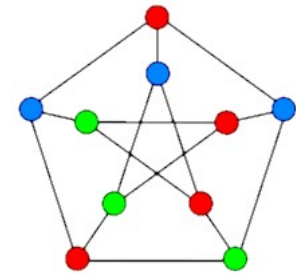
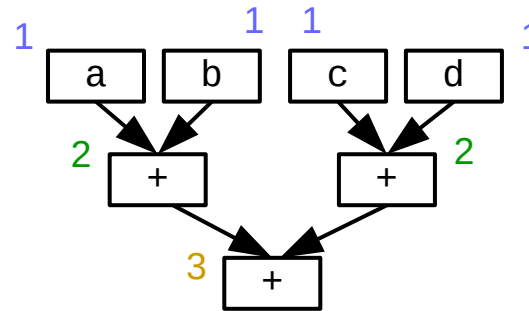


# CS 432

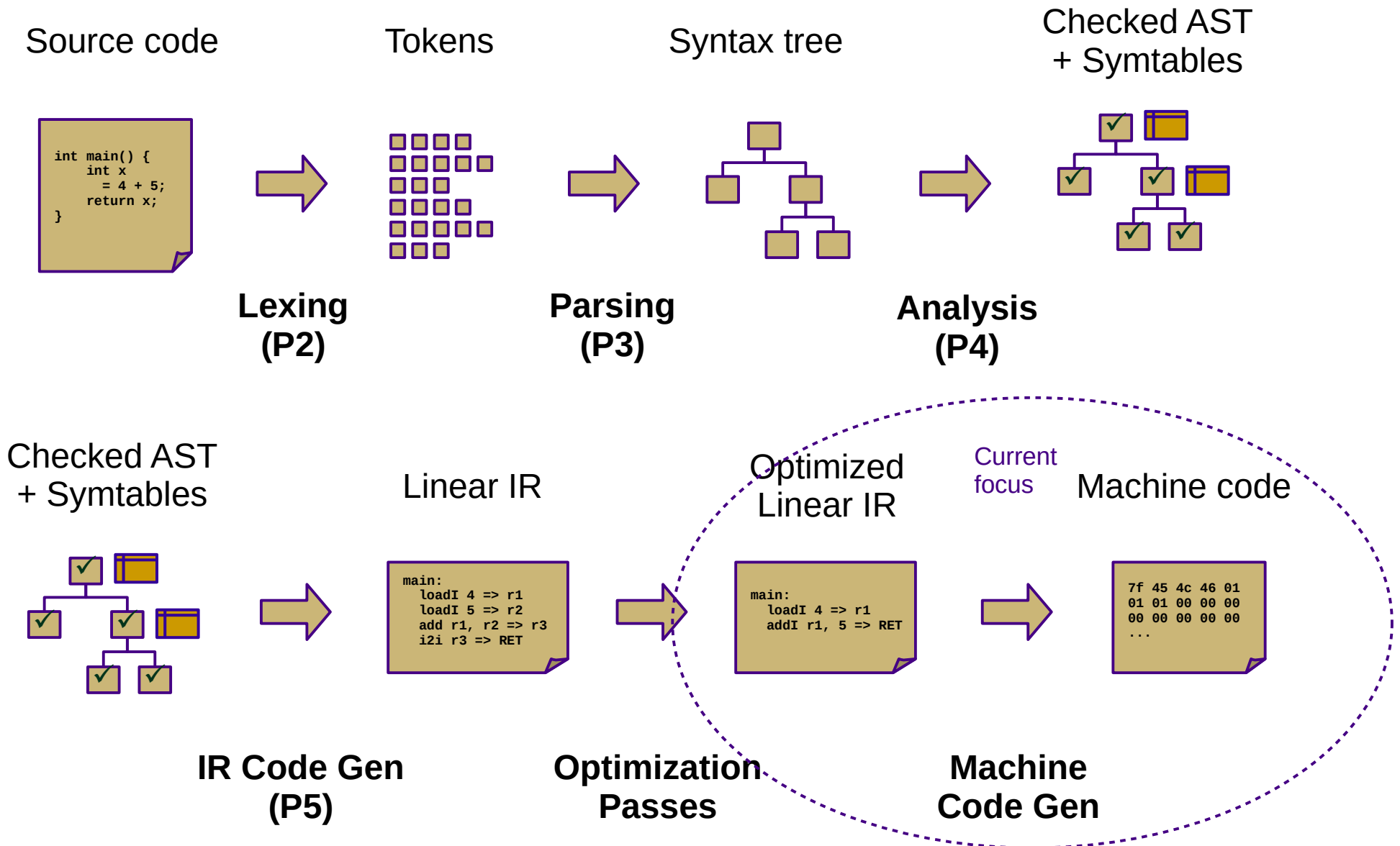
## Fall 2024

Mike Lam, Professor



# Register Allocation

# Compilers

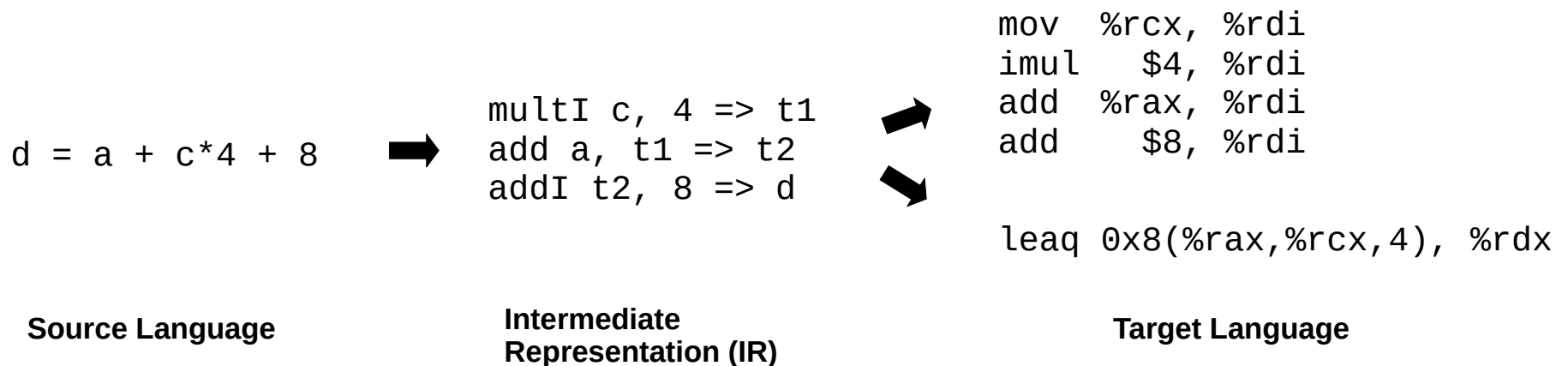


# Machine Code Gen (Ch. 11-13)

- Translate from (usually linear) IR to machine code
  - Often, compilers will 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)** - *example in y86.c provided w/ P5*
  - Arrangement of machine code instructions for optimal pipelining
    - **Instruction scheduling (Ch. 12)** - *algorithm next week; no implementation*
  - Assignment of registers to minimize memory accesses
    - **Register allocation (Ch. 13)** - *primary focus of P5*

# Instruction Selection

- Choose machine code instructions to replace IR
  - Complexity is highly dependent on target architecture
  - CISC provides more options than RISC (e.g., x86 vs. ARM)
    - Tradeoff: *(possible) performance improvement* vs. *compiler complexity*
- Algorithms:
  - **Treewalk** routine (similar to P4)
  - Tree-pattern **matching / tiling** (variant implemented in `y86.c` in P5)

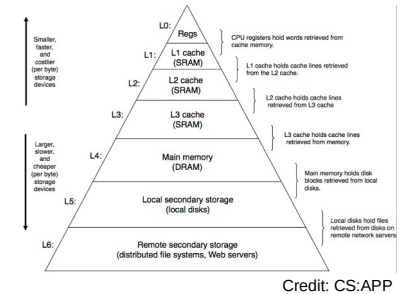


# Instruction Scheduling

- Modern CPUs expose many opportunities for optimization
  - Some instructions require fewer cycles
  - Instruction pipelining
  - Branch prediction and speculative execution
  - 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
    - Everything not in registers must be stored in cache or main memory
  - Need to reduce the # of registers used to match the target system
  - Program using  $n$  registers  $\Rightarrow$  Program using  $m$  registers ( $n \gg m$ )
- 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)



# Question

- Which virtual registers should be allocated to “real” physical registers and which must be allocated elsewhere?

```
add:
    loadAI [bp+16] => r0
    loadAI [bp+24] => r1
    add r0, r1 => r2
    i2i r2 => ret
    return
```

```
main:
    loadI 3 => r3
    storeAI r3 => [bp-8]
    loadAI [bp-8] => r4
    loadI 2 => r5
    param r5
    param r4
    call add
    i2i ret => r6
    i2i r6 => ret
    return
```

# 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
    - Especially in SSA form where registers aren't often re-used



# Local Allocation

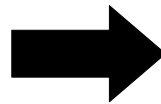
- **Bottom-up local register allocation**
  - Scan each block instruction-by-instruction
    - Essentially, simulate running the program
  - Maintain physical-to-virtual register mapping (“Name”)
    - Initialize registers to empty (“INVALID”) at beginning of block
  - For each instruction:
    - Assign virtual registers to physical registers
    - Ensure operands are in physical registers (load them if not)
    - Greedy algorithm: choose best allocation available at each instruction
      - Track next reference and free physical registers as soon as possible

# Example

Suppose we have three **physical** registers:

Name[R0] = ~~INVALID r0~~ ~~INVALID r2~~ INVALID  
Name[R1] = ~~INVALID r1~~ INVALID  
Name[R2] = INVALID

```
add:  
  loadAI [bp+16] => r0  
  loadAI [bp+24] => r1  
  add r0, r1 => r2  
  i2i r2 => ret  
  return
```

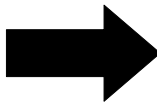


```
add:  
  loadAI [bp+16] => R0  
  loadAI [bp+24] => R1  
  add R0, R1 => R0  
  i2i R0 => ret  
  return
```

# Example

```
add:
  loadAI [bp+16] => r0
  loadAI [bp+24] => r1
  add r0, r1 => r2
  i2i r2 => ret
  return
```

```
main:
  loadI 3 => r3
  storeAI r3 => [bp-8]
  loadAI [bp-8] => r4
  loadI 2 => r5
  param r5
  param r4
  call add
  i2i ret => r6
  i2i r6 => ret
  return
```



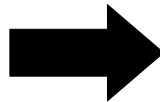
```
add:
  loadAI [bp+16] => R0
  loadAI [bp+24] => R1
  add R0, R1 => R0
  i2i R0 => ret
  return
```

```
main:
  loadI 3 => R0
  storeAI R0 => [bp-8]
  loadAI [bp-8] => R0
  loadI 2 => R1
  param R1
  param R0
  call add
  i2i ret => R0
  i2i R0 => ret
  return
```

Only needed **two** physical registers for this example!

# Example

```
gcd:
l1:
  loadAI [bp+24] => r0
  loadI 1 => r1
  cmp_GE r0, r1 => r2
  cbr r2 => l2, l3
l2:
  loadAI [bp+24] => r3
  loadI 0 => r4
  store r3 => [r4]
  loadAI [bp+16] => r5
  loadAI [bp+24] => r6
  div r5, r6 => r7
  mult r6, r7 => r8
  sub r5, r8 => r9
  storeAI r9 => [bp+24]
  loadI 0 => r10
  load [r10] => r11
  storeAI r11 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => r12
  i2i r12 => ret
  return
```



```
gcd:
l1:
  loadAI [bp+24] => R0
  loadI 1 => R1
  cmp_GE R0, R1 => R0
  cbr R0 => l2, l3
l2:
  loadAI [bp+24] => R0
  loadI 0 => R1
  store R0 => [R1]
  loadAI [bp+16] => R0
  loadAI [bp+24] => R1
  div R0, R1 => R2
  mult R1, R2 => R1
  sub R0, R1 => R0
  storeAI R0 => [bp+24]
  loadI 0 => R0
  load [R0] => R0
  storeAI R0 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => R0
  i2i R0 => ret
  return
```

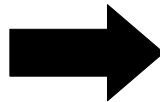
What if we  
only had two  
physical  
registers?

# Spilling

- If no physical registers are free, **spill** one!
  - Store its value to memory and re-load it later
  - For optimal results, spill register that will be accessed the furthest in the future
    - Store Next[pr] for this purpose or just re-calculate when needed
- This is the hardest part of P5 (leave it for last!)
  - Allocate slot in stack frame for each spilled register
    - It's essentially a new local variable
  - Track the offset for each virtual register
  - Emit load/store instructions as needed
    - **Significant helper code is provided!**

# Bottom-up local register allocation

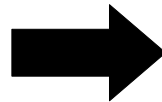
```
gcd:
l1:
  loadAI [bp+24] => r0
  loadI 1 => r1
  cmp_GE r0, r1 => r2
  cbr r2 => l2, l3
l2:
  loadAI [bp+24] => r3
  loadI 0 => r4
  store r3 => [r4]
  loadAI [bp+16] => r5
  loadAI [bp+24] => r6
  div r5, r6 => r7
  mult r6, r7 => r8
  sub r5, r8 => r9
  storeAI r9 => [bp+24]
  loadI 0 => r10
  load [r10] => r11
  storeAI r11 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => r12
  i2i r12 => ret
  return
```



```
gcd:
l1:
  loadAI [bp+24] => R0
  loadI 1 => R1
  cmp_GE R0, R1 => R0
  cbr R0 => l2, l3
l2:
  loadAI [bp+24] => R0
  loadI 0 => R1
  store R0 => [R1]
  loadAI [bp+16] => R0
  loadAI [bp+24] => R1
  div R0, R1 => ???
  mult R1, ??? => R1
  sub R0, R1 => R0
  storeAI R0 => [bp+24]
  loadI 0 => R0
  load [R0] => R0
  storeAI R0 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => R0
  i2i R0 => ret
  return
```

# Bottom-up local register allocation

```
gcd:
l1:
  loadAI [bp+24] => r0
  loadI 1 => r1
  cmp_GE r0, r1 => r2
  cbr r2 => l2, l3
l2:
  loadAI [bp+24] => r3
  loadI 0 => r4
  store r3 => [r4]
  loadAI [bp+16] => r5
  loadAI [bp+24] => r6
  div r5, r6 => r7
  mult r6, r7 => r8
  sub r5, r8 => r9
  storeAI r9 => [bp+24]
  loadI 0 => r10
  load [r10] => r11
  storeAI r11 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => r12
  i2i r12 => ret
  return
```



```
gcd:
l1:
  loadAI [bp+24] => R0
  loadI 1 => R1
  cmp_GE R0, R1 => R0
  cbr R0 => l2, l3
l2:
  loadAI [bp+24] => R0
  loadI 0 => R1
  store R0 => [R1]
  loadAI [bp+16] => R0
  loadAI [bp+24] => R1
  storeAI R0 => [bp-8] // store r5
  div R0, R1 => R0
  mult R1, R0 => R1
  loadAI [bp-8] => R0 // load r5
  sub R0, R1 => R0
  storeAI R0 => [bp+24]
  loadI 0 => R0
  load [R0] => R0
  storeAI R0 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => R0
  i2i R0 => ret
  return
```

# Full algorithm

for each instruction  $i$  in program:

for each  $vr$  read in  $i$ :

$pr = \mathbf{Ensure}(vr)$

replace  $vr$  with  $pr$  in  $i$

if  $vr$  is not needed after  $i$  then free  $pr$

for each  $vr$  written in  $i$ :

$pr = \mathbf{Allocate}(vr)$

replace  $vr$  with  $pr$  in  $i$

**Ensure**( $vr$ ):

if  $vr$  is in  $pr$ :

return  $pr$

else:

$pr = \mathbf{Allocate}(vr)$

emit load from  $vr$

return  $pr$

**Allocate**( $vr$ ):

if  $pr$  is available:

return  $pr$

else:

find furthest-used  $pr$  to spill

emit spill for  $pr$

return  $pr$



# Textbook vs. reference compiler

- Textbook algorithm uses a stack to store free registers
  - Must remember to add registers to stack when freeing them
  - $O(1)$  access to a free register if one is available
- Reference compiler scans physical registers for first free one
  - $O(k)$  where  $k$  is number of physical registers, which is essentially a small constant
  - Only need  $\text{Name}[pr]$ 
    - $pr$  is free if  $\text{Name}[pr] == \text{INVALID}$

TEXTBOOK:

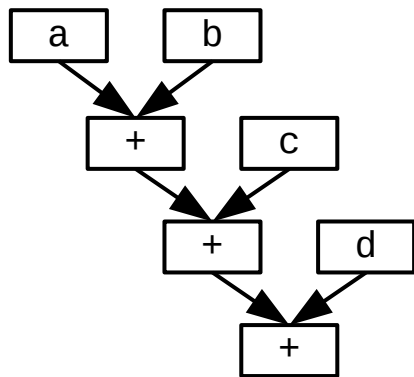
```
loadI 1 => R0
loadI 2 => R1
add R0, R1 => R1
```

REFERENCE:

```
loadI 1 => R0
loadI 2 => R1
add R0, R1 => R0
```

# Expression evaluation

- How many registers does it take to evaluate an arbitrary expression without any spilling?
  - Is there an easy way to determine this?

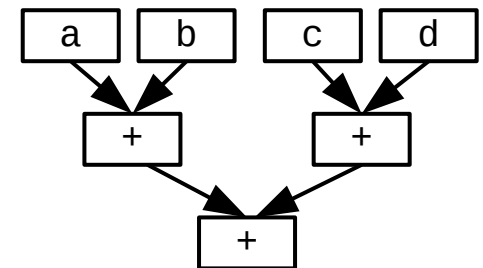


For example:

$a + b + c + d$

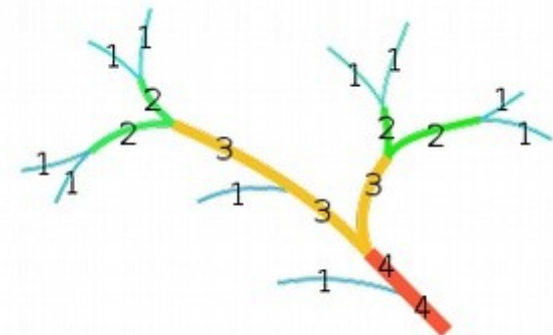
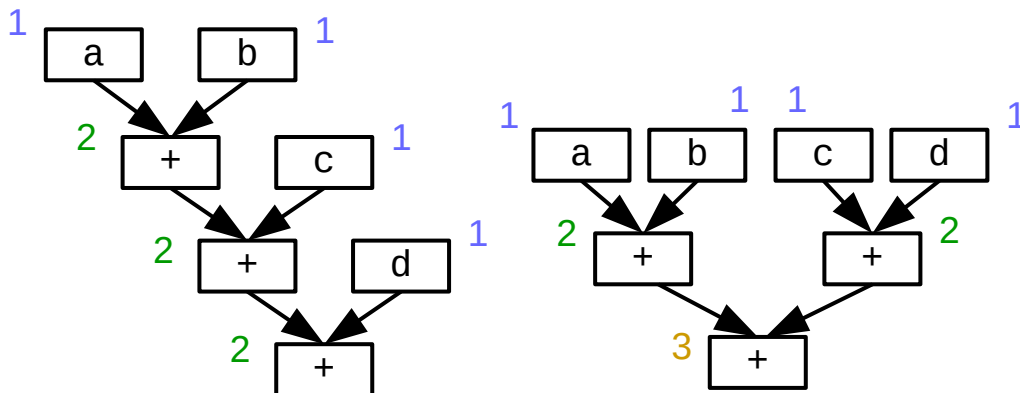
vs.

$(a+b) + (c+d)$



# Expression evaluation

- How many registers does it take to evaluate an an arbitrary expression without any spilling?
  - Examine the expression tree (e.g., parse tree)
  - Calculate the **Strahler number**:
    - If the node is a leaf (has no children), its Strahler number is one.
    - If the node has one child with Strahler number  $i$ , and all other children have Strahler numbers less than  $i$ , then the Strahler number of the node is  $i$ .
    - If the node has two or more children with Strahler number  $i$ , and no children with greater number, then the Strahler number of the node is  $i + 1$ .



# Systems design tradeoff

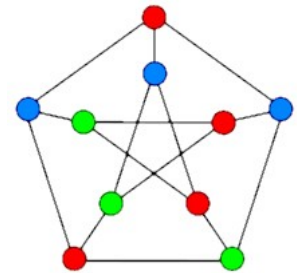
- Parallelism vs. register pressure
  - Balanced trees provide **more parallelism** and (as we'll see next week) **better pipelining**
    - However, more spills => **worse performance**
  - Unbalanced trees require fewer registers
    - Fewer spills => **better performance**
    - However, **fewer opportunities for parallelism and pipelining**
  - Usually the parallelism is worth the increased register pressure
    - Especially in the presence of forwarding and robust caches

# Local vs. global allocation

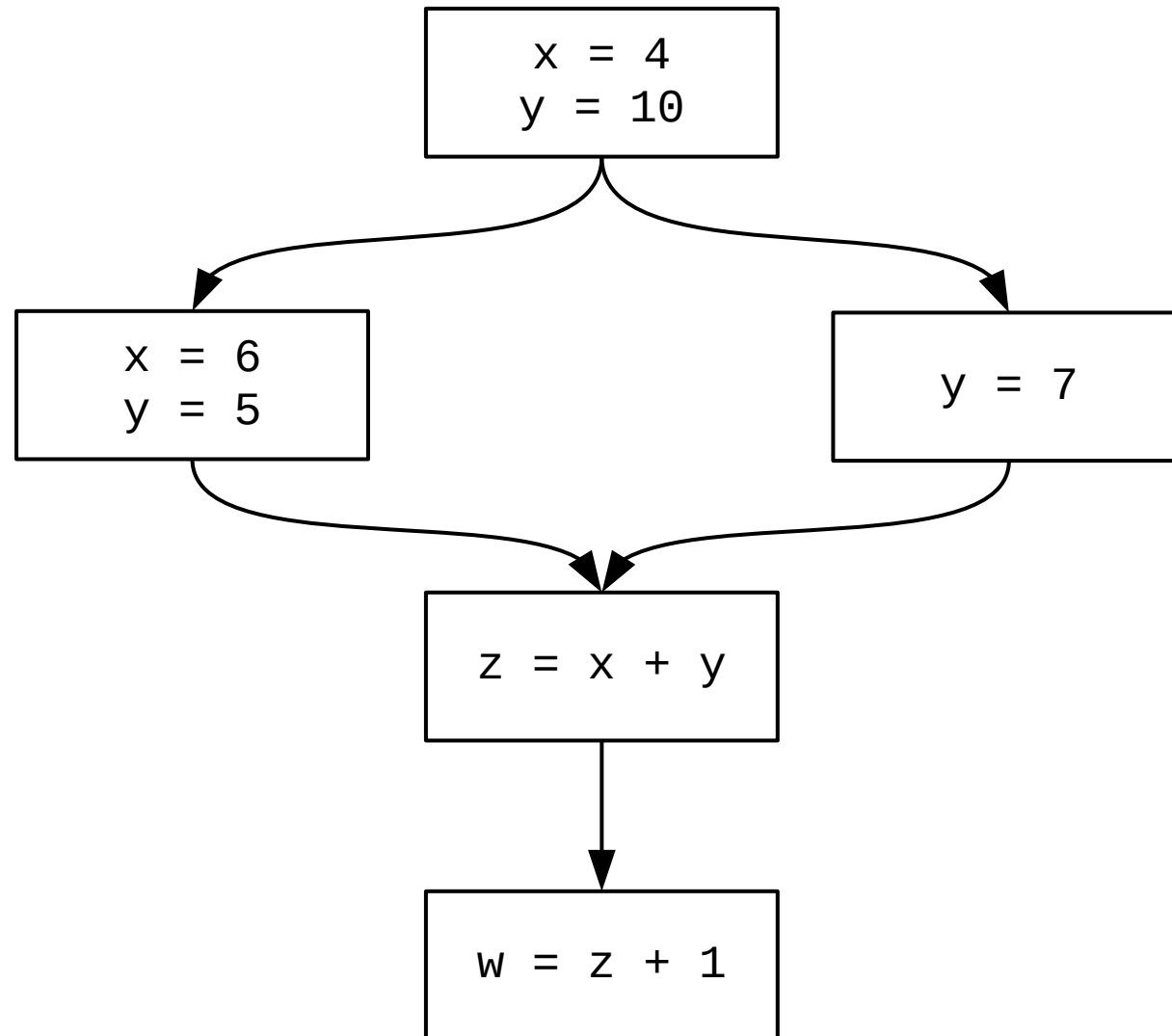
- Local allocation handles each basic block separately
  - Will miss inter-block dependencies
- Global allocation handles all basic blocks in a procedure
  - Does NOT consider inter-procedural dependencies
  - This is why calling conventions are important
    - I.e., caller-save vs. callee-save and return value
- Decaf project
  - Because we used SSA in P4 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 P5

# Global Allocation

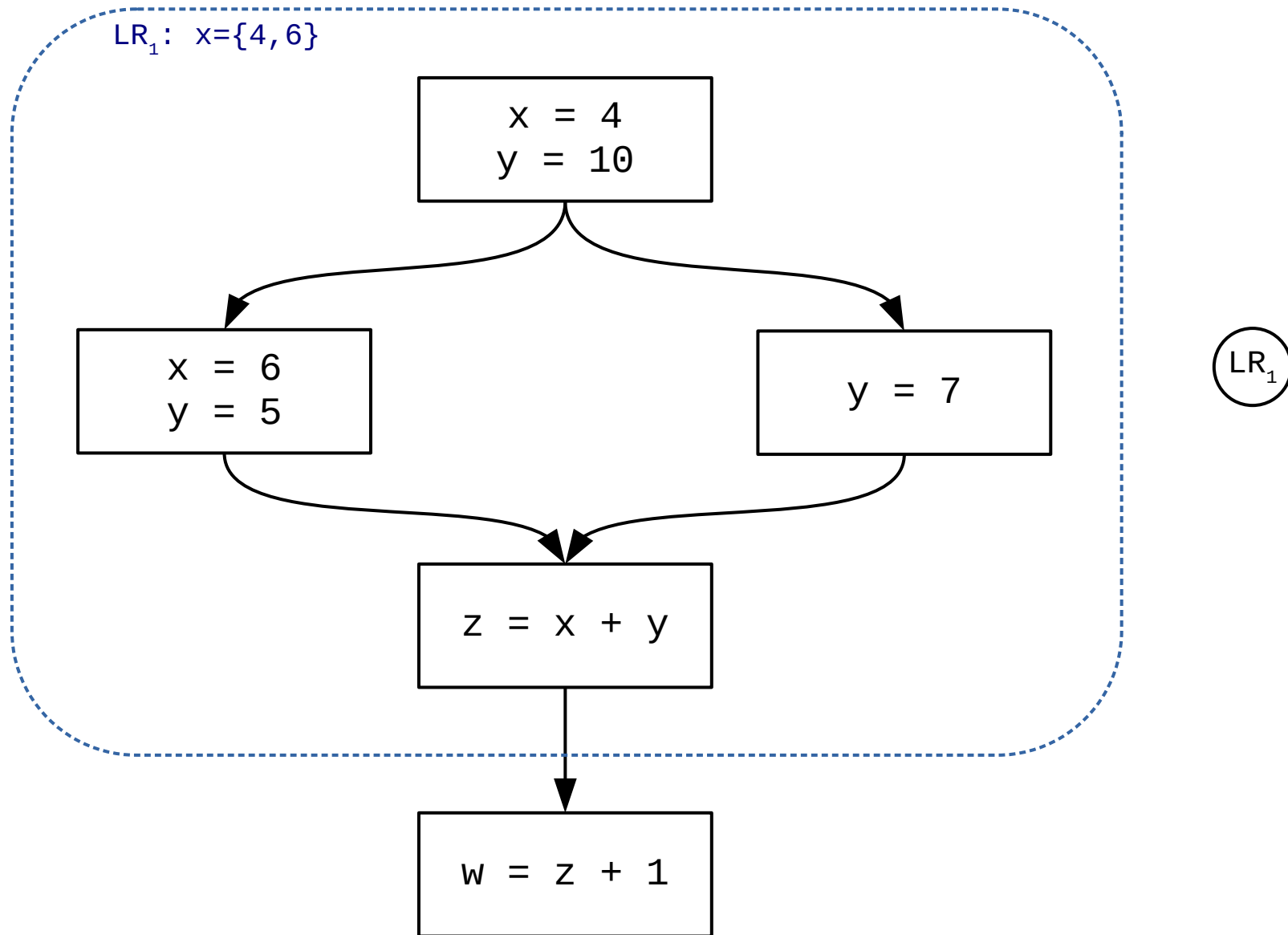
- Discover global **live ranges** of related uses and definitions
  - For each use, any reaching definitions must be in the same range
  - For each definition, any reachable uses must be in the same range
  - Simple disjoint-set union-find algorithm over SSA form
- Build **interference graph**
  - Node for each live range and edges between interfering live ranges
- Attempt to compute **graph  $k$ -coloring**
  - $k$  is the number of physical registers
  - Greedy algorithm: order the colors (registers)
    - For each vertex, choose smallest color not shared by neighbors
  - 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

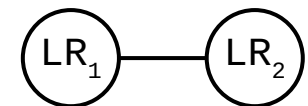
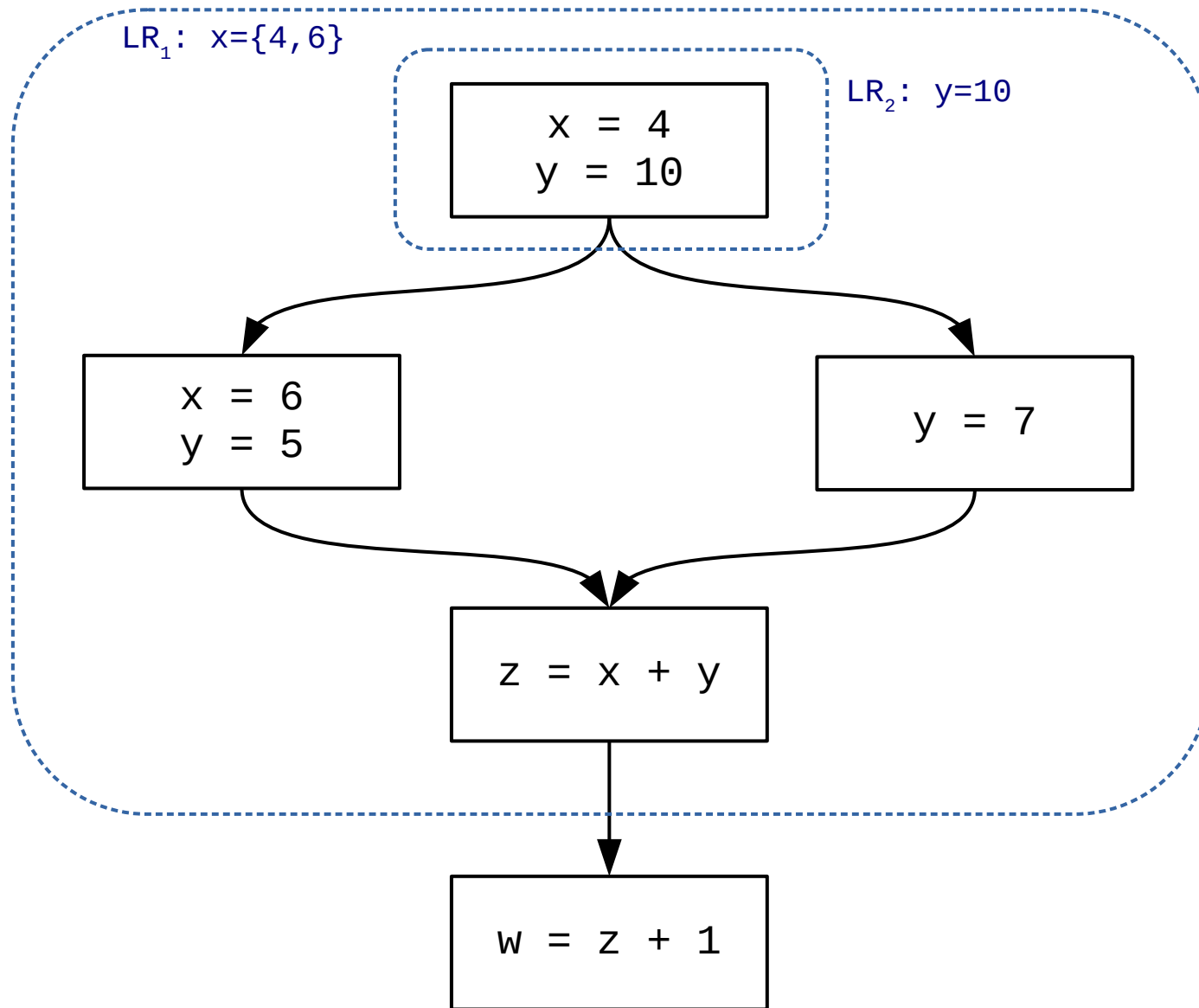


# Global Allocation

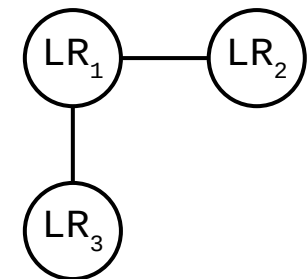
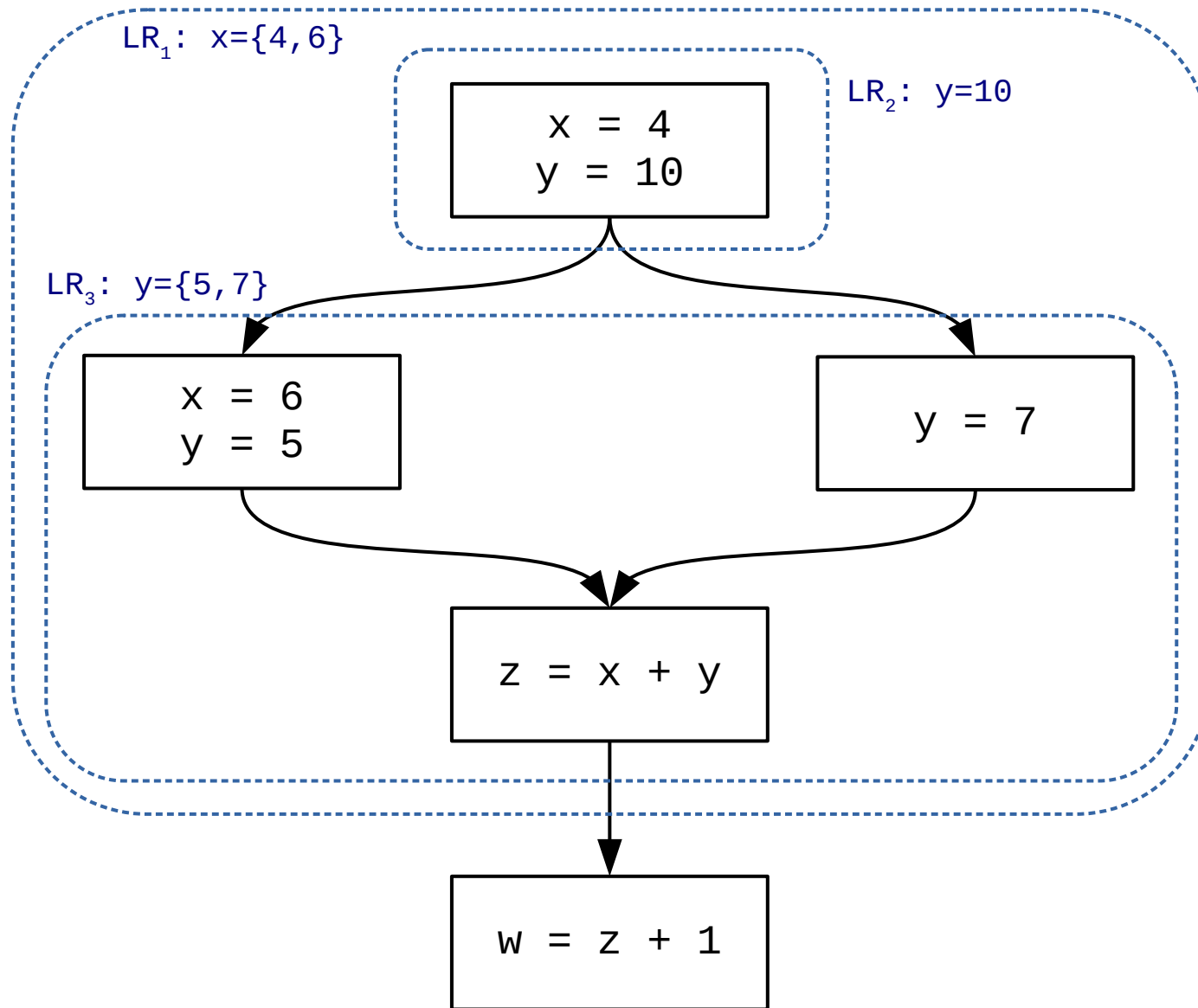




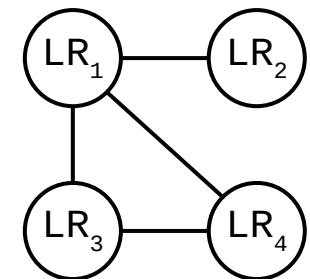
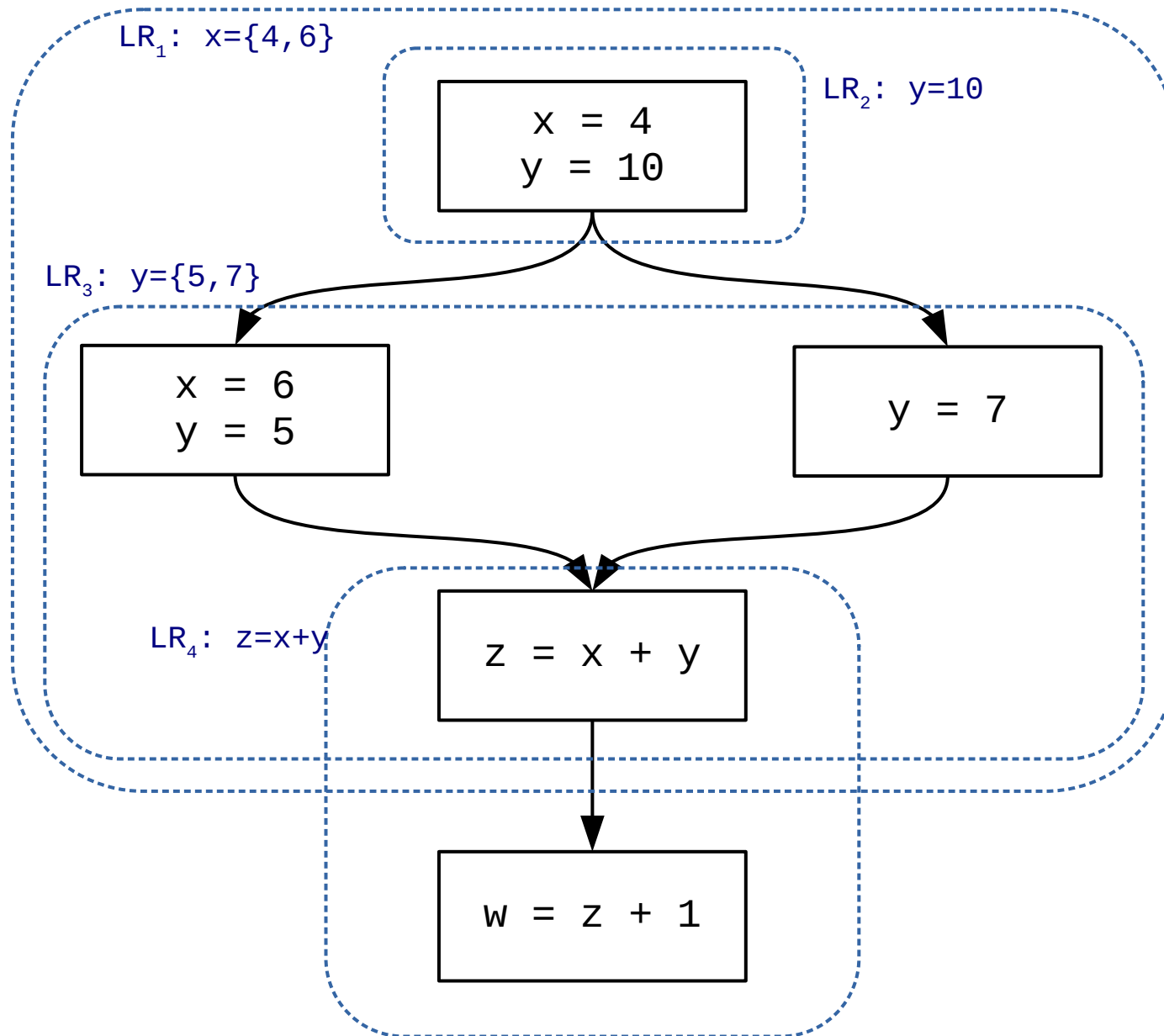
# Global Allocation



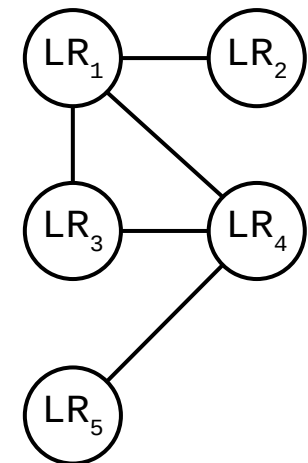
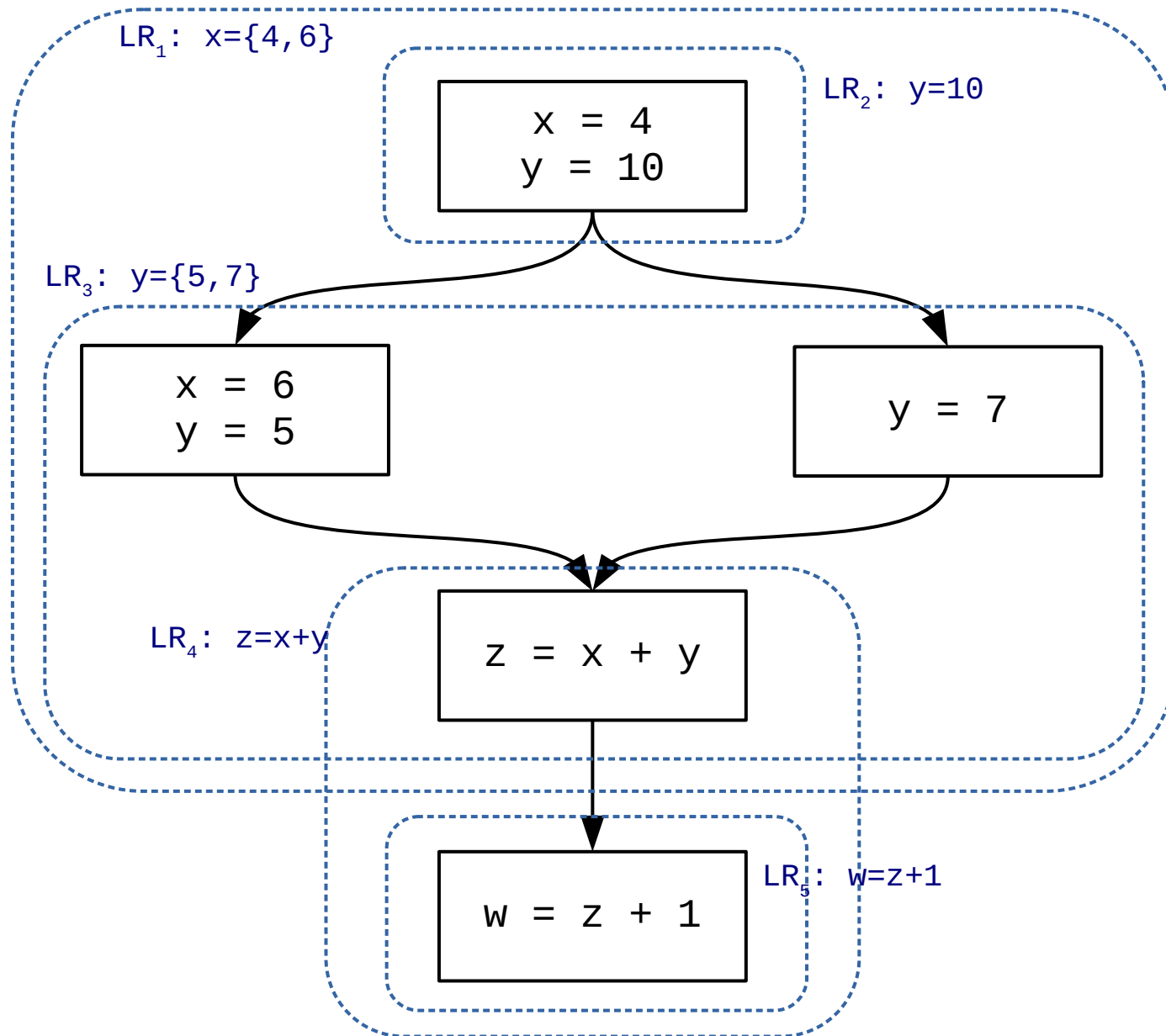
# Global Allocation



# Global Allocation



# Global Allocation



# Global Allocation

