i < 5){
	a += i;
	i += 1;
	}
	a += 2;
	}
	}

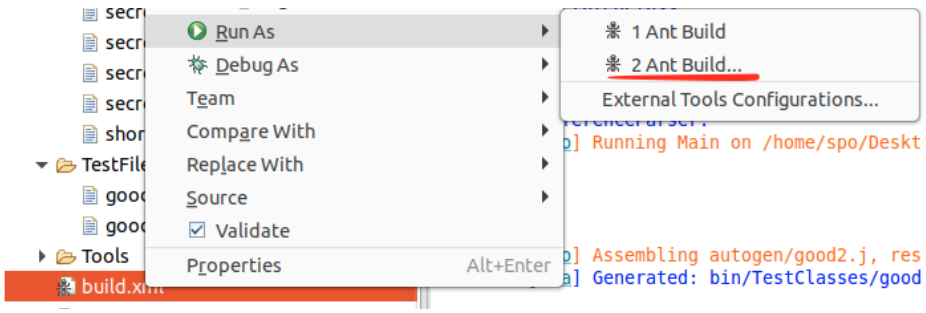
   Note that in the bytecode the loop predicate check is inverted (‘>=’ instead of ‘<’). If the inverted predicate evaluates to true, the loop is exited by jumping to **19.** (Can you see why this inversion might be useful?)

4. (optional but recommended) Follow the Lab associated with Crafting a CompilerChapter 11: Code Generation for a Virtual Machine

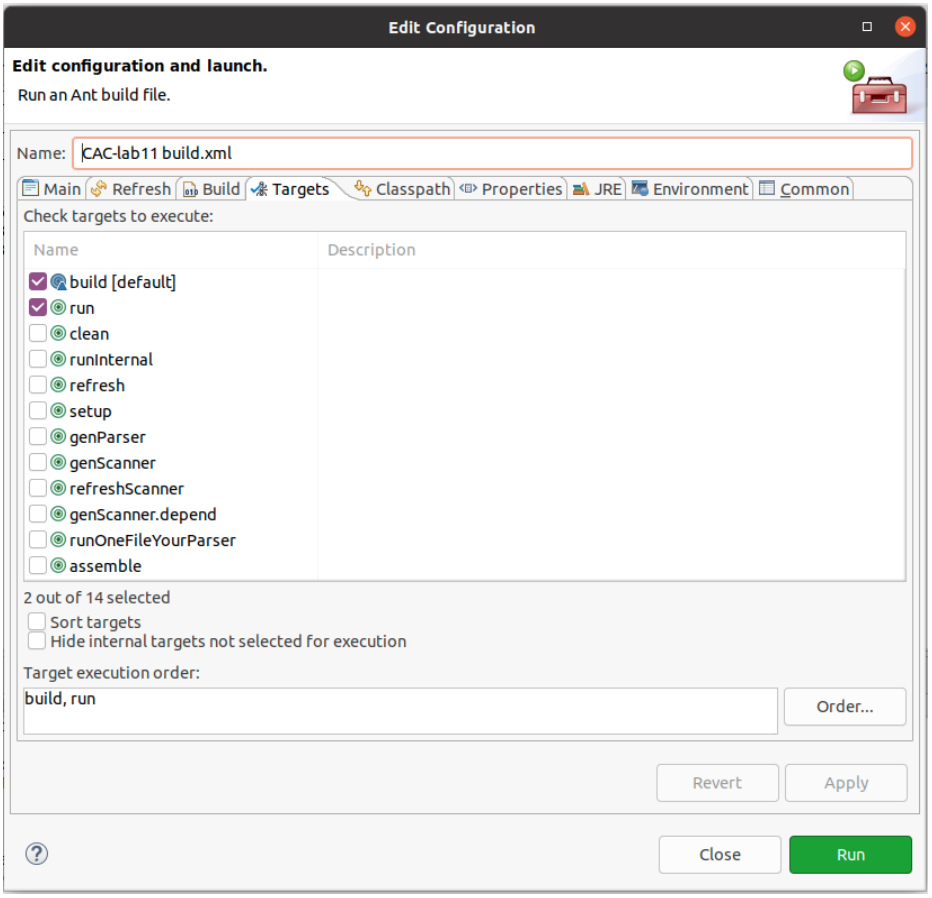
   The materials for this exercise are provided in the virtual machine found on moodle.

   Running the main:

   Right click the ANT build and click the second Ant build…:



   Check both ‘build’ and ‘run’ in the options. This will make sure that the code generator is first built and then run using the two files in the /TestFiles folder. Then click run.



The rest is explained in comments inside the ...src/submit/CodeGenVisitor.java.

NOTE: There is an error in the code provided by Fischer for this exercise. This prevents us from finding the main method. It can, however, be fixed by doing the following:

Replace the following code (inside /CAC-lab11/src/submit/CodeGenVisitor.java)

```
public void visit(ClassDeclaring c) {
    emitComment("CSE431 automatically generated code file");
    emitComment("");
    emit(c, ".class public " + c.getName());
    emit(".super java/lang/Object");
    emit("; standard initializer\n\n" + ".method public <init>()V\n"
    + "aload_0\n"
    + "invokenonvirtual java/lang/Object/<init>()V\n"
    + "return\n" + ".end method\n\n");
    emitComment("dummy main to call our main, we don't handle
arrays");
    skip(2);
    emit(".method public static main([Ljava/lang/String;)V\n"
```

```
    + ".limit locals 1\n" + ".limit stack3\n"
    + "invokestatic " + c.getName() + "/main431()V\n"
    + "return\n" + ".end method\n\n");
    visitChildren((AbstractNode) c);
}
```

With this code:

```
public void visit(ClassDeclaring c) {
    emitComment("CSE431 automatically generated code file");
    emitComment("");
    emit(c, ".class public TestClasses/" + c.getName());
    emit(".super java/lang/Object");
    emit("; standard initializer\n\n" + ".method public <init>()V\n"
    + "aload_0\n"
    + "invokenonvirtual java/lang/Object/<init>()V\n"
    + "return\n" + ".end method\n\n");
    emitComment("dummy main to call our main, we don't handle
arrays");
    skip(2);
    emit(".method public static main([Ljava/lang/String;)V\n"
    + ".limit locals 1\n" + ".limit stack3\n"
    + "invokestatic TestClasses/" + c.getName() +
    "/main431()V\n"
    + "return\n" + ".end method\n\n");
    visitChildren((AbstractNode) c);
}
```

5. (optional) In exercise 2 for Lecture 11 you created an AST for the little language defined in Figure 7.14 and in exercise 5, for lecture 14 you designed and implement a recursive interpreter for the language based on the Visitor Pattern. Design and implement a code generator visitor for the language generation JVM instructions (or rather Jasmin textual JVM programs)

   Example of a generation method for the E + T case of the Expression rule:

```
visit(PlusExprNode plusNode) {
      // Emit code that will push value of expression to TOS
      plusNode.getLeftExpression().accept(this);
      // Emit code that will push the value the right term to TOS
      plusNode.getRightTern.accept(this);
      emit("iadd");
```

Group Exercises - Lecture 16

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Do Fischer et al exercise 5, 8, 4, 10 on pages 441-443 (exercise 4, 6, 7, 12 on pages 473-476 in GE)

   5. Languages like Java and C allow conditional execution among statements using the if construct. Those languages also allow conditional selection of an expression value using the ternary operator, which chooses one of two expressions based on the truth of a predicate. For example, the expression b ? 3 : 5 has value 3 if b is true; otherwise, it has value 5.

   (a) Can the ternary operator be represented by an CondTesting node in the AST? If not, design an appropriate AST node for representing the ternary operator.
   Yes, a ternary operator selects between two expressions similar to how if else statements selects between two block statements.

   (b) How does the treatment of the ternary operator differ from the treatment of an if statement during semantic analysis?
   In most languages the ternary operator yields a value (ie. it is an expression), while an if-statement is just a statement.
   For this reason we must do type checking on the ternary operator to verify that the two branches return the same type, or atleast, type that can be coerced to the same type.
   Be aware that some languages (e.g., Scala and Rust) also treat if-else as expressions.

   (c) How does code generation differ for the two constructs?
   The main difference is, that code generation for the ternary expression must leave the value of the executed expression on TOS, whereas the if/else statement does not.

8. The semantics of a WhileTesting node accommodate Java and C's while constructs. Languages like Java have other syntax for iteration, such as Java's do-while construct. The body of the loop is executed, and the predicate is then tested. If the predicate is true, then the body is executed again. Execution continues to loop in this manner until the predicate becomes false. The sense of the predicate is consistent, in that iteration ceases when the predicate test is false. However, the do-while construct executes the loop body before the test. The CondTesting node represents constructs where the predicate is tested before the loop body is executed.

How would you accommodate the do-while statement? What changes are necessary across a compiler's phases, from syntax analysis to code generation?
The node could be annotated with a ConditionTestTime flag that signifies if the test is supposed to happen before or after the loop iteration.
When generating the code, you emit the loop label at the start of the loop body, and the conditional jump to the loop label at the end of the loop body.

4. Consider the code-generation strategy presented in Section 11.3.9. A conditional jump to false Label is taken, when the predicate is false, to skip over code that should execute only when the predicate is true. Rewrite the code-generation strategy so that the conditional jump is taken when the predicate is true, to skip over code that should execute only when the predicate is false.

11.3.9 Conditional Execution

```
/*   Visitor code for Marker (12)                                    */
procedure VISIT(CondTesting n)
    falseLabel ← GENLABEL()                                          (45)
    endLabel ← GENLABEL()
    call n.GETPREDICATE().ACCEPT(this)                               (46)
    predicateResult ← n.GETPREDICATE().GETRESULTLOCAL()
    call EMITBRANCHIFFALSE(predicateResult, falseLabel)              (47)
    call n.GETTRUEBRANCH().ACCEPT(this)                              (48)
    call EMITBRANCH(endLabel)
    call EMITLABELDEF(falseLabel)
    call n.GETFALSEBRANCH().ACCEPT(this)                             (49)
    call EMITLABELDEF(endLabel)
end
```

```
procedure visit(CondTesting n)
    trueLabel ← genLabel()
    endLabel ← genLabel()
    call n.getPredicate().accept(this)
    predicateResult ← n.getPredicate() .getResultLocal()
    call emitBranchIfTrue(predicateResult, trueLabel)
    call n.getFalseBranch() .accept(this)
    call emitBranch(endLabel)
    call emitLabelDef( trueLabel )
    call n.getTrueBranch().accept(this)
```

call emitLabelDef(endLabel)
end

10. Later versions of Java offer the so-called “enhanced for” statement. Investigate the syntax and semantics of that statement. Develop an AST node and suitable interface to represent the statement. Design and implement its code-generation visitor.
An enhanced for loop is an adaptation of a normal for loop for cases where the conditional check is trivial, i.e. looping over a collection, so its a for loop where the condition is implicit and not user accessible. Example:

```
List<Integers> integers = ...
for (Integer number: integers) { System.out.println(number) }
```

A common implementation technique is for a type to implement a specific interface so that it can return an iterator over its elements (e.g., Java’s iterable) which allows multiple iterations. Using this approach, the AST node proposed for loops in Section 11.3.10 can be reused with the predicate being an iterable type. For code generation an iterator for the type is retrieved using the iterable interface at the start of the loop, then at each iteration it is verified that the iterator has more elements using the hasNext method before next is called to retrieve an element which is assigned to the symbol used in the foreach loop. At the code generation level, the predicate condition would be generated based on the implementation of hasNext() method in the iterator code.

Java Overview: https://www.journaldev.com/13460/java-iterator
Java Iterable: https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html
Java Iterator: https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

3. Discuss and sketch a code generator for your language for the JVM
How do you translate your types into the ones for the JVM? How would some of your language constructs look in java bytecode, for example an if-stmt?