Tutorial 4

Code Generation

(to be covered in 2.5 tutorials)

Question 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?

```
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

You may assume that the stack machine has an instruction set as shown in Appendix A

Instructions	x	y	Stack	Jump?
	24	40		
`load x`			24	
`load y`			24 40	
`ifeq l2`				continue
`load x`			24	
`load y`			24 40	
`ifle l1`				goto l1
`load y`			40	
`load x`			40 24	
`sub`			16	
`store y`		16		
`goto l0`				goto l0
`load x`			24	
`load y`			24 16	
`ifeq l2`				continue

Instr	x	y	Stack	Jump?
	24	16		
load x			24	
load y			24 16	
ifle l1				continue
load x			24	
load y			24 16	
sub			8	
store x	8			
goto l0				goto l0
load x			8	
load y			8 16	
ifeq l2				continue
load x			8	
load y			8 16	
ifle l1				goto l1

Instr	x	y	Stack	Jump?
	8	16		
load y			16	
load x			16 8	
sub			8	
store y		8		
goto l0				goto l0
load x			8	
load y			8 8	
ifeq l2				goto l2
load x			8	

The value left on top of the stack after executing the instruction at l2 is 8

Question 1(b)

What does this code compute in general?

- Greatest Common Divisor (GCD) of x and y

Write Java code that performs the same computation

```
int solveGCD(int x, int y) {
    while (x != y)
        if (x > y)
            x = x-y;
        else
            y = y-x;
    return x;
}
```

Question 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

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

JVM bytecode (from javap)

```
int solveGCD(int, int);
  Code:
   Stack=2, Locals=3, Args_size=3
```

```
0:     iload_1
1:     iload_2
2:     if_icmpeq 24
5:     iload_1
6:     iload_2
7:     if_icmple 17
10:    iload_1
11:    iload_2
12:    isub
13:    istore_1
14:    goto 0
17:    iload_2
18:    iload_1
19:    isub
20:    istore_2
21:    goto 0
24:    iload_1
25:    ireturn
```

Question 2(a)

Comment on *three* main differences between the JVM bytecode for this example and the idealized stack machine code

1. In JVM bytecode, arguments and local variables are allocated local slots at the beginning of the stack frame and are addressed by their index: for instance (i.e. non-static) methods, value of "**this**" is in slot 0, first argument x is in slot 1, second argument y is in slot 2
2. In JVM bytecode, the target of a jump instruction is the offset in the code rather than a label (the operand is actually a signed offset which is added to the address of the current instruction to obtain the target, but javap simply displays the address of the target)
3. In JVM bytecode, nearly all instructions have a type, e.g. iload_1, isub, istore_1 for integer

Question 2(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

```
long solveGCD(long x, long y) {
    while (x != y)
        if (x > y)
            x = x-y;
        else
            y = y-x;
    return x;
  }
```

JVM bytecode (from javap)

```
long solveGCD(long, long);
  Code:
   Stack=4, Locals=5, Args_size=3
0:     lload_1
1:     lload_3
2:     lcmp
3:     ifeq  26
6:     lload_1
7:     lload_3
8:     lcmp
9:     ifle  19
12:    lload_1
13:    lload_3
14:    lsub
15:    lstore_1
16:    goto  0
19:    lload_3
20:    lload_1
21:    lsub
22:    lstore_3
23:    goto  0
26:    lload_1
27:    lreturn
```

Question 2(b)

Differences

- Type long requires 64 bits, so value of "**this**" is in slot 0, first argument x is in slot 1 (& 2), second argument y is in slot 3 (& 4)
- Operand stack size is now 4 rather than 2
- Instructions that operate on long are prefixed with `l`, not `i`
- Conditional jump instructions can only be used with type integer; for type long, need to use `lcmp` instruction:

  pop operands y and x off the stack, then compare them; if x > y, push (integer) 1; if x < y, push (integer) -1; otherwise push (integer) 0; can then use conditional jump on integer value

Question 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 (integer) value off the stack and jumps to the target if that value is not equal to `0`

Question 3

- First, evaluate `a - b` and if the value is `0`, then `a = b` and `a > b` is false, so we can eliminate the `a = b` case by using an `ifne` instruction on the result of `a - b`
  - Because the top of stack value is consumed by the `ifne` instruction, that value must be duplicated prior to the test (as we need it if `a` $\neq$ `b`)
  - Suppose the target of the conditional jump is `l1` – the code between the conditional jump and `l1` will be executed if `a = b`, and that code should pop the stack to eliminate the duplicated `a - b` value and then load the value of d on the stack

Question 3

- In the case where `a ≠ b`, because `a – b` is a two's-complement integer value, the leftmost bit of that value indicates whether the result is positive or negative
  - The value of `a – b` can be resolved into a `0` or `1` by a logical right shift of 31 bits
  - If the result is `0`, then `a > b`, so the value of `c` must be left on the stack
  - If the result is 1, then `a < b`, so the value of `d` must be left on the stack
  - An `ifne` instruction can be used to test between these cases
- For unconditional jumps, we load (say) the constant `1` onto the stack followed by a `ifne` to the desired target

```
// Assume  a,  b,  c,  d are  in  slots  0 – 3
// Compute  a – b
iload_0
iload_1
isub
dup
// We are using label here, not actual offset
ifne l1
// Case a = b, remove duplicated  value and load d
pop
iload_3
// Unconditional  jump  to  l3 (we’re done)
iconst_1
ifne l3
```

```
l1:
  // Case a ≠ b, shift  to  get  sign  bit
  // Load constant 31 from the constant pool
  ldc #index_of_31
  iushr
  ifne l2
  // Case a > b, load  c and jump  to  l3 (done)
  iload_2
  iconst_1
  ifne l3
l2:
  // Case a < b, load d and we are done
  iload_3
l3:
```

Question 4

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

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

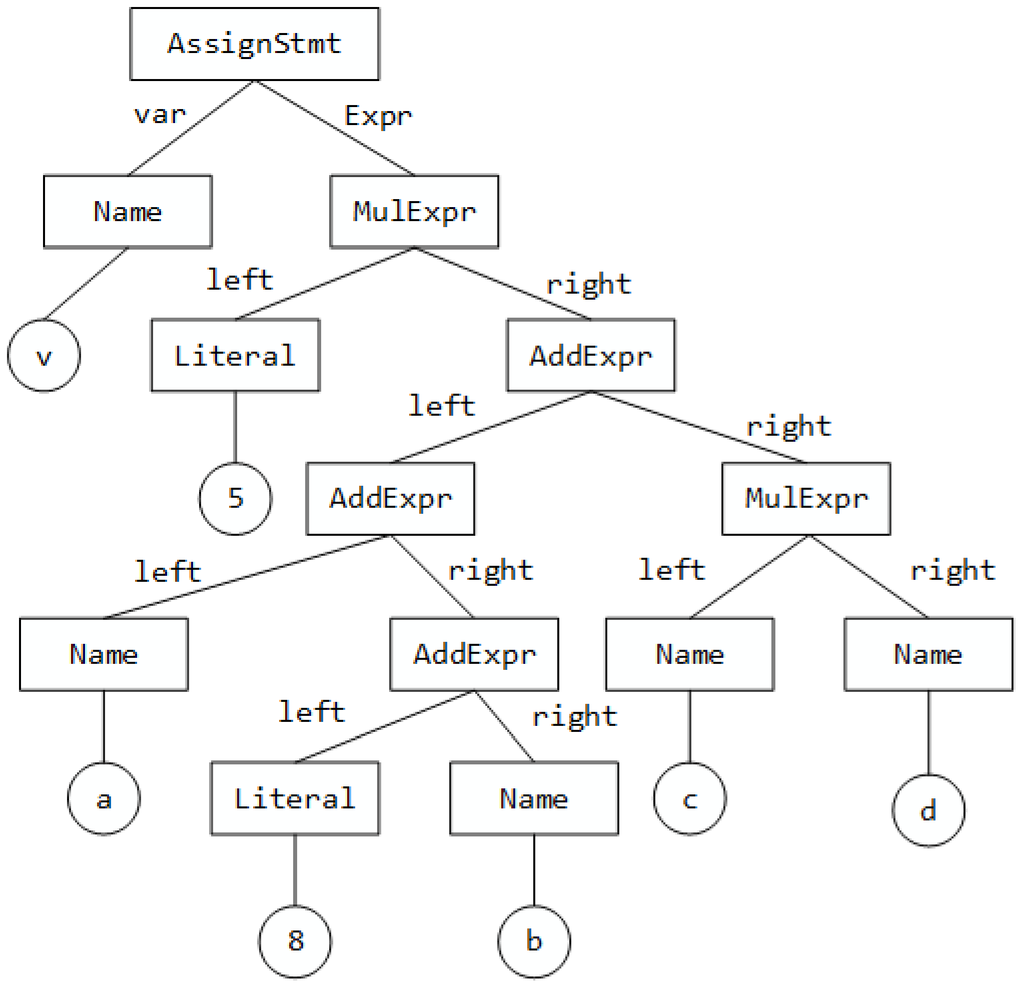
(b)
```
s = (-b + Math.sqrt(r))/(2*a);
```

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

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

Question 4

(e) Give Jimple-like 3-address code for the following AST:



Question 4

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

- Temporary variables will be shown as `$i0`, `$i1`, etc
- This is not strict Jimple (hence we say “Jimple-like”)
  - In strict Jimple, method calls should specify instance or static, the method reference, a base value (if not static) and a list of arguments

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

```
$i0 = f(21);
$i1 = $i0 + a;
x = $i1 * b;
```

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

```
$i0 = -b;
$i1 = Math.sqrt(r);
$i2 = $i0 + $i1;
$i3 = 2 * a;
s = $i2 / $i3;
```

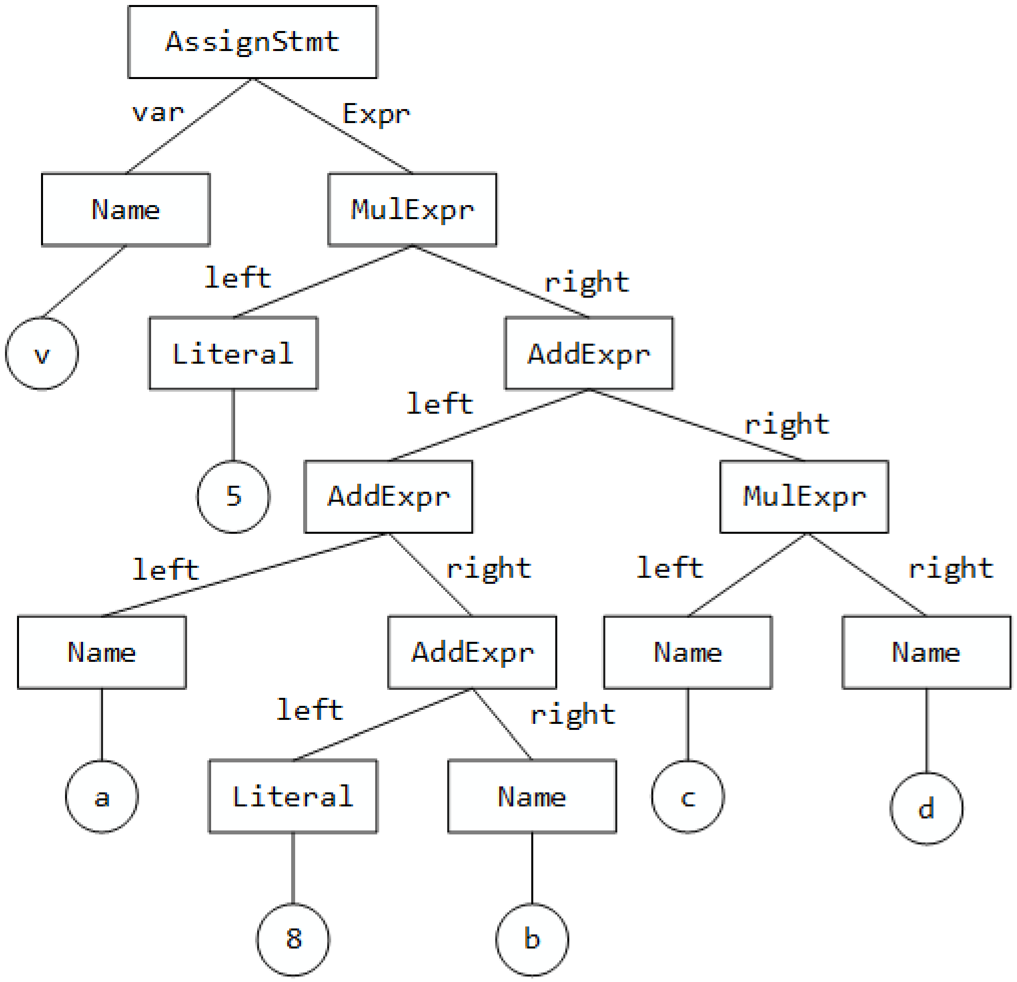
(c) `r = fib(n-2) + fib(n-1);`

```
$i0 = n-2;
$i1 = fib($i0);
$i2 = n-1;
$i3 = fib($i2);
r = $i1 + $i3;
```

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

```
   $i0 = length fib
   if n >= $i0 goto l1;
   $i1 = fib[n];
   if $i1 == 0 goto l1;
   return $i1;
l1:
```

(e)



```
$i0 = 8 + b;
$i1 = a + $i0;
$i2 = c*d;
$i3 = $i1 + $i2;
v = 5*$i3;
```

Question 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

Basic Idea

```
Java:

while(cond)
   body
```

=>

```
Jimple:

label0:
   // compute value of cond
   c = …
   if c == 0 goto label1
   // code for body
   goto label0
label1:
```

Question 5(a)

- There is a conditional jump to `label1` if the condition is false
  - As the target of this jump has not yet been generated, we need to use either backpatching or NOP padding
  - Generally, it is easier to use NOP padding and generate a NOP instruction to represent `label1`
- Also, the condition may be a complex expression which is translated recursively
  - It may involve assignments to temporary variables at different levels of recursion
  - It is not so easy to determine the unit that is the start of the condition
  - It is therefore also convenient to generate a NOP instruction to represent `label0`

Question 5(a)

- Outline of Code Generation for While Loop

```
label0 = new NOP statement
label1 = new NOP statement
emit statement label0
c = generate code for condition (storing in a
temporary variable if condition is a complex
expression)
emit statement if c == 0 goto label1
generate code for body
emit statement goto label0
emit statement label1
```

Question 5(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

Question 5(b)

- A break statement within the body should be translated into a jump to `label1`, while a continue statement should be translated to a jump to `label0`
- As `while` loops may be nested, we keep a mapping (HashMap) from `while` loops to their corresponding break targets (and similarly for continue targets)

```
label0 = new NOP statement
label1 = new NOP statement
// Assume nd is the AST node of the while loop
put (nd, label0) in HashMap for continue targets
put (nd, label1) in HashMap for break targets
// etc
```

Question 5(b)

- Code Generation for Break and Continue Statements
  - We assume that we can call `getEnclosingLoop()` in the AST to return the AST node of the closest enclosing `while` loop (defined during semantic analysis using an inherited attribute)
  - For a `break` statement, we look up the `while` loop node in the HashMap for break targets to get the corresponding target (`label1`) and generate a jump to that target
  - For a continue statement, we look up the `while` loop node in the HashMap for continue targets to get the corresponding target (`label0`) and generate a jump to that target

Question 6

Consider the following Java code

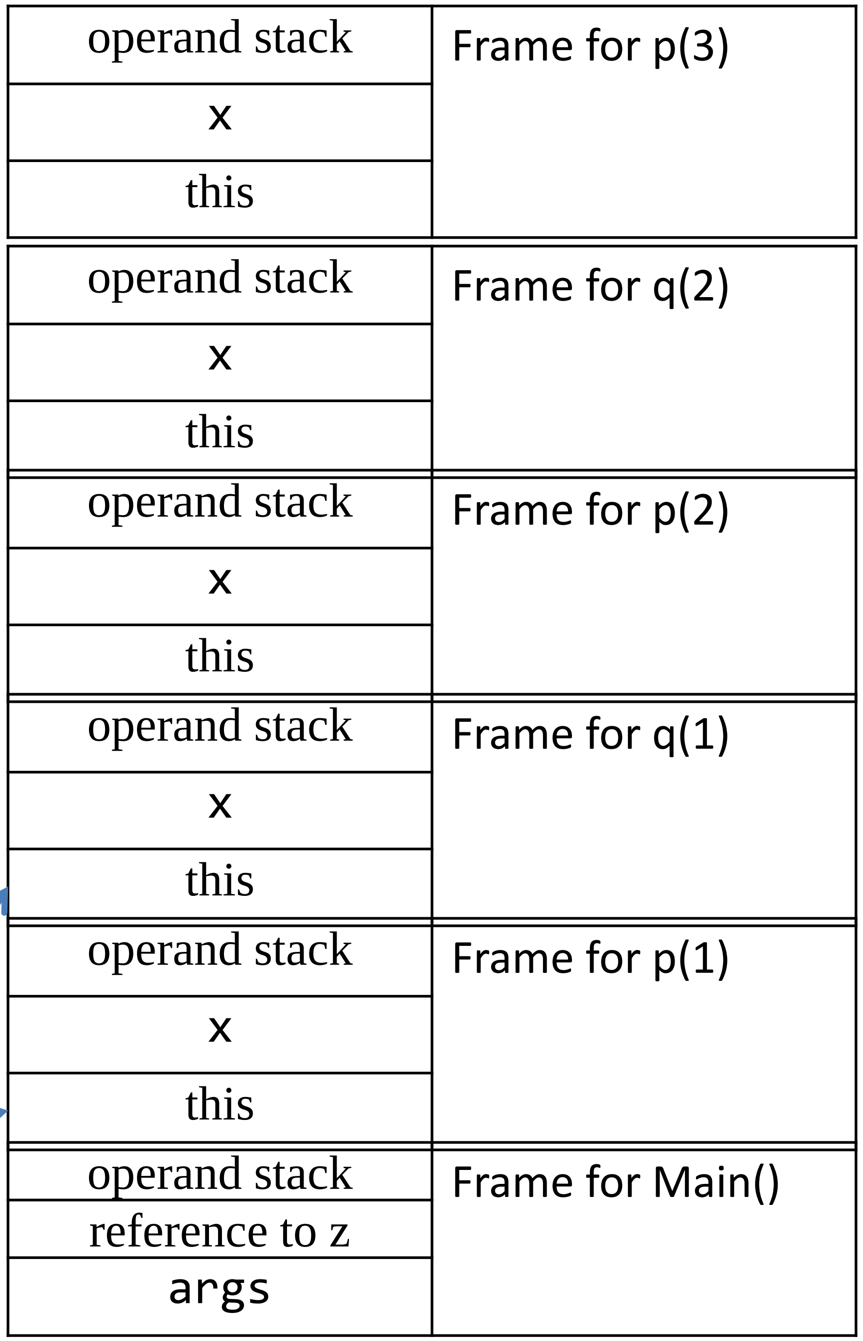
Show the sequence of frames on the stack when `r(3)` is executed, assuming the program starts by executing `main()` Indicate the arguments/locals and operand stack in each frame

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

```
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);
  }
}
```

Question 6

```
class Example {
    void r(int x)  {
      ...
    }

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

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

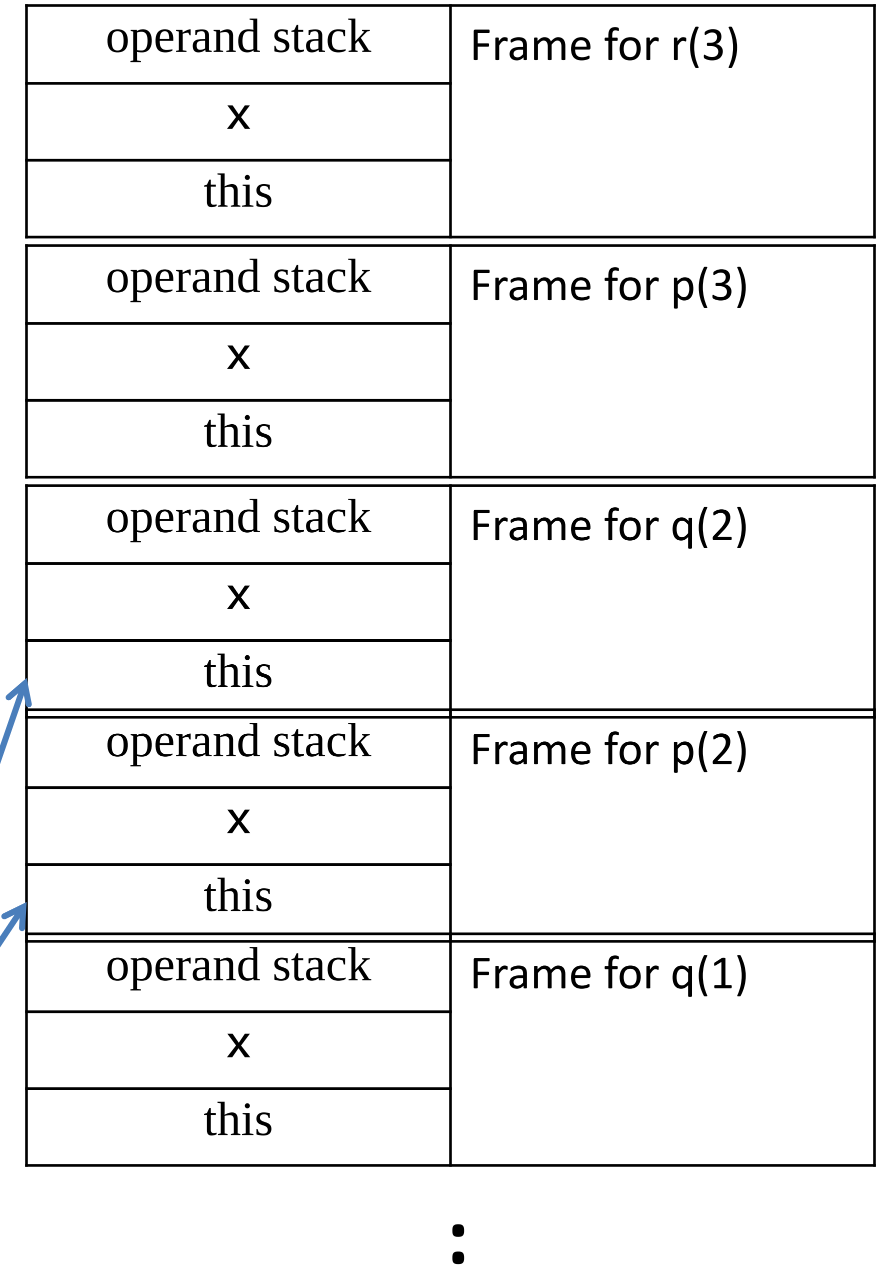


For non-static methods, reference "this" is stored in slot 0

Question 6

```
class Example {
  void r(int x)  {
    ...
  }

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

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

For non-static methods, reference "this" is stored in slot 0



Question 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

You may assume that the idealized stack machine has an instruction set as shown in Appendix A

- Direct Translation

```
x = y op z;
```

=>

```
load y
load z
op
store x
```

Question 7

(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

- store/load: a store instruction followed by a load instruction referring to the same local variable with no other uses – both the store and load instructions can be eliminated, and the value will simply remain on the stack
- store/load/load: a store instruction followed by two load instructions, all referring to the same local variable with no other uses – the three instructions can be eliminated and a dup instruction introduced to replace the second load by duplicating the value left on the stack after eliminating the store and the first load

Question 7

- Eliminating redundant patterns is trivial when the relevant instructions directly follow each other
- If there are intermediate instructions then some care must be taken to eliminate the pair safely
  - We compute the net stack height variation (nshv) and minimum stack height variation (mshv) for these intermediate instructions
  - Only if these are both equal to zero can we eliminate the pair (or triple)
  - If they are not zero, it may be possible to reorder the stack machine instructions to make them zero

Question 7(a)

```
y = x*x + 2*x + 1;
```

- Jimple-like 3-address code

```
$i0 = x * x;
$i1 = 2 * x;
$i2 = $i0 + $i1;
y = $i2 + 1;
```

- Naïve stack machine code

Instructions
load x load x mul store $i0
load 2 load x mul store $i1
load $i0 load $i1 add store $i2
load $i2 load 1 add store y

Question 7(a)

For `store $i0/load $i0`, nshv = 0 and mshv = 0, so this store/load pair may be eliminated

After `store $i0/load $i0` have been eliminated, `store $i1 /load $i1` are adjacent, so can be eliminated

`store $i2/load $i2` are adjacent, so can be eliminated

(shv = stack height variation)

Instructions	shv

`load x`
`load x`
`mul`
`store $i0`	

`load 2`
`load x`
`mul`
`store $i1`
	+1
	+2
	+1
	0

`load $i0`
`load $i1`
`add`
`store $i2`	

`load $i2`
`load 1`
`add`
`store y`	

Question 7(a)

```
y = x*x + 2*x + 1;
```

- Optimized stack machine code
- All temporary variables have been eliminated

Instructions
load x
load x
mul
load 2
load x
mul
add
load 1
add
store y

Question 7(b)

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

- Jimple-like 3-address code

```
$i0 = b * b;
$i1 = 1 - $i0;
a = $i1 * c;
```

- Naïve stack machine code

Instructions
load b load b mul store $i0
load 1 load $i0 sub store $i1
load $i1 load c mul store a

Question 7(b)

For `store $i0`/`load $i0`, nshv = 1 and mshv = 0, so this store/load pair may be not eliminated, as sub would wrongly subtract 1 from b*b

However, if we reorder the `load 1` to the beginning, then `store $i0`/`load $i0` are adjacent, so can be eliminated

`store $i1`/`load $i1` are adjacent, so can be eliminated

Instructions	**shv**

`load b`
`load b`
`mul`
`store $i0`	

`load 1`
`load $i0`
`sub`
`store $i1`
	+1

`load $i1`
`load c`
`mul`
`store a`	

(shv = stack height variation)

Question 7(b)

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

- Optimized stack machine code
- All temporary variables have been eliminated

Instructions
`load 1`
`load b`
`load b`
`mul`
`sub`
`load c`
`mul`
`store a`

Question 7(c)

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

- Jimple-like 3-address code

```
$i0 = r + s;
$i1 = $i0 * $i0;
z = $i1 – u;
```

- Naïve stack machine code

Instructions
load r load s add store $i0
load $i0 load $i0 mul store $i1
load $i1 load u sub store z

Question 7(c)

The triple `store $i0`/`load $i0` /`load $i0` `directly` follow each other and can be replaced by a dup

`store $i1`/`load $i1` are adjacent, so can be eliminated

No need to calculate shv

Instructions	**shv**

`load r`
`load s`
`add`
`store $i0`	

`load $i0`
`load $i0`
`mul`
`store $i1`	

`load $i1`
`load u`
`sub`
`store z`	

(shv = stack height variation)

Question 7(c)

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

- Optimized stack machine code
- All temporary variables have been eliminated

Instructions
load r
load s
add
dup
mul
load u
sub
store z

Question 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?

Question 8

- Register allocation with three registers r, s, t
- 3-colouring of the graph using *Chaitin's* algorithm

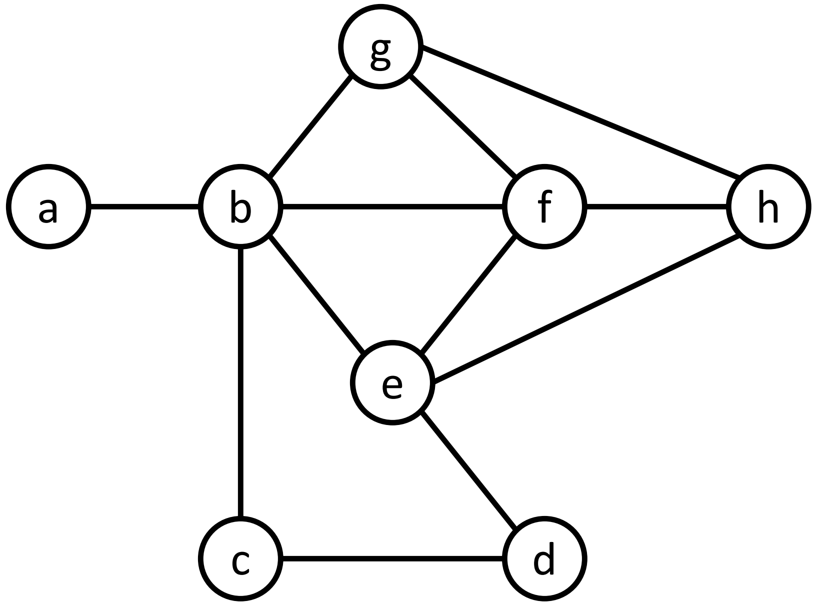
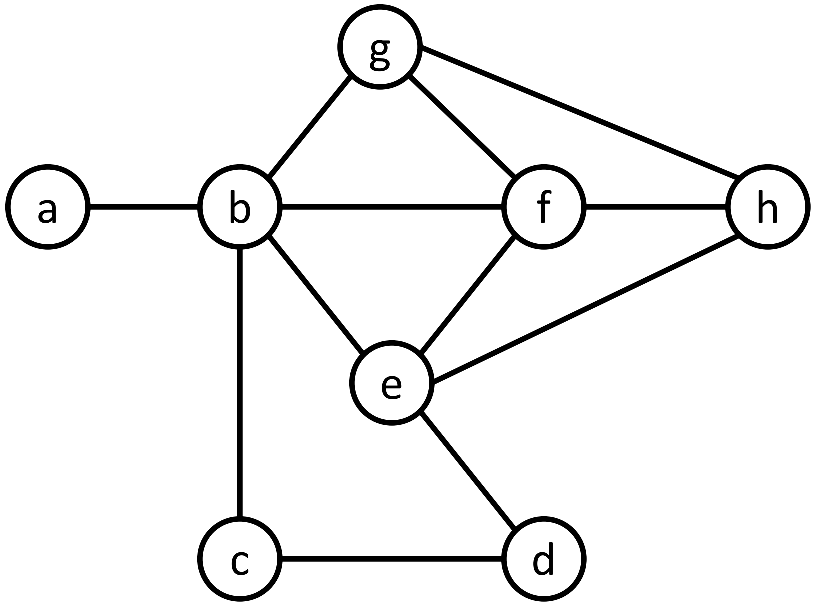


Figure Q8

Question 8

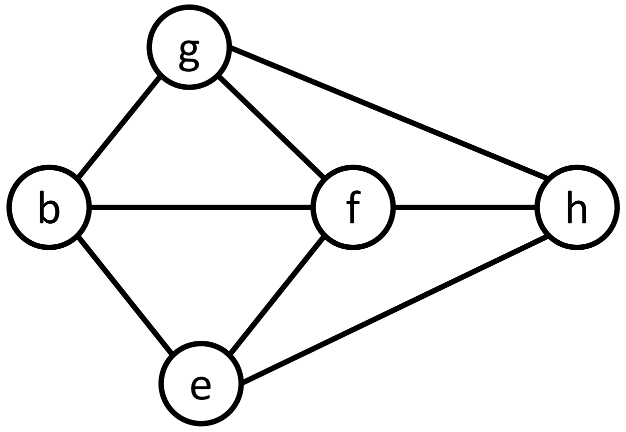
- Stack: empty



- Remove nodes with less than 3 neighbours (a, c, d) and push onto stack

Question 8

- Stack: a, c, d

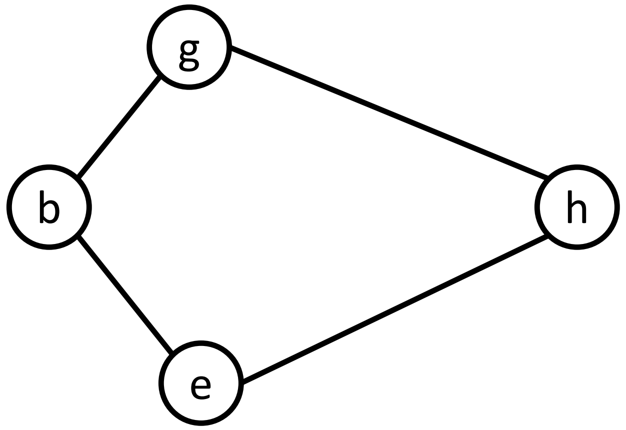


- All nodes have at least 3 neighbours, so choose as the spill candidate the node with the largest number of neighbours, remove and push onto stack: f is spill candidate (4 neighbours)

Question 8

- Stack: a, c, d, f*

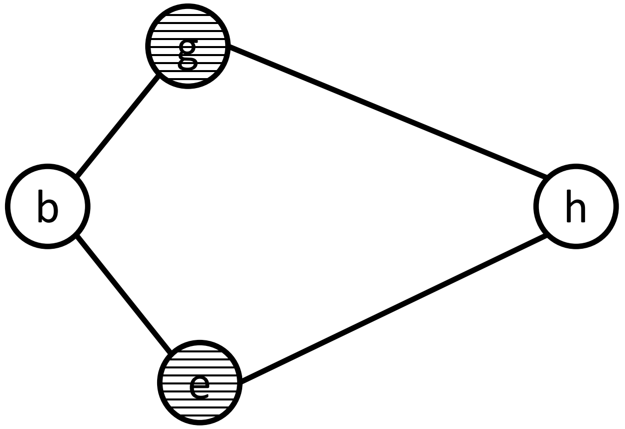
  * denotes spill candidate



- Remove nodes with less than 3 neighbours and push onto stack
- Stack: a, c, d, f*, b, e, g, h

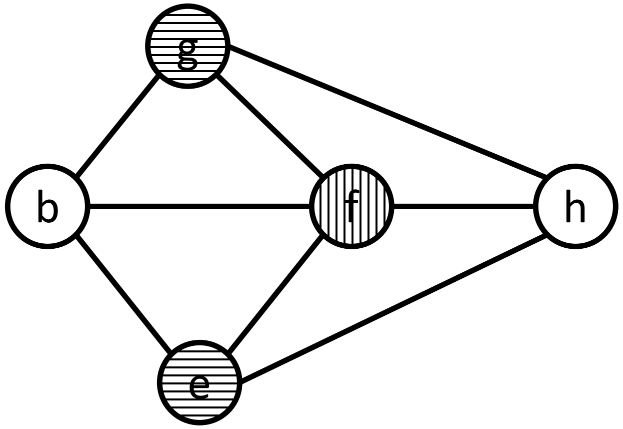
Question 8

- Stack: a, c, d, f*, b, e, g, h
- Pop nodes from stack one by one: h, g, e, b
- Allocate a register to the node not allocated to any of the neighbours
- Stack: a, c, d, f*



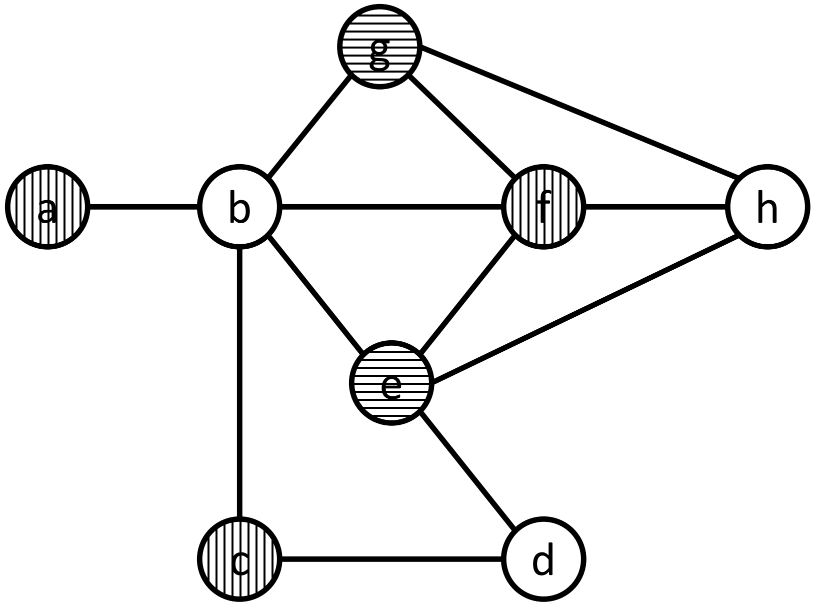
Question 8

- Stack: a, c, d, f*
- In this example, neighbours of the spill candidate only have two different colours, so can still find a colour for the spill candidate (generally this is not possible)
- Stack: a, c, d



Question 8

- Continue to pop remaining nodes: d, c, a
- Allocate a register not allocated to any of neighbours



- In this case, it is possible to allocate registers using 3 registers *r*, *s* and *t* such that there is no register spilling (i.e. no need to save/reload)