CMSC 430 (Spring’06)
Practice Problems 2

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

<table>
<thead>
<tr>
<th>3-addr Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>load R1 x</td>
<td>R1 ← x</td>
</tr>
<tr>
<td>store x R1</td>
<td>x ← R1</td>
</tr>
<tr>
<td>add R1 R2 R3</td>
<td>R1 ← R2 + R3</td>
</tr>
<tr>
<td>sub R1 R2 R3</td>
<td>R1 ← R2 - R3</td>
</tr>
<tr>
<td>mult R1 R2 R3</td>
<td>R1 ← R2 * R3</td>
</tr>
<tr>
<td>neg R1 R2</td>
<td>R1 ← -(R2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Java Stack Code</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>none</td>
</tr>
<tr>
<td>ldc_int c</td>
<td>push constant c onto stack</td>
</tr>
<tr>
<td>iload index(x)</td>
<td>push local variable X onto stack</td>
</tr>
<tr>
<td>istore index(x)</td>
<td>pop stack, store in local variable X</td>
</tr>
<tr>
<td>iadd</td>
<td>pop 2 elms off stack, add, push</td>
</tr>
<tr>
<td>isub</td>
<td>pop 2 elms off stack, subtract, push</td>
</tr>
<tr>
<td>imult</td>
<td>pop 2 elms off stack, multiply, push</td>
</tr>
<tr>
<td>ineg</td>
<td>pop stack, negate, push</td>
</tr>
<tr>
<td>goto L</td>
<td>jump to handle L</td>
</tr>
<tr>
<td>ifeq L</td>
<td>pop stack, jump to handle L if zero</td>
</tr>
<tr>
<td>if_icmpleq L</td>
<td>pop 2 elms, jump if less than or equal</td>
</tr>
<tr>
<td>if_icmpgt L</td>
<td>pop 2 elms, jump to L if greater</td>
</tr>
<tr>
<td>dup</td>
<td>duplicate top of stack</td>
</tr>
<tr>
<td>pop</td>
<td>pop top of stack</td>
</tr>
<tr>
<td>swap</td>
<td>swap top two positions of stack</td>
</tr>
</tbody>
</table>


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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Holds</th>
</tr>
</thead>
<tbody>
<tr>
<td>AstNode.code</td>
<td>list of instructions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>genInst( X )</td>
<td>create new instruction X</td>
</tr>
<tr>
<td>append( ... )</td>
<td>concatenates lists of instructions</td>
</tr>
</tbody>
</table>

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

```java
stmt → IF ( exp ) stmtList ;
{ stmt.code = ??; }
```

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

```java
stmt → FOR ( stmt ; exp ; stmt ) stmt ;
{ stmt.code = ??; }
```

(c) Write grammar actions needed to generate control code for an AND expression in the following production, using numerical value representation of booleans. Use short circuiting.

```java
exp → exp1 AND exp2
{ exp.code = ??; }
```

(d) Write grammar actions needed to generate control code for an NOR expression in the following production, using numerical value representation of booleans. Use short circuiting.

```java
exp → exp1 NOR exp2
{ exp.code = ??; }
```

(e) Write grammar actions needed to generate control code for an ≥ (GEQ) expression in the following production, using numerical value representation of booleans.

```java
exp → exp1 GEQ exp2
{ exp.code = ??; }
```

2. Complex code generation.

(a) Name two issues and solutions to generating code for function calls in C.

(b) Name two issues and solutions to generating code for array references in C.

(c) What code must the compiler generate for the code

```java
int i, a[ 100 ] ;
...
a[ i + 5 ] = 4 ;
```

(d) What code must the compiler generate for the code

```java
int foo ( int x ) ;
...
x = foo( i + 2 ) ;
```

   i. Assuming foo() is call-by-value?

   ii. Assuming foo() is call-by-reference?

3. Optimizations

(a) How can compiler transformations improve a program?

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

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

4. Local optimizations

Consider the following code.

```java
(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.

5. Control flow analysis

For the following problems, consider this code:
6. 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) In what direction is REACH calculated? I.e., does information flow forwards or backwards in the CFG?
(c) Calculate GEN, KILL for each basic block.
(d) What is a good initial value for REACH for each basic block?

(e) Solve the data-flow equations in reverse Postorder. Show your work.

7. Live variables

Live variables for a point in the program $p$ is defined as the set of variables $x$ for which there is some path from $p$ to a use of $x$ with no definition to $x$ on the path. Calculate live variables for the code in the control-flow graph problem.

(a) We define LIVE($b$) for a basic block $b$ to be the set of live variables at the end of $b$. What is the dataflow equation for LIVE?
(b) In what direction is LIVE calculated? I.e., does information flow forwards or backwards in the CFG?
(c) Show GEN, KILL for each basic block.
(d) What is a good initial value for LIVE for each basic block?
(e) Solve the data-flow equations in rPostorder. Show your work.

8. 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 is a good initial value for AVAIL for each basic block?
(d) Calculate AVAIL. Show all steps, including values for AVAIL and the order basic blocks are analyzed.

9. Data-flow lattices

Prove the following properties of lattices:

(a) Show that $a \leq b$ and $b \leq c$ implies $a \leq c$
(b) Show that $a \leq (b \wedge c)$ implies $a \leq b$
10. 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?

11. 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

\[ \langle \text{op} \rangle \langle \text{dst}, s1, s2 \rangle \]

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 VLIW processor.

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

12. 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.

```
1: read a  B1
2: read b

L1: 3: c = 2+a
4: d = c+a
5: if c < d goto L2

6: e = c
7: a = e+d

L2: 8: if a < 100 goto L1

9: a = a+b  B5

exit  B6
```

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.