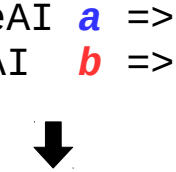
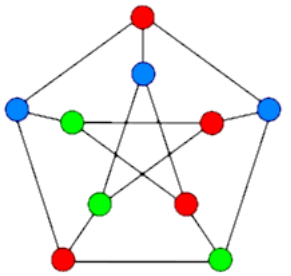
# CS 432
# Fall 2017

Mike Lam, Professor

```
storeAI a => b
loadAI  b => c
```



```
storeAI a => b
i2i     a => c
```



# Register Allocation

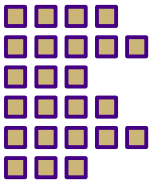
# Compilers

Source code

```
int main() {
    int x
      = 4 + 5;
    return x;
}
```

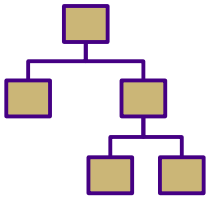
**Lexing (P2)**

Tokens



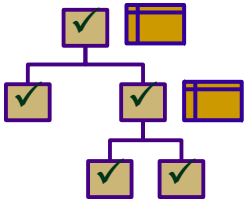
**Parsing (P3)**

Syntax tree

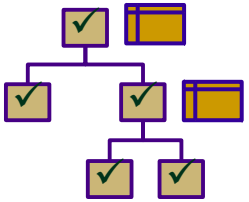


**Analysis (P4)**

Checked AST + Symtables



Checked AST + Symtables



**IR Code Gen (P5)**

Linear IR

```
main:
  loadI 4 => r1
  loadI 5 => r2
  add r1, r2 => r3
  i2i r3 => RET
```

**Optimization Passes**

Optimized Linear IR

```
main:
  loadI 4 => r1
  addI r1, 5 => RET
```

**Machine Code Gen**

Machine code

```
7f 45 4c 46 01
01 01 00 00 00
00 00 00 00 00
...
```

# Optimization (Ch. 8-10)

- **Local**
  - Local value numbering (8.4.1)
  - Tree-height balancing (8.4.2)
- **Regional**
  - Superlocal value numbering (8.5.1)
  - Loop unrolling (8.5.2)
- **Global**
  - Constant propagation (9.3.6, 10.7.1)
  - Dead code elimination (10.2)
  - Global code placement (8.6.2)
  - Lazy code motion (10.3)
- **Whole-program**
  - Inline substitution (8.7.1)
  - Procedure placement (8.7.2)

*Asides:*

Data-flow analysis (Ch. 9)
Liveness analysis (8.5.1, 9.2.2)
Single static assignment (9.3)

# Machine Code Gen (Ch. 11-13)

- Translate from linear IR to machine code
  - Often, we can just emit assembly
  - Use built-in system assembler and linker to create final executable
- Issues:
  - Translation from IR instructions to machine code instructions: **instruction selection** ***(Ch. 11)***
  - Arrangement of machine code instructions for optimal pipelining: **instruction scheduling** ***(Ch. 12)***
  - Assignment of registers to minimize memory and HDD accesses: **register allocation** ***(Ch. 13)***

# Instruction Selection

- Choose machine code instructions to replace IR
  - Complexity is highly dependent on target architecture
  - CISC provides more options than RISC
- Algorithms
  - Treewalk routine (similar to P5)
  - Tree-pattern matching / tiling
  - Peephole optimization

# Peephole Optimization

- Scan linear IR with sliding window ("peephole")
  - Look for common inefficient patterns
  - Replace with known equivalent sequences

**Example:**

```
storeAI r5 => [bp+8]
loadAI [bp+8] => r7
```

```
storeAI r5 => [bp+8]
i2i r5 => r7
```

**Generalized pattern:**

```
storeAI a => b
loadAI  b => c
```

```
storeAI a => b
i2i     a => c
```

# Instruction Scheduling

- Modern architectures expose many opportunities for optimization
  - Some instructions require fewer cycles
  - Instruction pipelining
  - Speculative execution / branch prediction
  - Multicore shared-memory processors
- Scheduling: re-order instructions to improve speed
  - Must not modify program semantics
  - Maximize utilization of CPU and memory resources
  - Main algorithm: **list scheduling** (next week!)

# Register Allocation

- Maximizing register use is very important
  - Registers are the lowest-latency memory locations
  - Issue: limited number of registers
  - Reduce the # of registers used to match the target system
  - Program using *n* registers => Program using *m* registers (*n* > *m*)
- Allocation vs. assignment
  - Allocation: map a virtual register space to a physical register space
    - This is hard (NP-complete for any realistic situation)
  - Assignment: map a valid allocation to actual register names
    - This is easy (linear or polynomial)

# Local Allocation

- Top-down local register allocation
  - Compute a priority for each virtual register
    - Frequency of access to that register
  - Sort by priority, highest to lowest
  - Assign registers in order, highest priority first
  - Rewrite the code
- General idea: prioritize most-often-accessed virtual registers
  - Allocate to physical registers in priority order
  - Very simple to implement
  - Static per-block allocations are not always optimal
    - Access patterns may change throughout block

# Local Allocation

- Bottom-up local register allocation
  - Scan each block instruction-by-instruction
  - For each instruction:
    - Examine virtual registers
    - Ensure operands are in physical registers (load them if they’re not)
    - Allocate physical register for result
  - May need to "spill" virtual registers
    - Save their values to the stack temporarily
    - This frees up a physical register

**THIS IS YOUR LAST DECAF PROJECT**

# Bottom-up local register allocation

for each physical register *reg*:
    name[*reg*] = -1  *# contents of reg (virtual id)*
    next[*reg*] = INF  *# index of next reference*

for each virtual register *vr*:
    loc[*vr*] = -1  *# location of vr (physical id or BP offset)*
    spilled[*vr*] = false  *# is vr currently spilled?*

for each instruction *i*:
    for each *vr* read in *i*:
        *reg* = **ensure**(vr)
        replace *vr* with *reg* in *i*
        if *vr* is not needed after *i* then **free**(*vr*)
    for each vr written in i:
        *reg* = **allocate**(*vr*)
        replace *vr* with *reg* in *i*
        if *vr* is not needed after *i* then **free**(*vr*)

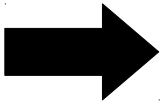
**ensure**(vr):
    if *vr* is in *reg*:
        return *reg*
    else:
        *reg* = **allocate**(*vr*)
        emit code "`reg ← vr`"
        return *reg*

**allocate**(vr):
    if *reg* is available:
        return *reg*
    else:
        find *reg* to spill
        **spill**(*reg*)
        return *reg*

# Bottom-up local register allocation

```
add:
  loadAI [bp+8] => r1
  loadAI [bp+12] => r2
  add r1, r2 => r3
  i2i r3 => ret
  return

main:
  loadI 3 => r4
  storeAI r4 => [bp-4]
  loadAI [bp-4] => r5
  loadI 2 => r6
  param r6
  param r5
  call add
  i2i ret => r7
  i2i r7 => ret
  return
```

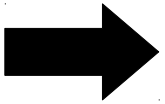


```
add:
  loadAI [bp+8] => r0
  loadAI [bp+12] => r1
  add r0, r1 => r0
  i2i r0 => ret
  return

main:
  loadI 3 => r0
  storeAI r0 => [bp-4]
  loadAI [bp-4] => r0
  loadI 2 => r1
  param r1
  param r0
  call add
  i2i ret => r0
  i2i r0 => ret
  return
```

# Bottom-up local register allocation

```
gcd:
l1:
  loadAI [bp+12] => r1
  loadI 1 => r2
  cmp_GE r1, r2 => r3
  cbr r3 => l2, l3
l2:
  loadAI [bp+12] => r4
  loadI 0 => r5
  store r4 => [r5]
  loadAI [bp+8] => r6
  loadAI [bp+12] => r7
  div r6, r7 => r8
  mult r7, r8 => r9
  sub r6, r9 => r10
  storeAI r10 => [bp+12]
  loadI 0 => r11
  load [r11] => r12
  storeAI r12 => [bp+8]
  jump l1
l3:
  loadAI [bp+8] => r13
  i2i r13 => ret
  return
```



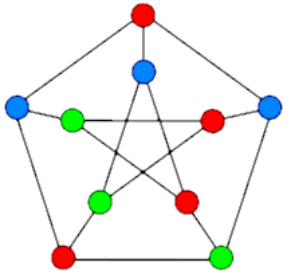
```
gcd:
l1:
  loadAI [bp+12] => r0
  loadI 1 => r1
  cmp_GE r0, r1 => r0
  cbr r0 => l2, l3
l2:
  loadAI [bp+12] => r0
  loadI 0 => r1
  store r0 => [r1]
  loadAI [bp+8] => r0
  loadAI [bp+12] => r1
  div r0, r1 => r2
  mult r1, r2 => r1
  sub r0, r1 => r0
  storeAI r0 => [bp+12]
  loadI 0 => r0
  load [r0] => r0
  storeAI r0 => [bp+8]
  jump l1
l3:
  loadAI [bp+8] => r0
  i2i r0 => ret
  return
```

# Local vs. global allocation

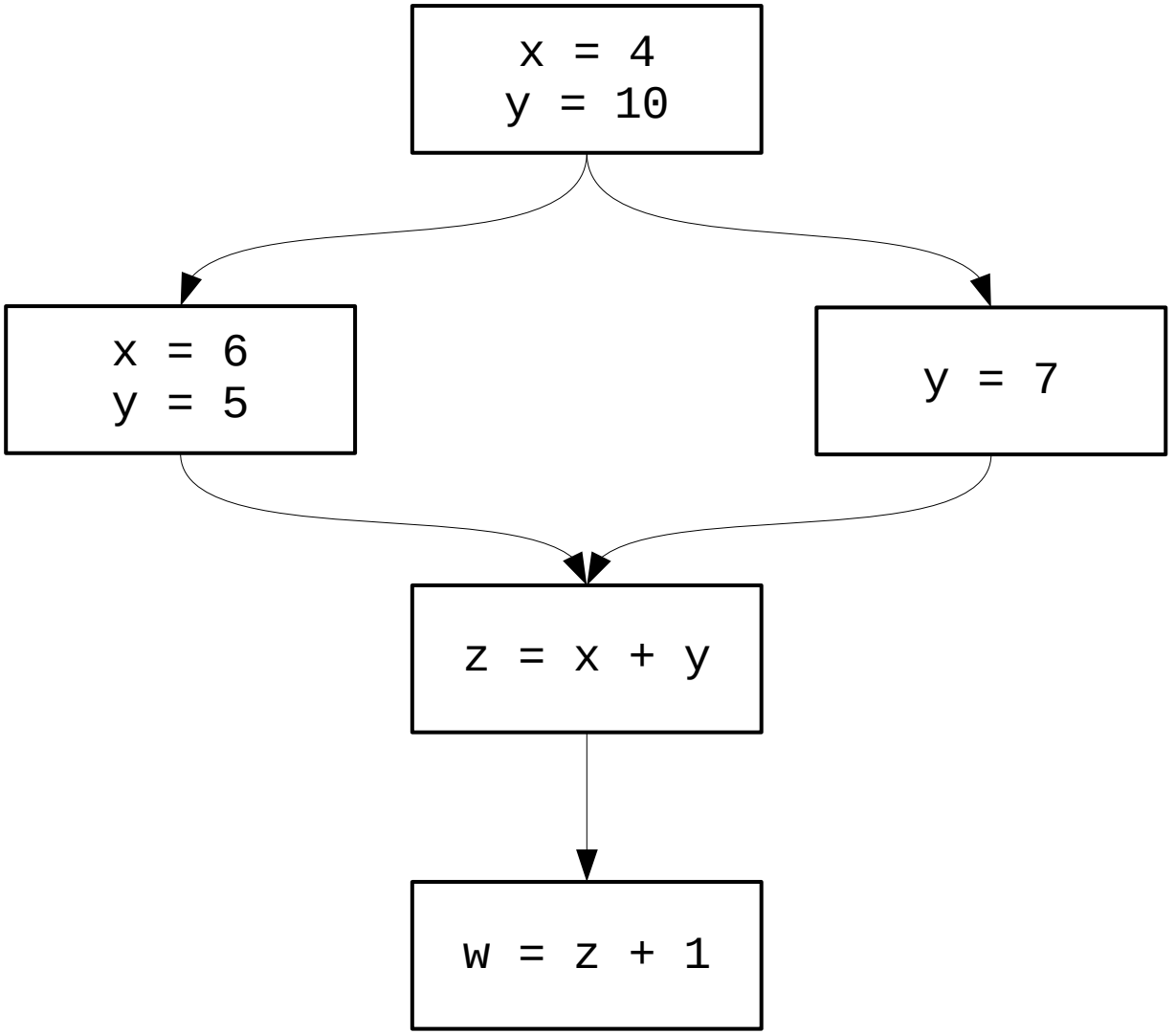
- Local allocation handles each basic block separately
  - Will miss inter-block dependencies
- Global allocation handles all basic blocks in a procedure
  - Does NOT consider inter-procedural dependencies
  - This is why calling conventions are important
    - I.e., caller-save vs. callee-save and return value
- Decaf project
  - Because we used SSA in P5 and always load/store to memory, no virtual registers will be live at the entrance or exit of any block (so no inter-block dependencies)
  - Thus, we can use local register allocation in P6

# Global Allocation

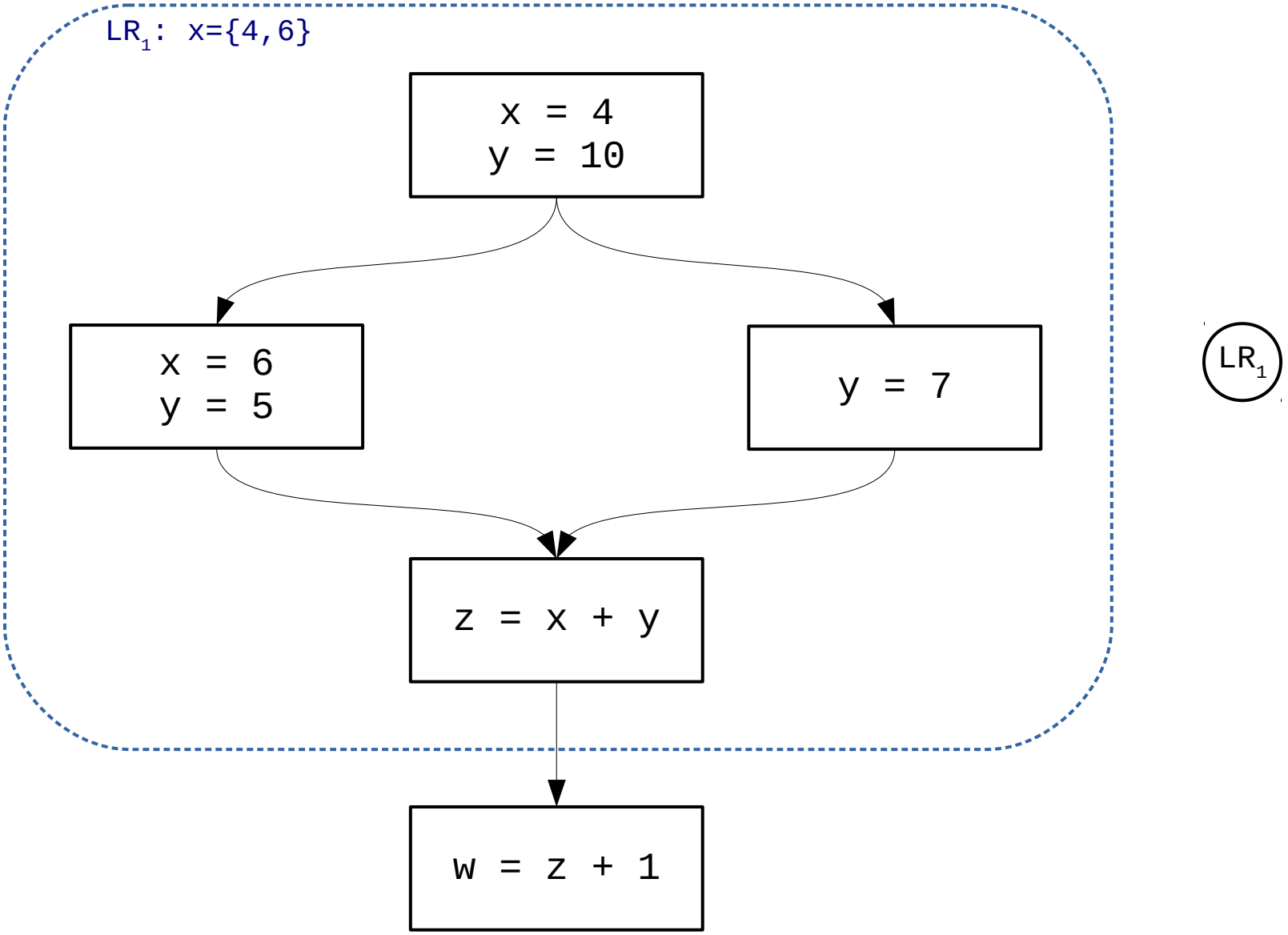
- Determine live range for each virtual register
  - Use results from liveness analysis
- Build interference graph
  - Node for each virtual register
  - Edges between registers with interfering live ranges
- Attempt to compute graph *k*-coloring
  - *k* is the number of physical registers
  - Top-down and bottom-up differ in coloring order
  - If successful, done!
  - If not successful, spill some values and try again
    - Need a robust way to pick which values to spill
    - Alternatively, split live ranges at carefully-chosen points



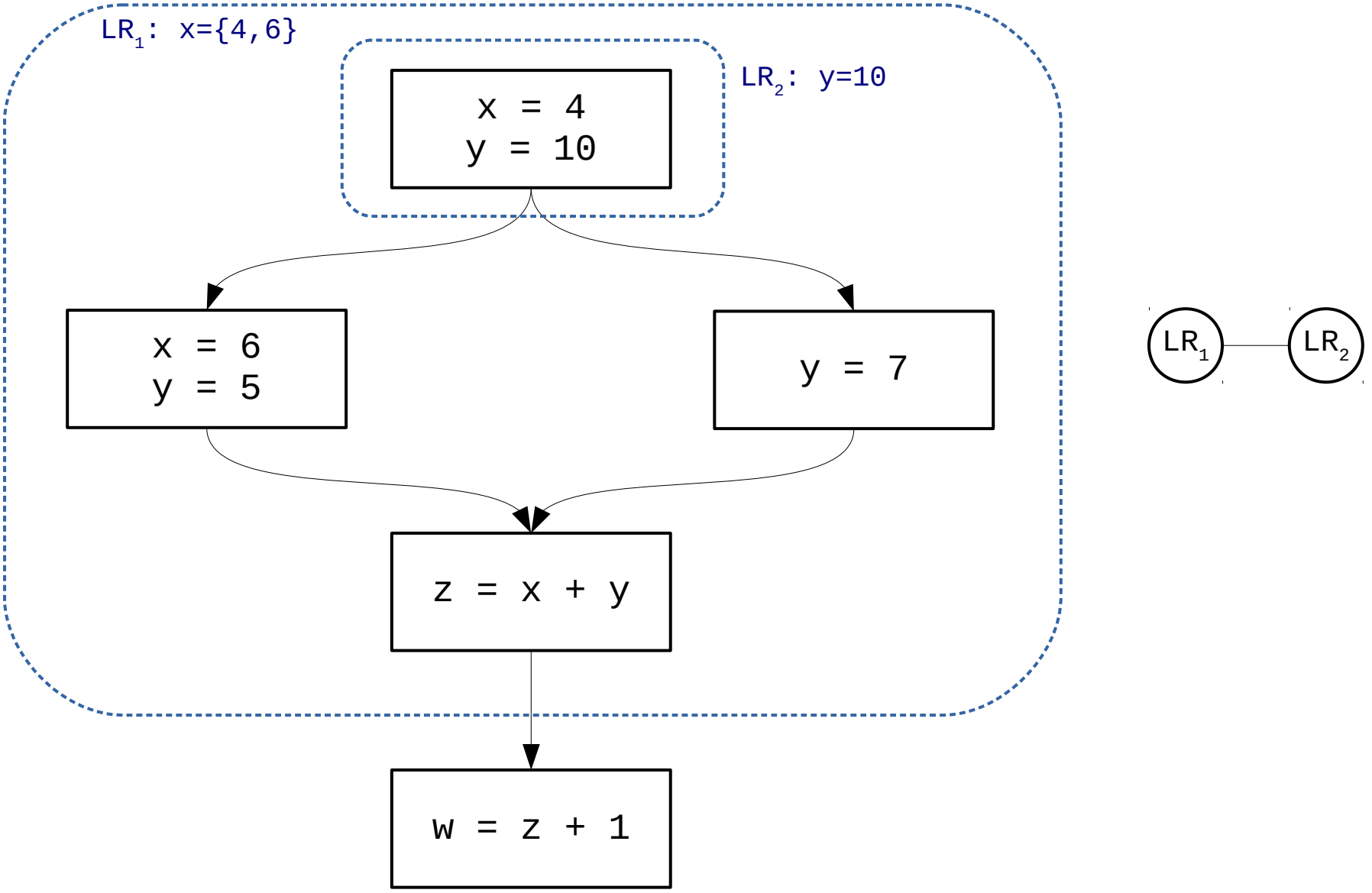
# Global Allocation



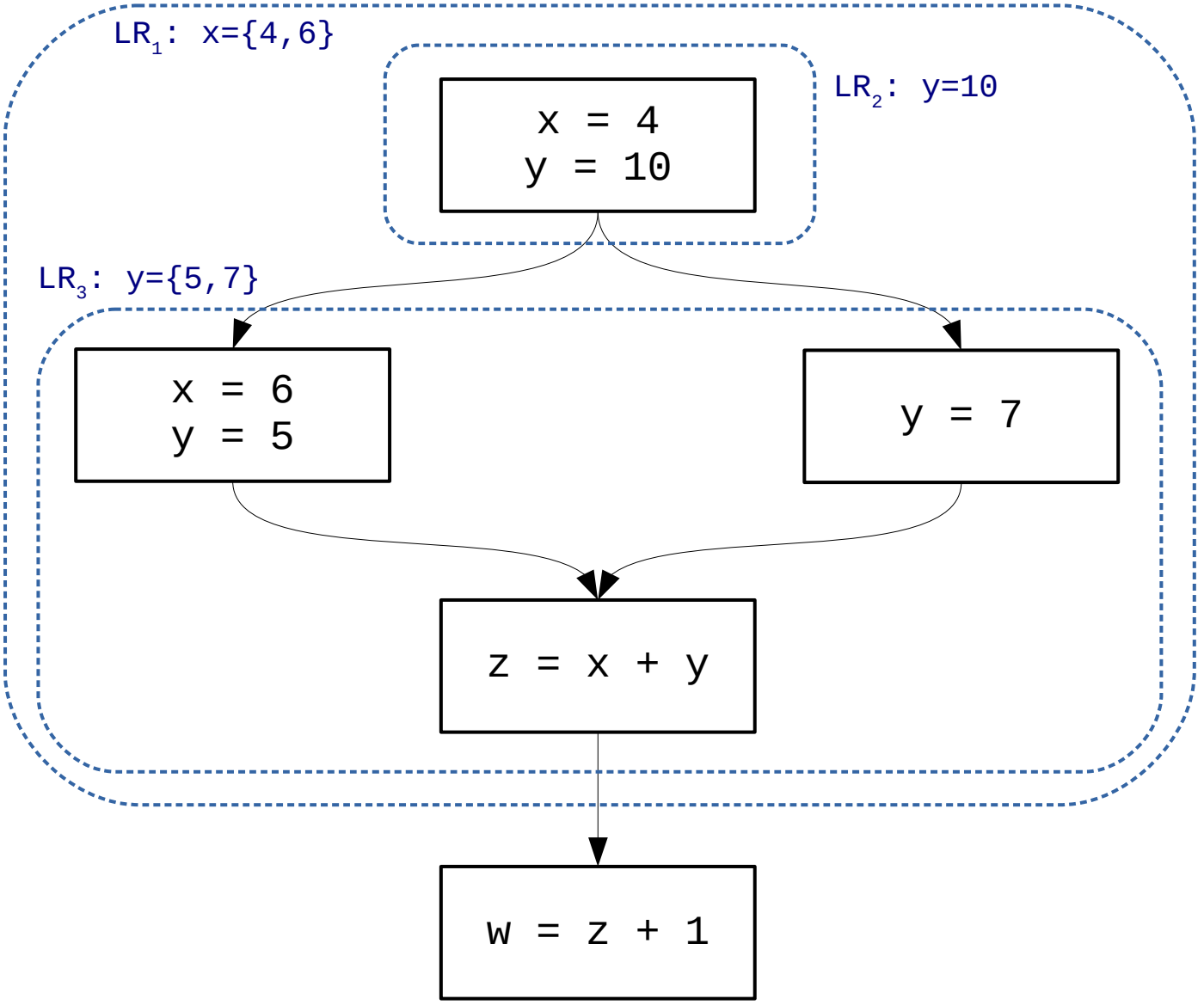
# Global Allocation

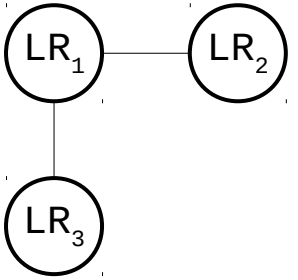


# Global Allocation

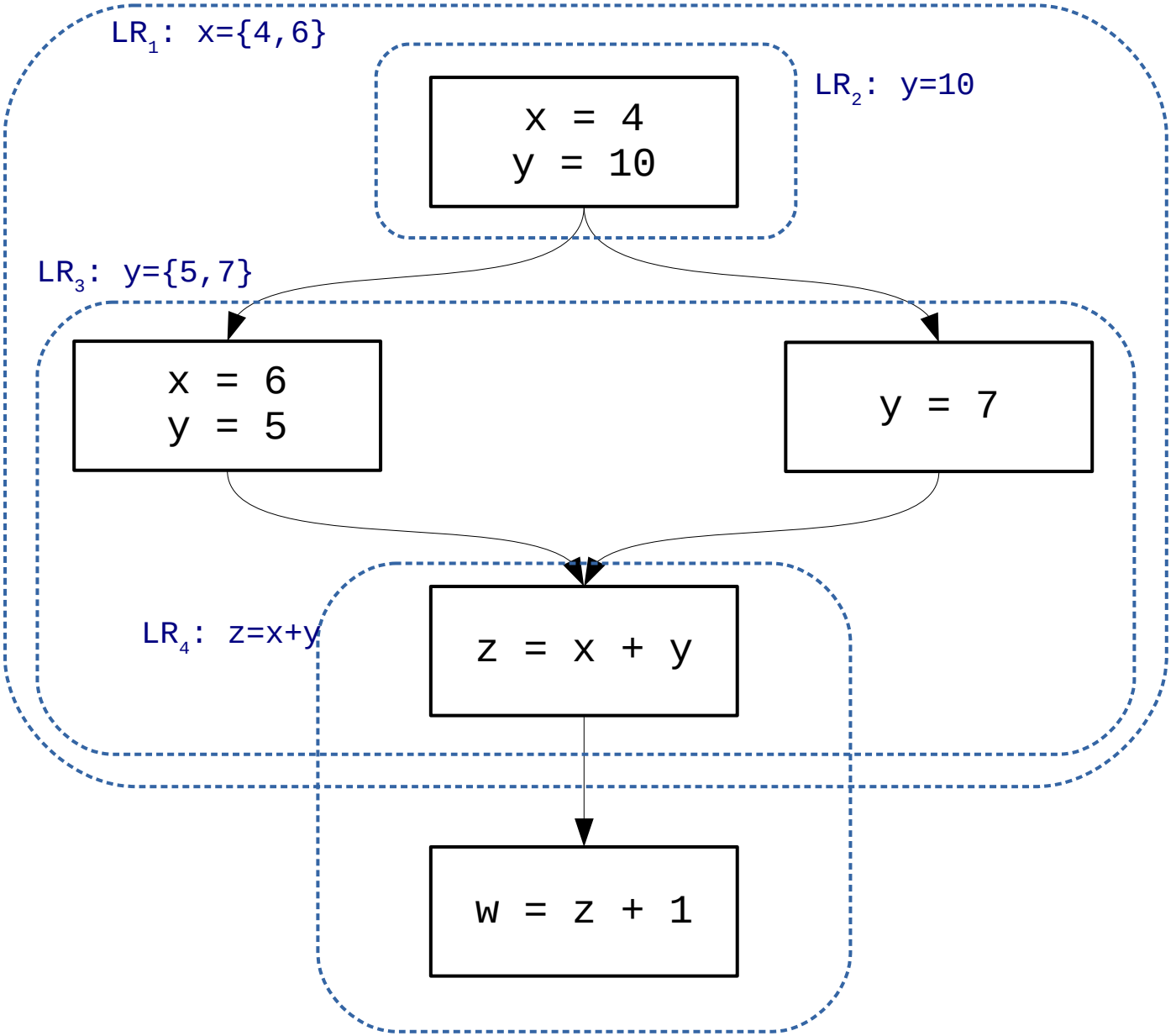


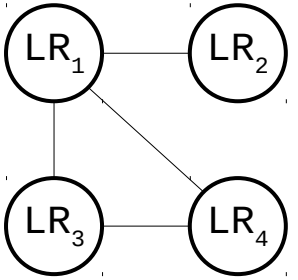
# Global Allocation



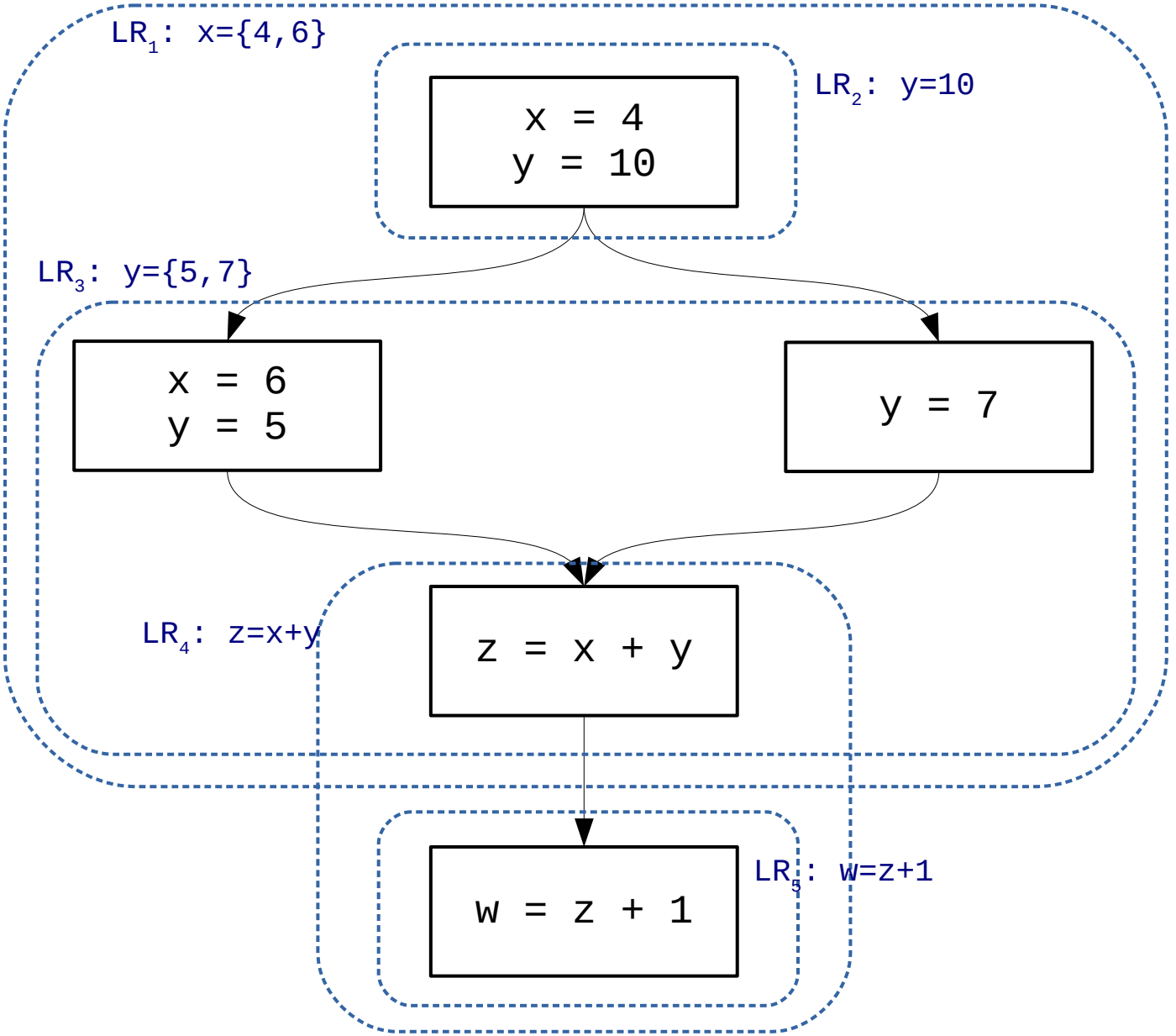



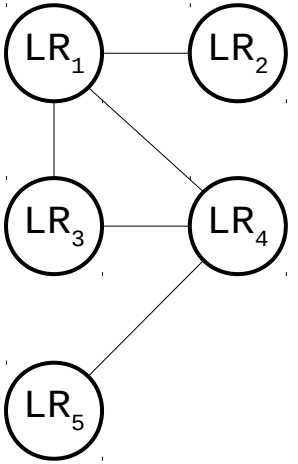
# Global Allocation



# Global Allocation



# Global Allocation