

CMSC 430 Practice Problems 2

Use 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	$R1 \leftarrow R2 * R3$
neg R1 R2	$R1 \leftarrow -(R2)$

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:

$$a * - (b + c)$$

- Translate it into an AST
- Translate it into 3-address code
- Translate it into Java stack code
- Compare the three different forms

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

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

$\text{stmt} \rightarrow \text{FOR} (E ; E ; E) \text{stmtList} | \dots$

- 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.

$\text{exp} \rightarrow \text{exp}_1 \text{NOT_EQUALS} \text{exp}_2$
 $\{ \text{exp.code} = ??; \}$

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:

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

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).

When generating code, start with register $R1$ and increment (i.e., $R1, R2, \dots$). 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:

- 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).
- Generate code for the expression using the Sethi-Ullman algorithm. How many stalls are present?
- 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.

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:

- Describe two distinguishing features of the language
- Sketch how these features are implemented using compiler and run-time support.
- Describe how these features affect performance, and strategies for improving their performance.

5. Optimizations

- How can compiler transformations improve a program?
- What does the compiler need to consider when applying optimizations?
- What are the different scopes of compiler optimizations? What are the tradeoffs when considering what scope of optimizations to use?

6. Local optimizations

Consider the following code.

- `a := 1`
- `b := f + a`
- `c := a`
- `d := f + a`
- `e := f + c`
- `f := b`
- `g := f + a`

- Build a DAG for the code.
- Perform value numbering for the code. What are in the CODE, SYMBOLS, AVAIL, and CONSTANTS tables?

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

```

- What are the basic blocks?
- What is the control flow graph?
- What is a reverse Postorder numbering of the basic blocks?

8. Reaching definitions

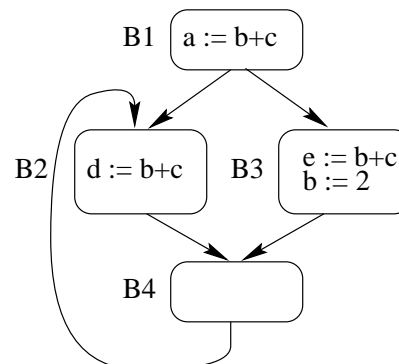
Reaching definitions for a point in the program p is defined as the set of definitions of a variable for which there is some path from the definition to p with no other definition of that variable. Calculate reaching definitions for the code in the control-flow graph problem.

- What is the dataflow equation for REACH?
- Show GEN, KILL for each basic block.
- What do you initialize REACH to for each basic block?
- Solve the data-flow equations in rPostorder. Show your work.

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?
- (b) Give GEN and KILL (needed by AVAIL) for each basic block.
- (c) What value do you initialize AVAIL to?
- (d) Calculate AVAIL. Show all steps, including values for AVAIL and the order basic blocks are analyzed.

10. Data-flow lattices

Prove the following properties of lattices:

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

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
 - ii. Reaching definitions
 - iii. Live variables
- (b) What properties are necessary to ensure an iterative data-flow analysis framework terminates?
- (c) What properties are necessary to ensure an iterative data-flow analysis framework terminates with the meet-over-all-paths solution?

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

```

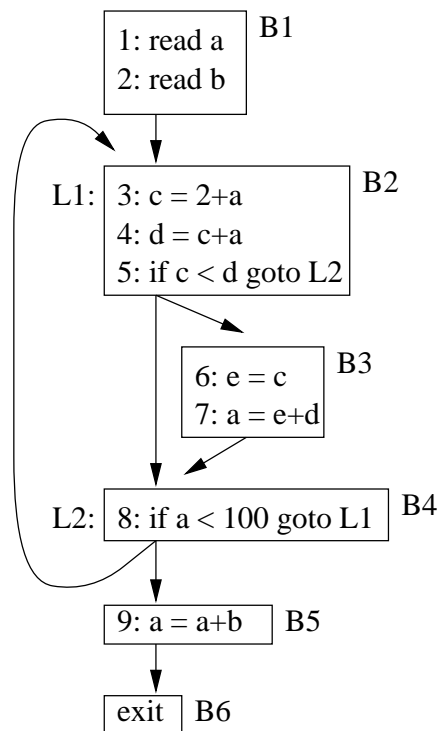
<op> <dst, s1, s2>
1  load  r1, a
2  add   r2, r1, #4
3  store x, r2
4  load  r3, b
5  mult  r4, r3, r2
6  load  r1, c
7  add   r5, r1, r3
8  store y, r5
9  load  r6, d
10 mult  r7, r5, #1
11 store z, r7
12 jmp

```

- (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.
- (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.
- (d) Schedule the instructions as above, for a two-issue processor.
- (e) How could you change register assignments to improve instruction schedules in the code?

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?

- (b) Draw the interference graph for the live ranges.
- (c) Use the graph-simplification method to find a coloring for this graph.
- (d) Can you color this graph with fewer colors?
- (e) If spilling is needed, which live range would be spilled first? Why?
- (f) Draw the spill code needed if the value for c is spilled.
- (g) What is rematerialization? Are there any such opportunities in the code?

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?
- (b) What reuse exists in the loop? For each, give type, reference(s), and loop carrying reuse.
- (c) Which loop permutation is preferred? Calculate by estimating number of cache lines accessed by each loop. Show your work.
- (d) Are either of the loops parallel? Why?