

CMSC430 Spring 2009 Midterm 2 (Solutions)

Instructions

- You have until 4:45pm to complete the midterm.
- Feel free to ask questions on the midterm.
- One sentence answers are sufficient for the ``essay" questions.
- Use only the following 3-address code and Java stack code instructions for answering code generation questions.

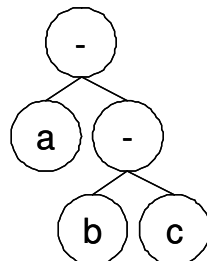
3-addr Instruction	Effect
load R1 x	$R1 \leftarrow x$
store x R1	$x \leftarrow R1$
add R1 R2 R3	$R1 \leftarrow R2 + R3$
sub R1 R2 R3	$R1 \leftarrow R2 - R3$
mult R1 R2 R3	$R1 \leftarrow R2 * R3$

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, subtract top from 2nd, 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
iflicmpeq L	pop 2 elems, jump to L if equal
iflicmpgt 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. (15 pts) 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
sub R3 R1 R2
load R4 a
sub R5 R4 R3
```

- c. Translate it into Java stack code

```
iload index(a)
iload index(b)
iload index(c)
isub
isub
```

- d. Name an advantage of using 3-address code instead of stack code.
Greater flexibility in reordering/eliminating instructions.

2. (16 pts) 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. A Ruby-style UPTO loop iterates $(\text{exp_2} - \text{exp_1} + 1)$ number of times. For example, *5 upto (2+6) do ... end* would iterate 4 times. Write syntax-directed actions needed to generate code for a UPTO loop for the following grammar production. Note you can only use the Java instructions provided at the beginning of the exam.

$$\text{stmt} \rightarrow \text{exp}_1 \text{ UPTO } \text{exp}_2 \text{ DO } \{ \text{stmtList} \} \text{ END}$$
$$\{ : \text{stmt.code} = ?? : \}$$

exp := exp₁ NAND exp₂

{:

Handle h1 = genInst(NOP); // end of UPTO loop

Handle h2 = genInst(NOP); // beginning of UPTO loop

exp.code = append(

exp₁.code,

genInst(ISTORE(index(1))),

exp₂.code,

```

    genInst( ISTORE( index(2) ) ),
    h2,
    genInst( ILOAD( index(1) ) ),
    genInst( ILOAD( index(2) ) ),
    genInst( ICMPGT( h1 ) ),
    stmtList.code,
    genInst( ISTORE( index(1) ) ),
    genInst( LDC_INT( 1 ) ),
    genInst( IADD() ),
    genInst( ISTORE( index(1) ) ),
    genInst( GOTO( h2 ) ),
    h1 );
:}

```

- b. The logical operator NAND is false only if BOTH of its operands are true. Write syntax-directed actions needed to generate code for a NAND expression in the following production. Your code should leave a 1 or 0 on the stack, depending on whether the resulting expression is true or false. Make sure you apply short circuiting.

$$exp \rightarrow exp_1 \text{ NAND } exp_2$$

$$\{ : exp.code = ?? : \}$$

```

exp := exp1 NAND exp2
{
    Handle h1 = genInst( NOP ); // NAND = true
    Handle h2 = genInst( NOP ); // end of NAND

    exp.code = append(
        exp1.code,
        genInst( IFEQ( h1 ) ),
        exp2.code,
        genInst( IFEQ( h1 ) ),
        genInst( LDC_INT( 0 ) ),
        genInst( GOTO( h2 ) ),
        h1;
        genInst( LDC_INT( 1 ) ),
        h2 );
:}

```

3. (8 pts) Compiling and optimizing high-level languages

Object-oriented programming languages such as C++ and Java use classes and inheritance to improve programmer productivity. Remember that inheritance allows objects of one class to be used in place of objects in a parent class.

- a. How are objects and classes implemented in the run-time environment?

Class records and object records.

- b. What compiler-generated code is needed to support inheritance?

Code to check class type via object tags.

- c. What is prefixing?

Assigning object fields and method references to the same index for subclasses, so dynamic checks of class type will not be needed.

4. (6 pts) Compiler optimizations

- a. Give an example of a machine-independent optimization, and explain why it is machine-independent

Eliminating redundancy, since executing fewer instructions usually improves performance

- b. Give an example of a machine-dependent optimization, and explain why it is machine-dependent

Register allocation, since number of registers is machine dependent

- c. Why would a compiler writer decide not to implement a particular optimization?

Insufficient improvement, or rarity of occurrence

5. (6 pts) Directed acyclic graphs

Consider the following code.

(1) $a := b + c$

(2) $b := b + c$

(3) $c := b + c$

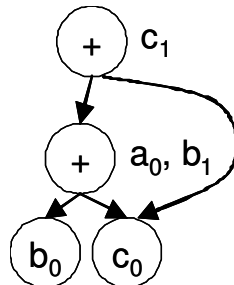
- a. Perform renaming for the code

$a_0 := b_0 + c_0$

$b_1 := b_0 + c_0$

$c_1 := b_1 + c_0$

- b. Build a DAG for the renamed code



- c. How do DAGs support optimizations?
By exposing common subexpressions

6. (6 pts) Control flow analysis

For the following problems, consider this code:

```
<S1>      a := b
<S2>  L1:  b := c
<S3>      if (...) goto L2
<S4>      c := d
<S5>      if (...) goto L1
<S6>  L2:  d := a
```

- a. What are the basic blocks?

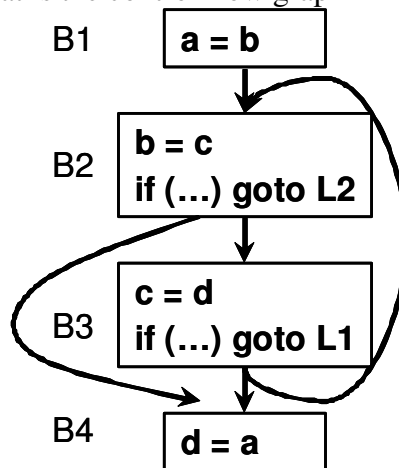
B1 = { S1 }

B2 = { S2, S3 }

B3 = { S4, S5 }

B4 = { S6 }

- b. What is the control flow graph



- c. What is a reverse Postorder numbering of the basic blocks?

B1, B2, B3, B4 or B1, B2, B4, B3

- d. Why do compilers use basic blocks?

To find regions of code with no control flow where local optimizations may be performed. To reduce the size of the control flow graph. To allow effects of multiple statements to be summarized together.

7. (6 pts) Local information

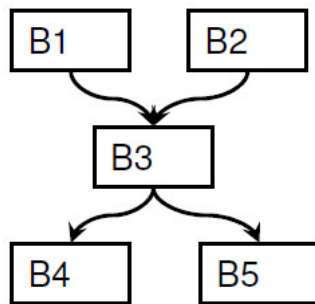
Consider the following basic block for live variables:

b = a
d = b
a = c
e = d
a = e

- What is GEN for the basic block?
{ a, c }
- What is KILL for the basic block?
{ a, b, d, e }

8. (6 pts) Data-flow analysis

Consider the following control flow graph:



Assume you are given IN/OUT for B1,B2,B4,B5, and GEN/KILL for B3. Show the data-flow equations for IN/OUT for B3 (e.g., $IN(B3) = OUT(B1)$).

- For forward data-flow problems

$$IN(B3) = OUT(B1) \wedge OUT(B2)$$

$$OUT(B3) = GEN(B3) \cup (IN(B3) - KILL(B3))$$
- For backwards data-flow problems?

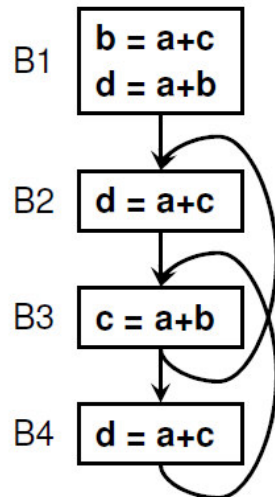
$$OUT(B3) = IN(B4) \wedge IN(B5)$$

$$IN(B3) = GEN(B3) \cup (OUT(B3) - KILL(B3))$$

9.

10. (20 pts) Available expressions

Consider the following control flow graph for available expressions:



- a. Calculate GEN/KILL for each basic block

	GEN	KILL
B1	a+c, a+b	a+b
B2	a+c	∅
B3	a+b	a+c
B4	a+c	∅

- b. Solve available expressions, showing IN/OUT for each pass

		Init	Pass1	Pass2	Pass3
B1	IN	∅	∅	∅	∅
	OUT	∅	a+c, a+b	a+c, a+b	a+c, a+b
B2	IN	∅	∅	a+b	a+b
	OUT	∅	a+c	a+c, a+b	a+c, a+b
B3	IN	∅	∅	a+c, a+b	a+c, a+b
	OUT	∅	a+b	a+b	a+b
B4	IN	∅	a+b	a+b	a+b
	OUT	∅	a+c, a+b	a+c, a+b	a+c, a+b

11. (12 pts) Data-flow analysis frameworks

Recall that $\wedge$ is used in data-flow iterative analysis to combine information where paths merge.

- a. Using properties of the $\wedge$ operator, prove $a \leq b$ and $b \leq a$ imply $a = b$.

$a \leq b$ implies $a \wedge b = a$, and $b \leq a$ implies $b \wedge a = b$.

Since $\wedge$ is commutative, $a \wedge b = b \wedge a$, so $a = a \wedge b = b \wedge a = b$, and $a = b$.

- b. For reaching definitions, pick values for a, b, c for which $a > b$ and $b > c$.

$a = \{a, b, c\}, b = \{a, b\}, c = \{b\}$

- c. For very busy expressions, pick values for a, b for which neither $a \leq b$ or $b \leq a$ are true?

$a = \{a+b\}, b = \{a+c\}$

- d. How is $\perp$ defined in terms of the $\wedge$ operator?

$\perp \wedge a = \perp$

- e. What is monotonicity and why is it important for iterative data-flow problems?

Monotonicity means $f(x \wedge y) \leq f(x) \wedge f(y)$, and is required for guaranteeing iterative data-flow problems will converge.