CC Lecture 3

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

Three Address Code

- In **three-address code**, there is **at most one operator** on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.
- Thus, a source-language expression like **x + y * z** might be translated into the sequence of three-address instructions

  t1 = y * z

  t2 = x + t1

  where t1 and t2 are compiler-generated temporary names.

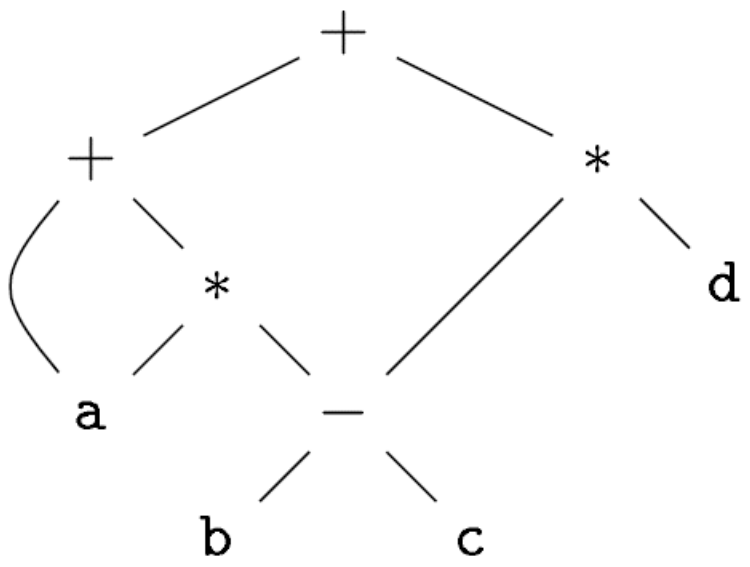
Benefits

- Desirable for target-code generation and optimization
  - multi-operator arithmetic expressions and of nested flow-of-control statements are unravelled

- The code can be rearranged easily
  - use of names for the intermediate values computed by a program

Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.

DAG (Directed Acyclic Graph)



Three address code

t1 = b - c

t2 = a * t1

t3 = a + t2

t4 = t1 * d

t5 = t3 + t4

Three address code = Address + Instructions

Address in Three address code

1. **A name.**
   - For convenience, we allow source-program names to appear as addresses in three-address code.
   - In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
2. **A constant.**
   - In practice, a compiler must deal with many different types of constants and variables.
   - Type conversions within expressions can be both implicit as well as explicit.

Address in Three address code

3. **A compiler-generated temporary.**
   - It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed.
   - These temporaries can be combined, if possible, when registers are allocated to variables.

Symbolic labels

- Symbolic labels will be used by instructions that alter the flow of control.
- A **symbolic label** represents the index of a three-address instruction in the sequence of instructions.
- Actual indexes can be substituted for the labels, either by making a separate pass or by “backpatching”.

Three-address instruction forms

1. **Assignment instructions** of the form **x = y op z**,
   - where op is a binary arithmetic or logical operation,
   - and x, y, and z are addresses.
2. **Assignments** of the form **x = op y**,
   - where op is a unary operation.
   - Essential unary operations include unary minus, logical negation, and conversion operators that, for example, convert an integer to a floating-point number.
3. **Copy instructions** of the form **x = y**,
   - where x is assigned the value of y.
4. **An unconditional jump goto L**.
   - The three-address instruction with label L is the next to be executed.

Three-address instruction forms

5. **Conditional jumps** of the form **if x goto L** and **ifFalse x goto L**.
   - These instructions execute the instruction with label L next if x is true and false, respectively.
   - Otherwise, the following three-address instruction in sequence is executed next, as usual.
6. **Conditional jumps** such as **if x relop y goto L**,
   - which apply a **relational operator** (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation relop to y.
   - If not, the three-address instruction following if x relop y goto L is executed next, in sequence.

Three-address instruction forms

7. **Procedure calls and returns** are implemented using the following instructions for procedure and function call
   - **param x for parameters**
   - **call p, n**
   - **y = call p, n**

e.g.

```
param x1
param x2
...
param xn
call p, n
```

- generated as part of a call of the procedure p(x1; x2; : : : ; xn).

Three-address instruction forms

- The integer n, indicating the number of actual parameters in **“call p, n,”** is not redundant because calls can be nested.

- That is, some of the first param statements could be parameters of a call that comes after p returns its value; that value becomes another parameter of the later call.

Three-address instruction forms

8. **Indexed copy instructions** of the form **x = y[i]** and **x[i]=y**.
   - The instruction x = y[i] sets x to the value in the location i memory units beyond location y.
   - The instruction x[i]=y sets the contents of the location i units beyond x to the value of y.

Three-address instruction forms

9. **Address and pointer assignments** of the form **x = &y**, **x = *y**, and ***** **x = y**.
   - The instruction x = &y sets the r-value of x to be the location (l-value) of y. Presumably y is a name, perhaps a temporary, that denotes an expression with an l-value such as A[i][j], and x is a pointer name or temporary.
   - In the instruction x = * y, presumably y is a pointer or a temporary whose r-value is a location. The r-value of x is made equal to the contents of that location.
   - Finally, * x = y sets the r-value of the object pointed to by x to the r-value of y.

```
void quicksort(int m, int n){
/* recursively sorts a[m] through a[n] */
int i, j; int v, x;
if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
            x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Generating Three address code for given code fragment

- **i = m-1;**
- **j = n;**
- **v = a[n];**

(1) i = m-1

(2) j = n

(3) t1 = 4 * n

(4) v = a[t1]

Here, it is assumed that int occupies 4 bytes.

Generating Three address code for given code fragment

```
do i = i + 1;
   while (a[i] < v);
```

```
(5) i = i + 1
(6) t2 = 4 * i
(7) t3 = a[t2]
(8) if t3<v goto (5)
```

```
do j = j - 1;
   while (a[j] > v);
```

```
(9) j = j - 1
(10) t4 = 4 * j
(11) t5 = a[t4]
(12) if t5>v goto (9)
```

Generating Three address code for given code fragment

if (i >= j) break;	(13) if i>=j goto **(23)**
x = a[i];	(14) t6 = 4 * i (15) x = a[t6]
a[i] = a[j];	(16) t7 = 4 * i (17) t8 = 4 * j (18) t9 = a[t8] (19) a[t7] = t9
a[j] = x; **/* swap a[i], a[j] */**	(20) t10 = 4 * j (21) a[t10] = x
} //closing of while(1)	(22) goto (5)

Generating Three address code for given code fragment

x = a[i];	(23) t11 = 4*i (24) x = a[t11]
a[i] = a[n];	(25) t12 = 4*i (26) t13 = 4*n (27) t14 = a[t13] (28) a[t12] = t14
a[n] = x; **/* swap a[i], a[n] */**	(29) t15 = 4*n (30) a[t15] = x

Three Address Code

(1) i = m-1
(2) j = n
(3) t1 = 4*n
(4) v = a[t1]
(5) i = i+1
(6) t2 = 4*i
(7) t3 = a[t2]
(8) if t3<v goto (5)
(9) j = j-1
(10) t4 = 4*j
(11) t5 = a[t4]
(12) if t5>v goto (9)
(13) if i>=j goto (23)
(14) t6 = 4*i
(15) x = a[t6]
(16) t7 = 4*i
(17) t8 = 4*j
(18) t9 = a[t8]
(19) a[t7] = t9
(20) t10 = 4*j
(21) a[t10] = x
(22) goto (5)
(23) t11 = 4*i
(24) x = a[t11]
(25) t12 = 4*i
(26) t13 = 4*n
(27) t14 = a[t13]
(28) a[t12] = t14
(29) t15 = 4*n
(30) a[t15] = x

Graph representation of intermediate code

1. Partition the intermediate code into basic blocks, which are maximal sequences 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. That is, there are no jumps into the middle of the block.
   b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.

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

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.

Algorithm 1: Partitioning three-address instructions into basic blocks.

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.

Algorithm 1: Partitioning three-address instructions into **basic blocks**.

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.

Example: Source code to set a 10 x 10 matrix to an identity matrix

```
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i; j] = 0:0;
for i from 1 to 10 do
    a[i; i] = 1:0;
```

Three address code

1. i = 1
2. j = 1
3. t_1 = 10 * i
4. t_2 = t_1 + j
5. t_3 = 8 * t_2
6. t_4 = t_3 – 88
7. a[t_4] = 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. t_5 = i – 1
14. t_6 = 88 * t5
15. a[t_6] = 1.0
16. i = i + 1
17. if i <= 10 goto (13)

Instruction 1 is a leader by rule (1) of Algorithm 1

1. **i = 1**
2. j = 1
3. $t_1 = 10 * i$
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. j = j + 1
9. if j <= 10 goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. $t_5 = i - 1$
14. $t_6 = 88 * t5$
15. $a[t_6] = 1.0$
16. i = i + 1
17. if i <= 10 goto (13)

To find the other leaders, we first need to find the jumps.

1. **i = 1**
2. j = 1
3. $t_1 = 10 * i$
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 – 88$
7. $a[t_4] = 0.0$
8. j = j + 1
9. if j <= 10 goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. $t_5 = i – 1$
14. $t_6 = 88 * t5$
15. $a[t_6] = 1.0$
16. i = i + 1
17. if i <= 10 goto (13)

By rule (2), the targets of these jumps are leaders

1. **i = 1**
2. **j = 1**
3. **t_1 = 10 * i**
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. j = j + 1
9. if j <= 10 goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. **t_5 = i – 1**
14. $t_6 = 88 * t5$
15. $a[t_6] = 1.0$
16. i = i + 1
17. if i <= 10 goto (13)

by rule (3), each instruction following a jump is a leader

1. **i = 1**
2. **j = 1**
3. **t_1 = 10 * i**
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 – 88$
7. $a[t_4] = 0.0$
8. j = j + 1
9. if j <= 10 goto (3)
10. **i = i + 1**
11. if i <= 10 goto (2)
12. **i = 1**
13. **t_5 = i – 1**
14. $t_6 = 88 * t5$
15. $a[t_6] = 1.0$
16. i = i + 1
17. if i <= 10 goto (13)

So, finally we have the leaders

1. **i = 1**
2. **j = 1**
3. **t_1 = 10 * i**
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. j = j + 1
9. if j <= 10 goto (3)
10. **i = i + 1**
11. if i <= 10 goto (2)
12. **i = 1**
13. **t_5 = i – 1**
14. $t_6 = 88 * t5$
15. $a[t_6] = 1.0$
16. i = i + 1
17. if i <= 10 goto (13)

Basic Blocks

1. **i = 1** **B1**

2. **j = 1** **B2**

3. **t_1 = 10 * i**

4. $t_2 = t_1 + j$

5. $t_3 = 8 * t_2$

6. $t_4 = t_3 - 88$ **B3**

7. $a[t_4] = 0.0$

8. j = j + 1

9. if j <= 10 goto (3)

10. **i = i + 1**

11. if i <= 10 goto (2) **B4**

12. **i = 1** **B5**

13. **t_5 = i – 1**

14. $t_6 = 88 * t5$

15. $a[t_6] = 1.0$ **B6**

16. i = i + 1

17. if i <= 10 goto (13)

Flow Graphs

- Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow 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 an edge could be justified:

1. There is a conditional or unconditional **jump** from the end of **B to** the beginning of **C**.
2. C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

- We say that B is a predecessor of C, and C is a successor of B.

Entry and Exit nodes

- There is an edge from the **entry** to the first executable node of the flow 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.

Flow graph for intermediate code to set a 10 x 10 matrix to an identity matrix:

1. $i = 1$ **B1**
2. $j = 1$ **B2**
3. $t_1 = 10 * i$
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 - 88$ **B3**
7. $a[t_4] = 0.0$
8. $j = j + 1$
9. if $j <= 10$ goto (3)
10. $i = i + 1$
11. if $i <= 10$ goto (2) **B4**
12. $i = 1$ **B5**
13. $t_5 = i - 1$
14. $t_6 = 88 * t5$
15. $a[t_6] = 1.0$ **B6**
16. $i = i + 1$
17. if $i <= 10$ goto (13)

Flow graph for intermediate code to set a 10 x 10 matrix to an identity matrix:

- The **entry points to basic block B_1**, since B_1 contains the first instruction of the program.

- The **only successor of B_1 is B_2**, because B_1 does not end in an unconditional jump, and the leader of B_2 immediately follows the end of B_1.

  ENTRY →B1 →B2

Flow graph for intermediate code to set a 10 x 10 matrix to an identity matrix:

- Block **B_3 has two successors**.
- One is **itself**, because the leader of B_3, instruction **3**, is the **target** of the conditional jump at the end of **B_3, instruction 9**.
- The **other successor is B_4**, because control can fall through the conditional jump at the end of B_3 and next enter the **leader of B_4**.

B3 → B3

↓

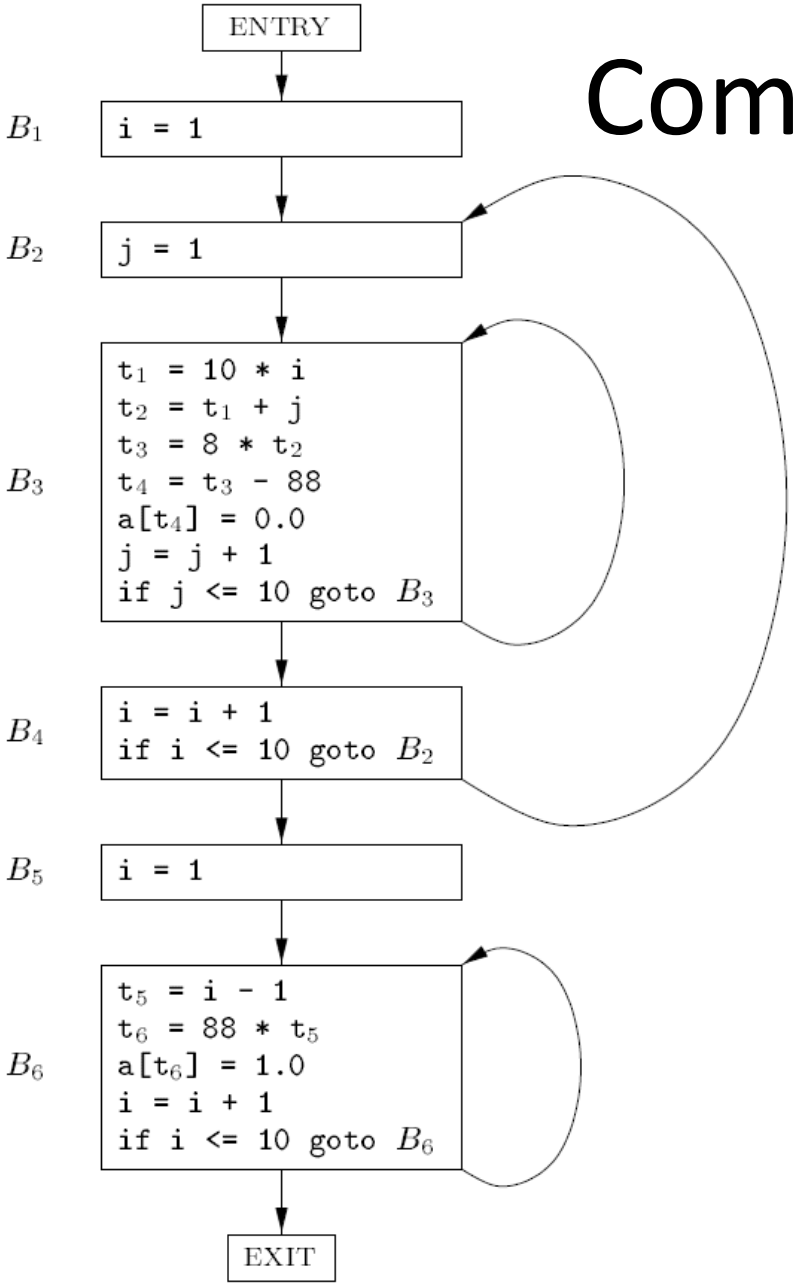
B4

Flow graph for intermediate code to set a 10 x 10 matrix to an identity matrix:

- Only **B_6 points to the exit** of the flow graph, since the only way to get to code that follows the program from which we constructed the flow graph is to fall through the conditional jump that ends B_6.

  B6 → EXIT

Complete Flow Graph



Three Address Code for Quicksort code fragment

(1) i = m-1
(2) j = n
(3) t1 = 4*n
(4) v = a[t1]
(5) i = i+1
(6) t2 = 4*i
(7) t3 = a[t2]
(8) if t3<v goto (5)
(9) j = j-1
(10) t4 = 4*j
(11) t5 = a[t4]
(12) if t5>v goto (9)
(13) if i>=j goto (23)
(14) t6 = 4*i
(15) x = a[t6]
(16) t7 = 4*i
(17) t8 = 4*j
(18) t9 = a[t8]
(19) a[t7] = t9
(20) t10 = 4*j
(21) a[t10] = x
(22) goto (5)
(23) t11 = 4*i
(24) x = a[t11]
(25) t12 = 4*i
(26) t13 = 4*n
(27) t14 = a[t13]
(28) a[t12] = t14
(29) t15 = 4*n
(30) a[t15] = x

What next??

- Find Leaders
- Decide the blocks
- Design flow graph by determining edges.

Leaders

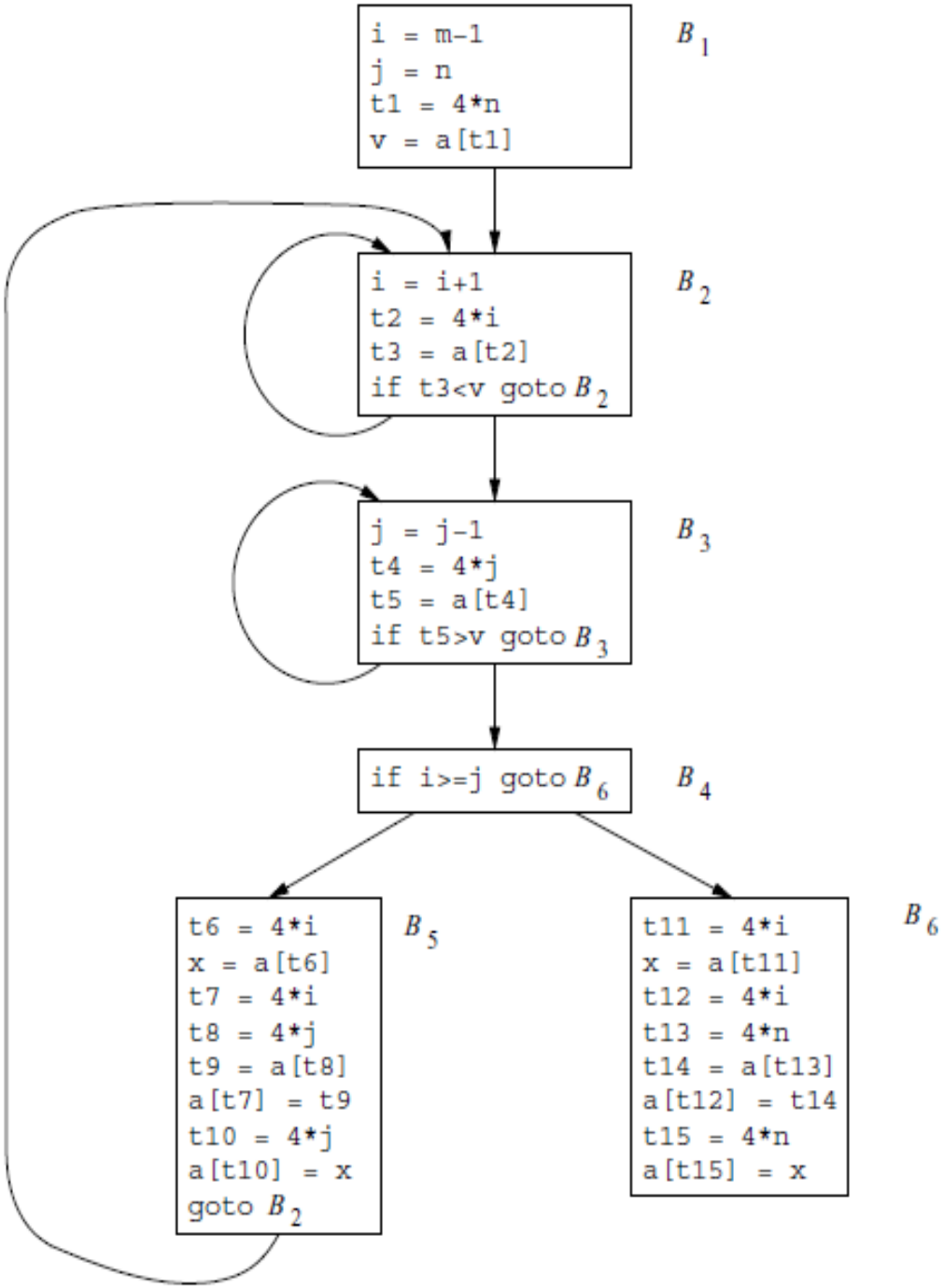
- First, instruction **1** is a leader by rule (1) of Algorithm 1.
- To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 8,12,13 and 22. By rule (2), the targets of these jumps are leaders; they are instructions **5**, **9**, and **23**, respectively.
- Then, by rule (3), each instruction following a jump is a leader; those are instructions **9, 13, 14** and **23**.

- We conclude that the leaders are instructions **1, 5, 9, 13, 14 and 23**

Blocks

- Block 1 is 1 to 4
- Block 2 is 5 to 8
- Block 3 is 9 to 12
- Block 4 is 13
- Block 5 is 14 to 22
- Block 6 is 23 to 30

Flow Graph


i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
B5
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
B6