CMPSC 160 – Practice Exam

1. Consider the following grammar:

```
P → declare D begin B end
D → D; int id | int id
B → B; S | S
S → id = E
E → E + T | E – T | T
T → (E) | id | num
```

(a) Convert the above grammar to an LL(1) grammar.

(b) Write a translation scheme for the resulting grammar for interpreting the programs written in the above language. The translation scheme you write should evaluate the values of the expressions and store the values of the variables using the procedures provided below.

```
// These procedures are given. You do not have to write them.
void enterId(String varName){ ... }
  // creates a storage for the identifier with name varName
void setValue(String varName, int value) { ... }
  // sets the value of the identifier with name varName to value
int getValue(String varName) {...}
  // returns the value of the identifier with name varName
```

2. The grammar given below is used for specifying the declarations in a simple programming language.

```
Program    → DeclList
DeclList   → DeclList Decl
           | Decl
Decl       → RecordDecl
           | BasicDecl
RecordDecl → struct id begin FieldList end;
FieldList  → FieldList BasicDecl
           | BasicDecl
BasicDecl  → int id;
           | real id;
```

You are asked to convert his grammar to a translation scheme for storage allocation. Use a global variable called *offset* to keep the address of the next available memory location (initialize it to 0). Use the procedure *enter_loc(id.name, mem_loc)* to store *mem_loc* as the address of the memory location for identifier *id.name* in the symbol table (we assume that the symbol table entry is created in a previous pass). The memory location of a record should be same as memory location of its first field. Assume that the size of *int* is 4 and the size of *real* is 8.

3. Consider the following grammar for binary numbers:

```
N  → L.L
   | L
L  → L B
   | B
B  → 0
   | 1
```

(a) Using only synthesized attributes, write the semantic rules to evaluate the value of the binary numbers generated by this grammar. For example, the value of the input string:

```
110.011
```

should be evaluated as: 6.375

(b) Below we give the LR parse table for the grammar given in part (a)

	Action				Goto	
State	.	0	1	$	L	B
s0		shift s1	shift s2		s4	s3
s1	reduce B→0	reduce B→0	reduce B→0	reduce B→0		
s2	reduce B→1	reduce B→1	reduce B→1	reduce B→1		
s3	reduce L→B	reduce L→B	reduce L→B	reduce L→B		
s4	shift s6	shift s1	shift s2	accept		s5
s5	reduce L→LB	reduce L→LB	reduce L→LB	reduce L→LB		
s6		shift s1	shift s2		s7	s3
s7		shift s1	shift s2	accept		s5

Using the semantic definitions you derived in part (a), show the evaluation of the synthesized attributes using the stack-based shift-reduce parsing algorithms (LR paring algorithm) for the input string:

```
10.101
```

Assume that synthesized attributes of each nonterminal is stored next to it in the parser stack. Your solution should show the contents of the stack, values of the attributes of each nonterminal symbol in the stack, and the production and the semantic rule used in computing each attribute. You should also show the final value computed for the *val* attribute of the start symbol ***N*** and the production and the semantic rule that computes it.

4. Write the semantic rules for type checking the following expression grammar:

```
E    →    E aop E
     |    E rop E
     |    E = E
     |    E lop E
     |    id
     |    bool
     |    num
```

The token *bool* is a boolean literal and the token *num* is an integer literal. Tokens *aop*, *rop*, and *lop* denote arithmetic, relational, and logical operators, respectively. Assume that the token *id* has an attribute called *type* which could be *integer* or *boolean*. Nonterminal *E* also has an attribute called type which could be *integer*, *boolean* or *type-error*. You should write the semantic rules to compute the *type* attribute for nonterminal *E*. The type checking rules are:

- Both operands of an arithmetic operator should be of integer type.
- Both operands of a logical operator should be of boolean type.
- Both operands of a relational operator should be of integer type.
- The two operands of the equality operator = should be of the same type.

5. Write the type-expressions for foo and bar in the following C code fragments.

(a)

```
int foo(int bar[10], char * x);
```

(b)

```
typeof struct{
  int a;
  char b;
} data, *pdata;

data foo[100];
pdata bar(int w, data y){ ... }
```

6. Consider the following program (with nested procedures):

```
procedure main
  float x;
  int a, b;
  procedure p1(int a)
    int b;
    float y;
    procedure p3()
      int c;
      float x;
      begin
        ...
      end
    begin
      ...
    end;

  procedure p2 (float x, float z)
    int a;
    begin
      ...
    end
  begin
    ...
  end;
```

(a) Show the contents of a lexically-scoped symbol table for this program. Also show the memory offset of each variable in the symbol table. Assume the following: Memory offsets for variables are computed in the order they appear in the procedure. *int* is 4 bytes and *float* is 8 bytes. Parameters are passed using call-by value. Parameters of a procedure are stored in the local data area of that procedure.

(b) Draw the activation tree for the following execution sequence:
`main calls p1, p1 returns, main calls p2, p2 calls p1, p1 calls p3, p3 calls p3, ... show the contents of the control stack at this point ..., p3 returns, p3 returns, p1 calls p2, p2 returns, p1 returns, p2 returns.`
Draw the contents of the control stack during the second activation of procedure p3. Show the access and control links in the activation records.

7. You are given the following instruction set for a stack machine.

`push value`	Pushes the *value* to the stack
`pop`	Pops top of the stack
`load loc`	Pushes value of the data location *loc* to the stack
`store loc`	Pops a value from the stack and stores it in the data location *loc*
`ifeq label`	Pops a value from the stack, if it is equal to zero jumps to statement *label*
`ifneq label`	Pops a value from the stack, if it is not equal to zero jumps to statement *label*
`goto label`	Jumps to a statement *label*
`add`	Pops two values from the stack, adds them, and pushes the result back onto the stack
`if_cmplt label`	Pops the top two values of the stack, if the value popped last is less than the value popped first then it jumps to statement *label*

(a) Write a semantic rule for generating stack machine code for statements defined by the following production:

```
ForStmt → for (Expr; Expr; Expr) Stmt
```

Write a semantic rule to generate the code attribute of *ForStmt*. Assume that *Expr* and *Stmt* have *code* attributes which hold the code generated for them. Your job here is to compute the *code* attribute for *ForStmt* using the *code* attributes of *Expr* and *Stmt* nonterminals. Assume that the code generated for a boolean expression stores a value in the top of the stack which is 1 if the expression is true, 0 if it is false. The stack machine instructions can be labeled as "Label: instruction". You can use operator `||` to concatenate generated code, function *newlabel()* to get a new label.

(b) Based on the semantic rules you give in part (a), write the stack machine code for the following program segment assuming that variable *i* is stored in data location 10, variable *count* is stored in data location 11, and variable *x* is stored in data location 12. Here you must also write the stack machine code for *Expr.code* and *Stmt.code*.

```
for (i = 0; i < 5; i++)
  count = count + x
```