Code Generation

Part I

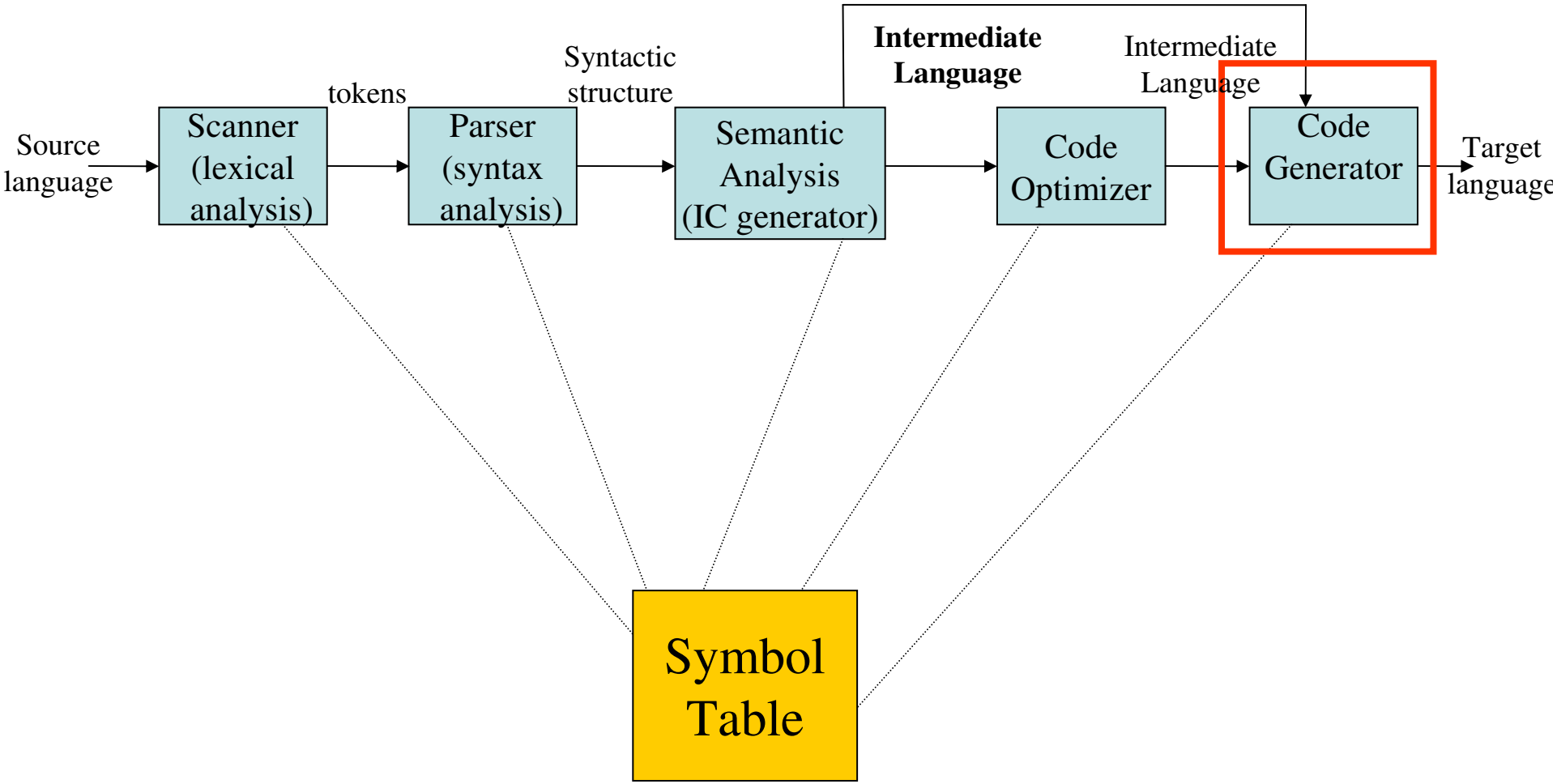
Code Generation

The code generation problem is the task of mapping intermediate code to machine code.

Requirements:

- Correctness
  - Must preserve semantic meaning of source program
- Efficiency
  - Make effective use of available resources
  - Code Generator itself must run efficiently

Compiler Architecture



Code Generation

- Want ***optimal*** code sequences?

  NP-Complete

  Generate all correct code sequences

  ... and see which is best

- We should be content with heuristic techniques that generate good code

- **Optimal?**

  The target program...

  ... executes **faster**

  ... takes less memory

Tasks of a Code Generator:

- Input language:
  - intermediate code (optimized or not)
    - Syntax and semantic errors have been removed
    - Type checking done and type conversion operators inserted
  - Information in the symbol table
- Target architecture: must be **well** understood
  - Significantly influences the difficulty of code generation
  - RISC, CISC
- Interplay between
  - Instruction Selection
  - Register Allocation
  - Instruction Scheduling

Instruction Selection

- There may be a large number of ‘candidate’ machine instructions for a given IR instruction
  - Level of IR
    - High: Each IR translates into many machine instructions
    - Low: Reflects many low-level details of machine
  - Nature of the instruction set
    - Uniformity and completeness
  - Each has own cost and constraints
    - Accurate cost information is difficult to obtain
  - Cost may be influenced by surrounding context

Instruction selection: Machine Idioms

IR Code: `x := x + 5`

Target Code:

```
mov   x,r0
add   5,r0
mov   r0,x
```

IR Code: `x := x + 1`

Target Code:

```
mov   x,r0
add   1,r0
mov   r0,x
```

Target Code:

```
mov   x,r0
inc   r0
mov   r0,x
```

Target Code:

```
inc   x
```

Register Allocation

- How to best use the bounded number of registers.
- Use or registers
  - Register allocation
    - We select a set of variables that will reside in registers at each point in the program
  - Register assignment
    - We pick the specific register that a variable will reside in.
- Complications:
  - special purpose registers
  - operators requiring multiple registers.
- Optimal assignment is NP-complete

Register Allocation

Multiply Instruction

```
mul    y,r4
```

Must specify an even numbered register

r5 × y → [r4,r5]

Multiply Instruction

```
div    y,r4
```

Must specify an even numbered register

[r4,r5] ÷ y ⇒ [r4,r5]

SRDA: Shift Right Double Arithmetic

```
srda   32,r6
```

Register Allocation

IR Code:

```
t := a + b
t := t * c
t := t / d
```

Target Code:

```
mov   a,r1
add   b,r1
mul   c,r0
div   d,r0
mov   r1,t
```

IR Code:

```
t := a + b
t := t + c
t := t / d
```

Target Code:

```
mov   a,r0
add   b,r0
add   c,r0
srda  32,r0
div   d,r0
mov   r1,t
```

Conclusion:

Where you put the result of t:=a+b (either r0 or r1) depends on how it will be used later!!!

[A "chicken-and-egg" problem]

Instruction Scheduling

- Choosing the order of instructions to best utilize resources
- Picking the optimal order is NP-complete problem

- Simplest Approach
  - Don’t mess with re-ordering.
  - Target code will perform all operations in the same order as the IR code
- Trickier Approach
  - Consider re-ordering operations
  - May produce better code

    ... Get operands into registers just before they are needed

    ... May use registers more efficiently

Moving Results Back to Memory

- When to move results from registers back into memory?
  - After an operation, the result will be in a register.
- **Immediately**
  - Move data back to memory just after it is computed.
  - May make more registers available for use elsewhere.
- ***Wait as long as possible* before moving it back**
  - Only move data back to memory “at the end”
    - or “when absolutely necessary”
  - May be able to avoid re-loading it later!

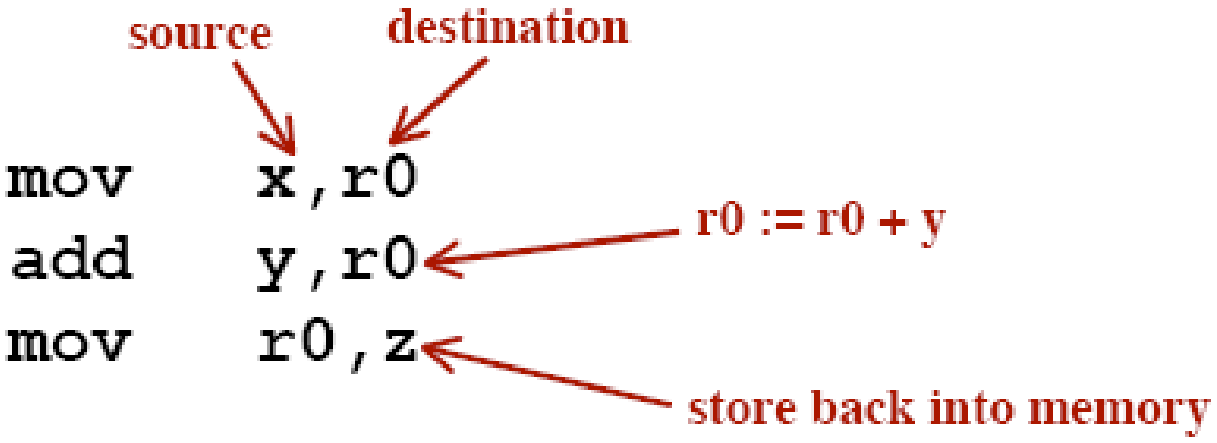
Code Generation Algorithm #1

Simple code generation algorithm:

Define a target code sequence to each intermediate code statement type.

Example Target Machine

2-Address Architecture



Code Generation Algorithm #1

Statement-by-statement generation
Code for each IR instruction is
generated independently of all other IR instructions.

IR Code:

```
a := b + c
d := a + e
```

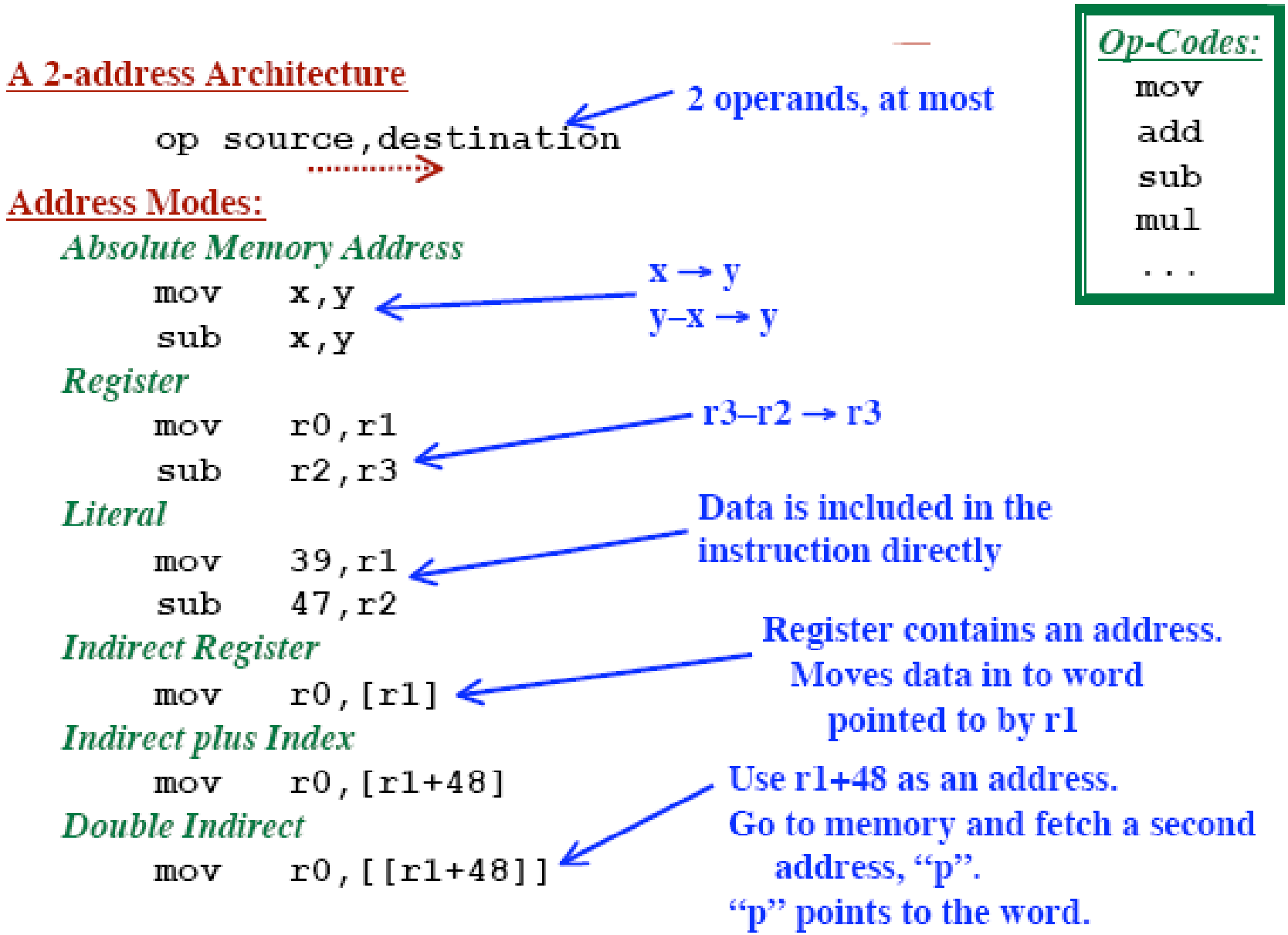
Target Code:

```
...
mov   b,r0
add   c,r0      a := b + c
mov   r0,a
mov   a,r0
add   e,r0      d := a + e
mov   r0,d
...
```

ALSO: Registers are not used effectively.

This instruction is totally unnecessary!!!

Example Target Machine



Evaluating A Potential Code Sequence

- Each instruction has a "*cost*"

  Cost = Execution Time

- Execution Time is difficult to predict.

  Pipelining, Branches, Delay Slots, etc.

- **Goal:** Approximate the real cost

  A "***Cost Model***"

Simplest Cost Model:

Code Length ≈ Execution Time

Just count the instructions!

A Better Cost Model

Look at each instruction.
Compute a cost (in "units").
Count the number of memory accesses.

Cost = 1 + Cost-of-operand-1 + Cost-of-operand-2 + Cost-of-result

	example	cost
Absolute Memory Address	`x`	1
Register	`r0`	0
Literal	`39`	0
Indirect Register	`[r1]`	1
Indirect plus Index	`[r1+48]`	1
Double Indirect	`[[r1+48]]`	2

Example: `sub 97,r5` r5 – 97 → r5
Cost = 1 + 0 + 0 + 0 = 1

Example: `sub 97,[r5]` [r5] – 97 → [r5]
Cost = 1 + 1 + 0 + 1 = 3

Example: `sub [r1],[[r5+48]]` [[r5+48]] – [r1] → [[r5+48]]
Cost = 1 + 2 + 1 + 2 = 6

Cost Generation Example

IR Code: `x := y + z`

Translation #1:

```
mov   y,x   3
add   z,x   4
```

} Cost = 7

Translation #2:

```
mov   y,r1  2
add   z,r1  2
mov   r1,x  2
```

} Cost = 6

Lesson #1:
Use Registers

Translation #3:

Assume "y" is in r1 and "z" is in r2
Assume "y" will not be needed again

```
add   r2,r1  1
mov   r1,x   2
```

} Cost = 3

Lesson #2:
Keep variables in registers

Lesson #3:
Avoid or delay storing into memory.

Translation #4:

Assume "y" is in r1 and "z" is in r2
Assume "y" will not be needed again.
Assume we can keep "x" in a register.

```
add   r2,r1  1
```

} Cost = 1

Lesson #4: *(not illustrated)*
Use different addressing modes effectively.

Basic Blocks

Break IR code into blocks such that...

The block contains NO transfer-of-control instructions

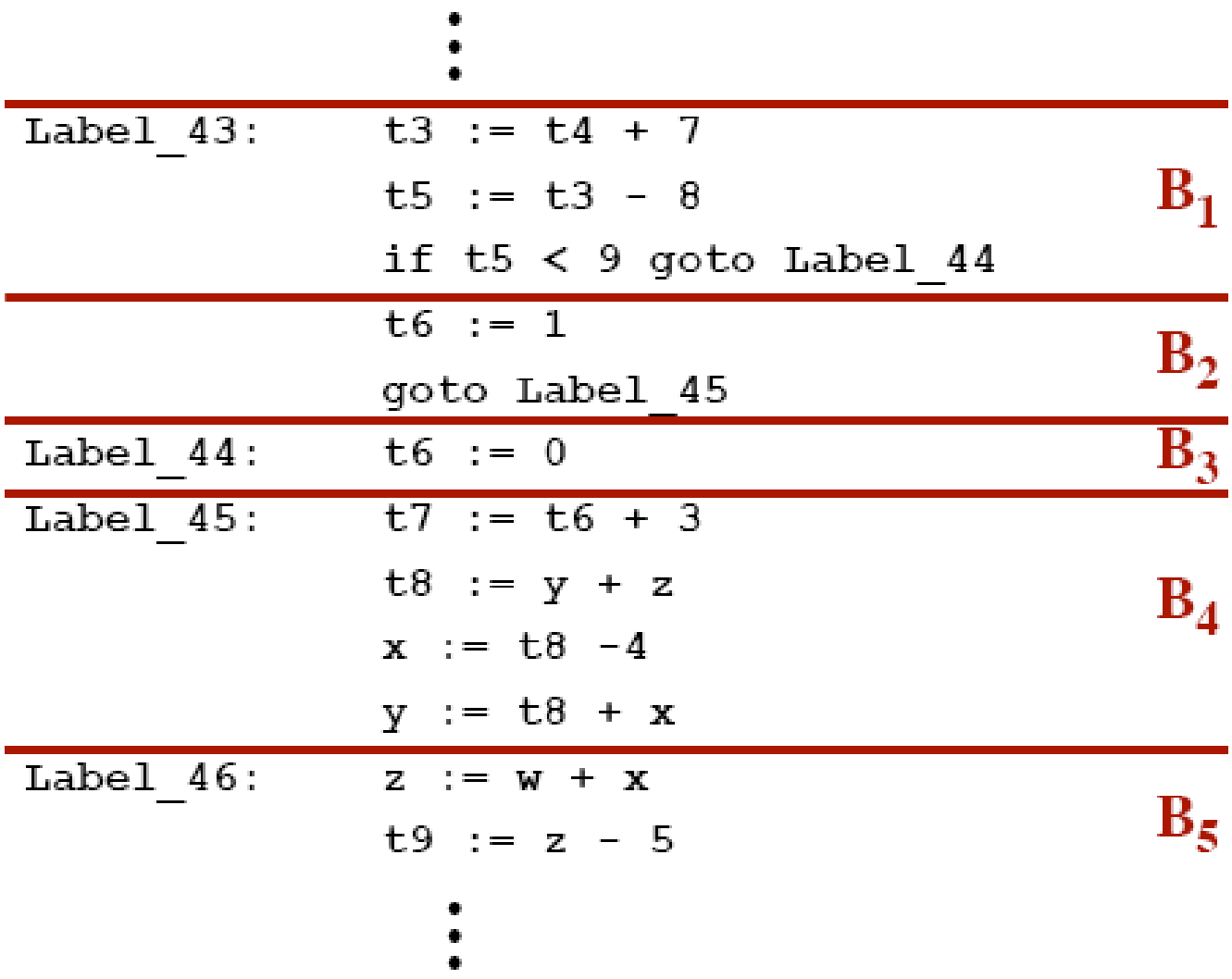
... except as the last instruction

- **A sequence of consecutive statements.**
- **Control enters only at the beginning.**
- **Control leaves only at the end.**

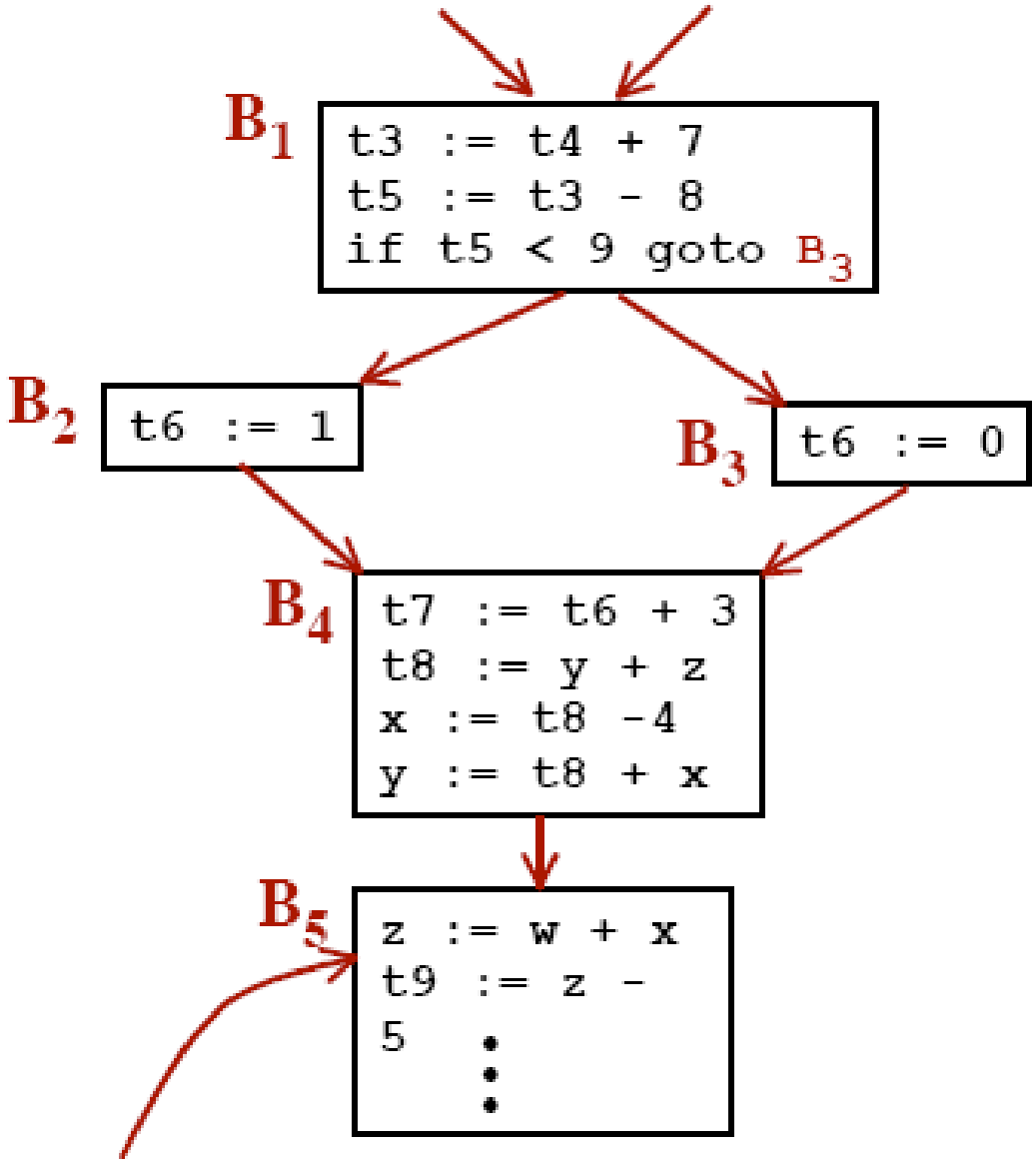
Basic Blocks

```
                    ⋮
Label_43:     t3 := t4 + 7
              t5 := t3 - 8
              if t5 < 9 goto Label_44
              t6 := 1
              goto Label_45
Label_44:     t6 := 0
Label_45:     t7 := t6 + 3
              t8 := y + z
              x := t8 -4
              y := t8 + x
Label_46:     z := w + x
              t9 := z - 5
                    ⋮
```

Basic Blocks



Control Flow Graph



Algorithm to Partition Instructions into Basic Blocks

Concept: "Leader"

The first instruction in a basic block

Idea:

Identify "leaders"

- **The first instruction of each routine is a leader.**
- **Any statement that is the target of a branch / goto is a leader.**
- **Any statement that immediately follows**

  a branch / goto

  a call instruction

  ... is a leader

A Basic Block consists of

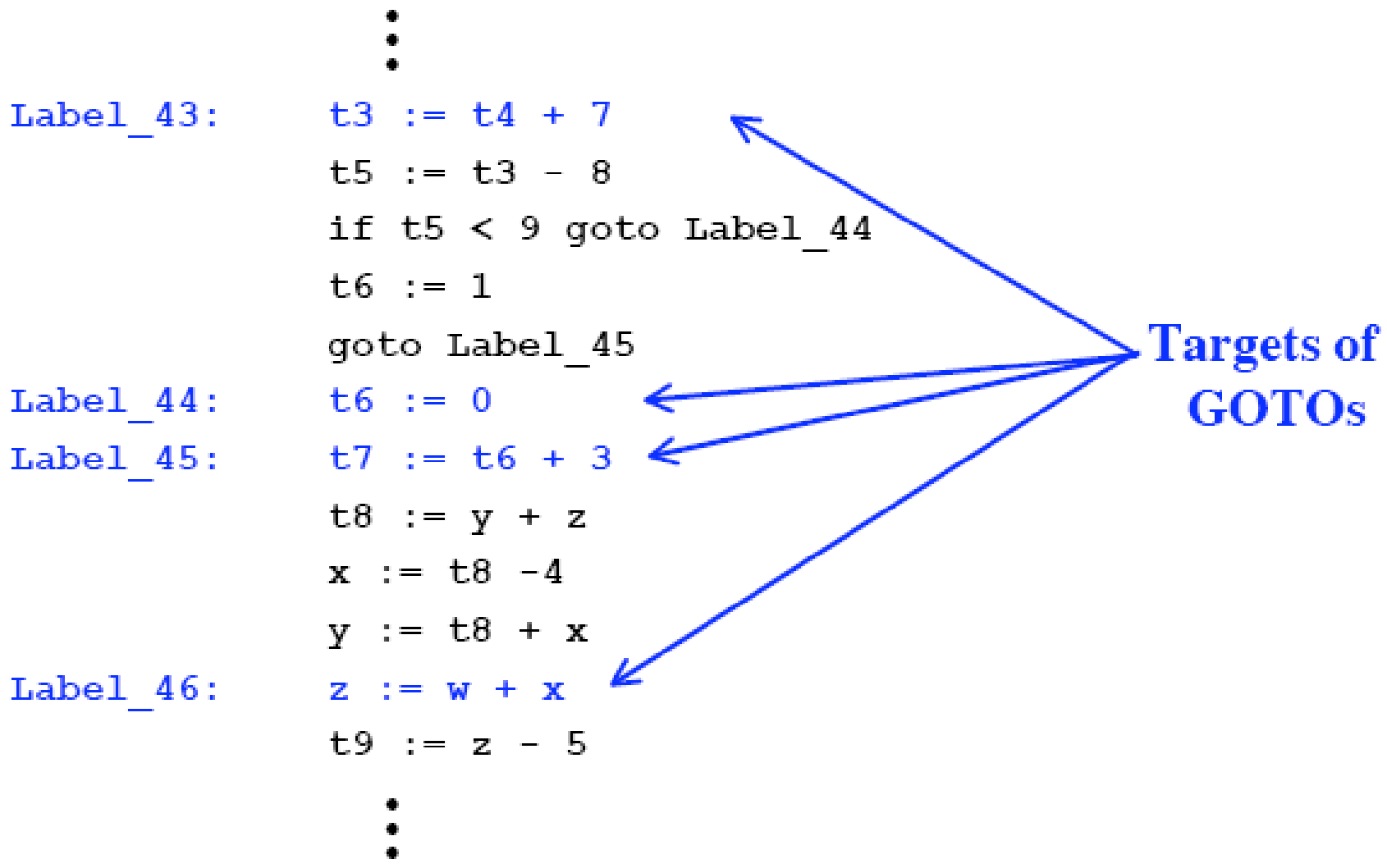
A leader and all statements that follow it

... up to, but not including, the next leader

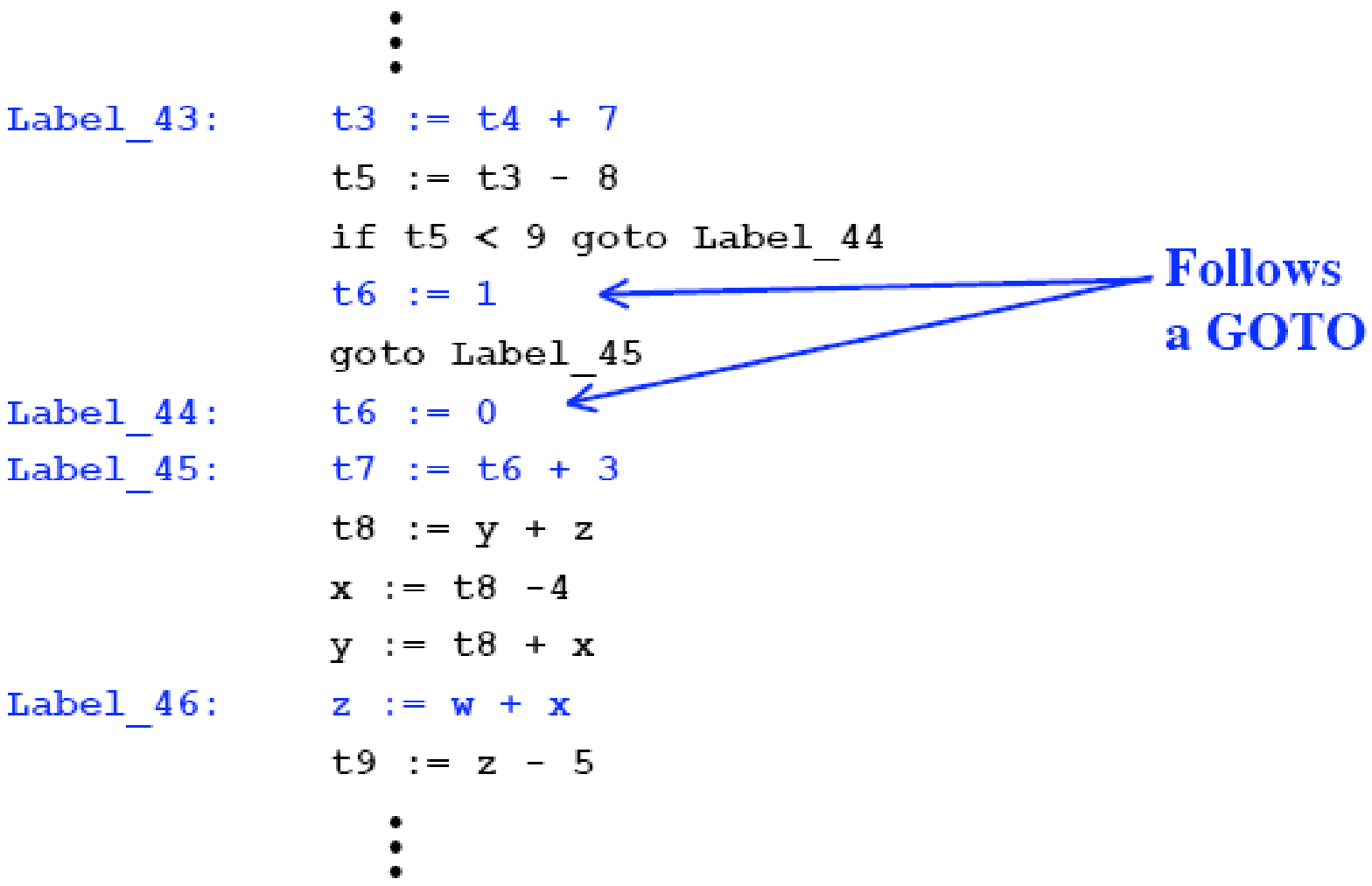
Identify Leaders

```
                    .
                    .
                    .
Label_43:     t3 := t4 + 7
              t5 := t3 - 8
              if t5 < 9 goto Label_44
              t6 := 1
              goto Label_45
Label_44:     t6 := 0
Label_45:     t7 := t6 + 3
              t8 := y + z
              x := t8 -4
              y := t8 + x
Label_46:     z := w + x
              t9 := z - 5
                    .
                    .
                    .
```

Identify Leaders



Identify Leaders



Identify Leaders

```
                    :
                    :
                    :
Label_43:     t3 := t4 + 7
              t5 := t3 - 8
              if t5 < 9 goto Label_44
              t6 := 1
              goto Label_45
Label_44:     t6 := 0
Label_45:     t7 := t6 + 3
              t8 := y + z
              x := t8 -4
              y := t8 + x
Label_46:     z := w + x
              t9 := z - 5
                    :
                    :
                    :
```

Look at Each Basic Block in Isolation

Use (B)

The set of variables used (i.e., read) by the Basic Block

(... before being written / updated)

The "inputs" to the BB

Def (B)

The set of variables in the Basic Block that are written / assigned to.

The "outputs" of the BB

B_7

```
x := y + v
z := x * y
v := z + 5
if w < v goto B9
```

Use (B_7) = y, v, w

Def (B_7) = x, z, v

View the basic block as a function

< x, z, v > := f (y, v, w)

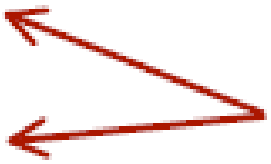
Okay to transform the block!

(as long as it computes the same function)

Common Sub-Expression Elimination

Transform:

```
x := b + c
y := a - d
d := b + c
```



We compute "b+c" twice!

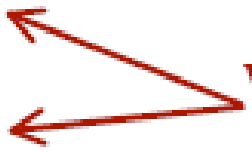
Into:

```
x := b + c
y := a - d
d := x
```

Common Sub-Expression Elimination

Transform:

```
x := b + c
y := a - d
d := b + c
z := a - d
```



What about "a-d"...
Do we need recompute?

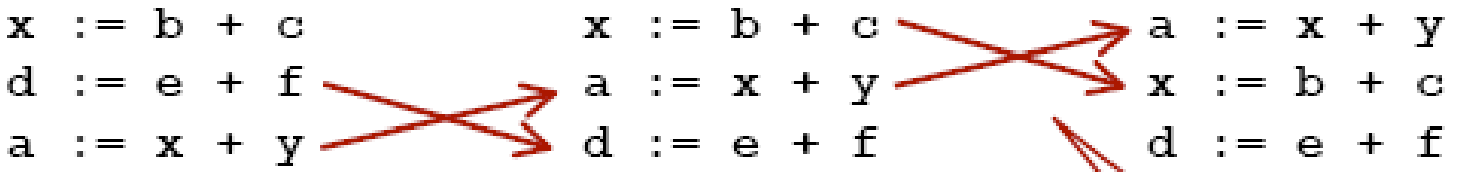
Into:

```
x := b + c
y := a - d
d := x
z := a - d
```

Yes!
"d" has been changed since "a-d" computed!
Now, "a-d" may compute a different value!

Reordering Instructions in a Basic Block

Sometimes we can change the order of instructions...



But some changes would change the program!

When can we exchange these two instructions?

```
x := ....v1...v2...
y := ....v3...v4...
```

If and only if...

- $v_1 \neq y$
- $v_2 \neq y$
- $v_3 \neq x$
- $v_4 \neq x$

Any variables (including possibly "x" and "y")

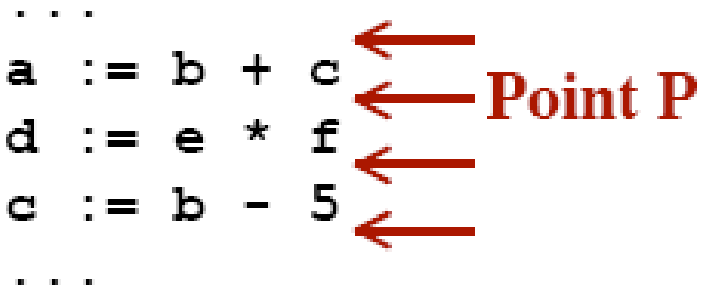
Live Variables

"Is some variable x live at some point P in the program?"

Could the value of "x" at point P ever be needed later in the execution?

"Point in a program"

A point in a program occurs between two statements.



Is it possible that the program will ever read from x along a path from P?

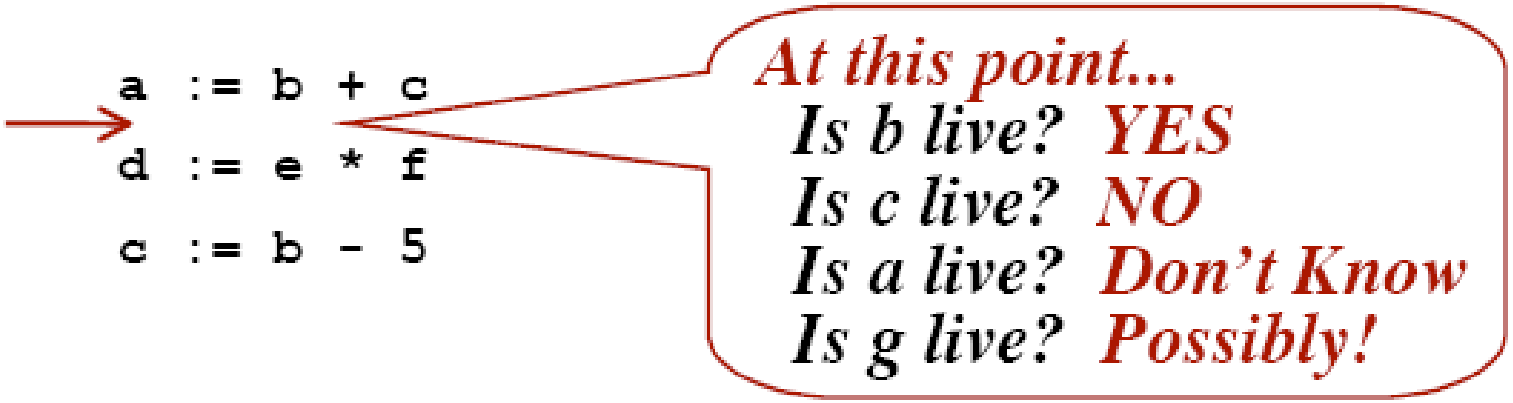
[... before "x" is written / stored into]

Dead Variables

A Variable is "*Dead at point P*"
= Not Live

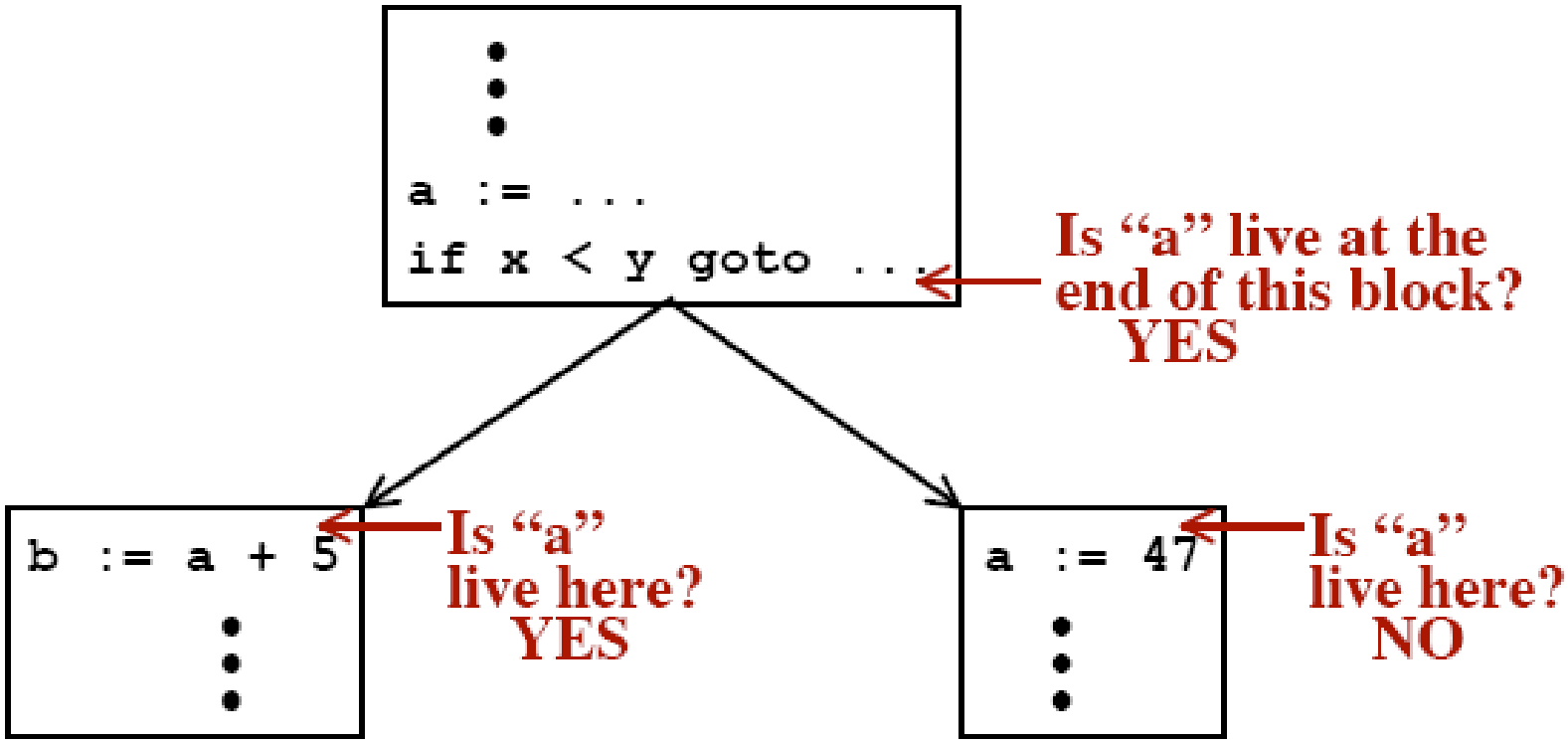
Value will definitely never be used.
No need to compute it!
If value is in register, no need to store it!

Liveness Example



Liveness Example

Must look at the whole "control flow graph" to determine liveness.



Live Variable Analysis

Input:

The Control Flow Graph

$\text{Use}(B_i)$, $\text{Def}(B_i)$ } for all B_i

Output:

$\text{Live}(B_i)$ = a list of all variables live at the end of B_i

Live Variable Analysis missing?

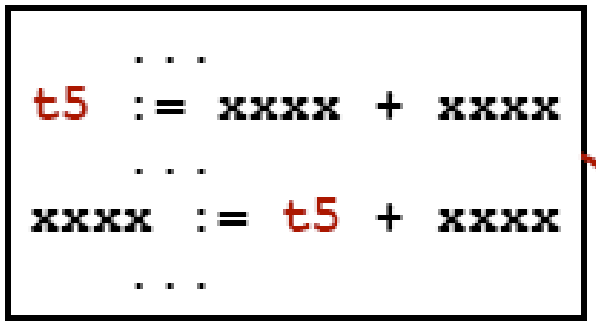
Assume all variables are live at the end of each basic block.

Temporaries

Assumption:

Each temporary is used in only one basic block

(True of temps for expression evaluation)



More precisely:
No temp will ever be in $Use(B_i)$ for any BB

Conclusion:

Temps are never live at the end of a basic block.

If Live-Variable-Analysis is missing...

this assumption can at least identify many dead variables.

Dead Code

"Dead Code" (first meaning)

Any code that cannot be reached.
(Will never be executed.)

```
        x := y + z
        goto Label_45
        a := b + c
        d := e * f
    Label_45:
        z := x - a
```

Dead Code (unreachable)

"Dead Code" (second meaning)

A statement which computes a dead variable.

Example:

```
        b := x * y
        a := b + c
        ...
```

← If "a" is not live here...

Then eliminate this statement!!!

Temporaries

If you can identify a variable which is
not in Use(B_i) for any basic block
(e.g., a temporary used only in this basic block)

Then you may...

- **Rename the variable**
- **Keep the variable in a register instead of in memory**
- **Eliminate it entirely (during some optimization)**

Must be careful that the variable
is not used in other routines
(i.e., accessed as a non-local from another routine)

Algebraic Transformation

Watch for special cases.
Replace with equivalent instructions
... that execute with a lower cost.

Examples

```
x := y + 0    ⇒    x := y
x := y * 1    ⇒    x := y
x := y ** 2   ⇒    x := y * y
x := y + 1    ⇒    x := incr(y)
x := y - 1    ⇒    x := decr(y)
...etc...
```

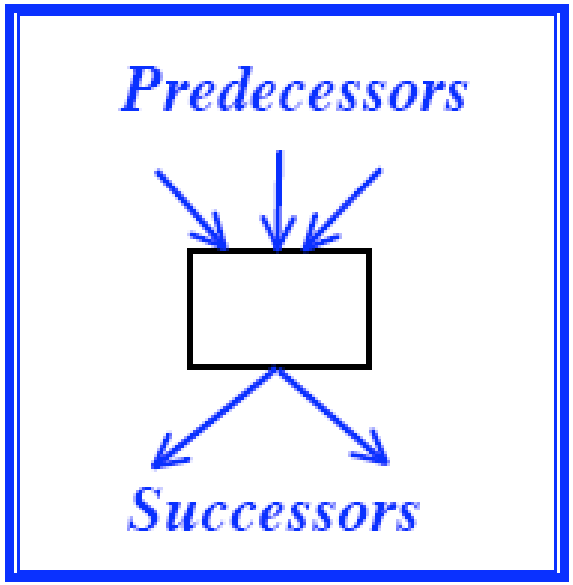
May do some transformations during "Peephole Optimization."

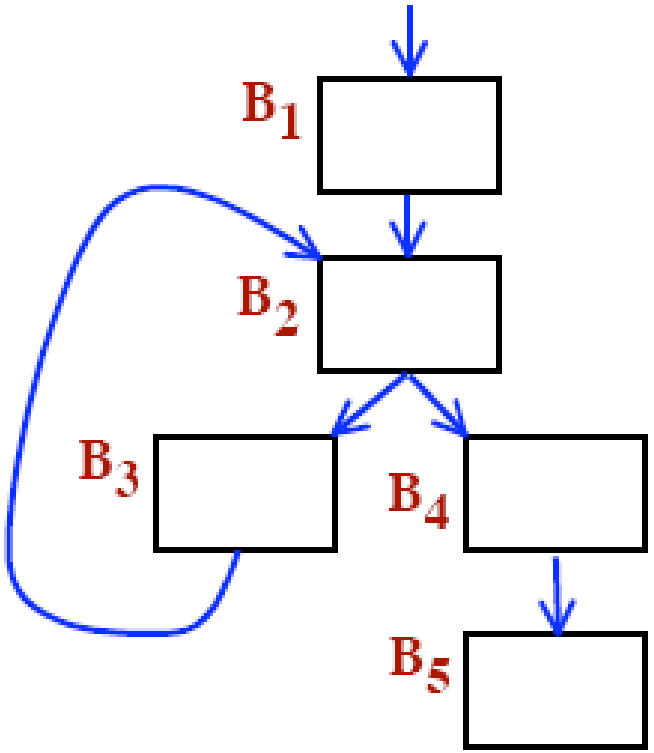
Other transformations may be Target Architecture Dependent
(use your "cost model" to determine when to transform)

Control Flow Graphs

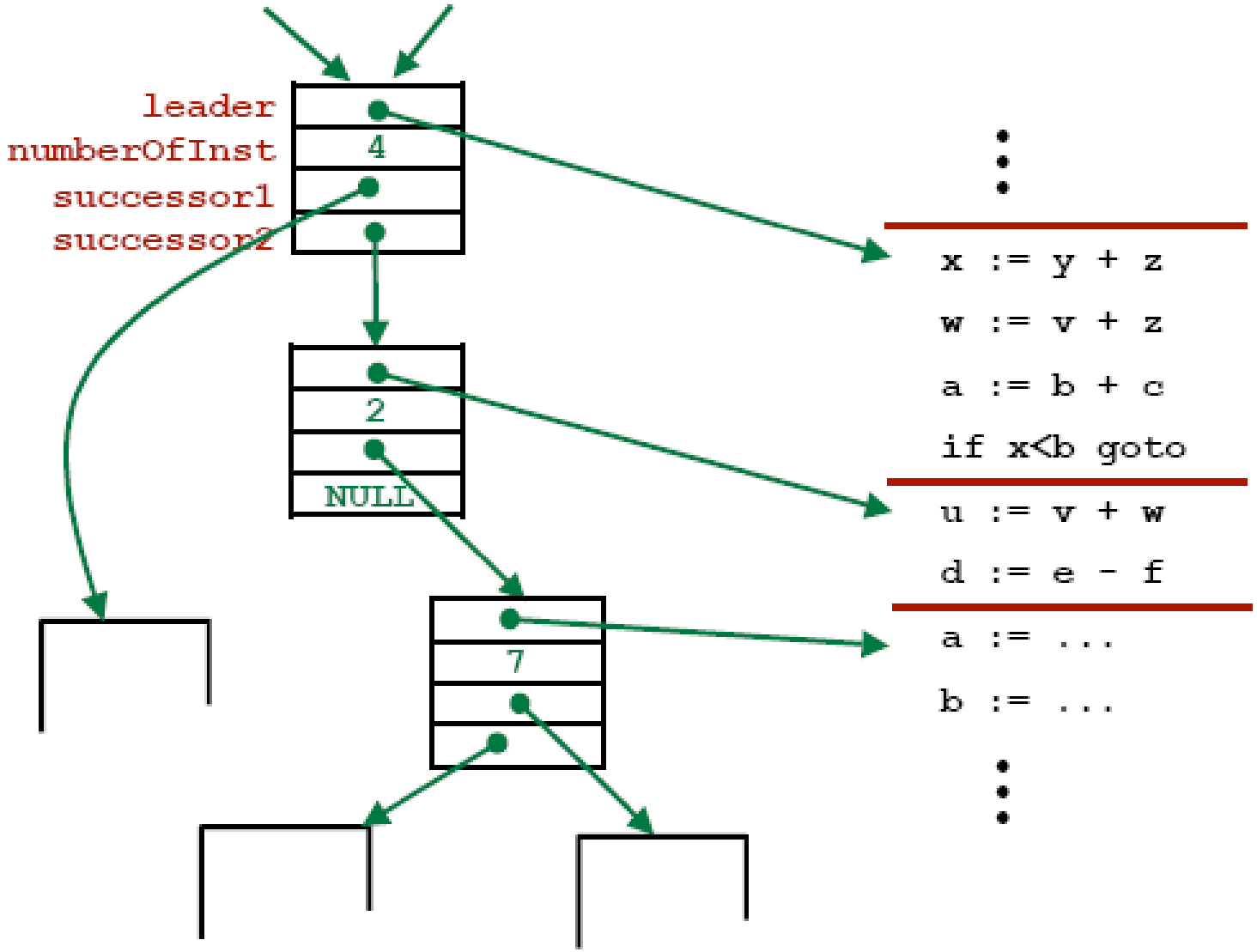
Definitions:

- **Initial Block**
- **Predecessor Blocks**
- **Successor Blocks**

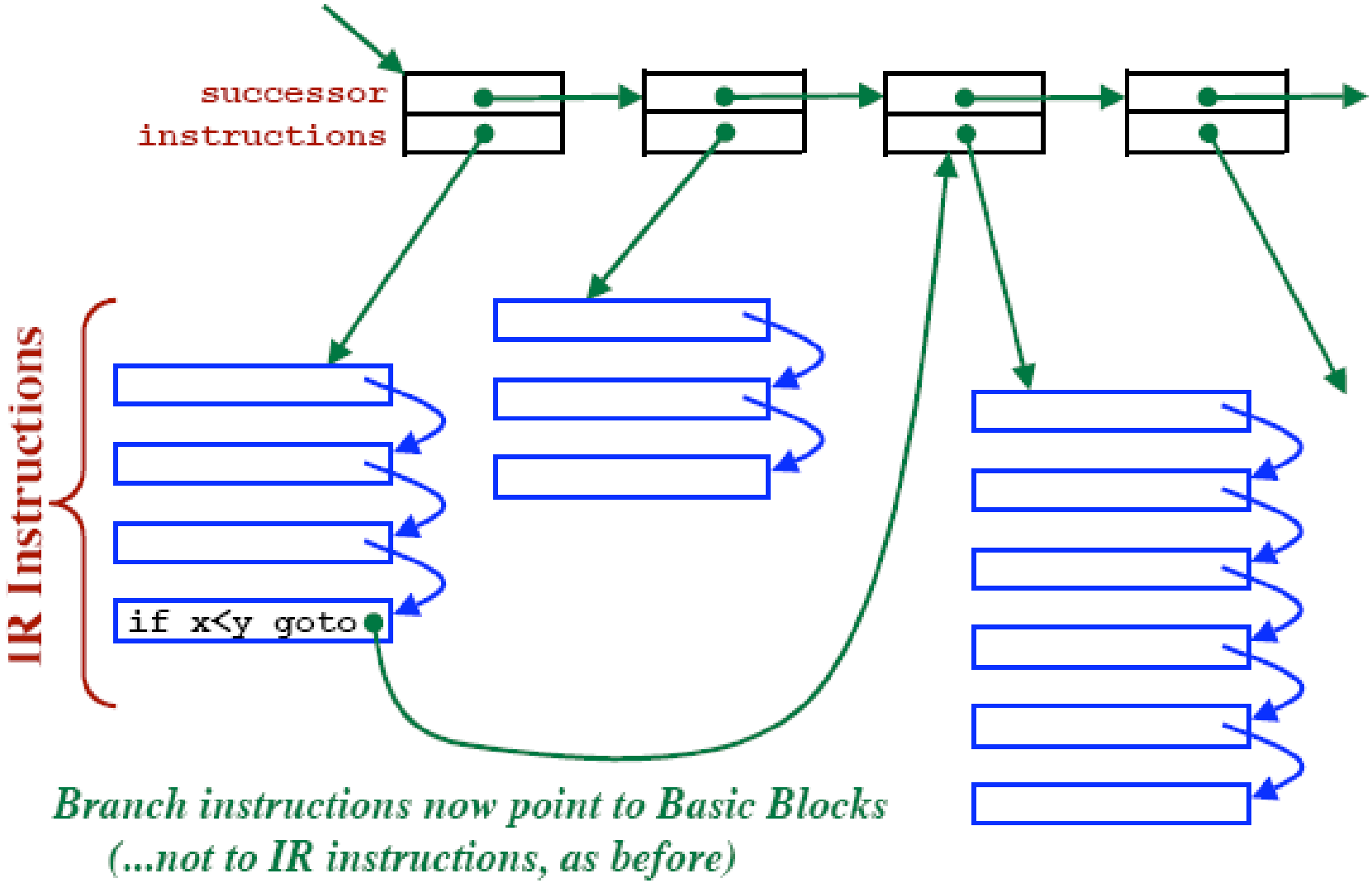




Representing Flow Graphs: Idea 1

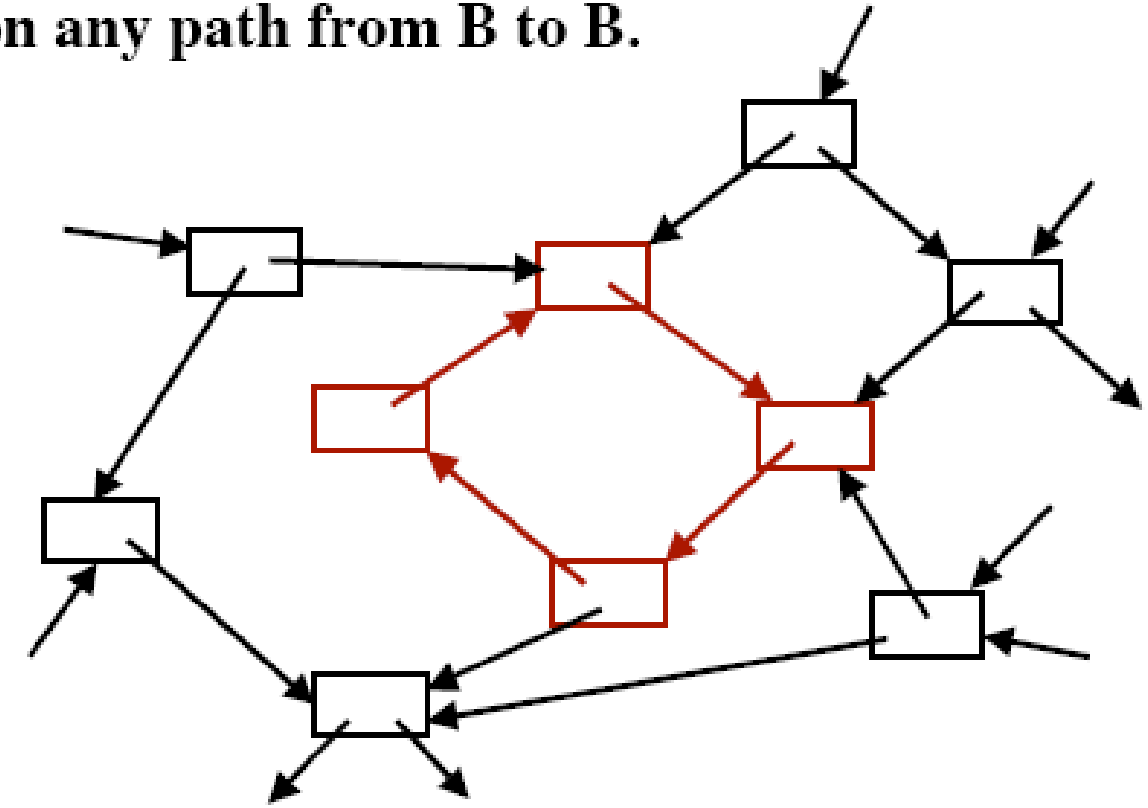


Representing Flow Graphs: Idea 2



What is Loop?

A *cycle* in the flow graph.
Can go from B back to B.
A *path* from B to B.
All blocks on any path from B to B.



Natural Loop

Each loop has a unique entry (its "Header Block")

To reach any block in the loop (from outside the loop) you must first go through the header block

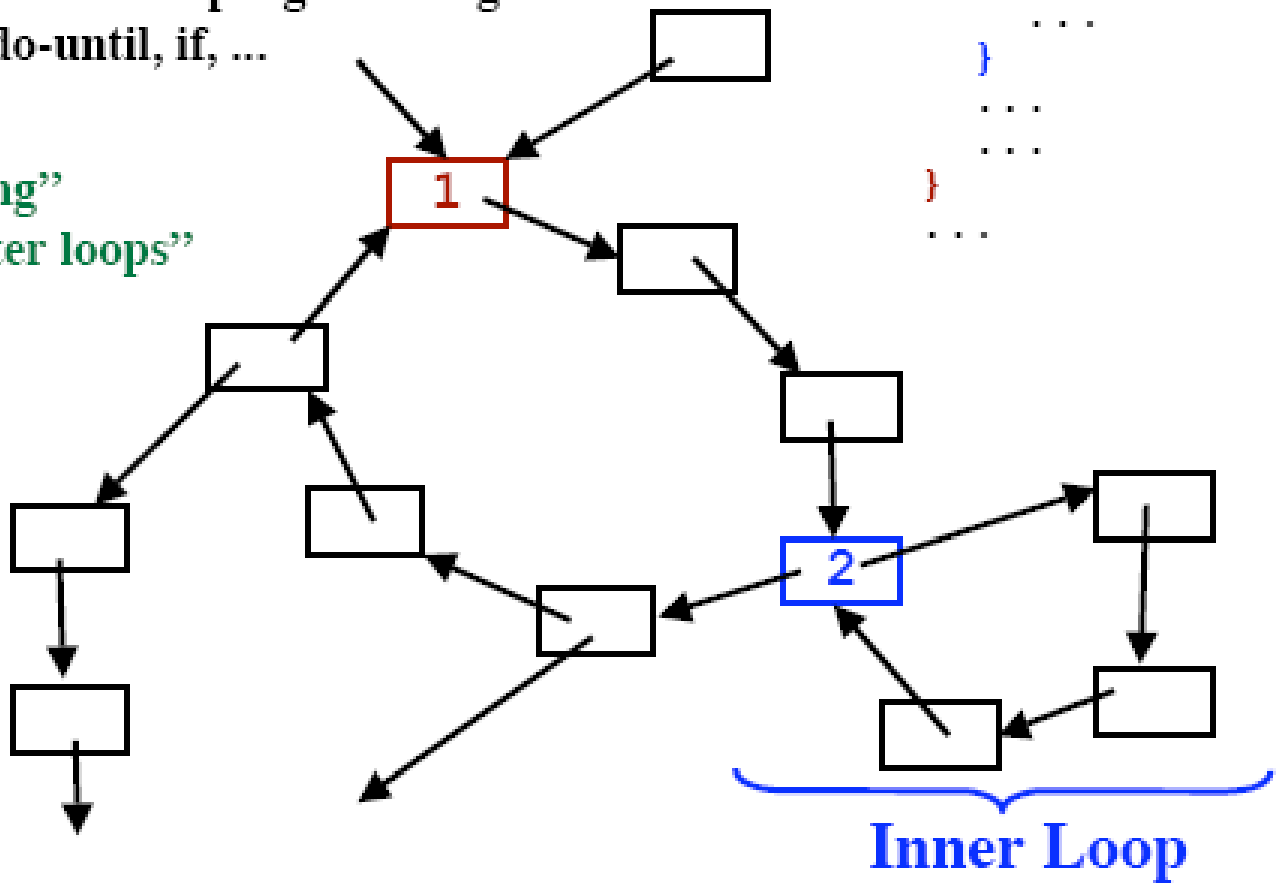
Result from "structured programming" constructs
while, for, do-until, if, ...

Concepts:

"loop nesting"

"inner / outer loops"

```
...
...
while(...) {
   ...
   ...
   while(...) {
      ...
      ...
   }
   ...
   ...
}
...
```



Loops with Multiple Entries

- **Very un-natural**
- **Rare in assembly language programs**
- **Impossible in many programming languages**

```
goto Lab45;
...
while (x<y) {
   ...
   Lab45:
   ...
}
```