INTRODUCTION TO COMPILERS

FINAL EXAM QUESTIONS

PRESETED TO:
MULHERN

PRESENTED BY:
BRIAN DESARMO
SANTIAGO MELENDRO

UNIVERSITY OF WISCONSIN
NOVEMBER 2009

All relevant files are be located in the svn repository: /u/d/e/desarmo/shared/cs536-final/

Question 2

Optimizations in Javac

While not an aggressively optimizing compiler, javac implements some simple optimizations, including a constant folding optimization. Explore the optimizations that it does perform. Accumulate examples of these optimizations by compiling Java examples to bytecode files and using javap to disassemble the bytecode files and examine the code. Contrast the optimized code with the code that would result if javac performed no optimizations whatsoever. Briefly discuss how these optimizations might be performed.

The following optimizations have been identified:

Constant Folding

This optimization evaluates constant expressions and generates simplicied bytecode. Removing the unecessary operations at compile time. Doing this results in fewer cycles freeing processing time.

A simple example is presented on file ConstantFolding.java

```
public class ConstantFolding {

    static int doSomeMath() {
        return  0 + 1 + 2 + 3 + 4 + 5 +6 + 7 + 8 + 9;
    }
}
```

After compilation and dissasembly. (ConstantFolding.dissasembled)

```
Compiled from "ConstantFolding.java"
public class ConstantFolding extends java.lang.Object{
public ConstantFolding();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

static int doSomeMath();
  Code:
   0:   bipush  45
   2:   ireturn

}
```

it can be seen that at compile time javac pre-calculated all constant operations and replaces them with the appropriate values.
t can be seen that the optimization removed the code in the foo function leaving only the return instruction.

To implement this optimization javac is likely using DFS to traverse the AST and evaluating the subtrees rooted at expression nodes. Closer look at javac reveals that if fails to simplify the expression it stops any additional attempts for that expression.

Example code is presented on sm_ConstantFolding.java

```
public class sm_ConstantFolding {


    static int doSomeMath(int x) {
        return  0 + 1 + 2 + 3 + 4 + 5 + x + 6 + 7 + 8 + 9;
    }
}
```

which evaluates to:

```
Compiled from "sm_ConstantFolding.java"
public class sm_ConstantFolding extends java.lang.Object{
public sm_ConstantFolding();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

static int doSomeMath(int);
  Code:
   0:   bipush  15
   2:   iload_0
   3:   iadd
   4:   bipush  6
   6:   iadd
   7:   bipush  7
   9:   iadd
   10:  bipush  8
   12:  iadd
   13:  bipush  9
   15:  iadd
   16:  ireturn

}
```

The first part of the expression 0+1+2+3+4+5 was simplified by Constant Folding to bipush 15. However the part that follows the argument 6+7+8+9 could also be simplified at compile time reducing the number of instructions.

Dead Code Elimination

This optimization examines code in conditional statements. If the condition will always evaluate to false the code will never be executed and therefore it can be removed. Overall the optimization reduces the size of the binary file as well as improving efficient by avoiding the irrelevant conditional statement.

A simple example is presented on DeadCode.java

```
public class DeadCode {

    public static void foo() {

        if (false) {
            System.out.println("This should be eliminated");
        }
    }
}
```

After compilation and disassembly. (DeadCode.dissasembled)

```
Compiled from "DeadCode.java"
public class DeadCode extends java.lang.Object{
public DeadCode();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void foo();
  Code:
   0:   return

}
```

Question 16

Compile all Field Definitions to Nullary Methods : Implementation Project

The Java language prohibits forward references between fields and so does the simpleJ language. This is due to constraints in the way fields must have their values set in the static initializer.

Change your compiler so that it compiles all fields to methods which contain no arguments.
So,

```
integer field1 = field2;
integer field2 = field1;
```

becomes equivalent to:

```
integer field1 () { return with field2() }
integer field2 () { return with field1() }
```

Note that this would have the surprising effect for a user that field accesses are potentially non-terminating. But it does allow any field to have a forward reference to another field.

Clazz.g has been modified to create fields as methods allowing forward reference. At this point it needs additional validation but basic test case appears to work.

Input:

```
class Test1 {
    integer method1 = method2;
    integer method2 = method1;
}
```

Output:

```
class Test1 {

  integer method1 () {
    return with method2() }


  integer method2 () {
    return with method1() }

}
```

Question 17

Force a StackOverflowError during execution of a Java class with no main method

Construct a simple Java class, Overflower with no main methods such that invoking the Java interpreter on the class file will result in a StackOverflow-Error. Disassemble the corresponding class file and use the structure of the class file to explain why the error occurs.

Below is the java class for Overflower

```
public class Overflower {
    static void overflow() {
        overflow();
    }
}
```

This is the resulting byte codes when decompiled using javap

```
Compiled from "Overflower.java"
public class Overflower extends java.lang.Object{
public Overflower();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

static void overflow();
  Code:
   0:   invokestatic    #2; //Method overflow:()V
   3:   return

}
```

overflow() is calling itself never reaching the return code. For each recursive call to itself a new stack frame is pushed on to the call stack until a StackOverflowError occurs.

To implement this optimization javac is likely constant folding the condition in the statements and if the resulting condition is constant it no longer checks for the condition (eliminates it) when resulting condition is true. Or, it completely removes the statement when the resulting condition is false.