Project 2: Local Value Numbering (LLVM)

Points:
- 10 out of 35 (total of project)

Due:
- Feb 11th (Thu) 23:59

Objectives:
- review the ideas of value numbering
- learn how to implement a basic program analysis pass

Tasks:
- **Step-1**: Read the instructions in the **HelloPass-LLVM** repo (see below) and understand how it works. use the LLVM documents for more references about the APIs.

- **Step-2**: Implement the basic local value numbering (LVN), with the following specifications:
    - find all LVN-identifiable redundant computations in the given function;
    - no need to remove the redundant computations, but **only identify** them;
    - your implementation only needs to handle a **basic scenario**, where
        - all variables are *local* variables;
        - all variables are of *primitive* data types;
        - no branching statements (including if-else and loops);
        - only redundant arithmetic operations (`+`,`-`,`*`,`/`) like `a + b` need to be considered;
        - no need to handle commutative cases (like `a + b` vs. `b + a`).

  Tips:
    - *As redundancy removal is not required, renaming the variables is not necessary;*
    - *`load/store` operations in the IR can be treated as copy operations (like `b = a`);*

- **Step-3**: Test your implementation to make sure it works correctly. Some test cases will be provided.

Delivery:
- A PDF report:
    - first summarizes the algorithm and major data structures
    - then explains the implementation details: source code pieces + clear explanation
- A source code package (following the same structure as in **HelloPass-LLVM**)
- A video demo with voice (3-5 mins, you may use Zoom for recording) showing:
    - your source code and its compilation;
    - loading the `ValueNumbering` pass and running it on test cases;
    - successfully identifying redundant computations in the test cases.

Grading Criteria:
- Correctness of the implementation (instructor may test your implementation with more test cases);
- The clarity of the report (details matter) and the video demo.

Reference:
- **HelloPass-LLVM**: https://github.com/ufarooq/HelloPass-LLVM
- The LLVM Compiler Infrastructure
- Writing an LLVM Pass

Input Format:

- `test.ll` files generated using `clang -S -emit-llvm test.c -o test.ll`

Output Format:

- For `load/store` and arithmetic operations, print out their LLVM IR instructions, followed by the value numbers of operands and defined variables. Label each redundant computation.

Example (LLVM 8.0.0):

- Input (`test.ll`)

```
...
define i32 @test(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  %e = alloca i32, align 4
  %f = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %c, align 4
```

```
  %2 = load i32, i32* %b.addr, align 4
  %3 = load i32, i32* %c, align 4
  %add1 = add nsw i32 %2, %3
  store i32 %add1, i32* %d, align 4
  %4 = load i32, i32* %a.addr, align 4
  %5 = load i32, i32* %b.addr, align 4
  %add2 = add nsw i32 %4, %5
  store i32 %add2, i32* %e, align 4
  %6 = load i32, i32* %b.addr, align 4
  %7 = load i32, i32* %e, align 4
  %add3 = add nsw i32 %6, %7
  store i32 %add3, i32* %f, align 4
  %8 = load i32, i32* %f, align 4
  ret i32 %8
}
...
```

- Output

```
ValueNumbering: test()
   store i32 %a, i32* %a.addr, align 4    1 = 1
   store i32 %b, i32* %b.addr, align 4    2 = 2
   %0 = load i32, i32* %a.addr, align 4   1 = 1
   %1 = load i32, i32* %b.addr, align 4   2 = 2
   %add = add nsw i32 %0, %1              3 = 1 add 2
   store i32 %add, i32* %c, align 4       3 = 3
   %2 = load i32, i32* %b.addr, align 4   2 = 2
   %3 = load i32, i32* %c, align 4        3 = 3
   %add1 = add nsw i32 %2, %3             4 = 2 add 3
   store i32 %add1, i32* %d, align 4      4 = 4
   %4 = load i32, i32* %a.addr, align 4   1 = 1
   %5 = load i32, i32* %b.addr, align 4   2 = 2
   %add2 = add nsw i32 %4, %5             3 = 1 add 2 (redundant)
   store i32 %add2, i32* %e, align 4      3 = 3
   %6 = load i32, i32* %b.addr, align 4   2 = 2
   %7 = load i32, i32* %e, align 4        3 = 3
   %add3 = add nsw i32 %6, %7             4 = 2 add 3 (redundant)
   store i32 %add3, i32* %f, align 4      4 = 4
   %8 = load i32, i32* %f, align 4        4 = 4
```

Tip: use `formatv()` for formatting output as in `errs() << formatv("{0,-40}", inst);`

- Output for test1.c

```
ValueNumbering: test1()
   store i32 %a, i32* %a.addr, align 4    1 = 1
   store i32 %b, i32* %b.addr, align 4    2 = 2
   store i32 %c, i32* %c.addr, align 4    3 = 3
   store i32 %d, i32* %d.addr, align 4    4 = 4
   store i32 %e, i32* %e.addr, align 4    5 = 5
   store i32 %f, i32* %f.addr, align 4    6 = 6
   store i32 %g, i32* %g.addr, align 4    7 = 7
   %0 = load i32, i32* %a.addr, align 4   1 = 1
   %1 = load i32, i32* %b.addr, align 4   2 = 2
   %add = add nsw i32 %0, %1              8 = 1 add 2
   store i32 %add, i32* %c.addr, align 4  8 = 8
   %2 = load i32, i32* %c.addr, align 4   8 = 8
   %add1 = add nsw i32 %2, 5              10 = 8 add 9
   store i32 %add1, i32* %d.addr, align 4 10 = 10
   %3 = load i32, i32* %a.addr, align 4   1 = 1
   %4 = load i32, i32* %b.addr, align 4   2 = 2
   %add2 = add nsw i32 %3, %4             8 = 1 add 2 (redundant)
   store i32 %add2, i32* %e.addr, align 4 8 = 8
   %5 = load i32, i32* %e.addr, align 4   8 = 8
   %add3 = add nsw i32 %5, 5              10 = 8 add 9 (redundant)
   store i32 %add3, i32* %f.addr, align 4 10 = 10
   %6 = load i32, i32* %d.addr, align 4   10 = 10
   %7 = load i32, i32* %f.addr, align 4   10 = 10
   %add4 = add nsw i32 %6, %7             11 = 10 add 10
   store i32 %add4, i32* %g.addr, align 4 11 = 11
```

- Output for test2.c

```
ValueNumbering: test2
   store i32 %a, i32* %a.addr, align 4    1 = 1
   store i32 %b, i32* %b.addr, align 4    2 = 2
   store i32 %c, i32* %c.addr, align 4    3 = 3
   store i32 %x, i32* %x.addr, align 4    4 = 4
   store i32 %y, i32* %y.addr, align 4    5 = 5
   %0 = load i32, i32* %x.addr, align 4   4 = 4
   %1 = load i32, i32* %y.addr, align 4   5 = 5
   %add = add nsw i32 %0, %1              6 = 4 add 5
   store i32 %add, i32* %b.addr, align 4  6 = 6
   %2 = load i32, i32* %x.addr, align 4   4 = 4
   %3 = load i32, i32* %y.addr, align 4   5 = 5
   %add1 = add nsw i32 %2, %3             6 = 4 add 5 (redundant)
   store i32 %add1, i32* %a.addr, align 4 6 = 6
   store i32 17, i32* %a.addr, align 4    7 = 7
   %4 = load i32, i32* %x.addr, align 4   4 = 4
   %5 = load i32, i32* %y.addr, align 4   5 = 5
   %add2 = add nsw i32 %4, %5             6 = 4 add 5 (redundant)
   store i32 %add2, i32* %c.addr, align 4 6 = 6
```