BASIC BLOCKS

- Our first job is to partition a sequence of three-address instructions into basic blocks. We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction.
- In the absence of jumps and labels, control proceeds sequentially from one instruction to the next. This idea is formalized in the following algorithm.

Algorithm

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are leaders that is the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

- In generating the intermediate code, we have assumed that the real-valued array elements take 8bytes each, and that the matrix a is stored in row-major form.

Next-Use Information:

- Knowing when the value of a variable will be used next is essential for generating good code. If the value of a variable that is currently in a register will never be referenced subsequently, then that register can be assigned to another variable.
- The use of a name in a three-address statement is defined as follows. Suppose three-address statement i assigns a value to x. If statement j has x as an operand, and control can ow from statement i to j along a

path that has no intervening assignments to x, then we say statement j uses the value of x computed at statement i. We further say that x is live at statement i.

Algorithm

INPUT: A basic block B of three-address statements. We assume that the symbol table initially shows all non-temporary variables in B as being live on exit.

OUTPUT: At each statement i: x = y + z in B, we attach to i the liveness and next-use information of x, y, and z.

METHOD: We start at the last statement in B and scan backwards to the beginning of B. At each statement i: x = y + z in B, we do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
2. In the symbol table, set x to \not live" and \no next use."
3. In the symbol table, set y and z to \live" and the next uses of y and z to i

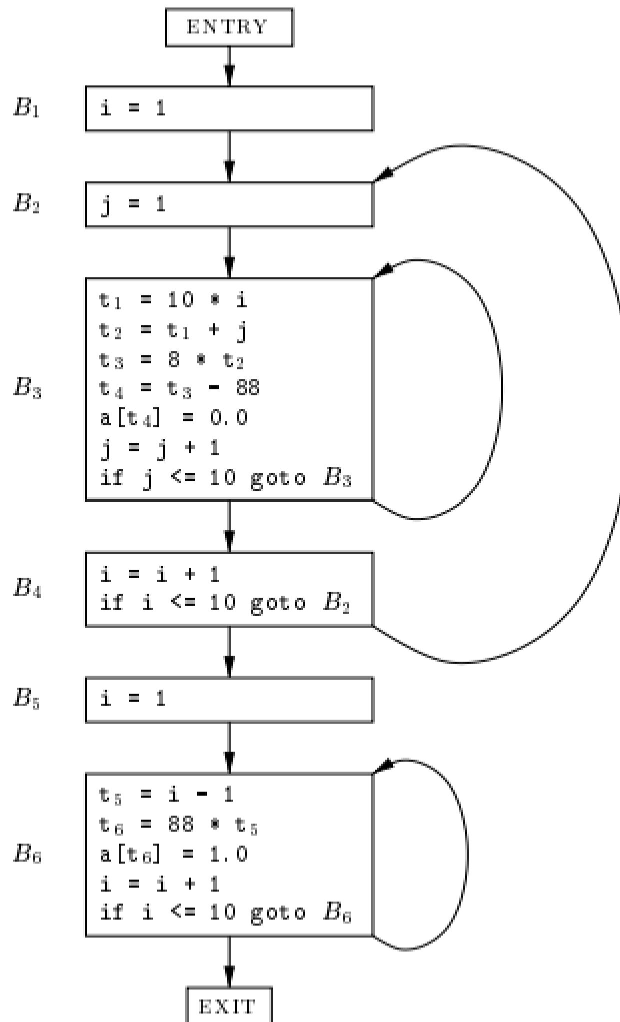
Here we have used + as a symbol representing any operator. If the three-address statement i is of the form x = + y or x = y, the steps are the same as above, ignoring z.

FLOW GRAPHS

- Once an intermediate-code program is partitioned into basic blocks, we represent the ow of control between them by a ow graph. The nodes of the flow graph are the basic blocks.
- There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B. There are two ways that such an edge could be justified:
- There is a conditional or unconditional jump from the end of B to the beginning of C.
- C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

Often, we add two nodes, called the entry and exit, that do not correspond to executable intermediate instructions. There is an edge from the entry to the first executable node of the ow graph, that is, to the basic block that comes from the first instruction of the intermediate code. There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program.

If the final instruction of the program is not an unconditional jump, then the block containing the final instruction of the program is one predecessor of the exit, but so is any basic block that has a jump to code that is not part of the program.



- The entry points to basic block B1, since B1 contains the first instruction of the program.
- The only successor of B1 is B2, because B1 does not end in an unconditional jump, and the leader of B2 immediately follows the end of B1.
- Block B3 has two successors. One is itself, because the leader of B3, instruction 3, is the target of the conditional jump at the end of B3, instruction 9.
- The other successor is B4, because control can fall through the conditional jump at the end of B3 and next enter the leader of B4.
- Only B6 points to the exit of the ow graph, since the only way to get to code that follows the program from which we constructed the ow graph is to fall through the conditional jump that ends B6.

Representation of flow graphs:

- Flow graphs, being quite ordinary graphs, can be represented by any of the data structures appropriate for graphs. The content of nodes (basic blocks) needs their own representation.
- We might represent the content of a node by a pointer to the leader in the array of three-address instructions, together with account of the number of instructions or a second pointer to the last instruction.

- However, since we may be changing the number of instructions in a basic block frequently, it is likely to be more efficient to create a linked list of instructions for each basic block.

Loops:

- Programming-language constructs like while-statements, do-while-statements, and for-statements naturally give rise to loops in programs.
- Since virtually every program spends most of its time in executing its loops, it is especially important for a compiler to generate good code for loops.
- Many code transformations depend upon the identification of "loops" in a ow graph. We say that a set of nodes L in a ow graph is a loop if L contains a node e called the loop entry, such that:
    1. e is not ENTRY, the entry of the entire ow graph.
    2. No node in L besides e has a predecessor outside L. That is, every path from ENTRY to any node in L goes through e.
    3. Every node in L has a nonempty path, completely within L, to e.