CMSC430 Spring 2011 Midterm 2

Name _____

Instructions

- You have 50 minutes to take this exam.
- This exam has a total of 100 points. An average of 30 seconds per point.
- This is a closed book exam. No notes or other aids are allowed.
- If you have a question, please raise your hand and wait for the instructor.
- Answer essay questions concisely using 1-2 sentences. Longer answers are not necessary and a penalty may be applied.
- In order to be eligible for partial credit, show all of your work and clearly indicate your answers.
- Write neatly. Credit cannot be given for illegible answers.

	Problem	Score	Max Score
1	Attribute grammars & syntax-directed translation		24
2	Syntax-directed translation & type checking		10
3	Symbol tables		6
4	Run-time environments		12
5	Intermediate representations		8
6	Code generation		40
	Total		100

1. (24 pts) Attribute grammars & syntax directed translation

   a. (4 pts) Explain why techniques such as attribute grammars & syntax directed translation are needed in compilers.

   Consider the following attribute grammar for calculating values of signed binary numbers.

Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← *0* *If Sign.neg* *then Number.val* ← *– List.val* *else Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← *false*
	|	–	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1$*.pos* ← $List_0$*.pos + 1* *Bit.pos* ← $List_0$*.pos* $List_0$*.val* ← $List_1$*.val + Bit.val*
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← *0*
	|	1	*Bit.val* ← $2^{Bit.pos}$

   b. (4 pts) List its inherited attributes.

   c. (4 pts) List its derived attributes.

d. (12 pts) Show how the attribute grammar can calculate the value for "-10" by annotating its parse tree, showing how each attribute is calculated.

2. (10 pts) Syntax directed translation & type checking

   Consider the following grammar fragment for a statement list & Ruby-style TIMES loop:

   stmtList → *stmtList ; stmt* | …
   stmt → *exp TIMES { stmtList }* | …

   Assume that the nonterminals *exp, stmt*, and *stmtList* all have an attribute *type* that can be assigned a range of values, including *typeBoolean*, *typeInt*, *typeFloat*, *typeError*, etc…

   Add syntax-directed type checking rules to the grammar fragment to enforce the following:

   a. (4 pts) The type of a *stmtList* is the type of the last statement in the list.

   b. (6 pts) The type of a TIMES statement is the type of the *stmtList* if *exp.type* = *typeBoolean*, otherwise it is *typeError*.

3. (6 pts) Symbol tables
    a. (6pts) Show the state of lexically nested symbol tables when the compiler reaches the point marked HERE in the following code fragment:

```
int a;
void bar( ) { int b; }
void foo( ) { int c; { int d; } { int e; /* HERE */ } }
```

4. (12 pts) Run-time environment
    a. (4 pts) Explain why local variables can be efficiently allocated in a frame.

    b. (4 pts) Explain why the frame contains a *static link* to the frame of the lexically enclosing scope, instead of the existing link to the frame of the calling procedure.

    c. (4 pts) Explain why local variables allocated in a frame are represented as a (level, offset) pair, using the answers to the first two questions.

Use only the following 3-address code and Java stack code instructions for answering the following intermediate representation & 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

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
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

For code generation, 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

5. (8 pts) Intermediate representations.
   Consider the arithmetic expression:

   $(b + a) - a$

   a. (2 pts) Translate it into an abstract syntax tree (AST)

   b. (2 pts) Translate it into a directed acyclic graph (DAG)

   c. (2 pts) Translate it into 3-address code

   d. (2 pts) Translate it into Java stack code

6. (40 pts) Code generation.

You are generating code for a Java virtual machine. Your code generator follows the convention that code for Boolean expressions leave a 1 on the stack for TRUE, and a 0 for FALSE. Logical operators also use *short circuiting*, where the 2nd operand is evaluated only if it may affect the result.

The logical operator CCC returns true only if its 1st operand is TRUE and its 2nd operand is FALSE. Consider the following grammar production for the CCC operator.

$$exp \rightarrow exp_1\ CCC\ exp_2$$
$$\{: exp.code = ??\ :\}$$

a. (8 pts) Write Java byte code that would implement the CCC operator. Assume $exp_1.code$ and $exp_2.code$ are available and may be inserted in your code sequence.

b. (12 pts) Write syntax-directed actions needed to generate code for a CCC expression in the production:

A Ruby-style TIMES loop iterates (exp) number of times. For example, *(1+2).times { ... }* would iterate 3 times. Consider the following grammar production for the TIMES loop. You may assume that exp evaluates to a non-negative integer value.

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

c. (10 pts) Write Java byte code that would implement a TIMES loop. Assume *exp.code* and *stmtList.code* are available and may be inserted in your code sequence.

d. (10 pts) Write syntax-directed actions needed to generate code for a TIMES loop.