CMSC 430 Practice 2
Solutions

Use the following 3-address code and and Java stack code instructions for answering code generation questions.

3-addr Instruction	Effect
load R1 x	R1 ← x
store x R1	x ← R1
add R1 R2 R3	R1 ← R2 + R3
sub R1 R2 R3	R1 ← R2 - R3
mult R1 R2 R3	R1 ← R2 * R3
neg R1 R2	R1 ← -(R2)

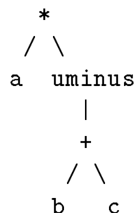
Java Stack Code	Effect
nop	none
ldc_int c	push constant *c* onto stack
iload index(x)	push local variable X onto stack
istore index(x)	pop stack, store in local variable X
iadd	pop 2 elems off stack, add, push
isub	pop 2 elems off stack, subtract, push
imult	pop 2 elems off stack, multiply, push
ineg	pop stack, negate, push
goto L	jump to handle L
ifeq L	pop stack, jump to handle L if zero
if_icmpeq L	pop 2 elems, jump to L if equal
if_icmpgt L	pop 2 elems, jump to L if 1st greater
dup	duplicate top of stack
pop	pop top of stack
swap	swap top two positions of stack

1. Intermediate representations.

   Consider the arithmetic expression:

   a * - (b + c)

   (a) Translate it into an AST



   (b) Translate it into 3-address code

```
load R1 b
load R2 c
add R3 R1 R2
neg R4 R3
load R5 a
mult R6 R5 R4
```

   (c) Translate it into Java stack code

```
iload index(a)
iload index(b)
iload index(c)
iadd
ineg
imult
```

   (d) Compare the three different forms

   ASTs are high-level representations, while 3-address code and Java byte codes are low-level representations. 3-address code operations on registers, while Java byte codes assume an implicit stack.

2. Code generation.

   You are generating code for a Java stack machine. You are given the following grammar attributes and helper functions:

Attribute	Holds
AstNode.code	list of instructions
Function	**Effect**
genInst(X)	create new instruction X returns handle to instruction
append(...)	concatenates lists of instructions

   (a) What grammar actions needed to generate code for a C-style FOR loop in the following production?

   stmt → FOR (E ; E ; E) stmtList | ...

```
forStmt := for (E1; E2; E3) stmt
{:
   Handle h1 = genInst( NOP );
   Handle h2 = genInst( NOP );
   forStmt.code = append(
        E1.code,
        h1,
        E2.code,
        genInst( IFEQ(h2) ),
        stmt.code,
        E3.code,
        genInst( GOTO(h1) ),
        h2);
:}
```

   (b) What grammar actions needed to generate code for the NOT_EQUALS expression in the following production? Your code should leave a 1 or 0 on the stack, depending on whether the expression is true or false. Note you can only use the Java instructions provided on the midterm.

   exp → exp_1 NOT_EQUALS exp_2
   { exp.code = ??; }

```
exp := exp1 NOT_EQUALS exp2
{:
   Handle h1 = genInst( NOP );
   Handle h2 = genInst( NOP );
   exp.code = append(
        exp1.code,
        exp2.code,
        genInst( IF_ICMPEQ(h1) ),
        genInst( LDC_INT(1) ),
        genInst( GOTO(h2) ),
        h1,
        genInst( LDC_INT(0) ),
        h2);
:}
```

3. Improved code generation.

   Frequently code generation and optimization are combined. Two examples are the Sethi-Ullman and DLS code generation algorithms for expression trees. Assume you are generating 3-address instructions.

   Consider the following two expressions:

   (a) a + b * c
   (b) (a * -b) + (c - (d + e))

   Generate 3-address code for each expression using the Sethi-Ullman and DLS algorithms, assuming a 1-cycle latency for loads (e.g, they take 2 cycles to complete). In case of ties, generate code for the left child first.

   When generating code, start with register R1 and increment (i.e., R1, R2, ...). When deciding which register to use, always use the lowest numbered register available. (I.e., always choose "add R1 R1 R2" instead of "add R2 R1 R2").

   For each expression, perform the following:

   (a) Build the abstract syntax tree for the expression and use the Sethi-Ullman labeling algorithm to mark the number of registers needed for each node of the tree (recall you are generating code for a load-store architecture, so each variable must be put into a register).
   (b) Generate code for the expression using the Sethi-Ullman algorithm. How many stalls are present?
   (c) How many registers are needed by the DLS algorithm to eliminate all stalls for loads (relative to Sethi-Ullman)?
   (d) Generate code for the expression using the DLS algorithm, using the minimal number of registers required to eliminate all stalls for loads.

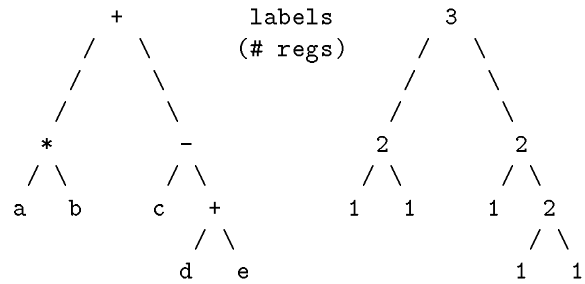
```
For:    a + (b * c)
```

```
Sethi-Ullman:               DLS:

  load R1 b                 load R1 b
  load R2 c                 load R2 c
* mult R1 R1 R2             load R3 a
  load R2 a                 mult R1 R1 R2
* add R1 R1 R2              add R1 R3 R1

  * --> 2 stalls
```

```
For:    (a * b) + (c - (d + e))
```




```
Sethi-Ullman:               DLS:

  load R1 a                 load R1 a
  load R2 b                 load R2 b
* mult R1 R1 R2             load R3 d
  load R2 d                 load R4 e
  load R3 e                 mult R1 R1 R2
* add R2 R2 R3              load R2 c
  load R3 c                 add R3 R3 R4
* sub R2 R3 R2              sub R2 R2 R3
  add R1 R1 R2              add R1 R1 R2

  * --> 3 stalls
```

Overall, DLS uses 1 more register than Sethi-Ullman to eliminate interlocks.

4. Compiling high-level languages

   Three classes of high-level programming language are Java, object-oriented (OO), and Functional programming (FP). For each of these classes answer the following questions:

   (a) Describe two distinguishing features of the language

   Java: implicit stack, objects
   OO: encapsulation, inheritance
   FP: higher-order functions, no side effects

   (b) Sketch how these features are implemented using compiler and run-time support.

   Java: generate stack code, run-time object tags
   OO: object tags, class records
   FP: function pointers, function environments on heap, copying

   (c) Describe how these features affect performance, and strategies for improving their performance.

   Java: map stack to registers, inline method calls
   OO: inlining, prefixing parent data/methods
   FP: inlining, exploiting tail recursion, updates

5. Optimizations

(a) How can compiler transformations improve a program?

By reducing program size or number of instructions executed, taking advantage of redundant computations, customizing general purpose code, or undoing high-level abstractions.

(b) What does the compiler need to consider when applying optimizations?

Safety, profitability, and applicability.

(c) What are the different scopes of compiler optimizations? What are the tradeoffs when considering what scope of optimizations to use?

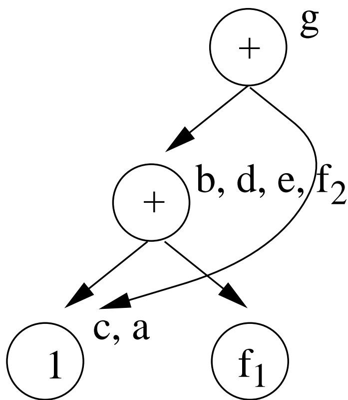
Peephole, local, global, or interprocedural. The basic tradeoff is more complexity and compile time for optimizations with greater scope.

6. Local optimizations

Consider the following code.

```
(1) a := 1
(2) b := f + a
(3) c := a
(4) d := f + a
(5) e := f + c
(6) f := b
(7) g := f + a
```

(a) Build a DAG for the code.



Note that assignment gives a node multiple labels, and may require renaming variables.

(b) Perform value numbering for the code. What are in the CODE, SYMBOLS, AVAIL, and CONSTANTS tables?

CODE			
Instructions		*resultValNum*	*isConst*
(1)	*a := 1*	*1*	*yes*
(2)	*b := f + a*	*3*	*no*
(3)	*c := a*	*1*	*yes*
(4)	*d := f + a*	*3*	*no*
(5)	*e := f + c*	*3*	*no*
(6)	*f := b*	*3*	*no*
(7)	*g := f + a*	*4*	*no*

SYMBOLS		
name	*ValNum*	*isConst*
a	*1*	*yes*
f	$2 \rightarrow 3$	*no*
b	*3*	*no*
c	*1*	*yes*
d	*3*	*no*
e	*3*	*no*
g	*4*	*no*

CONSTANTS	
ValNum	*bits*
1	*1*

AVAIL				
lhsValNum	*op*	*rhsValNum*	*resultValNum*	*Instr*
2	+	*1*	*3*	*(2)*
3	+	*1*	*4*	*(7)*

Note that expressions are stored as operations on value numbers.

7. Control flow analysis

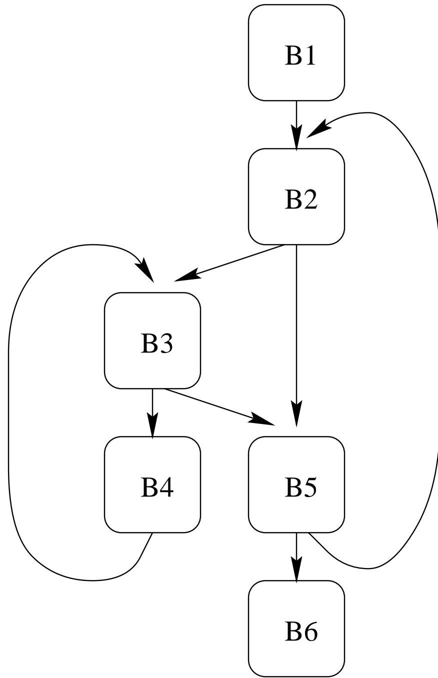
For the following problems, consider this code:

```
<S1>          a := 1
<S2>          b := 2
<S3>    L1:   c := a + b
<S4>          d := c - a
<S5>          if (...) goto L3
<S6>    L2:   d := b * d
<S7>          if (...) goto L3
<S8>          d := a + b
<S9>          e := e + 1
<S10>         goto L2
<S11>   L3:   b := a + b
<S12>         e := c - a
<S13>         if (...) goto L1
<S14>         a := b * d
<S15>         b := a - d
```

(a) What are the basic blocks?

B1 = { *S1, S2* }
B2 = { *S3, S4, S5* }
B3 = { *S6, S7* }
B4 = { *S8, S9, S10* }
B5 = { *S11, S12, S13* }
B6 = { *S14, S15* }

(b) What is the control flow graph?



(c) What is a reverse Postorder numbering of the basic blocks?

B1, B2, B3, B4, B5, B6

8. Reaching definitions

Reaching definitions for a point in the program p is defined as the set of definitions of a variable for which there is some path from the definition to p with no other definition of that variable. Calculate reaching definitions for the code in the control-flow graph problem.

(a) What is the dataflow equation for REACH?

(b) Show GEN, KILL for each basic block.

Note: For reaching definitions, we append to each variable definition its location to distinguish between multiple definitions to the same variable.

Block	GEN	KILL
B1	a1,b2	a1,b2,b11,a14,b15
B2	c3,d4	c3,d4,d6,d8
B3	d6	d4,d6,d8
B4	d8,e9	d4,d6,d8,e9,e12
B5	b11,e12	b2,e9,b11,e12,b15
B6	a14,b15	a1,b2,b11,a14,b15

(c) What do you initialize REACH to for each basic block?

REACH for each basic block is initialized to ∅ (no reaching definitions).

(d) Solve the data-flow equations in rPostorder. Show your work.

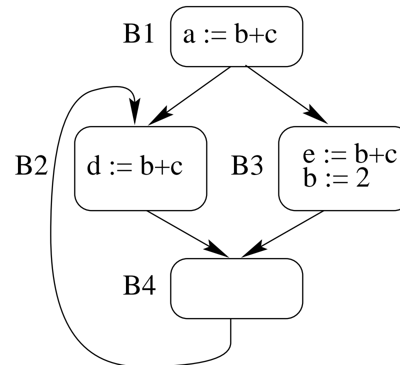
Solving the data flow equations in reverse postorder (B1,B2,B3,B4,B5,B6), we get the following:

Block	Data	Initial	Pass1	Pass2
B1	IN	∅	∅	∅
	OUT	∅	a1,b2	a1,b2
B2	IN	∅	a1,b2	a1,b2,c3,d4,d6,b11,e12
	OUT	∅	a1,b2,c3,d4	a1,b2,c3,d4,b11,e12
B3	IN	∅	a1,b2,c3,d4	a1,b2,c3,d4,d8,e9,b11,e12
	OUT	∅	a1,b2,c3,d6	a1,b2,c3,d6,e9,b11,e12
B4	IN	∅	a1,b2,c3,d6	a1,b2,c3,d6,e9,b11,e12
	OUT	∅	a1,b2,c3,d8,e9	a1,b2,c3,d8,e9,b11
B5	IN	∅	a1,b2,c3,d4,d6	a1,b2,c3,d4,d6,e9,b11,e12
	OUT	∅	a1,c3,d4,d6,b11,e12	a1,c3,d4,d6,b11,e12
B6	IN	∅	a1,c3,d4,d6,b11,e12	a1,c3,d4,d6,b11,e12
	OUT	∅	c3,d4,d6,e12,a14,b15	c3,d4,d6,e12,a14,b15

9. Available expressions

Available expressions is a data-flow analysis problem whose solution is used to guide global common subexpression. It calculates AVAIL, the expressions available at the beginning of each basic block.

Consider the following code. Assume that `b+c` is the only expression of interest:



(a) What is the data-flow equation for AVAIL?

$$AVAIL(b) = \bigcap_{x \in pred(b)} (GEN(x) \cup (AVAIL(x) - KILL(x)))$$

(b) Give GEN and KILL (needed by AVAIL) for each basic block.

Block	*GEN*	*KILL*
B1	*b+c*	∅
B2	*b+c*	∅
B3	∅	*b+c*
B4	∅	∅

(c) What value do you initialize AVAIL to?

Initialize AVAIL to ∅ for all basic blocks,

(d) Calculate AVAIL. Show all steps, including values for AVAIL and the order basic blocks are analyzed.

Pass1

$\text{AVAIL}(B1) = \emptyset$

$\text{AVAIL}(B2) = (\text{GEN}(B1) \cup (\text{AVAIL}(B1) - \text{KILL}(B1)))$
$\bigcap (\text{GEN}(B4) \cup (\text{AVAIL}(B4) - \text{KILL}(B4)))$

$$= (\{b+c\} \cup (\emptyset - \emptyset)) \bigcap (\emptyset \cup (\emptyset - \emptyset))$$
$$= \{b+c\} \bigcap \emptyset = \emptyset$$

$$\text{AVAIL}(B3) = (\text{GEN}(B1) \cup (\text{AVAIL}(B1) - \text{KILL}(B1)))$$
$$= (\{b+c\} \cup (\emptyset - \emptyset)) = \{b+c\}$$

$$\text{AVAIL}(B4) = (\text{GEN}(B2) \cup (\text{AVAIL}(B2) - \text{KILL}(B2))) \bigcap (\text{GEN}(B3) \cup (\text{AVAIL}(B3) - \text{KILL}(B3)))$$
$$= (\{b+c\} \cup (\emptyset - \emptyset)) \bigcap (\emptyset \cup (\{b+c\} - \{b+c\}))$$
$$= \{b+c\} \bigcap \emptyset = \emptyset$$

AVAIL(B3) changed, repeat

$\text{AVAIL}(B1) = \emptyset$

$$\text{AVAIL}(B2) = (\text{GEN}(B1) \cup (\text{AVAIL}(B1) - \text{KILL}(B1))) \bigcap (\text{GEN}(B4) \cup (\text{AVAIL}(B4) - \text{KILL}(B4)))$$
$$= (\{b+c\} \cup (\emptyset - \emptyset)) \bigcap (\emptyset \cup (\emptyset - \emptyset))$$
$$= \{b+c\} \bigcap \emptyset = \emptyset$$

$$\text{AVAIL}(B3) = (\text{GEN}(B1) \cup (\text{AVAIL}(B1) - \text{KILL}(B1)))$$
$$= (\{b+c\} \cup (\emptyset - \emptyset)) = \{b+c\}$$

$$\text{AVAIL}(B4) = (\text{GEN}(B2) \cup (\text{AVAIL}(B2) - \text{KILL}(B2))) \bigcap (\text{GEN}(B3) \cup (\text{AVAIL}(B3) - \text{KILL}(B3)))$$
$$= (\{b+c\} \cup (\emptyset - \emptyset)) \bigcap (\emptyset \cup (\{b+c\} - \{b+c\}))$$
$$= \{b+c\} \bigcap \emptyset = \emptyset$$

Pass2 yields no changes to AVAIL, stop

Summary of results:

	Init	*Pass1*	*Pass2*
AVAIL(B1)	$\emptyset$	$\emptyset$	$\emptyset$
AVAIL(B2)	$\emptyset$	$\emptyset$	$\emptyset$
AVAIL(B3)	$\emptyset$	$b+c$	$b+c$
AVAIL(B4)	$\emptyset$	$\emptyset$	$\emptyset$

10. Data-flow lattices

Prove the following properties of lattices:

(a) Show that $a \leq b$ and $b \leq c$ implies $a \leq c$

First, by definition of $\leq$*, we see* $a \leq b$ *gives us* $a \wedge b = a$.

Similarly, from $b \leq c$ *we get* $b \wedge c = b$.

Taking $a \wedge b = a$ *and replacing* b *with* $(b \wedge c)$*, we get* $a \wedge (b \wedge c) = a$.

Since $\wedge$ *is associative, we find* $a \wedge (b \wedge c) = (a \wedge b) \wedge c = a \wedge c$.

But since we also have $a \wedge (b \wedge c) = a$*, this implies* $a \wedge c = a$.

By definition of $\leq$*, this implies* $a \leq c$.

(b) Show that $a \leq (b \wedge c)$ implies $a \leq b$

First, we show $(b \wedge c) \leq b$.

By definition of $\leq$*, this is true if* $b \wedge (b \wedge c) = b \wedge c$.

Since $\wedge$ *is associative, we prove it using* $b \wedge (b \wedge c) = (b \wedge b) \wedge c = b \wedge c$.

Now combining $a \leq (b \wedge c)$ *and* $(b \wedge c) \leq b$ *gives us* $a \leq b$*, using the solution to the previous problem.*

11. Data-flow frameworks

(a) When estimating each of the following sets, tell whether too-large or too-small estimates are conservative. Explain your answer in terms of the intended use of information.

i. Available expressions
Too-small estimates are conservative ($\bot = \emptyset$)*. Common subexpression elimination would not replace some expressions if the estimate is too small, but the answer would be still correct. In comparison, performing CSE when an expression is not available may yield incorrect results.*

ii. Reaching definitions
Too-large estimates are conservative ($\bot = \{$ *all copy statements* $\}$ *).*
Copy propagation would be restricted if the estimate of copy statements reaching a point is too large, but the result would still be correct. In comparison, estimating too few copy statements reach a given point may enable copy propagation when the right-hand side of a missing copy statement differs from the right-hand side of the copy statements found, yielding incorrect results.

iii. Live variables
Too-small estimates are conservative ($\bot = \emptyset$)*. The opposite of variables changed by a procedure. Variables not modified by a procedure are left out of the KILL set for available expressions. Adding too few variables would result in a too-small estimate for available expressions, yielding a conservative approach to common subexpression elimination.*

(b) What properties are necessary to ensure an iterative data-flow analysis framework terminates?
The dataflow problem must be monotonic, I.e., $f(x \wedge y) \leq f(x) \wedge f(y)$*. All chains (sequences of less-than orderings) in the lattice must also have finite height (true for any lattice with finite height).*

(c) What properties are necessary to ensure an iterative data-flow analysis framework terminates with the meet-over-all-paths solution?
The dataflow problem must be distributive. I.e., $f(x \wedge y) = f(x) \wedge f(y)$.

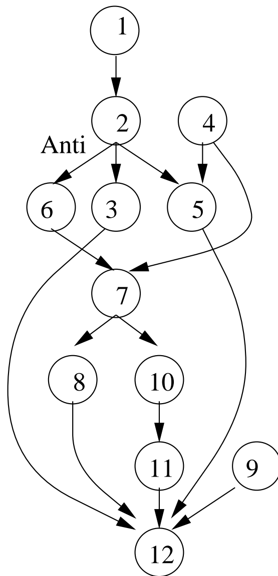
12. Instruction scheduling

Consider scheduling the code below using list scheduling. All instructions must complete before executing the *jmp* instruction. Assume the following instruction latencies:

- 2-cycle latency for load
- 1-cycle latency otherwise

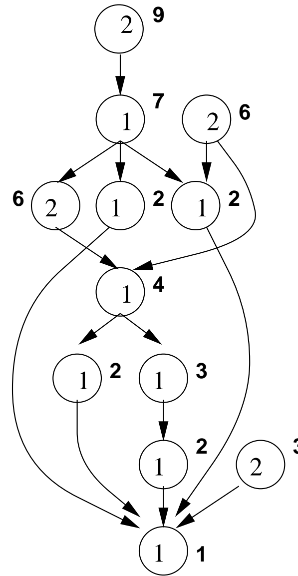
```
    <op>  <dst, s1, s2>
 1   load    r1, a
 2   add     r2, r1, #4
 3   store   x, r2
 4   load    r3, b
 5   mult    r4, r3, r2
 6   load    r1, c
 7   add     r5, r1, r3
 8   store   y, r5
 9   load    r6, d
10   mult    r7, r5, #1
11   store   z, r7
12   jmp
```

(a) Build the precedence graph for the instructions. Mark dependences as flow, anti, or output. You can ignore transitive dependences.



(b) Calculate the critical path for the instructions.

The following graph shows the critical path for a node as a number next to the node. The number inside a node indicates the latency of its instruction.



(c) Schedule the instructions for a single-issue processor, using forward list scheduling. Showing candidates instructions at each cycle. Prioritize candidates using 1) critical path, 2) latency of instruction, 3) number of children.

	1-issue		*2-issue*	
Time	*Cand*	*Inst*	*Cand*	*Inst*
1	*1,4,9*	*1*	*1,4,9*	*1,4*
2	*4,9*	*4*	*9*	*9*
3	*2,9*	*2*	*2*	*2*
4	*3,5,6,9*	*6*	*3,5,6*	*6,3*
5	*3,5,9*	*9*	*5*	*5*
6	*3,5,7*	*7*	*7*	*7*
7	*3,5,8,10*	*10*	*8,10*	*8,10*
8	*3,5,8,11*	*11*	*11*	*11*
9	*3,5,8*	*3*	*12*	*12*
10	*5,8*	*5*		
11	*8*	*8*		
12	*12*	*12*		

(d) Schedule the instructions as above, for a two-issue VLID processor.

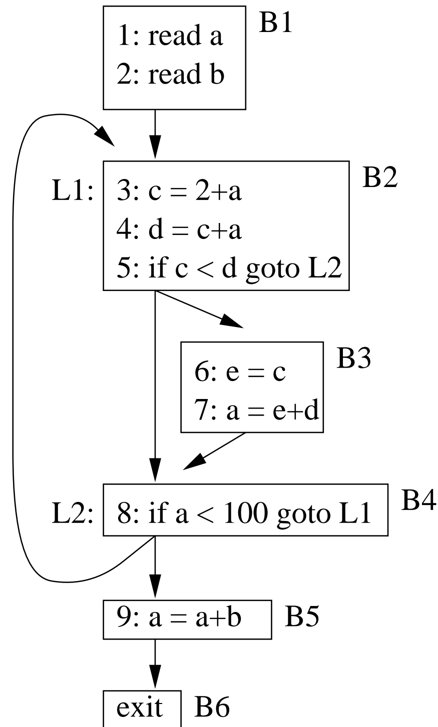
See above. Results show that issuing two instructions per cycle is sufficient to achieve maximum performance (equal to critical path).

(e) How could you change register assignments to improve instruction schedules in the code?

Using a new register instead of r1 in instruction 6 & 7 would avoid an antidependence between instruction 2 & 6.

13. Register allocation

Consider the flow graph below. Each statement is labeled by its statement number and each basic block is labeled in the upper right hand corner.



Use statement numbers or basic block numbers to indicate live ranges as appropriate.

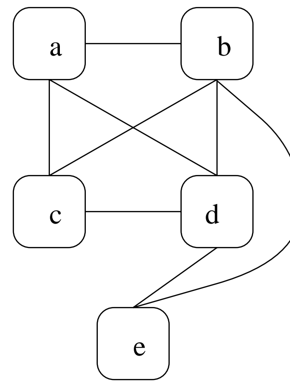
(a) What are the live ranges for a global top-down allocator?

a	*1–5, 7–9*
b	*2–9*
c	*3–6*
d	*4–7*
e	*6–7*

(b) Draw the interference graph for the live ranges.

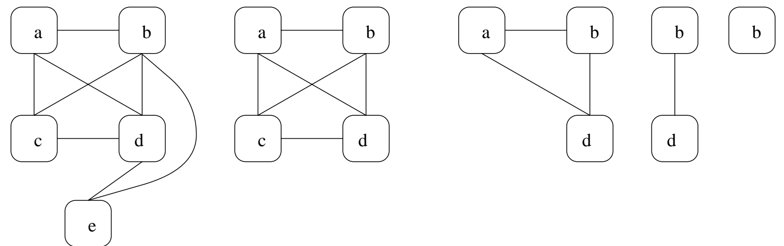
We assume that the compiler analysis is powerful enough to realize that there is no interference between the live range for a and b in the following example:

```
a = ...
b = a
  = b
```



(c) Use the graph-simplification method to find a coloring for this graph.

Assume we need 4 registers (since each node has at least 3 neighbors). Begin simplification of the graph by iteratively removing all nodes with fewer than 4 neighbors as follows:



Since we simplify the graph down to a single node, we know 4 colors are sufficient. We can now assign colors to nodes in reverse order. For instance, one assignment would be: b=r1, d=r2, a=r3, c=r4, e=r4.

(d) Can you color this graph with fewer colors?

No, since nodes a, b, c, d are completely connected. They will thus require at least 4 colors.

(e) If spilling is needed, which live range would be spilled first? Why?

Live range for b, since its spill cost is lowest due to its references not being in the loop (spill cost of instruction containing use or def is multiplied by loop nesting depth of instruction).

(f) Draw the spill code needed if the value for c is spilled.

```
3: c = 2+a
   store mem[c], c  // spill c
   load c, mem[c]   // reload c
4: d = c+a
   load c, mem[c]   // reload c
5: if c<d goto L2
   load c, mem[c]   // reload c
6: e = c
```

(g) What is rematerialization? Are there any such opportunities in the code?

Rematerialization refers to reducing the amount of spill code by recomputing a constant or near-constant value, rather than spilling and reloading it from memory. There are no opportunities in this example, since all values are non-constant.

14. Dependence analysis

Consider the following loop in Fortran:

```
A(N,N), B(N)
do i = 1,N
   do j = 2,N
      A(i,j) = A(i,j-1)+B(j)
   enddo
enddo
```

(a) What are the dependences in the loop?

From both references to A, there is a loop-carried dependence on the j loop with distance vector (0,1), since different iterations will access the same elements of A, with at least one of the references being a write.

Similarly, for the reference to B there is a loop-carried dependence on the i loop with distance vector (1,0), since different iterations of i access the same iterations of J each time..

(b) What reuse exists in the loop? For each, give type, reference(s), and loop carrying reuse.

Since arrays are stored column-major in Fortran, consecutive column elements are stored next to each other. As a result there is group-temporal reuse for A and spatial reuse for B for the j loop. There is spatial reuse for A and temporal reuse for B on the i loop.

(c) Which loop permutation is prefered? Calculate by estimating number of cache lines accessed by each loop. Show your work.

	loop i	loop j
A	$N/4 * N$	$N * N$
B	$1 * N$	$N/4 * N$
total	$N^2/4 + N$	$5N^2/4$

The preferred permutation is j, i, since loop i accesses fewer cache lines (has more locality).

(d) Are either of the loops parallel? Why?

The j loop is not parallel because it carries a dependence with distance vector (0,1). The i loop is also not parallel because it carries a dependence with distance vector (1,0).