You need to extend your solution for Project 3 to translate Sage AST expressions to intermediate code. As with Project 3, this will be done by traversing the Sage AST in ROSE, extracting the necessary information from the AST, and constructing a string that contains the resulting transformed program. This string will be printed by `main` (already provided in Project 3; do not change it). The result should be a valid C program whose behavior is equivalent to the input program. The project is due by October 28, 11:59 pm.

**Input Language**

Assume that the input is a C program that is a valid input for Project 3, with the following additional restrictions on all

- **Expression statements** of the form `expression;`
- **Return statements** of the form `return expression;`

In both cases, the expression is restricted to contain *only* the following operators

- Assignment `=` and all its variations (e.g., `+=`, `>>=`, etc.)
- Increment `++` and decrement `--` (prefix and postfix)
- Arithmetic operators: add, subtract, multiply, divide, modulo, unary plus/minus
- Bitwise operators: shift left/right, bitwise-and, bitwise-or, etc.
- Array-subscript operator `[ ]`

Thus, these expressions do not have equality operators (equal-to/not-equal-to), relational operators (less-than, etc.), logical operators (and, or, negation), the pointer-dereference operator `*`, and the address-of operator `&`.

For simplicity, you can also assume that the following additional restrictions are satisfied by the input program:

- The program does not contain any pointer-typed variables. Note that this restriction eliminates the use of `modfft2.c` as a test program, since it contains both pointer-typed variables and pointer dereference expressions.
- There are no multi-dimensional arrays: every array is one-dimensional.
- All variable declarations for local variables are in the outermost scope in the function body – that is, block statements inside the function body do not contain variable declarations.

**Translation**

Consider every expression statement in the input program. In the Sage AST in ROSE, all such statements are represented by `SgExprStatement` nodes. You have to generate intermediate code for each such statement, using the approach discussed in class. The intermediate language is the one described in the lecture notes. Since this intermediate language is a proper subset of C, your output program will be a C program. For example, if the input program is

```c
void main(){ float a, b, c, d; a = 1; b = 2; c = 3; d = a + b + c; }
```

the resulting program could be something like this

```c
void main(){ float _t1; ... float a; ... a = 1; b = 2; c = 3; _t1 = a; ... d = ...; }
```

Note that you need to create the temporary variables as regular C variables, with the appropriate declarations. The location of these declarations in the output program is irrelevant (e.g., they could all come at the very beginning of the function, before anything else), as long as the output is a valid C program that is a correct translation of the input program.
You also need to translate each `return statement` of the form `return expression;` to `return x;` where `x` is a single address (i.e., a program variable, a temporary variable, or a constant) corresponding to `expr.addr` from the code generation rules discussed in class.

**Output**

The result should be a single string value that is returned back to the `main` function provided for Project 3 (do not change this `main`; it will be used by the grader). This string should be a valid C program that can be compiled (by `gcc` on `stdlinux`) and executed. This output program should be a correct translation of the original program, based on the code generation rules discussed in class.

**Details**

- Your submission must compile and run on `stdlinux`, using the ROSE installation in `/class/cse756/...`
- During the analysis of the AST, do not print directly to `cout`. The only printing to `cout` must be done by `main`.
- Do not use the ROSE methods `unparseToString` and `unparseToCompleteString`.
- Your submission must work correctly on the `modfft1.c` test program from Project 3. As mentioned earlier, `modfft2.c` is not a valid test program due to the use of pointer-typed variables.
- You can name your temporaries `_t1`, `_t2`, `_t3`, etc. Assume that the input program does not use such names.
- You need to transform only expression statements and return statements. For everything else in the input program, just use your printing from Project 3. For example, you should not modify in any way the expressions in variable declarations – e.g., `float w=x+y+z;` should remain the same. Similarly, you should not modify the expressions in loops and ifs – e.g., `for(w=x+y+z;w<p+q+r;w+=a+b+c)` should remain the same. Note that the conditions of loops/ifs and the initializations of loops (e.g., `w=x+y+z` and `w<p+q+r` above) are represented in the Sage AST as `SgExprStatement` nodes. Unlike the “normal” `SgExprStatement` nodes for expression/return statements, these nodes should not be translated to three-address code.

**Project Submission**

On or before 11:59 pm on the due date, you should submit a single file `intermediate1.cpp` containing all of your code; it should work with `main` from Project 3. Submit your project using

```
submit c5343aa lab4 intermediate1.cpp
```

If the timestamp on your electronic submission is 12:00 am on the next day or later, you will receive 10% reduction per day, for up to three days. If your submission is later than 3 days after the deadline, it will not be accepted and you will receive zero points for this project. If you resubmit your project, this will override any previous submissions and only the latest submission will be considered – resubmit at your own risk.

**Academic Integrity**

The project you submit must be entirely your own work. Minor consultations with others in the class are OK, but they should be at a very high level, without any specific details. The work on the project should be entirely your own: all the design, programming, testing, and debugging should be done only by you, independently and from scratch. Sharing your code with others is not acceptable. Submissions that show excessive similarities will be taken as evidence of cheating and dealt with accordingly; this includes any similarities with projects submitted in previous instances of this course. Academic misconduct is an extremely serious offense with severe consequences. Additional details on academic integrity are available from the Committee on Academic Misconduct (see http://oaa.osu.edu/coamresources.html). I strongly recommend that you check this URL. If you have any questions about university policies or what constitutes academic misconduct in this course, please contact me immediately.