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.

There are no soft milestones for this assignment. Please submit your complete solution before the final due date.

This assignment can be completed by a team of 2 (you can form your own team, teams of 1 are okay as well). Submit the assignment individually. Submit the same code, but write individual reports.

Submit a ZIP or compressed tarball containing all your code. Your assignments will be tested on CSUG/CS machines (cycle1), and must run on them. **A skeleton LLVM pass and instructions to use it are enclosed in this assignment as llvm-pass-skeleton.tar.gz**. You can use that as a starting point for your assignment.

The overall goal of this assignment is to learn to use LLVM, by writing an LLVM pass to perform a loop dependence analysis. We will *not* be writing LLVM transformations.

I strongly recommend that you read the Writing an LLVM Pass LLVM documentation, and make sure you are able to perform the steps in "Quick Start" section. *Do this as quickly as possible and report back on Piazza.*

(Optional, but recommended) Using the LLVM `opt` tool, run existing passes that correspond to the passes you wrote in Python – domfrontier, domtree, dot-cfg, dot-dom-only, dot-dom-only. Explore their source code. Rather than give you detailed instructions, this exercise requires that you explore LLVM's documentation. Feel free to post to Piazza if you can't figure something out.

Exercise 1

(Using existing passes to identify and canonicalize induction variables) Read the list of passes in the LLVM documentation of LLVM's Analysis and Transform Passes, and write a pass that writes out the list of induction variables.

Hint: Recall the LLVM tutorial, and the advice to focus on Module/Function/BasicBlock passes? This exercise requires that you don't follow that advice.

(This exercise is intended as a warmup and will not be checked. However, it may be a good check to see if your team partner is up to speed.)

Exercise 2

(Generate an ILP for dependence analysis) Your LLVM pass will take in a piece of C code, and for the loop nest in the program (we'll stick to one loop nest per program, for now), generate an integer linear program in a form that can be solved using the `lp_solve` command (which is part of the GNU MathProg suite.)

Your pass will be invoked as:

```
opt OPTARGS -load=ilpdep.so -ilpdep -ilpoutput=output.ilp input.c.
```

It should read *`input.c`* and output the ILP to *`output.ilp`*. Use the LLVM CommandLine library to implement `-ilpoutput`. Note that the name for your pass is `ilpdep`. Setup the CMakeLists so that it is compiled to *ilpdep.so*.

If you provide an entirely optional file called `OPTARGS` in your submission, we will read that file and pass its entire contents to `opt` as additional arguments (*`OPTARGS`*). For example if your `OPTARGS` file contains (one word per line):

```
-print-after-all
-indvars
```

Then we will call `opt` as `opt -print-after-all -indvars -load=ilpdep.so ....`

Then, GLPK will be invoked thusly to solve the ILP:

```
glpsol --math output.ilp
```

See the GNU Mathprog documentation for the syntax of the `output.ilp` file.

END.