1 Procedure Calls

1. We do not know the size of the stack frame; in fact, in our conventions the frame size changes due to changing number of temporary values (e.g. value of a left operand in addition) in a computation we need to keep track of.

To support putting the stack frame on top of the stack, we need to pre-allocate stack slots for each temporary value we would encounter in the code of the function.\(^1\)

Effectively, we are introducing extra variables. For example,

\[
a = x + y - z;
\]

would be transformed into

\[
t = x + y;
a = t - z;
\]

Note that in this case the code for function body should be generated before generating the prologue and epilogue for the function.

A good take home question: What are the implications if you also make the arguments to a function temporary variables in the caller’s frame?

2. We need to first detect such consecutive calls first, by traversing the parse tree.

For all such calls detected perform saving registers at the beginning of all calls and restoring the registers after all functions are called. This implies that we are performing caller-save instead of callee-save.

Note that this technique in general have only a few impact on generated code due to very few occurrences of such case. Note that if there is any computation between two function calls (e.g. computing the value of an argument) this would not work.

3. First the key for the symbol table should be changed from using function names to function signatures (i.e. function name + list of type of parameters).

For example, suppose we have a symbol table of the following structure:

```
// {function name : (list of types of arguments, local symbol table)}
map<string, pair<vector<string>, map<string, string>>> symbolTable;
```

we can change it to

\(^1\)This might seem to be a waste of stack space, but it does help a lot when optimizing the code, especially in register allocation.
// {function name : {list of types of arguments: local symbol table}}
map<string, map<vector<string>, map<string, string>>> symbolTable;

The type checking for procedure declarations changes to:

if function_name exists in symbolTable and
argument_list exists in symbolTable[function_name] then
    ERROR
else
    symbolTable[function_name][argument_list] = map of local variables
end if

When generating the code for the function itself, we should encode function signature information into the label.\(^2\) For example, `foo(int, int *)` could use label `ProcedurefooIP` where I stands for integer and P stands for (integer) pointer.

## 2 Optimization

### 2.1 Constant Folding, Constant Propagation and Dead Code Elimination

The eliminated code is commented out below.

```cpp
int f1(int x, int y){
    // if (x + y < INT_MAX){ // This is always true
        return y;
    // } else {
        // return x;
    // }
    // return -1; // Eliminated since never reachable
}
```

```cpp
int f2(int x){
    int i;
    int c; // Note that c is not defined
    for (i = 0; i < 10; i++){
        c += i * 2;
    }
    // So cannot constant propagate c -> 90 here
    return c + x;
}
```

```cpp
int f3(){
    int k = 5;
    int i = 0;
    int j = 1;
    int &rj = j;
    int &ri = i;
    while(k != 0){ // constant propagation will convert this into
        // 5 != 0 (since k never changed); then fold this
```

\(^2\)This technique is called name mangling; you do not need to know this word for the final.
// expression into true. Infinite loop!
int t = rj;
rj = ri + rj;
ri = t;
}
// return j;
// return i;
}

2.2 Common Subexpression Elimination

Any expressions that contain side effects should not be eliminated as a result of common subexpression elimination. For example: i++ + i++.