CMSC430 Spring 2011 Midterm 2 Solutions

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.

   Possible answers:
   - **To obtain information about the program not possible using regular expressions (scanners) and context free grammars (parsers).**
   - **To perform type checking, build symbol tables, build ASTs, etc…**

   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*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos ← List.pos* *List.val ← Bit.val*
Bit	→	0	*Bit.val ← 0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

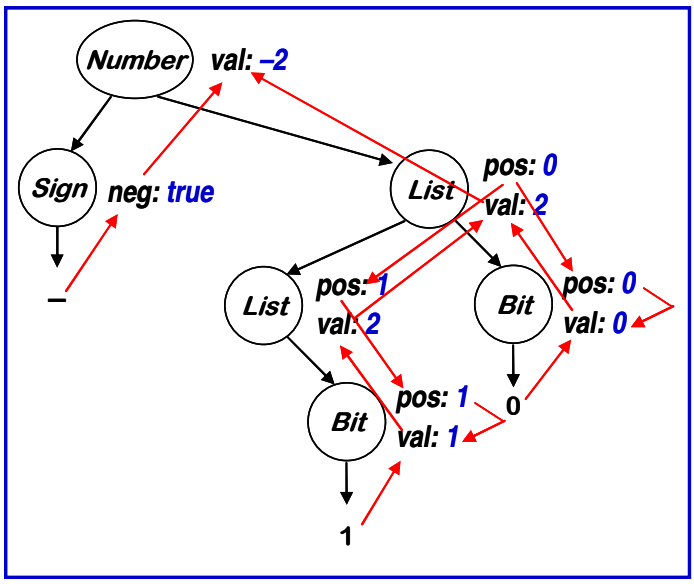
   b. (4 pts) List its inherited attributes.

   Pos, since inherited attributes flow from the top down (parents to child) or across (between siblings) in the parse tree.

   c. (4 pts) List its derived (synthesized) attributes.

   Val & neg, since derived (synthesized) attributes flow up (from children to parent) in the parse tree.

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*$_1$ *; 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.

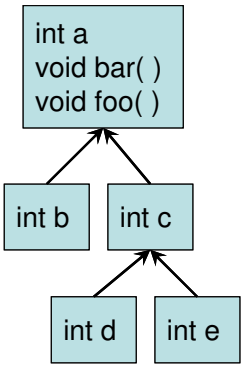
***stmtList* → *stmtList*$_1$ *; stmt* { *stmtList.type* = *stmt.type* }**

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

stmt* → *exp TIMES { stmtList }
{ *stmt.type* = (*exp.type* == *typeBoolean*) ? *stmtList.type* : *typeError* } OR
{ if (*exp.type* == *typeBoolean*) *stmt.type* = *stmtList.type*
else *stmt.type* = *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.

      Because local variables only need to be accessible while the procedure is active, and allocating them in the frame allows them to be accessed as an offset to the frame (stack) pointer.

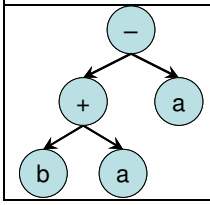
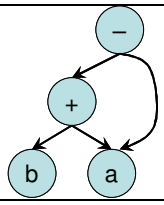
   b. (4 pts) Explain why the frame contains a *static link* to the frame of the lexically enclosing scope, instead of the existing link to the frame of the calling procedure.

      Because with static lexical scoping, free variables are bound to the nearest lexically enclosing scope, not the most recently executed (calling) scope.

   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.

      Because the level is used to determine how many static links to traverse to find the appropriate frame, and the offset is added to the frame address to find the address of the local variable.

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

AST	DAG	3-address code	Java stack code
		**load R1 b** **load R2 a** **add R3 R1 R2** **sub R4 R3 R2**	**iload index(b)** **iload index(a)** **iadd** **iload index(a)** **isub**

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.

exp_1.code	**exp_1.code**	**exp_1.code**
ifeq h1	**ifeq h2**	**dup**
exp_2.code	**exp_2.code**	**ifeq h3**
ifeq h2	**ifeq h1**	**pop**
h1: ldc_int 0	**goto h2**	**exp_2.code**
goto h3	**h1: ldc_int 1**	**ifeq h2**
h2: ldc_int 1	**goto h3**	**h1: ldc_int 0**
h3:	**h2: ldc_int 0**	**goto h3**
	h3:	**h2: ldc_int 1**
		h3:

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

exp := exp_1CCC exp_2	**exp := exp_1CCC exp_2**	**exp := exp_1CCC exp_2**
{:	**{:**	**{:**
Handle h1 = genInst(NOP); // false	**Handle h1 = genInst(NOP); // true**	**Handle h1 = genInst(NOP); // T**
Handle h2 = genInst(NOP); // true	**Handle h2 = genInst(NOP); // false**	**Handle h2 = genInst(NOP); // F**
Handle h3 = genInst(NOP); // exit	**Handle h3 = genInst(NOP); // exit**	**Handle h3 = genInst(NOP); // ex**
exp.code = append(	**exp.code = append(**	**exp.code = append(**
exp_1.code,	**exp_1.code,**	**exp_1.code,**
genInst(IFEQ(h1)),	**genInst(IFEQ(h2)),**	**genInst(DUP( )),**
exp_2.code,	**exp_2.code,**	**genInst(IFEQ(h3)),**
genInst(IFEQ(h2)),	**genInst(IFEQ(h1)),**	**genInst(POP( )),**
h1,	**genInst(GOTO(h2)),**	**exp_2.code,**
genInst(LDC_INT(0)),	**h1,**	**genInst(IFEQ(h2)),**
genInst(GOTO(h3)),	**genInst(LDC_INT(1)),**	**h1,**
h2,	**genInst(GOTO(h3)),**	**genInst(LDC_INT(0)),**
genInst(LDC_INT(1)),	**h2,**	**genInst(GOTO(h3)),**
h3);	**genInst(LDC_INT(0)),**	**h2,**
:}	**h3);**	**genInst(LDC_INT(1)),**
	:}	**h3);**
		:}

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.

```
      exp1.code,      // 1 copy of count on stack
h2:                   // beginning of TIMES loop
      dup             // 2 copies of count on stack
      ifeq h1         // 1 copy of count on stack
      ldc_int 1
      isub            // decrement count by 1
      stmtList.code,
      goto h2
h1:   pop             // pop last copy of count
```

```
      exp1.code,      // calculate count
      istore index(t) // store count at t
h2:                   // beginning of TIMES loop
      iload index(t)  // load count from t
      ifeq h1         // exit if count = 0
      iload index(t)
      ldc_int 1
      isub            // decrement count by 1
      istore index(t) // store new count at t
      stmtList.code,
      goto h2
h1:                   // end of TIMES loop
```

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

```
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 ) ),
    h1 );
 :}
```