CMSC430 Spring 2009 Final (Partial Solutions)

1. (9 pts) Compilers
   a. Describe the difference in the how one would implement compiler front ends and back ends when writing a compiler from scratch. Explain the difference

      Front end implemented via specifications to tools, back end by hand because of greater complexity.
   b. Describe what you think current compilers do well. Explain your answer.

      Front end (scanning/parsing/type checking) and code generation/optimization for classic processor architectures.
   c. Describe where you think compilers need to improve most, in order to meet future challenges. Explain your answer.

      Improved support for parallelism, software productivity, and security.

2. (15 pts) LR items

   Consider the following grammar

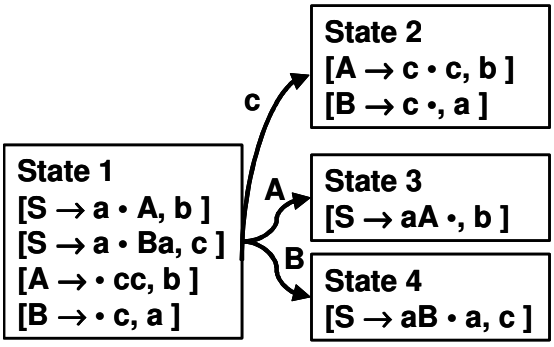
   S' → S

   S → Sa | b
   a. Construct the canonical set of LR(1) items for the grammar
   b. Is the grammar LR(1)? Explain.

      Yes, since there are no conflicts in the canonical set of LR(1) items.
   c. Is the grammar LL(1)? Explain.

      No, since the grammar is left recursive.

3. (14 pts) LR parsing
   a. (10 pts) Construct the ACTION/GOTO table for the following set of LR(1) items



Answer:

State	ACTION			GOTO	
	a	b	c	A	B
1			Shift 2	3	4
2	Reduce B → c				
3		Reduce S → aA			
4					

b. (4 pts) Given the following ACTION/GOTO table, show the parse if the current stack contents are 0 b 4 B 1 (i.e., current state is 1) and the remaining input is "d$".

State	ACTION			GOTO	
	d	e	$	S	B
0	Shift 2	Reduce S → Bb	Reduce S → bB	2	1
1	Shift 3	Reduce B → d	Accept	3	4
2	Shift 1	Shift 4	Accept	2	4
3	Reduce S → Bd	Reduce B → d	Reduce S → bBd	1	2
4	Reduce B → e	Shift 3	Reduce S → bSe	4	3

Answer:

Stack	**Input**	**Action**
0 b 4 B 1	**d $**	**Shift 3**
0 b 4 B 1 d 3	**$**	Reduce S → bBd
0 S	**$**	**Goto 2**
0 S 2	**$**	**Accept**

4. (15 pts) Code generation.

You are generating code for a Java virtual machine which supports the following Java byte codes:

You are given the following grammar attributes and helper functions:

A Ruby-style TIMES loop iterates (exp) number of times. For example, *(1+2).times { ... }* would iterate 3 times. Write syntax-directed actions needed to generate code for a TIMES loop for the following grammar production. You may assume that exp evaluates to a non-negative integer value. Note you can only use the Java instructions provided.

stmt → exp TIMES { stmtList }
{: stmt.code = ?? :}

```
stmt := exp TIMES { stmtList }
 {:
  Handle h1 = genInst( NOP ); // end of TIMES loop
  Handle h2 = genInst( NOP ); // beginning of TIMES loop

  exp.code = append(
   exp1.code,      // 1 copy of count on stack
   h2,
   genInst( DUP() ),        // 2 copies of count on stack
   genInst( IFEQ( h1 ) ),  // 1 copy of count on stack
   genInst( LDC_INT( 1 ) ),
   genInst( ISUB() ),       // decrement count by 1
   stmtList.code,
   genInst( GOTO( h2 ) ),
   h1,
   genInst( POP( ) )  );    // pop last copy of count
 :}
```

```
stmt := exp TIMES { stmtList }
 {:
  Handle h1 = genInst( NOP ); // end of TIMES loop
  Handle h2 = genInst( NOP ); // beginning of TIMES loop

  exp.code = append(
   exp1.code,
   genInst( ISTORE( index(1) ) ),
   h2,
   genInst( ILOAD( index(1) ) ),
   genInst( IFEQ( h1 ) ),
   genInst( ILOAD( index(1) ) ),
   genInst( LDC_INT( 1 ) ),
   genInst( ISUB() ),
   genInst( ISTORE( index(1) ) ),
   stmtList.code,
   genInst( GOTO( h2 ) ),
 :}
```

5. (4 pts) Local information

   Consider the following basic block for available expressions:

```
x = a+b
y = b+c
a = c+d
d = a+c
```

   a. What is GEN for the basic block?

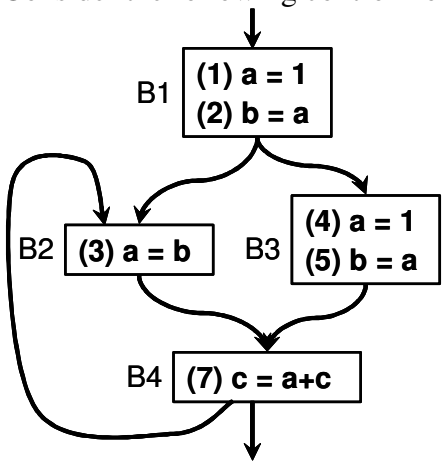
      { b+c, a+c }

   b. What is KILL for the basic block?

      { a+b, a+c, c+d }

6. (22 pts) Live variables

   Consider the following control flow graph for live variables:



a. (8 pts) Calculate GEN/KILL for each basic block

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

b. (14 pts) Initializing IN/OUT to ∅, solve live variables, showing IN/OUT for each pass

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

7. (18 pts) Register allocation

Consider the code in the control flow graph in the previous problem:

a. (8 pts) What are the live ranges for a global top-down allocator? Provide the variable name and statement numbers for reach live range.

	Stmts
a_1	**1**
a_2	**3,5,7**
b	**2,3,5,7**
c	**1,2,3,4,5,7**

b. (5 pts) Draw the interference graph for the live range.

c. (3 pts) If spilling is needed, which live range would be spilled? Explain.

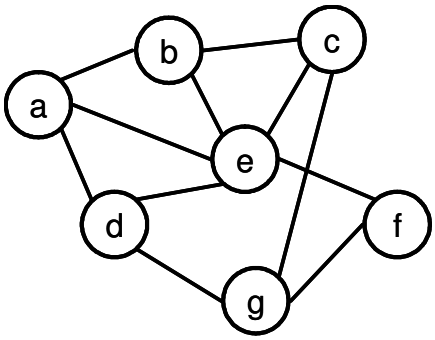
Decide considering spill cost (loop nest depth, # of refs) and node degree

d. (2 pts) How is code modified when a live range is spilled?

Insert load before each use, insert store after each definition.

8. (10 pts) Graph simplification

Consider the following interference graph:



a. (6 pts) Apply graph simplification for a machine with 3 registers. Is spilling required? Show your work.
b. (4 pts) Show how to use the results of graph simplification to color the graph with 3 colors 1,2,3.

9. (33 pts) 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 <dest, src1, src2>
1	load r1, a
2	add r2, r1, r1
3	load r3, b
4	add r4, r2 r3
5	load r1, c
6	add r3, r1, r1
7	add r5, r4, r3
8	jmp

a. (10 pts) Build the precedence graph for the instructions. Mark dependences as flow, anti, or output. You can ignore transitive dependences.

??

b. (6 pts) Calculate the critical path for the instructions

??

c. (10 pts) Schedule the instructions for a single-issue processor, using forward list scheduling. Show candidates instructions at each cycle. Prioritize candidates using 1) critical path, 2) latency of instruction, 3) number of children.

Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Candidates	**1,3**	**3**	**2**	**4,5**	**4**	**6**	**7**	**8**
Schedule	**1**	**3**	**2**	**5**	**4**	**6**	**7**	**8**

d. (7 pts) Schedule the instructions as above, but for a two-issue VLIW processor

Cycle	1	2	3	4	5	6	7	8
Candidates	1,3		2	4,5		6	7	8
Schedule	1,3		2	4,5		6	7	8

10. (20 pts) Dependence analysis

Consider the following loop in Fortran. Recall that Fortran is column-major, so columns are stored contiguously in memory. For both A & B, assume that 4 array elements fit on each cache line.

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

a. (3 pts) Find any dependences in the loop nest. Give the distance vector for each dependence.

Loop-carried true/flow dependence with distance vector (1,0) from A(i+1,j) to A(i,j).

b. (8 pts) What reuse exists in the loop? For each reuse, give type, reference(s), and loop carrying reuse.

	Loop i	Loop j
A	spatial & Group-temporal	group-spatial
B		spatial

c. (5 pts) Which loop permutation is preferred? Calculate by estimating number of cache lines accessed by each loop. Show your work.

	Loop i	Loop j
A	N * N	N * N
B	N * N	N * N
Total	N	N

d. (2 pts) Is it legal to interchange the i & j loops to make j the outer loop and i the inner loop? Explain.

Yes, since the distance vector changes to (0,1) after loop interchange and is not lexicographically negative.

e. (2 pts) Are either of the loops parallel? Explain.

The j loop is parallel since it does not carry any dependences. The dependence (1,0) is carried on the i loop.