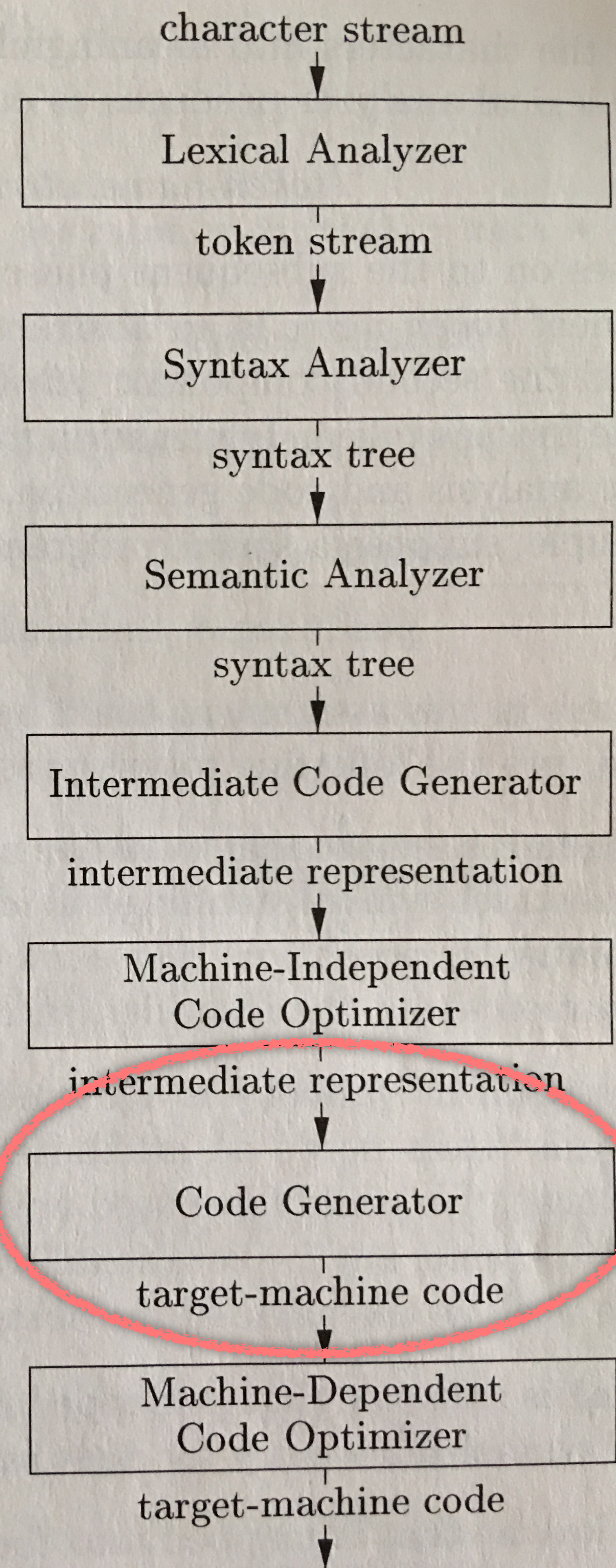
CSE443 Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Phases of a compiler



Target machine code generation

Figure 1.6, page 5 of text

Basic Blocks

&

Flow Graphs

Basic blocks and flow graphs

- To help us analyze the intermediate code we will group instructions from our program into "basic blocks".

Basic Block

A basic block is a "maximal sequence of consecutive three-address instructions with the properties that,

a) the flow of control can only enter the basic block through the first instruction in the block [...]

b) control will leave the block without halting or branching, except possibly at the last instruction in the block"

[p. 526]

Flow Graph

"The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks."

[p 526]

Partitioning IR into BB

"Algorithm 8.5 [p. 526]
INPUT: a sequence B of three-address instructions.
OUTPUT: a list of basic blocks for B, in which each instruction is assigned to exactly one basic block
METHOD: First, find leaders (see below).
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." [lightly edited from original]

"The rules for finding leaders are:
1) The first three address instruction (3AI) 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." [lightly edited from original]

Example

Figure 8.8 [p. 527]

```
for (i=1; i<=10; i=i+1) {
    for (j=1; j<=10; j=j+1) {
        a[i,j] = 0.0;
    }
}
for (i=1; i<=10; i=i+1) {
    a[i,i] = 1.0;
}
```

This code initializes a 10x10 real matrix to the identity matrix (1's along the main diagonal).

Assumptions:
matrix is of size 10x10 containing reals
a real occupies 8 bytes
matrix is stored in row-major form (see p. 382)

a[1,1]
a[1,2]
a[1,3]
a[1,4]
a[1,5]
a[1,6]
a[1,7]
a[1,8]
a[1,9]
a[1,10]
a[2,1]
a[2,2]
a[2,3]
a[2,4]
a[2,5]
a[2,6]
a[2,7]
a[2,8]
a[2,9]
a[2,10]

Example

Figure 8.7 [p. 527]

```
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)
```

A possible three-address code translation of the high-level program.

Identifying leaders

```
L  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)
```

Leaders are:

1. first instruction

Identifying leaders

```
L  1) i = 1
L  2) j = 1
L  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
L  13) t5 = i - 1
   14) t6 = 88 * t5
   15) a[t6] = 1.0
   16) i = i + 1
   17) if i <= 10 goto (13)
```

Leaders are:

1. first instruction
2. the target of any jump

Identifying leaders

```
L   1) i = 1
L   2) j = 1
L   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)
L   10)i = i + 1
    11)if i <= 10 goto (2)
L   12)i = 1
L   13)t5 = i - 1
    14)t6 = 88 * t5
    15)a[t6] = 1.0
    16)i = i + 1
    17)if i <= 10 goto (13)
```

Leaders are:

1. first instruction
2. the target of any jump
3. the instruction immediately after any jump

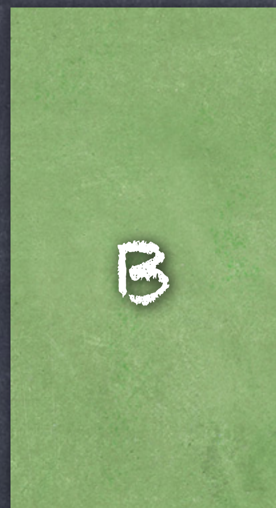
8.4.3 Flow Graphs

Each basic block is a node in the flow graph.

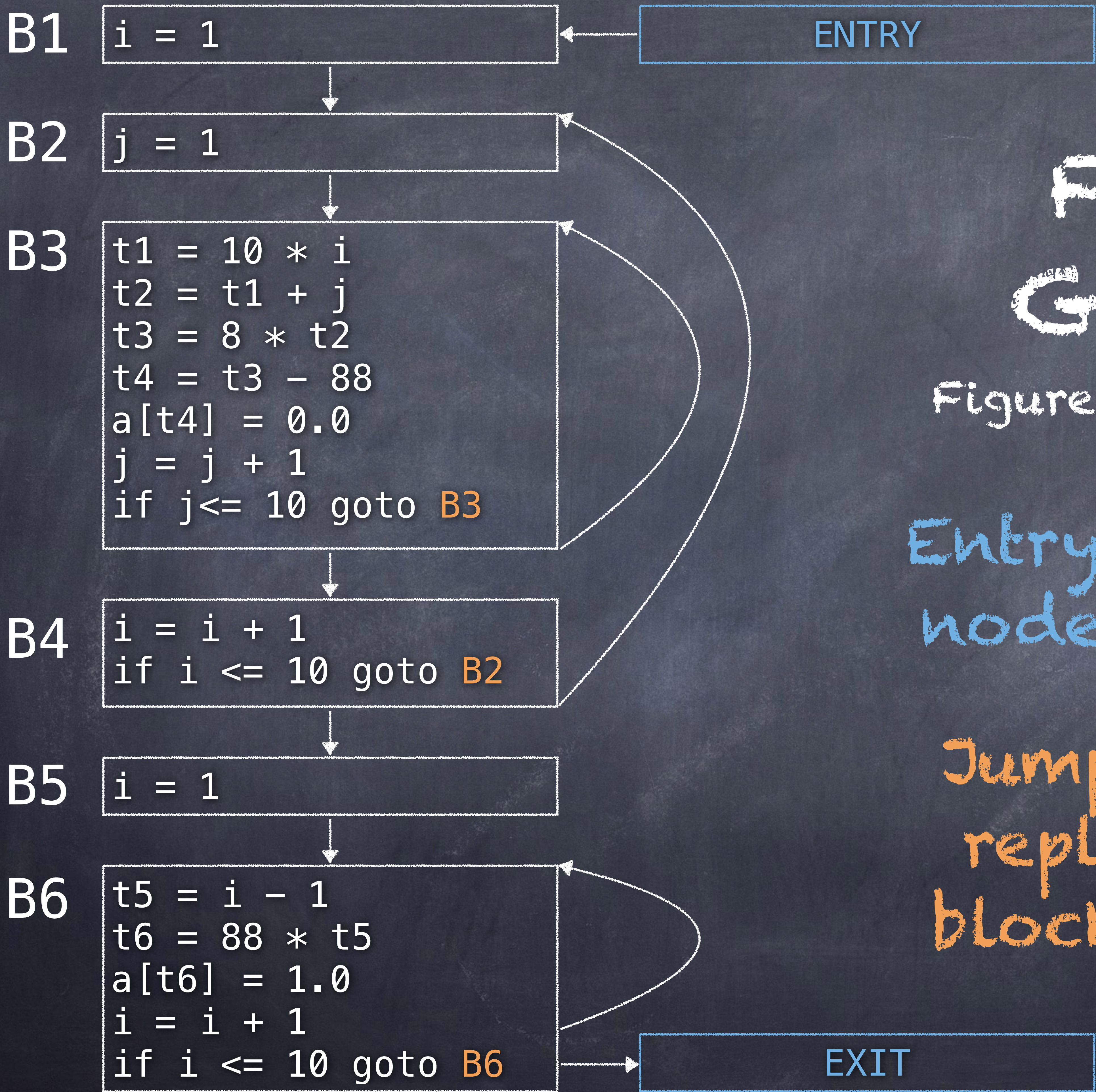
There is an edge between blocks B and C of the flow graph if:

1. there is a (conditional or unconditional) jump from from the end of B to the start of C, or
2. C immediately follows B and B does not end with an unconditional jump.

Terminology



- B is a predecessor of C
- C is a successor of B



Flow Graph

Figure 8.9 [p. 530]

Entry and exit nodes added.

Jump targets replaced by block names.

8.4.2 Liveness and next-use

"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."

8.4.2 Liveness and next-use

```
i:    x = ...
        .
        .
        .
j:    ... = x op ...
```

assuming there are no assignments to x between i and j

If statement j uses x, then x is live at i. Since we need the value of x we should try to keep it in a register.

8.4.2 Liveness and next-use

```
i:    ... x ...
       .
       .
       .
j:    x = ...
```

assuming there is no use of x between i and j

Statement j overwrites old value of x; we say x is dead at I. This means we need not preserve that value in a register.

Algorithm 8.7 [p. 528]
Determining the liveness and next-use information for each statement in a basic block.

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

Not this instruction specifically, but instructions of the form x = y op z, x = op y, or x = y.

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

Code Transformations on basic blocks

- Local optimizations can be performed on code inside basic blocks.
- Represent code *inside* a basic block as a DAG.
- The basic blocks will themselves be connected to form a flow graph.

Constructing DAG for basic blocks [p. 533]

1. For each variable in the block, create a node representing the variable's initial value.

2. For each statement s in the block, create a node N.

   "The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s."

Constructing DAG for basic blocks

3. For each node representing a statement, label it with the operator applied.

4. For each node representing a statement, attach a list of the variables for which it is the last definition within the block.

Constructing DAG for basic blocks

5. For each node representing a statement, its children are the nodes that are the last definitions of the operands used in the statement.

6. Identify as output nodes those whose variables are live on exit from the block ("their values may be used later, in another block of the flow graph")

Example 8.10 [p. 534]

1) a = b + c
2) b = a - d
3) c = b + c
4) d = a - d

Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

1) $a = b + c$
2) $b = a - d$
3) $c = b + c$
4) $d = a - d$

b_0 c_0

1. For each variable in the block, create a node representing the variable's initial value.

Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

1) a = b + c
2) b = a - d
3) c = b + c
4) d = a - d

2. For each statement s in the block, create a node N.

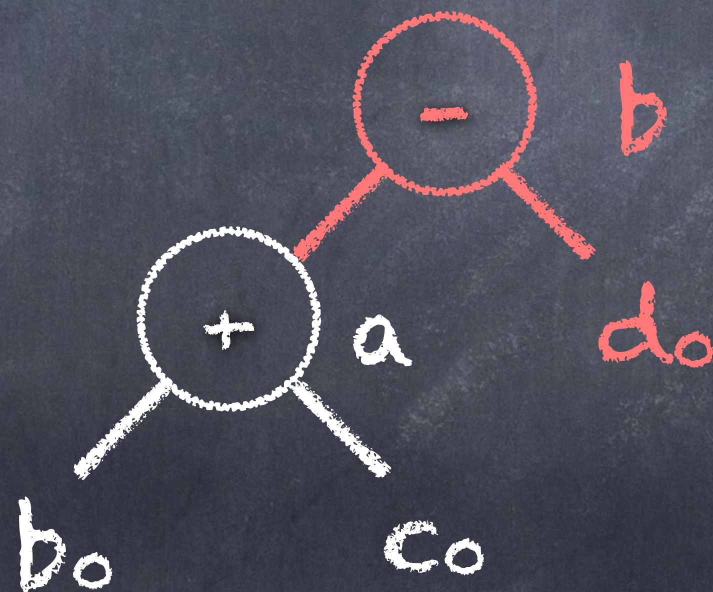
3. For each node representing a statement, label it with the operator applied.

4. For each node representing a statement, attach a list of the variables for which it is the last definition within the block.

Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

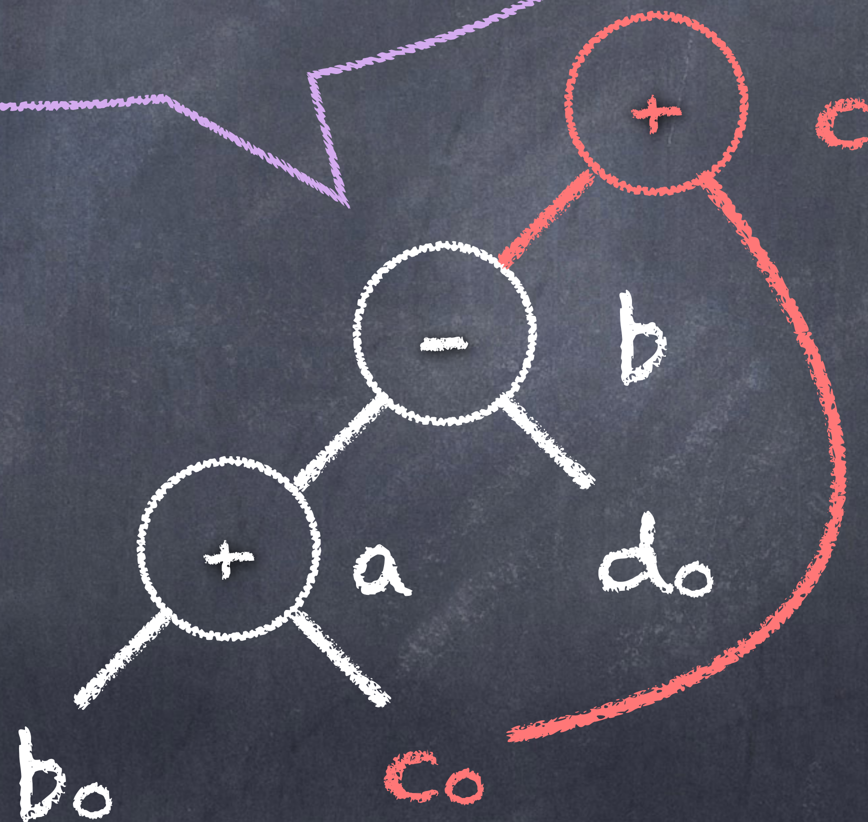
1) $a = b + c$
2) $b = a - d$
3) $c = b + c$
4) $d = a - d$



Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

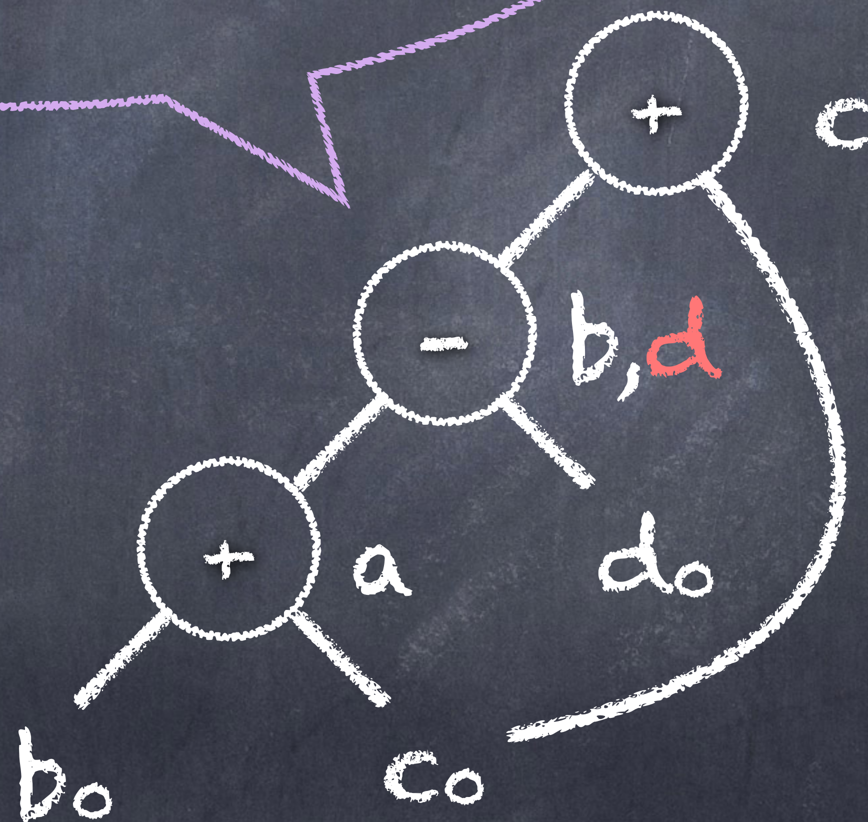
1) a = b + c
2) b = a - d
3) c = b + c
4) d = a - d



Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

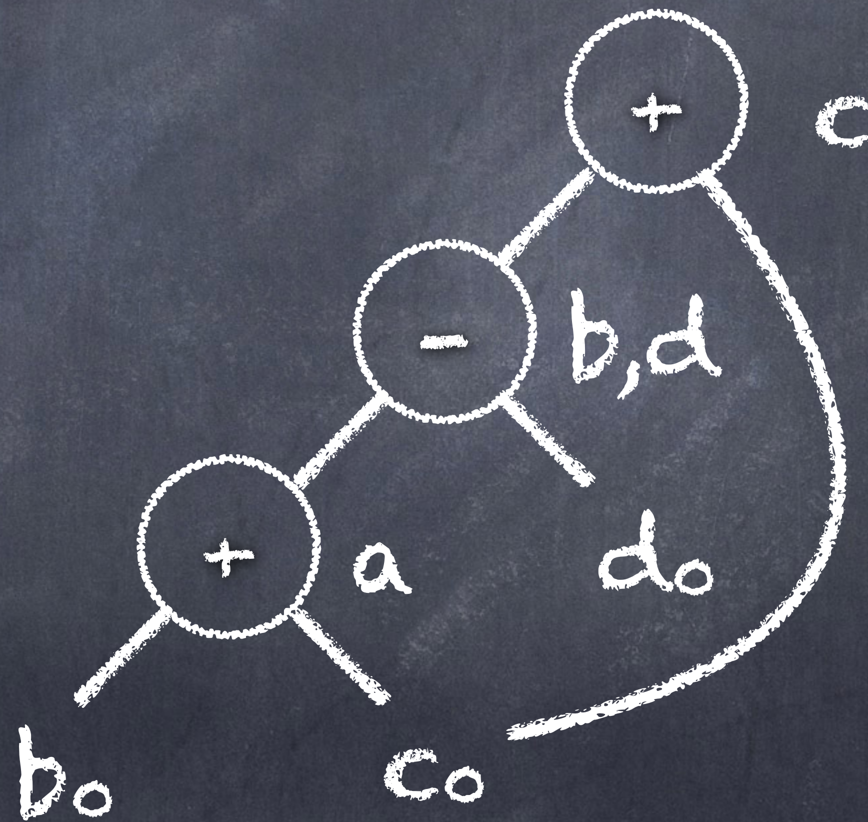
1) a = b + c
2) b = a - d
3) c = b + c
4) d = a - d



Example 8.10 [p. 534]

If b is live on exit:

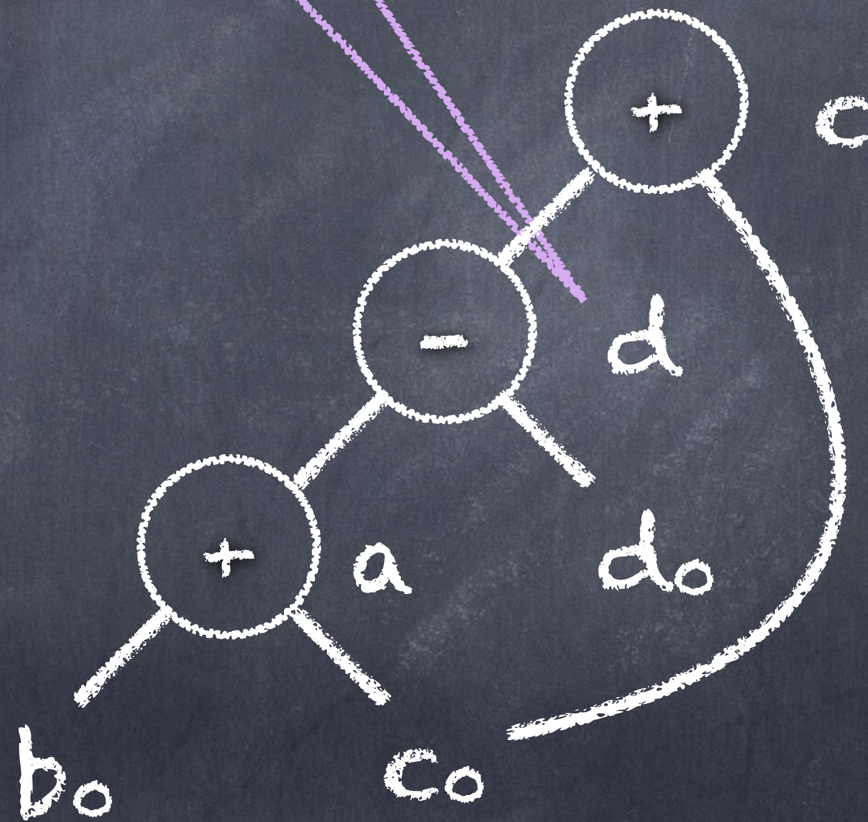
1) $a = b + c$
2) $b = a - d$
3) $c = b + c$
4) $d = b$



Example 8.10 [p. 534]

If b is not live on exit:

```
1) a = b + c
2) d = a - d
3) c = d + c
```



8.6 A Simple Code Generator [p. 542]

- algorithm focuses on generation of code for a single basic block
- generates code for each three address code instruction
- manages register allocations/assignment to avoid redundant loads/stores

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Principal uses of registers

- operator operands must be in registers
- temporaries needed within block
- variables that span multiple blocks
- stack pointer
- function arguments

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

"We [...] assume that for each operator, there is exactly one machine instruction that takes the necessary operands in registers and performs that operation, leaving the result in a register. The machine instructions are of the form:

- LD reg, mem
- ST mem, reg
- OP reg, reg, reg" [p. 543]

```
movl    MEM, REG
movl    REG, MEM
addl    REG, REG
```

x86 assembly resources (will add more as we go along)
https://en.wikipedia.org/wiki/X86_assembly_language
https://gcc-renesas.com/pdf/manuals/Assembler.pdf
man as   <-- at OS prompt

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

8.6.1 Register and Address Descriptors

A three-address instruction of the form:

```
v = a op b
```

we generate:

```
LD Rx, a
LD Ry, b
OP Rx, Rx, Ry
ST Rx, v
```

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

8.6.1 Register and Address Descriptors

A three-address instruction of the form:

v = a op b

we generate:

LD Rx, a
LD Ry, b
OP Rx, Rx, Ry
ST Rx, v

where a, b, and v are int

```
v = a + b
```

in x86 asm

```
movl    -4(%rbp), %edx
movl    -8(%rbp), %eax
addl    %edx, %eax
movl    %eax, -12(%rbp)
```

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent