CMSC430 Spring 2014 Midterm 2 Solutions

1. (12 pts) Syntax directed translation & type checking

   Consider the following grammar fragment for an expression for C--:

   $exp \rightarrow$ CONST | IDENT_1 | IDENT_2 [exp_1]

   Assume that the nonterminal *exp* and terminals IDENT and CONST have an attribute *type* that can be assigned a range of type values, including *typeBoolean*, *typeInt*, *typeError*, etc. In addition, the terminal IDENT has an attribute *isArray* that can be true or false. Add syntax-directed type checking rules to the production $exp \rightarrow \text{IDENT}_2$ [exp_1] to enforce the following:

   a. (2 pts) IDENT_2 must be an array. Otherwise call parser.msg("Misuse of array").

   b. (2 pts) The subscript expression exp_1 must be *typeInt*. Otherwise call parser.msg("Integer required").

   c. (2 pts) The expression type is assigned the type of IDENT_2 if both (a) and (b) are true, else it is assigned *typeError*.

Answer:

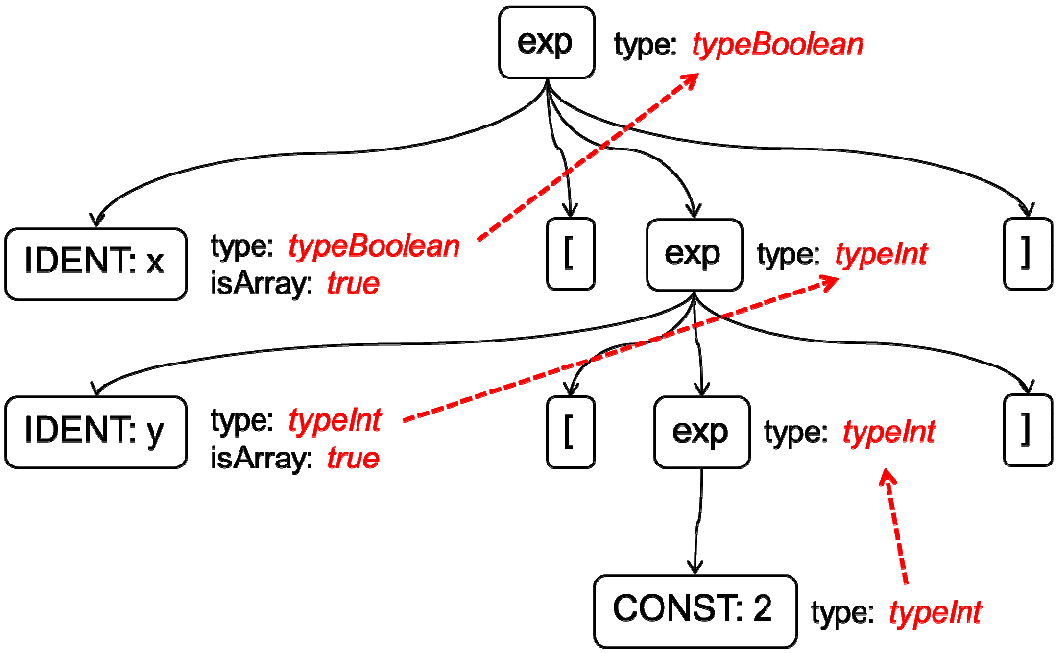
$exp \rightarrow \text{IDENT}_2$ [exp_1]

{:

```
exp.type = IDENT2.type;
if (!IDENT2.isArray) {
        parser.msg("Misuse of array");
        exp.type = typeError;
}
if (exp1.type != typeInt) {
        parser.msg("Integer required");
        exp.type = typeError;
}
```

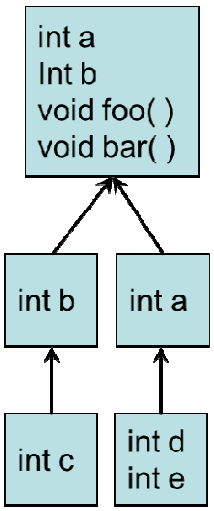
:}

d. (6 pts) Given the parse tree for x[y[2]], where x is a BOOL array, y is an INT array, and 2 is an INT, show how syntax directed translation can calculate the type of the expression using your type checking rules by annotating the parse tree. Show the value calculated for each attribute, and draw arrows linking the attributes used in the calculation.



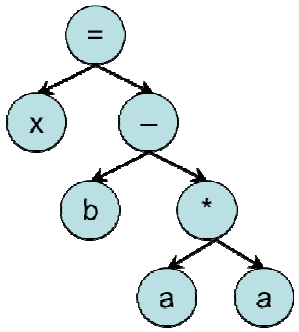
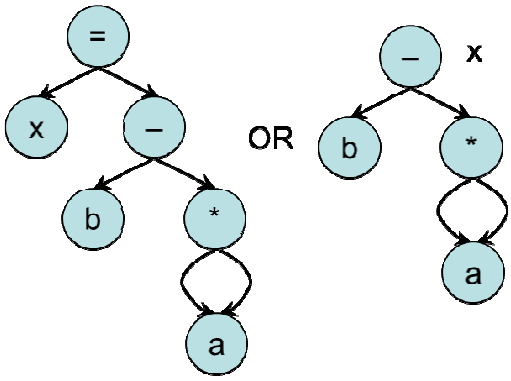
2. (4 pts) Symbol tables
Show the state of lexically nested symbol tables when the compiler reaches the point marked HERE in the following code fragment:

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



3. (12 pts) Intermediate Representations

Translate the following arithmetic expression: x = b – (a * a)

(2 pts) abstract syntax tree (AST)	(2 pts) directed acyclic graph (DAG)
	
(4 pts) 3-address code **load R1 b** **load R2 a** **mult R3 R2 R2** **sub R4 R1 R3** **store x R4**	(4 pts) Java stack code **iload index(b)** **iload index(a)** **iload index(a) // or dup** **imult** **isub** **istore index(x)**

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

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

Attribute	Holds
AstNode.code	list of instructions
Function	**Effect**
genInst(X)	create new instruction X returns handle to instruction
append(...)	concatenates lists of instructions

if stack is | … X Y (Y is at top of stack) then
if_icmpeq L → pop X, Y from stack, jump to L if X = Y
if_icmpgt L → pop X, Y from stack, jump to L if X > Y
if_icmplt L → pop X, Y from stack, jump to L if X < Y

3.step(2,2) { puts "x" } → ""
3.step(3,2) { puts "x" } → "x"
3.step(4,2) { puts "x" } → "x"
3.step(5,2) { puts "x" } → "xx"
3.step(6,2) { puts "x" } → "xx"

4. (26 pts) Code generation.

Your compiler generates Java byte codes using attributes (code) and helper functions (genInst, append) found in the table on previous page. Add support in your compiler for a Ruby style STEP loop, where x.step(y,z) { … } will iterate with *index* starting at x, incrementing *index* by z each iteration, and repeating while *index* ≤ y. Note the loop will not be executed if x > y. Write syntax-directed actions needed to generate code for a Ruby style STEP loop:

stmt → exp_1 . *STEP* (exp_2 , exp_3) { *stmtList* }
{: *stmt.code = ?? :}*

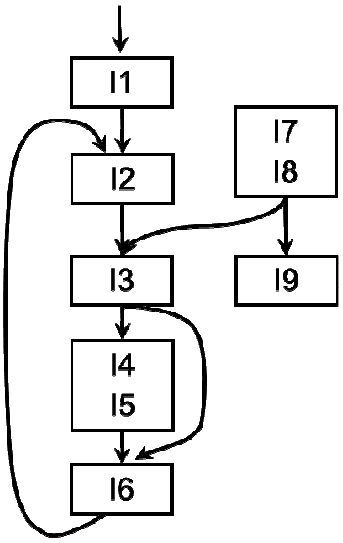
```
stmt →  exp1 . STEP ( exp2 , exp3 ) { stmtList }
 {:
   Handle h1 = genInst( NOP ); // end of STEP loop
   Handle h2 = genInst( NOP ); // beginning of STEP loop

   stmt.code = append(
     exp1.code,                  // 1 copy of index on stack
     h2,
     genInst( DUP() ),           // 2 copies of index on stack
     exp2.code,                  // exp2 = upper bound
     genInst( ICMPGT( h1 ) ),    // branch if index > exp2
     exp3.code,                  // exp3 = step
     genInst( IADD () ),         // increment index by exp3
     stmtList.code,              // body of loop
     genInst( GOTO( h2 ) ),
     h1,
     genInst( POP( ) )  );       // pop last copy of index
 :}
```

5. (10 pts) Control flow analysis

Consider the following code:

```
<I1>        a := b
<I2>   L1:  b := c
<I3>   L2:  if (...) goto L4
<I4>        c := b
<I5>   L3:  d := a
<I6>   L4:  goto L1
<I7>   L5:  b := a
<I8>   L6:  if (...) goto L2
<I9>        c := a
```



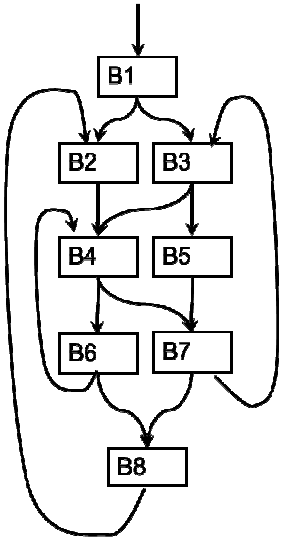
a. (8 pts) Find basic blocks and draw the control flow graph (CFG).

b. (2 pts) What is the compiler term used to describe statement I7, I8, and I9?

Unreachable / dead code.

6. (12 pts) CFG ordering, dominators, loops

Consider the following control flow graph (CFG), where B1 is the start of the program.



a. (2 pts) List all basic blocks dominated by B3.

B3, B5.

b. (2 pts) List all basic blocks dominated by B4.

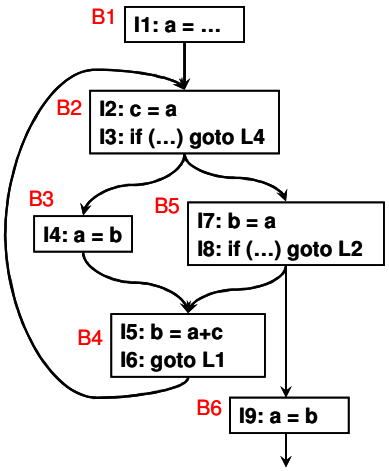
B4, B6.

c. (4 pts) List all the loops in the control flow graph. For each loop list all basic blocks in the loop, as well any back edges in the loop.

The only loop is B4, B6, with loop back edge B6->B4. Other cycles at B2 and B3 are not loops since no node dominates all other nodes in the loop (can enter loop bypassing loop header).

d. (4 pts) Give TWO reverse Postorder numberings of the CFG for Problem 7 on the next page.

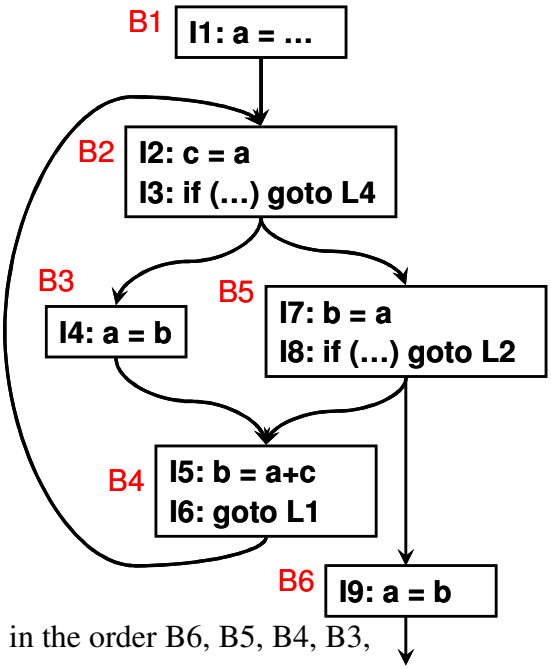
Possible rPostorder traversals: 1,2,3,5,4,6
1,2,3,5,6,4
1,2,5,6,3,4



7. (24 pts) Dataflow analysis

Calculate Live Variables for the following code:

```
<I1>          a := …
<I2>    L1:   c := a
<I3>          if (...) goto L4
<I4>          a := b
<I5>    L2:   b := a+c
<I6>          goto L1
<I7>    L4:   b := a
<I8>          if (...) goto L2
<I9>          a := b
```



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

b. (16 pts) Initializing IN/OUT to ∅, solve live variables in the order B6, B5, B4, B3, B2, B1, showing IN/OUT for each pass. Stop when a fixed point is reached.

Basic Block	GEN	KILL
B1	∅	**a**
B2	**a**	**c**
B3	**b**	**a**
B4	**a, c**	**b**
B5	**a**	**b**
B6	**b**	**a**

Basic Block		Initial	Pass 1	Pass 2	Pass 3	Pass 4
B6	OUT	∅	∅	∅	∅	
	IN	∅	**b**	**b**	**b**	
B5	OUT	∅	**b**	**a, b, c**	**a, b, c**	
	IN	∅	**a**	**a, c**	**a, c**	
B4	OUT	∅	∅	**a, b**	**a, b**	
	IN	∅	**a, c**	**a, c**	**a, c**	
B3	OUT	∅	**a, c**	**a, c**	**a, c**	
	IN	∅	**b, c**	**b, c**	**b, c**	
B2	OUT	∅	**a, b, c**	**a, b, c**	**a, b, c**	
	IN	∅	**a, b**	**a, b**	**a, b**	
B1	OUT	∅	**a, b**	**a, b**	**a, b**	
	IN	∅	**b**	**b**	**b**	