CS 363 – Homework #2 – Finding Loops
Due Monday November 6, 2017

The purpose of this assignment is to detect loops in a program's control flow. Loop detection is an essential part of compiler optimization because most of the CPU time is spent in the body of loops.

To simplify this assignment, the input will not be source code, but rather just the list of blocks that make up a computer program. A program consists of one or more functions, and each function consists of one or more blocks. Blocks do consist of instructions, but for the purpose of this assignment, the basic block is the lowest level of analysis we need in order to find loops. A compiler can figure out where the blocks begin and end based on where it generates branch and jump instructions. Your task is to find the loops in each function. A loop will consist of one or more blocks contained within the same function.

The program will run interactively. Ask the user for the name of the input file. An example input file is provided below, but note that each line of the file will look like one of these possibilities:

- Blank – just skip the line
- "function" followed by the name of the function – introduces a new function
- Starts with a number – this represents a block. If the number appears alone, then the block has no successors. Otherwise you will see "→" followed by the successors.

The process of finding loops usually follows these steps:

1. Find the (immediate) successors of each block. This is given to you in the input file. You may assume that a block will have no more than 2 successors.
2. Find the (immediate) predecessors of each block. Note that in general there is no limit to how many predecessor a block may have.
3. Find the dominators of each block. See the algorithm below.
4. Use the dominator information to find back edges. A back edge exists when a block can say that one of its successors is also a dominator. Then the block becomes the "tail" and the successor is the "header" of the loop. You may assume that at most *one* of the successors will be the header of a loop (i.e. You'll never have a situation where there are 2 successors and both are loop headers.)
5. Create a "loop" based on this back edge, and add other blocks to the loop. See the algorithm below.

To help you with the implementation, here are a couple of algorithms I found in another book (*Compilers: Principles, Techniques and Tools* by Aho, Sethi and Ullman, a.k.a. the "dragon" book because the cover features a computer game with a life-size dragon).

Here is the algorithm to find dominators. For each basic block b, we calculate D(b), the dominators of b. A block "a" dominates "b" if every execution path reaching b must go through a along the way. Assume that block 1 is the initial block of the function.

```
D(1) = { 1 }
for each b in the function except #1 do
  D(b) = all blocks in the function
while changes to any D(b) occur do
  for each b except #1 do
    D(b) = intersection of dominator sets of b's immediate predecessors,
             unioned with b itself
```

Once you have found a back edge, here is an algorithm to add all blocks to the loop containing that back edge. Let's say the back edge goes from n to d. From the dragon book: "*Beginning with node n, we consider each node m != d that we know is in the loop. Each node in the loop, except for d, is placed once on a stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors and thus find only those nodes that reach n without going through d*." Anyway, here is the algorithm:

```
  procedure insert(m)
    if m is not in loop then
      add m to loop, and push m onto stack
  main( )
    stack = empty
    loop = { d }
    insert(n)
    while stack is not empty do
      pop m; and for each predecessor p of m, insert(p)
```

Now, there is just more one thing to do for loop creation. The algorithm above only adds blocks to a loop according to a single back edge. But it is possible for a loop to have multiple back edges. What is unique about each loop is its *header block*. So, you need to traverse your set of loops that you created with the above algorithm, and see if there are any that have the same header. If so, combine them.

The output from your program should be a list of all the loops, according to which function they are in. For each loop, list its constituent blocks in ascending numerical order. Also state which block is the header of the loop. Note that the header is not always the lowest numbered block. When testing your program, you may also find it convenient to print out the predecessors and especially the dominators of all the blocks.

To simplify the implementation, you may assume that there are no more than 30 blocks in a function. This way, you can store the list of all the blocks in a loop or function in a single integer. An integer is represented as 32 bits, and each bit can represent whether a block is (1) or is not (0) present.

Here's an example input file:

```
function g
1 -> 2
2 -> 3 5
3 -> 4
4 -> 2
5 -> 6
6 -> 2 7
7
```

In this case your program should report that there is a loop in function g with header block 2 consisting of blocks 2 through 6.