All exercises must be done by yourself. You may discuss questions and potential solutions with your classmates, but you may not look at their code or their solutions. If in doubt, ask the instructor.

Acknowledge all sources you found useful. Use comments in code when appropriate.

Assignments must be submitted online to Gradescope. You are allowed to make unlimited submissions. Only the **last** submission before the final assignment submission will count towards your grade.

All code in the exercises will assume C99 semantics which is the version supported by `pycparser`. For this assignment, test cases will involve the following subset of the C language:

1. Integer (`int`, `long`, `char`) types, both signed and unsigned.
2. Pointers
3. Arithmetic, Boolean, Logical operators
4. Function calls
5. Conditional execution (`if`, `switch`)
6. Loops (`while`, `do while`)
7. Control flow (`break`, `continue`, `return`)

You will use the same 3-address code format as in Homework 1.

The overall goal of this assignment is to write a compiler that reduces C programs to 3 address code, identifies basic blocks in 3 address code, and ultimately generates a control flow graph (CFG).

I've given a *suggested* list of steps to follow below for each exercise, but they are hints and non-normative.

Exercise 1

Write a program `gen3ac.py` that uses pycparser to parse C code and rewrites the AST to 3 address form. Note that the 3 address form we're using in this course is also valid C code, so use pycparser's AST structures and code generator to generate the output file.

Your program will be called as:

```
python3 gen3ac.py preprocessed-input.c output3ac.c
```

The input supplied will be preprocessed. You should write your output to *`output3ac.c`*.

SUGGESTED STEPS:

1. Accept command line arguments in Python (use argparse)
2. Use pycparser to parse input file
3. Walk the AST and output 3 address code to a file using `print` or similar. Use this step to debug your 3 address translation (note that since we're using labels and not addresses for branches, forward references will *not* require a fixup pass).
4. Walk the AST, and instead of printing the code, *generate* a new AST containing 3 address code.
5. Write this newly generated AST to a output file using pycparser's generate (see example).

Exercise 2

Write a program `genbb.py` that takes the output file generated by `gen3ac.py` and identifies all the basic blocks in the source code.

Note this exercise requires you to output the list of basic blocks as a correctness check, but you should to maintain internal structures to track the basic blocks so you can create a control flow graph in the next exercise.

Your program will be called as:

```
python3 genbb.py output-from-3ac.c output.txt
```

Your output should consist of the following lines for each basic block:

```
BBID:
    instructions in basic block
```

where BBID is a string of the form “BB%03d” (e.g. BB001, BB002, etc.).

SUGGESTED STEPS:

1. Working one function at a time, identify all possible “entry instructions” (i.e. first instruction, or target of branch instruction)
2. Identify all ending instructions, (i.e. last instruction, or instruction before entry, or branch instruction).
3. Create internal data structure of basic blocks – ID and list of instructions.
4. Output in the required format.

Exercise 3

Write a program `cfg.py` that constructs a CFG. Use the basic block information from the previous exercise.

Again, while this exercise requires you to output the control flow graph, this is just a sanity check. Maintain internal data structures so that the CFG you build can be used in future assignments.

Your program will be called as:

```
python3 cfg.py output-from-3ac.c edgelist.txt output.dot
```

The file `edgelist.txt` should be produced by `cfg.py` and contain an edge of the CFG, one per line, as follows:

```
ENTRY BB001
BB001 EXIT
```

The `output.dot` file contains a graph specification in graphviz directed graph format. You can use the Python `graphviz` library to generate the dot file.

You can create a picture from `output.dot` using the command `dot -Tsvg output.dot -o output.svg`. The resulting SVG file can be viewed in any browser.

Note the input to `cfg.py` is the output of `gen3ac.py`.

SUGGESTED STEPS:

1. Write data structure code to build and store a graph.
2. Map all the basic blocks identified to nodes in this graph.
3. Create edges from a node that contains a jump instruction to the destination basic block.
4. Add ENTRY and EXIT nodes.
5. Walk the CFG to generate the output file.

Exercise 4

Write a program `vn.py` that computes local value numbers to a CFG to eliminate redundant computation.

Your program will be called as:

```
python3 vn.py output-from-3ac.c output.c
```

The code for `vn.py` will be after processing by `gen3ac.py`.

Build a CFG and apply value numbering to each basic block, eliminating as many redundant expressions as you can.

Write the transformed AST to *`output.c`*, note this requires a special format. See the `README-vn.html` file in the assignment for more information. **IMPORTANT:** Pass the labels from `input-from-3ac.c` to `output.c` unchanged.

END.