Problem 1 — Lexical/Dynamic Scoping

Assume variable names written as **capital** letters use **dynamic** scoping and variable names written as **lower case** letters use **static** (lexical) scoping. Assume that procedures return when execution reaches their last statement. Assume that all procedure names are resolved using static (lexical) scoping. Show the output of the entire program execution. Label the output with the location of the print statement (e.g.: (*2*): ...)).

```
program main()
{
    int A, b;
    procedure f()
    {
        int c;
        procedure g()
        {
            int c;
            c = 33;
            ... = ...b...  //<<<<-------- (*A*)
            print A,b,c;  //<<<<-------- (*1*)
            end g;
        }
        print A,b;  //<<<<-------- (*2*)
        A = 1; b = 2; c = 3;
        call g();
        print c;  //<<<<-------- (*3*)
        end f;
    }
    procedure g()
    {
        int A,b;
        A = 4; b = 9;
        call f();
        print A,b;  //<<<<--------(*4*)
        end g;
    }
    A = 5; b = 3;
    print A,b;  //<<<<-------- (*5*)
    call g();
    print A,b;  //<<<<--------(*6*)
    end main;
}
```
Problem 2 – Lexical Scoping Code Generation

Assume that all variables are lexically scoped.

```
program main()
{ int a, b;
  procedure f()
  { int c;
    procedure g()
    {
      ... = b + c  //<<<-------- (*A*)
      print a,b,c;
      end g;
    }
    a = 0; c = 1;
    ... = b + c  //<<<-------- (*B*)
    call g();
    print c;
    end f;
  }
  procedure g()
  { int a,b;
    a = 3; b = 7;
    call f();
    print a,b;
    end g;
  }
  a = 2; b = 3;
  print a,b;
  call g();
  print a,b;
  end main;
}
```

1. Show the runtime stack with its stack frames, access and control links, and local variables when the execution reaches program point (*A*).

2. Give the ILOC RISC code for the expressions at program points (*A*) and (*B*). The value of the expressions need to be loaded into a register. The particular register numbers are not important here.
Problem 3 – Parameter Passing

```plaintext
program foo()
{
    a, b integer;
    procedure bar(integer x, integer y)
    {
        z: integer;
        ------ /* 0 */
        z = 3;     /* 1 */
        x = x + y + z; /* 2 */
        y = 1;     /* 3 */
    }
    // statement body of foo
    a = 3;
    b = 6;
    call bar(a, b);
    print a, b;
}
```

Use the RISC machine instructions `loadI`, `loadAI`, `storeAI`, add to show the code that needs to be generated for the body of procedure bar (statements /*1*/ through /*3*/). Assume that

1. Register r0 contains the frame pointer (fp) value.

2. Formal parameter x is **call-by-reference**, and formal parameter y is **call-by-value**. Assume that bar’s parameters x and y have been correctly initialized as part of the procedure call of bar.

3. Use the stack frame layout as shown above. The figure shows the runtime stack when the program execution reaches program point /*0*/ in procedure bar.

What values for a and b does the program print?
Problem 4 – Parameter Passing

Assume that you don’t know what particular parameter passing style a programming language is using. In order to find out, you are asked to write a short test program that will print a different output depending on whether a call-by-value, call-by-reference, or call-by-value-result parameter passing style is used. Your test program must have the following form:

```plaintext
program main()
{
    x integer;
    procedure bar(integer a)
    {
        // statement body of foo
    }

    // statement body of main
    x = 1;
    call bar(x);
    print x;
}
```

The body of procedure `bar` must only contain assignment statements. For instance, you are not allowed to add any new variable declarations.

1. Write the body of procedure `bar` such that print `x` in the `main` program will print different values for the different parameter passing styles.

2. Give the output of your test program and explain why your solution works.