CSE 5343, Programming Project 6: Analysis of Three-Address Code
Due Tuesday, April 9, 11:59 pm (30 points)

The goal of this project is to generate and analyze a control-flow graph for the three-address code generated by your Project 5. You can assume that a correct implementation of Project 5 has produced a compilable C program, and that program is the input to your Project 6 (with some simplifying assumptions, as described later).

The starting point of your project will be your implementation of **Project 2** (with necessary bug fixes). You will modify the scanner and the parser to accept some new program elements, as described later. Then you will treat the resulting AST as the representation of the three-address code and will construct a control-flow graph for this representation. Create a directory `p6` under `proj`. Copy your Project 2 to `p6` and start from there. Do all work for Project 6 in `p6`.

Changes to the front end

The output from Project 5 contains several language features that are not handled by Project 2. Specifically, we have (1) `goto` statements, (2) labels, and (3) "not" expressions using the unary `!` operator. Make the following changes to handle them:
Scanner: add a new keyword `goto` and new symbols "colon" `:` and "not" `!`
Parser: add two new productions for <stmt>:

- `goto ident ;`
- `ident :`

The first one represents a `goto` statement. The second one represents a label. Strictly speaking, a label is not a statement, but for simplicity we will treat it as a statement in the AST. Also add a production for expressions of the form `!` <expr>
AST: add classes `GotoStmt extends Stmt`, `Label extends Stmt`, and `NotExpr extends Expr`

Simplifying assumptions. To simplify your implementation, assume that the input program satisfies the following restrictions. First, the code contains exactly one return statement, and that statement appears as the very last statement in the code. That is, list sList in class Program will contain only one ReturnStmt, which will be the last element in that list. Second, the program does not contain dead code: in the resulting CFG, all nodes will be reachable from the artificial ENTRY node and will be able to reach the artificial EXIT node.

You do **not** have to check these properties; just assume that they are always true. When you write your test cases, make sure they satisfy these restrictions.

Example. Suppose the input is the following C code
```
int f()
{
  int x;
  int y;
  int _t1;
  int _t2;
  x = 1;               // sList(0)
  y = 2;               // sList(1)
  _t1 = x < y;         // sList(2)
  if (!t1) goto _l2;   // sList(3)
  _t2 = x;             // sList(4)
  goto _l1;            // sList(5)
  _l2:                 // sList(6)
  _t2 = y;             // sList(7)
  _l1:                 // sList(8)
  return _t2;          // sList(9)
}
```

This input will produce Program.sList with five ExpressionStmt objects, one IfStmt object, one GotoStmt object, two Label objects, and one ReturnStmt (the last list element). The IfStmt object will have as children a NotExpr object (for `!t1`) and a GotoStmt object (for `goto _l2`). For illustration, sList(0) … sList(9) above are used to denote positions in Program.sList.

CFG construction

After parsing, call `astRoot.cfgAnalysis()` from `main` to perform control-flow analysis. First, create the CFG as described in class. The CFG should contain an artificial ENTRY node and an artificial EXIT node. Use some implementation of adjacency lists to be able to traverse CFG edges in both forward and backward direction during the subsequent CFG analysis.

Each CFG node, except for ENTRY and EXIT, corresponds to a basic block. The three-address instructions in the basic block define a list of Stmt objects (but none of these objects will be Label objects, since labels are not three-address instructions). For the example from above, there are 4 basic blocks: B1 = sList(0), sList(1), sList(2), sList(3); B2 = sList(4), sList(5); B3 = sList(7); B4 = sList(9). Therefore, there are 6 CFG nodes: ENTRY, B1, B2, B3, B4, and EXIT. There are 6 CFG edges: ENTRY → B1, B1 → B2, B1 → B3, B2 → B4, B3 → B4, B4 → EXIT.

At the end of CFG construction, print the number of nodes and edges in the graph (including ENTRY and EXIT), in the following format:

```
CFG NODES: …
CFG EDGES: …
```

For example, for the input code shown earlier, your project should print

```
CFG NODES: 6
CFG EDGES: 6
```

Print in this format **exactly** (including spaces and upper case) to simplify grading.

Note: Keep in mind that it is possible to have a sequence of labels in the input code: e.g.

```
 …
 _l2:
 _l3:
 _t3 = y < 20;
 …
```

CFG analysis

After the CFG is constructed, perform a simple loop analysis. Specifically, identify the back edges using depth-first traversal of the graph. Assume that the CFG is reducible – thus, the set of *back edges* is the same as the set of *retreating edges* identified during the traversal (see slides 22-23). For each back edge, find its natural loop (slide 19). Print the following:

```
BACK EDGES: …
NODES IN LOOPS: …
```

The first number is the number of identified back edges. The second one is the total number of CFG nodes that belong to at least one natural loop. If a node belongs to several natural loops, it should be counted only once.

Testing

Write many test cases with three-address code and test your implementation with them. Or, you can run your Project 5 to generate the three-address code – but if you do so, you must modify it to conform to the simplifying assumptions described earlier. Submit at least 5 test cases with your submission. The test cases you submit will not affect your score for the project. Put them in the same location as the provided file t1.c and name them t2.c, …

Your submission must work correctly on the three-address code for test program lpc.c (but you need to remove the trailing semicolon produced by your Project 5). *You can expect that a substantial number of points in the grading will be related to this test case.*

Submission

After completing your project, do

```
cd p6
make clean
cd ..
tar -cvzf p6.tar.gz p6
```

Then submit `p6.tar.gz` in Carmen.

General rules (copied from the course syllabus)

Your submissions must be uploaded via Carmen by midnight on the due date. The projects must compile and run on **stdlinux**. Some students prefer to implement the projects on a different machine, and then port them to stdlinux. If you decide to use a different machine, it is entirely your responsibility to make the code compile and run correctly on stdlinux before the deadline. In the past many students have tried to port to stdlinux too close to the deadline, leading to last-minute problems and missed deadlines.

Projects should be done independently. General high-level discussion of projects with other students in the class is allowed, but **you must do all design, programming, testing, and debugging independently.** Projects that show excessive similarities will be taken as evidence of cheating and dealt with accordingly. Code plagiarism tools may be used to detect cheating. See the syllabus under “Academic Integrity”.

The projects are due by 11:59 pm on the due day. You can submit up to 24 hours after the deadline; if you do so, your score will be reduced by 10%. **ONLY THE LAST SUBMITTED VERSION WILL BE CONSIDERED. Triple-check carefully that you have submitted the correct version. If you submit the wrong version of your code, and you get a low score (or zero score), I will NOT consider resubmissions – the original low/zero score will be assigned WITHOUT DISCUSSION.**

If you submit more than 24 hours after the deadline, the submission will not be accepted. NO EXCEPTIONS TO THIS RULE WILL BE CONSIDERED. NO REQUESTS FOR RESUBMISSION WILL BE CONSIDERED. MAKE SURE YOU SUBMIT THE CORRECT CODE VERSION.

Read the project description **very carefully, several times, start-to-end.** If you need any clarifications, contact me immediately (do **not** wait until the last minute). **Test extensively.**

Accommodations for sickness and other special circumstances will be made based on university guidelines. Please contact me **ahead of time** to arrange for such accommodations.