Question 2. (30 POINTS)

Consider adding *forward function declarations* to the Little language. A forward function declaration is a function header (including its return type and formal parameters) without a body. The normal declaration (including the body) comes later in the code. For example:

```
int f (int x, double y);      // forward declaration of f
int f (int x, double y) { }   // normal declaration of f
```

The reason for adding forward function declarations is to allow mutually recursive functions. For example, the code above could include a (normal) declaration of a function `g` between the two declarations of `f`, and the body of `g` could include a call to `f` (since the symbol table would have an entry for `f`).

The following table lists the errors that name analysis must detect for forward and normal function declarations, and the corresponding error message numbers.

ERROR	MESSAGE NUMBER
function declaration (forward or normal) with the same name as a previously declared global variable	1
forward function declaration with the same name as a previous forward or normal function declaration	2
normal function declaration with the same name as a previous normal function declaration	3
forward declaration with **no** normal declaration later in the code	4
normal declaration's return type and/or parameter list don't match those of previous forward declaration	5

Think about how to change Little name analysis to handle forward function declarations. Assume the following:

- You can create different kinds of symbol-table entries (Sym objects) for variables and for functions, and given a Sym object, you can tell whether it is for a variable or for a function.
- For a forward or normal function declaration, you can create a Sym object that includes the return type and a list of the parameter types.
- For a forward or normal function declaration, you can compare the declaration's return type and list of parameters with the return type and parameter list in an existing Sym object that includes those values.

On the rest of this page and on the next 3 pages, describe each of the following:

(a) What changes (if any) would you make to the over-all approach to name analysis?

(b) What are the steps that would be done by the name-analysis method for a forward function declaration? Be sure to say which of the error messages listed in the table on the previous page might be given.

(c) What are the steps that would be done by the name-analysis method for a normal function declaration. Be sure to say which of the error messages listed in the table on the previous page might be given.

(d) If one or more of the error messages listed in the table on the previous page would ***not*** be given by the name-analysis methods for forward or for normal function declarations, when and how would those error messages be given.

(a) What changes (if any) would you make to the over-all approach to name analysis?

(b) Steps for name analysis for a forward function declaration (include error msg numbers)

(c) Steps for name analysis for a normal function declaration (include error msg numbers)

(d) When and how any "missing" error messages would be given

Question 3. (20 POINTS)

When a function is called, the following tasks may be done to set up the called function's Activation Record:

Task 1. set the value of the access link field (if access links are being used to access non-local variables)

Task 2. push the values of the actual parameters

Task 3. set the value of the save-display field, and set the appropriate element of the display to point to the new Activation Record (if a display is being used to access non-local variables)

Task 4. set the value of the return-address field (to the address of the instruction that follows the *jal* instruction in the calling function)

Task 5. set the value of the control-link field (to a copy of the value in the Frame Pointer)

Task 6. set the Frame Pointer to point to the bottom of the new AR

Task 7. leave space in the new AR for local variables (by subtracting the appropriate number from the Stack Pointer).

Part (a): In class (and in the on-line notes) we said that tasks 1 and 2 are done by the *calling* function (as part of the code generated for a function call), and tasks 3 – 7 are done by the *called* function (as part of the code generated for the "function prefix"). However, this is not necessarily the only option.

For each of the seven tasks, circle the correct answer below to say whether it could be done the other way around. Assume that the symbol-table entry for a function ID includes its nesting level, a list of the types of its formal parameters, and the total number of bytes needed for its local variables.

Task 1 could be done by the *called* function: YES NO

Task 2 could be done by the *called* function: YES NO

Task 3 could be done by the *calling* function: YES NO

Task 4 could be done by the *calling* function: YES NO

Task 5 could be done by the *calling* function: YES NO

Task 6 could be done by the *calling* function: YES NO

Task 7 could be done by the *calling* function: YES NO

Part (b): Some of the 7 tasks can be performed by either the calling function or the called function. Is there a reason for assigning those tasks to one or the other? Consider both execution time and the size of the generated code.

Question 4. (15 POINTS)

Assume that we have extended the Little language by adding ***break*** statements:

- A *break* is valid only inside a while loop.
- When executed, a *break* causes a jump to the code that follows the loop (i.e., to the place that is jumped to when the loop's condition is false).

Assume that the type checker has verified that all break statements are inside while loops. Consider the changes that would need to be made to the code-generation phase of the compiler. Below is the `codeGen` method for while loops for the original Little language, and an empty method for break statements. You are to make any changes necessary to the first method in order to handle break statements, and you are to complete the second method. You may change the method headers. If you want to add new fields to the `ASTnode` class, declare them below, before the start of the first `codeGen` method. Don't forget that loops can be nested!

```
// codeGen for WhileStmtNode
public void codeGen() {

    String loopLabel = Codegen.nextLabel();

    String falseLabel = Codegen.nextLabel();


    Codegen.genLabel(loopLabel);

    myExp.codeGen();

    Codegen.genPop(Codegen.T0, 4);

    Codegen.generate("beq", Codegen.T0, Codegen.FALSE, falseLabel);

    myStmtList.codeGen();

    Codegen.generate("b", loopLabel);

    Codegen.genLabel(falseLabel);

}
```

```
// codeGen for BreakStmtNode
public void codeGen( ) {









}
```

Question 5. (25 POINTS)

Consider the following Little program:

```
int k;

int h(int b) {
  k++;
  return b;
}

void g(int a) {
    k = a;
    printf("%d", a);
    printf("%d", k);
}

void f(int x) {
    x = x - 3;
    g(h(k));
    k = k - 4;
    printf("%d", x);
    printf("%d", k);
}

void main() {
    k = 10;
    f(k);
    printf("%d", k);
}
```

Part (a)

This program may produce different output depending on which parameter-passing modes are used for `f`'s parameter `x`, `g's` parameter `a`, and `h`'s parameter `b`. Fill in the table below, providing the output that corresponds to the specified modes.

x	a	b	output				
value	value	value	a:	k:	x:	k:	k:
reference	value	reference	a:	k:	x:	k:	k:
value-result	value	value	a:	k:	x:	k:	k:
value	name	value	a:	k:	x:	k:	k: