Tutorial 4 (Code Generation)
(to be covered in 2.5 tutorials)

1. (a) An idealized stack machine executes the code shown in Figure Q1. Assume that the initial values of `x` and `y` are `24` and `40` respectively. What is the value on top of the stack after the instruction at `l2` has been executed? You may assume that the stack machine has an instruction set as shown in Appendix A.

```
l0: load x
    load y
    ifeq l2
    load x
    load y
    ifle l1
    load x
    load y
    sub
    store x
    goto l0
l1: load y
    load x
    sub
    store y
    goto l0
l2: load x
    ret
```

Figure Q1

(b) What does this code compute in general? Write a piece of Java code that performs the same computation.

2. (a) Rewrite the idealized stack machine code shown in Figure Q1 as actual JVM bytecode. Comment on *three* main differences between the JVM bytecode for this example and the idealized stack machine code.

(b) How would the JVM bytecode for this example differ if the calculation used type `long` for variables `x` and `y` and the result rather than type `int`? Explain the reasons for your answer.

(Hint: you can check your answer to (a) and (b) by compiling the appropriate Java code and using `javap` to display the JVM bytecode in readable form).

3. Consider the Java ternary expression:

```
(a > b) ? c : d
```

which leaves the value of either `c` or `d` on top of the stack, depending on whether the condition is `true` or `false`. Assume that all variables are of type `int`, but do not assume that any of them are `0`. Explain how you would generate JVM bytecode for this ternary expression that uses only the `ifne` instruction for jumping (i.e. no other conditional jump or goto instructions are allowed). Note that in JVM bytecode, `ifne` pops one value off the stack and jumps to the target if that value is not equal to `0`.

4. Give Jimple-like 3-address code for the following Java statements and AST, identifying any temporary variables introduced.

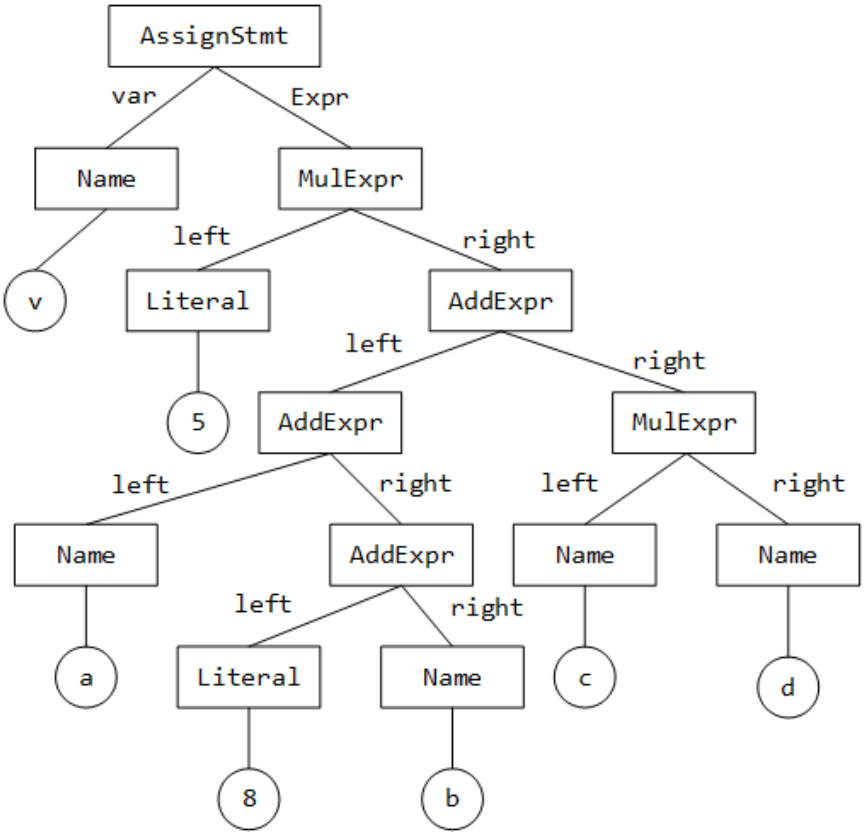
   (a) `x = (f(21) + a)*b;`

   (b) `s = (-b + Math.sqrt(r))/(2*a);`

   (c) `r = fib(n - 2) + fib(n - 1);`

   (d) `if (n<fib.length && fib[n] != 0) return fib[n];`

   (e)



5. (a) Discuss the generation of Jimple code for a Java `while` loop. Give an outline of the main steps involved in the code generation process. There is no need to give the actual statements to generate Jimple.

   (b) Discuss how to handle the generation of code for the `break` and `continue` statements in Java when used within a `while` loop. A `break` statement

terminates the closest enclosing loop and a `continue` statement skips the rest of the current iteration of the closest enclosing loop and re-evaluates the condition. There is no need to give the actual statements to generate Jimple.

6. Consider the following Java code. Show the sequence of frames on the stack when `r(3)` is executed, assuming the program starts by executing `main()`. Indicate the arguments/locals and operand stack in each frame.

```
class Example {
  void r(int x) {
    System.out.println("r(" + x + ")");
  }

  void q(int x) {
    p(x+1);
  }

  void p(int x) {
    if (x <= 2)
      q(x);
    else r(x);
  }
}

public class TestMain {
  public static void main(String[] args) {
    Example z = new Example();
    z.p(1);
  }
}
```

7. For each of the following three Java statements, (i) give Jimple-like 3-address code corresponding to the statement, identifying any temporary variables used, (ii) show a direct translation from the 3-address code to naïve code for an idealized stack machine, (iii) optimize the naïve stack machine code by eliminating redundant store/load pairs, explaining clearly in each case why it is (or is not) safe to perform the elimination. You may assume that the idealized stack machine has an instruction set as shown in Appendix A.

   (a) `y = x*x + 2*x + 1;`

   (b) `a = (1 - b*b) * c;`

   (c) `z  = (r + s)*(r + s) - u;`

8. Given the *interference graph* shown in Figure Q8, use *Chaitin's* heuristic algorithm to attempt to find a 3-colouring of the graph. Show the graph and the contents of the

stack at important stages of the algorithm and clearly indicate any nodes that are identified as spill candidates. Is it possible to allocate registers to the variables using only three registers r, s and t such that no register spilling is necessary?

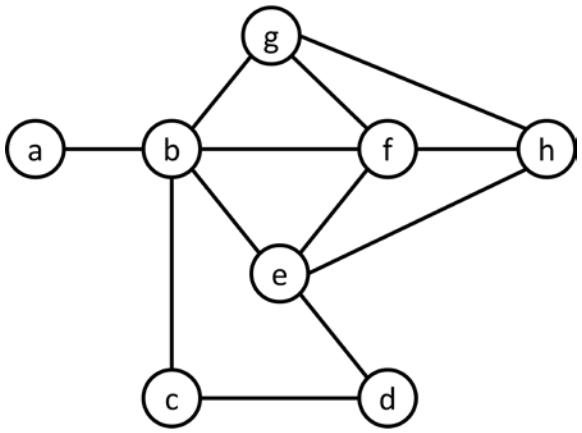


Figure Q8

Appendix A: Instruction Set of Idealized Stack Machine

An idealized stack machine has an instruction set that includes the following instructions:

`add`	pop top two values off the stack, add them, push the result
`dup`	duplicate the value on top of the stack
`ifeq`	let `y` be the value on top of the stack, `x` the one below it; pop `y` and `x` off the stack; if `x = y`, jump to the given label, otherwise continue execution at the next instruction
`ifle`	let `y` be the value on top of the stack, `x` the one below it; pop `y` and `x` off the stack; if $x \leq y$, jump to the given label, otherwise continue execution at the next instruction
`goto`	continue execution at the given label
`load`	load (push) given value or variable onto the stack
`mul`	pop top two values off the stack, multiply them, push the result
`ret`	pop top value off the stack and return it
`store`	pop top value off the stack and store it into the given variable
`sub`	let `y` be the value on top of the stack, `x` the one below it; pop `y` and `x` off the stack; calculate `x-y` and push onto the stack