Principles of Compiler Design
Practice Questions

---

1. This grammar generates binary numbers with a decimal point:

   S → L . L | L
   L → L B | B
   B → 0 | 1

   Design an L-attributed definition to compute S.val, the decimal number value of the input string. For example, the translation of string 101.101 should be the decimal number 5.625. Draw parse tree for string 101.101 and show how the rules compute the value.

   DO NOT TRANSFORM THE GRAMMAR IN ANY WAY.

---

2. Assume following grammar for array types:

   $T \rightarrow B\ C$
   $B \rightarrow int$
   $B \rightarrow float$
   $C \rightarrow [\ num\ ]\ C_1$
   $C \rightarrow \epsilon$

   (A) Write rules to compute attributes *type* (capturing the type expression), and *width* (capturing the number of bytes required) of all the grammar symbols. Assume that integer takes 4 bytes and float takes 8 bytes. You may use both inherited and synthesized attributes. However, make sure that the attribute equations are put in the right places so that evaluation order of the attribute equations remain correct.

   (B) Make parse tree of the string *int*[4][5][6] and show the value of attributes at each of the nodes using the above grammar and the rules developed in part (A). You will not get credit for this part if the answer to the part (A) is not correct.

---

3. What is printed by the program shown below, For each of the following parameter passing mechanism:
   (a) call-by-value (b) call-by-reference
   (c) call-by-name (d) copy-restore

```
void p(x, y, z)  {
    y = y + 1;
    z = z + x;
    printf("p: %d, %d, %d", x, y, z);
    return;
}
main()  {
    int a = 2;
    int b = 3;
    p(a+b, a, a);
    printf("main: %d", a);
}
```

---

4. Consider grammar and rules given below for array address translation and generating 3 address code for array references:

```
E → E1 + E2 {E.addr = newtemp();
              gen(E.addr '=' E1.addr '+' E2.addr);}

E → E1 * E2 {E.addr = newtemp();
             gen(E.addr '=' E1.addr '*' E2.addr);}

  | id        {E.addr = id.lexeme; }

  | L         {E.addr = newtemp();
               gen(E.addr '=' L.array.basname '[' L.addr ']'); }

L → id [ E ]  {L.array = id.lexeme;   L.type = L.array.typeofelement;
               L.addr=newtemp();
               gen(L.addr '=' E.addr '*' L.type.width);}

  | L1 [ E ]  {L.array = L1.array;   L.type = L1.type.typeofelement;
               t = newtemp();   L.addr = newtemp();
               gen(t '=' E.addr '*' L.type.width);
               gen(L.addr '=' L1.addr '+' t); }
```

Function newtemp() returns a new temporary name
L.array.basename means name of the array
L.array.typeofelement means type of the element of the array
L.type.width means width of L.type
Assume size of integer to be 4 bytes, and lower bound of the arrays to be 0

Let A, B and C be 10×5, 5×7, and 10×7 arrays of integers respectively. Let i, j, and k be integers.

Construct an annotated parse tree for the expression C[i][j]+A[i][k]*B[k][j] and show the 3-address code sequence generated for the expression.

5. This grammar generates binary numbers with an optional negative sign (−) and optional decimal point:

$$\begin{array}{rcl} S & \rightarrow & N \mid -N \\ N & \rightarrow & L\,.\,L \mid L \\ L & \rightarrow & L\,B \mid B \\ B & \rightarrow & 0 \mid 1 \end{array}$$

(a) Design an L-attributed definition to compute S.val, the decimal number value of the input string. For example, the translation of 101.101 should be the decimal number 5.625.

(b) Draw parse tree for string −101.101 and annotate the nodes with values computed by the rules.

DO NOT TRANSFORM THE GRAMMAR IN ANY WAY.

6. Given below is a grammar for switch statement.

```
SwitchStmt  →  switch Expr { Cases }
     Cases  →  Cases Case | Case
      Case  →  case Const :  Stmt
```

We would like to generate 3-address code for switch statements. The nature of the code that we would like to generate is illustrated below:

```
switch E {                  1. code to evaluate E in t
                            2. if t <> V1 goto 5
  case V1: S1               3. code for S1
  case V2: S2               4. goto 9
                            5. if t <> V2 goto 8
}                           6. code for S2
                            7. goto 9
                            8. nop
                            9. ...
```

Give **L-attributed** definitions to generate 3-address code for switch statements. Clearly describe all the attributes that you will be using. Assume that code has already been generated for `Expr` and `Stmt`. Use the familiar functions *gen* (or *emit*), *backpatch*, *nextquad* etc. You can also use other functions, but explain what they do.

Note: (i) `nop` statement is just a placeholder, and can be generated by *gen(nop)*.
(ii) If you can not give L-attributed definition, you can attempt giving a general SDT scheme for partial marks.

End of Practice Questions