

# CSE 401 – Compilers

Compiler Backend Survey

Hal Perkins

Winter 2015

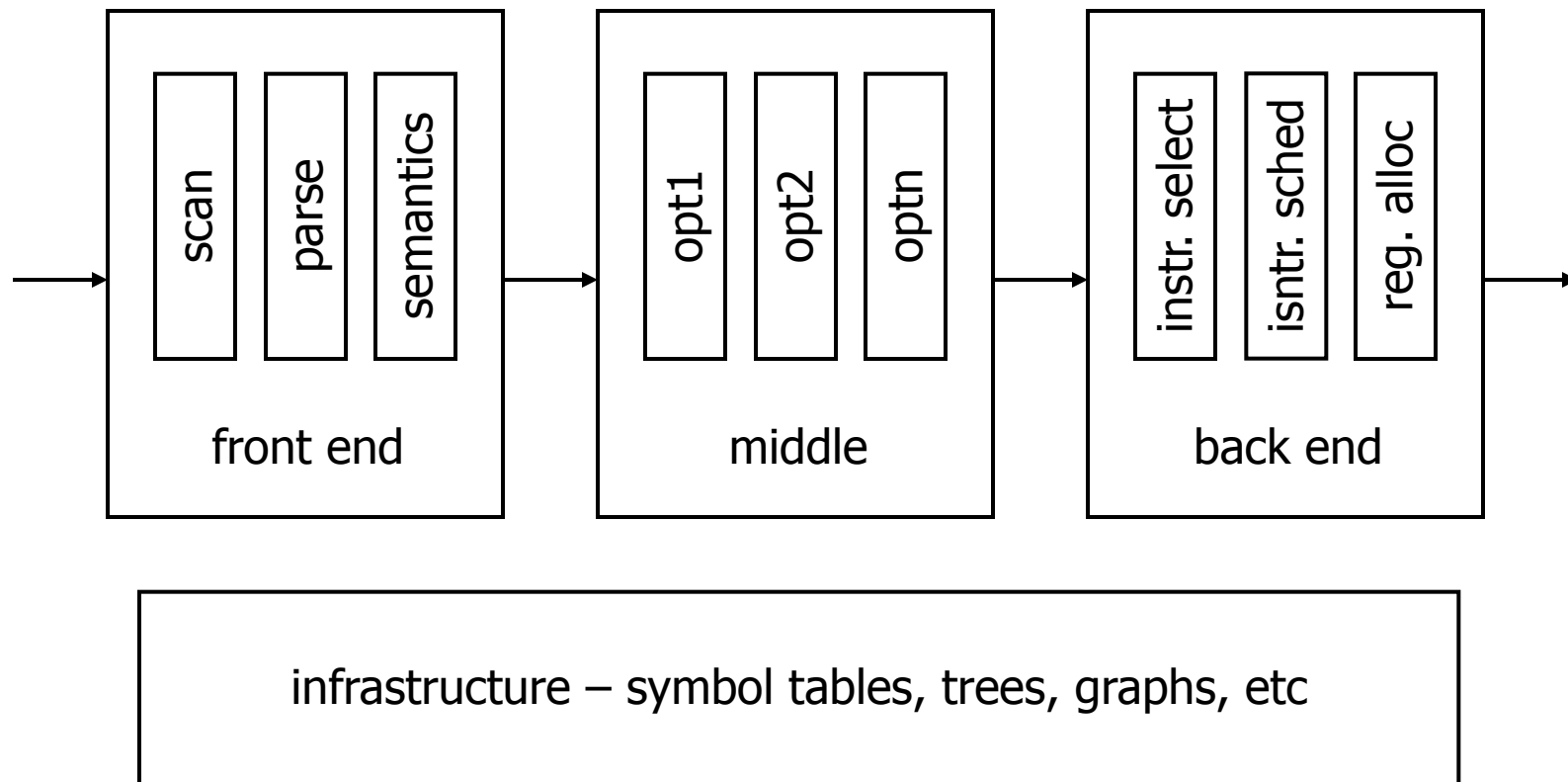
# Administrivia

- Wrapup for compiler project: short report due Saturday night after classes over (*no* late days)
- We'll grade the compiler additions, then do an overall evaluation of each project
  - Please fix up any loose ends or lingering problems from earlier phases before final submission
- Course evals: please do them this week
- Final exam: next Tuesday, 2:30
  - Review Monday, 4:30, MOR 230
  - Topic list available on the course web now

# Agenda

- Survey major pieces of a compiler back end
  - Instruction selection
  - Instruction scheduling
  - Register allocation
- And three particularly neat algorithms
  - Instruction selection by tree pattern matching
  - Instruction list scheduling
  - Register allocation by graph coloring

# Compiler Organization



# Big Picture

- Compiler consists of lots of fast stuff followed by hard problems
  - Scanner:  $O(n)$
  - Parser:  $O(n)$
  - Analysis & Optimization:  $\sim O(n \log n)$
  - Instruction selection: fast or NP-Complete
  - Instruction scheduling: NP-Complete
  - Register allocation: NP-Complete

# IR for Code Generation

- Assume a low-level RISC-like IR
  - 3 address, register-register instructions plus load/store
    - $r1 \leftarrow r2 \text{ op } r3$
  - Could be tree structure or linear
  - Expose as much detail as possible
- Assume “enough” (i.e.,  $\infty$ ) registers
  - Invent new temporaries for intermediate results
  - Map to actual registers towards the end

# Overview: Instruction Selection

- Map IR into assembly code
- Assume known storage layout and code shape
  - i.e., the optimization phases have already done their thing
- Combine low-level IR operations into machine instructions (take advantage of addressing modes, etc.)

# Overview: Instruction Scheduling

- Reorder instructions to minimize execution time
  - hide latencies – processor function units, memory/cache stalls
  - Originally invented for supercomputers (60s)
  - Required to get reasonable (or correct!) RISC code
  - Still important on most machines
    - Even non-RISC machines, e.g., x86 family
    - Even if processor reorders on the fly
- Good schedules help processor do a better job
- Assume fixed program at this point



# Overview: Register Allocation

- Map values to actual registers
  - Previous phases change need for registers
- Add code to spill values to temporaries in memory and reload as needed, etc.
- Usually worth doing another pass of instruction scheduling afterwards if spill code inserted

# Conventional Wisdom

- We typically lose little by solving these independently
  - But not always; depends on architecture
- Instruction selection
  - Use some form of pattern matching
  - $\infty$  virtual registers – create as needed
- Instruction scheduling
  - Within a block, list scheduling is close to optimal
  - Across blocks: extended basic blocks or trace scheduling if list scheduling not good enough
- Register allocation
  - Start with unlimited virtual registers and map to some subset of  $K$  real registers

# Instruction Selection

- Map IR into assembly code
- Assume known storage layout and code shape

# A Simple Low-Level IR (1)

- This example is from Appel, but details aren't really important. What matters is to get a feel for the level of detail involved.
- Expressions:
  - $\text{CONST}(i)$  – integer constant  $i$
  - $\text{TEMP}(t)$  – temporary  $t$  (i.e., register)
  - $\text{BINOP}(op, e1, e2)$  – application of  $op$  to  $e1, e2$
  - $\text{MEM}(e)$  – contents of memory at address  $e$ 
    - Means value when used in an expression
    - Means address when used as target of assignment
  - $\text{CALL}(f, args)$  – apply function  $f$  to argument list  $args$

# Simple Low-Level IR (2)

- Statements
  - MOVE(TEMP t, e) – evaluate e and store in temporary t
  - MOVE(MEM(e1), e2) – evaluate e1 to yield address a; evaluate e2 and store at a
  - EXP(e) – evaluate expressions e and discard result
  - SEQ(s1,s2) – execute s1 followed by s2
  - NAME(n) – assembly language label n
  - JUMP(e) – jump to e, which can be a NAME label, or more complex (e.g., switch)
  - CJUMP(op,e1,e2,t,f) – evaluate e1 op e2; if true jump to label t, otherwise jump to f
  - LABEL(n) – defines location of label n in the code

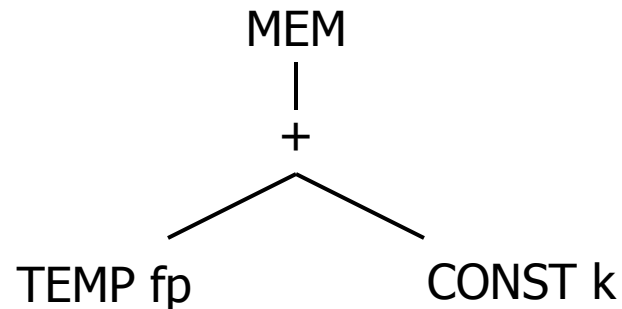
# Low-Level IR Example (1)

- Access a local variable at a known offset  $k$  from the frame pointer  $fp$

- Linear

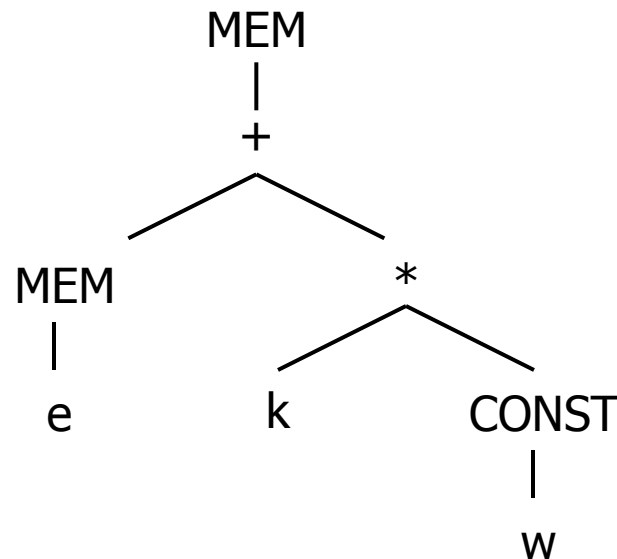
MEM(BINOP(PLUS, TEMP  $fp$ , CONST  $k$ ))

- Tree



# Low-Level IR Example (2)

- Access an array element  $e[k]$ , where each element takes up  $w$  storage locations



# Instruction Selection Issues

- Given the low-level IR, there are many possible code sequences that implement it correctly
  - e.g. to set `eax` to 0 on x86-64 (did we miss some?)

<code>movq \$0,%rax</code>	<code>salq 64,%rax</code>
<code>subq %rax,%rax</code>	<code>shrq 64,%rax</code>
<code>xorq %rax,%rax</code>	<code>imulq \$0,%rax</code>
  - Many machine instructions do several things at once – e.g., register arithmetic and effective address calculation, e.g.,  
`leaq offset(%rbase, %rindex, scale), %rdest`



# Instruction Selection Criteria

- Several possibilities
  - Fastest
  - Smallest
  - Minimize power consumption (ex: don't use a function unit if leaving it powered-down is a win)
- Sometimes not obvious
  - e.g., if one of the function units in the processor is idle and we can select an instruction that uses that unit, it effectively executes for free, even if that instruction wouldn't be chosen normally
    - (Some interaction with scheduling here...)
    - (and it might consume extra power, so bad if that matters)

# Tree Pattern Matching

- Goal: find a sequence of machine instructions that perform the computation described by the program IR code
  - Describe machine instructions using same low-level IR used for program, then
  - Use tree pattern matching to pick instructions that match fragments of the program IR tree; use a combination of these to cover the whole IR tree

# An Example Target Machine (1)

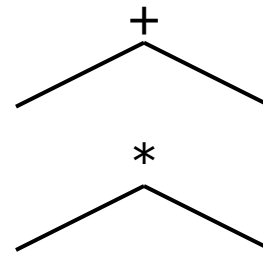
- Arithmetic Instructions

- (unnamed)  $r_i$  TEMP

- ADD  $r_i \leftarrow r_j + r_k$

- MUL  $r_i \leftarrow r_j * r_k$

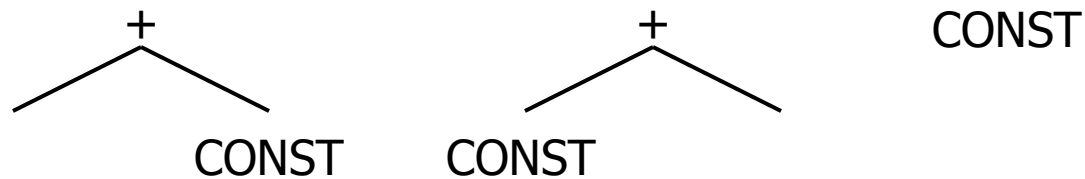
- SUB and DIV are similar



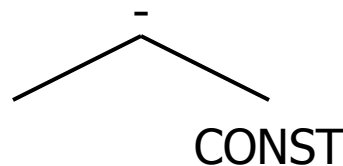
# An Example Target Machine (2)

- Immediate Instructions

- ADDI  $r_i \leftarrow r_j + c$

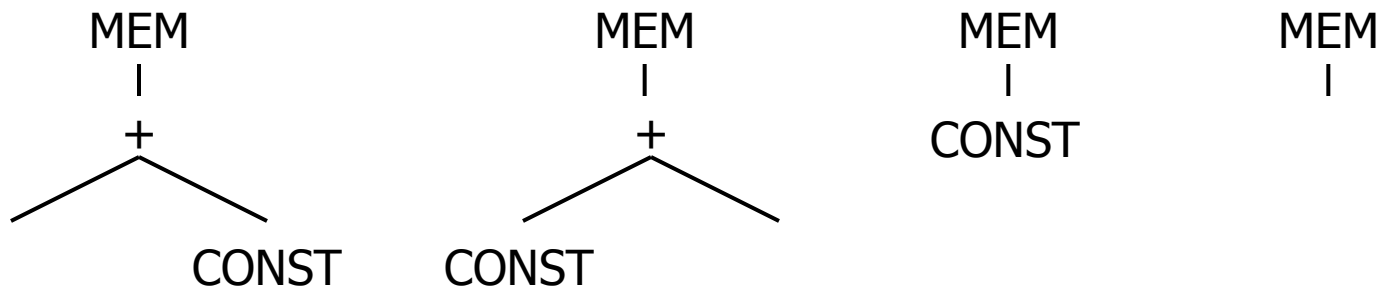


- SUBI  $r_i \leftarrow r_j - c$



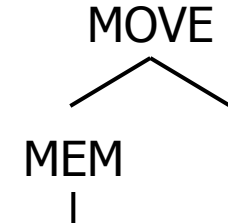
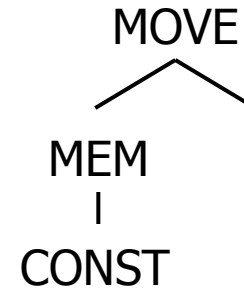
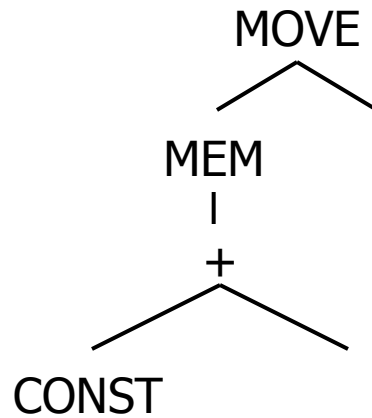
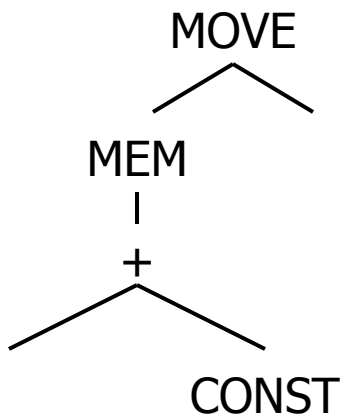
# An Example Target Machine (3)

- Load
  - LOAD  $r_i \leftarrow M[r_j + c]$



# An Example Target Machine (4)

- Store
  - STORE  $M[rj + c] \leftarrow r_i$



# Tree Pattern Matching (1)

- Goal: Tile the low-level tree with operation (instruction) trees
- A *tiling* is a collection of  $\langle \text{node}, \text{op} \rangle$  pairs
  - node is a node in the tree
  - op is an operation tree
  - $\langle \text{node}, \text{op} \rangle$  means that op could implement the subtree at node

# Tree Pattern Matching (2)

- A tiling “implements” a tree if it covers every node in the tree and the overlap between any two tiles (trees) is limited to a single node
  - If  $\langle \text{node}, \text{op} \rangle$  is in the tiling, then node is also covered by a leaf of another operation tree in the tiling – unless it is the root
  - Where two operation trees meet, they must be compatible (i.e., expect the same value in the same location)

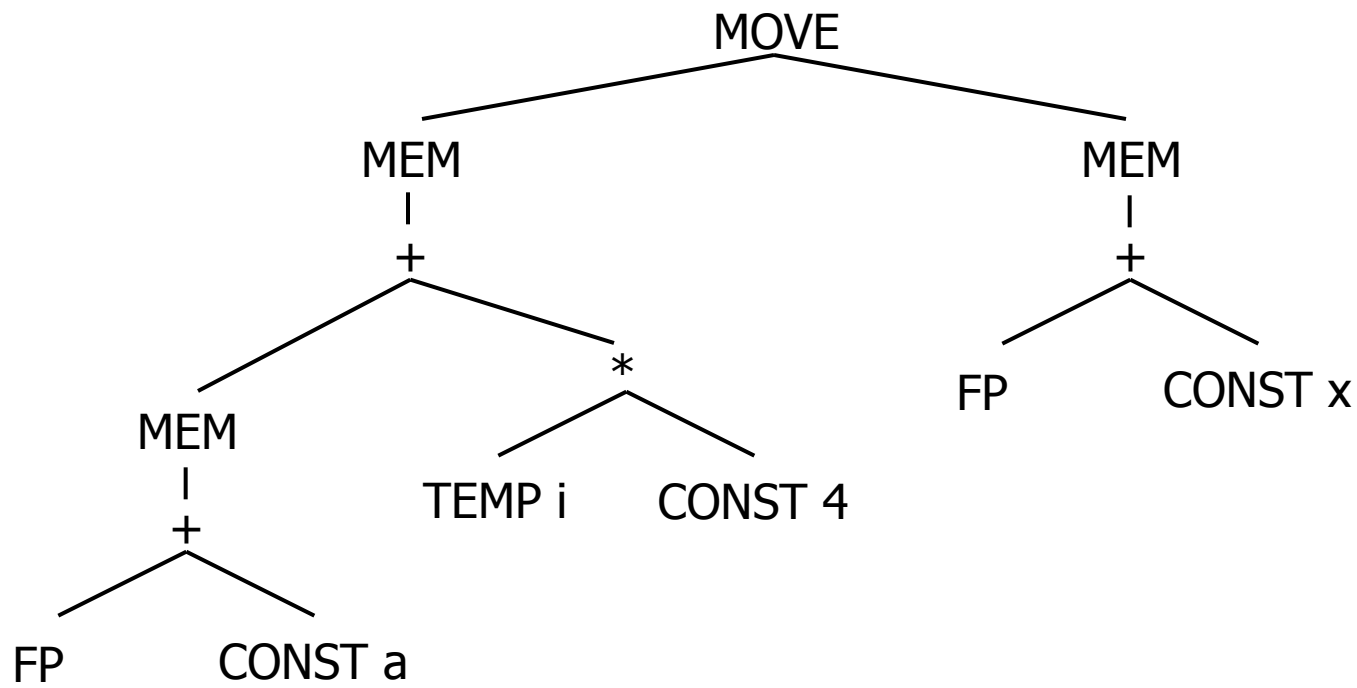


# Generating Tilings

Two common algorithms

- Maximal munch:
  - Top-down tree walk.
  - Find largest tile that fits each node
- Dynamic programming:
  - Assign costs to each node in the tree  
cost = cost of individual node + subtree costs
  - Try all possible combinations bottom-up and pick cheapest

# Example – Tree for $a[i]:=x$



# Generating Code

Given a tiled tree, to generate code

- Do a postorder treewalk with node-dependant order for children
- Each tile corresponds to a code sequence; emit code sequences in order
- Connect tiles by using same register name to tie boundaries together

# Instruction Scheduling

- Reorder instructions to minimize execution time given instruction and operand latencies
- Assume fixed program at this point

# Some Scheduling Issues

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
  - Want to take advantage of multiple function units on chip
- Loads & Stores may or may not block
  - may be cycles after load/store start for other useful work
- Branch costs vary
- Modern processors have heuristics to predict whether branches are taken and try to keep pipelines full

GOAL: Scheduler should reorder instructions to hide latencies, take advantage of multiple function units and delay slots, and help the processor effectively pipeline execution

# Latencies for a Simple Example Machine

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1
BRANCH	0 TO 8

# Example: $w = w * 2 * x * y * z;$

## Simple schedule

```
1  LOAD    r1 <- w
4  ADD     r1 <- r1,r1
5  LOAD    r2 <- x
8  MULT    r1 <- r1,r2
9  LOAD    r2 <- y
12 MULT    r1 <- r1,r2
13 LOAD    r2 <- z
16 MULT    r1 <- r1,r2
18 STORE   w <- r1
21 r1 free
```

2 registers, 20 cycles

## Loads early

```
1  LOAD    r1 <- w
2  LOAD    r2 <- x
3  LOAD    r3 <- y
4  ADD     r1 <- r1,r1
5  MULT    r1 <- r1,r2
6  LOAD    r2 <- z
7  MULT    r1 <- r1,r3
9  MULT    r1 <- r1,r2
11 STORE   w <- r1
14 r1 is free
```

3 registers, 13 cycles

# List Scheduling Algorithm Overview

- Build a precedence graph  $P$  of instructions, labeled with priorities (usually number of cycles on critical path to the end)
- Use list scheduling to construct a schedule, one cycle at a time
- Rename registers to avoid false dependencies and conflicts



# Precedence Graph

- Nodes  $n$  are operations
- Attributes of each node
  - type – kind of operation
  - delay – latency until end of graph
- If node  $n_2$  uses the result of node  $n_1$ , there is an edge  $e = (n_1, n_2)$  from  $n_1$  to  $n_2$  in the graph

# List Scheduling

- Construct a schedule, one cycle at a time
  - Keep a list of operations that are ready to execute
  - At each cycle, chose a ready operation and schedule it
    - Best pick: one that is on the “critical path” – i.e., an instruction that has longest latency from end of graph
  - Update ready list, deleting scheduled op and add ones that will be ready on next cycle

# Example

- Code
  - a LOAD      r1 <- w
  - b ADD        r1 <- r1,r1
  - c LOAD        r2 <- x
  - d MULT       r1 <- r1,r2
  - e LOAD        r2 <- y
  - f MULT        r1 <- r1,r2
  - g LOAD        r2 <- z
  - h MULT        r1 <- r1,r2
  - i STORE       w <- r1

# Forward vs Backwards

- Alternative: backward list scheduling
  - Instead of starting at the leaves, work from the root to the leaves
  - Schedules instructions from end to beginning of the block
- In practice, compilers try both and pick the result that minimizes costs
  - Little extra expense since the precedence graph and other information can be reused
  - Different directions win in different cases

# Register Allocation by Graph Coloring

- How to convert the infinite sequence of temporary data references,  $t_1, t_2, \dots$  into assignments to finite number of actual registers
- Goal: Use available registers with minimum spilling
- Problem: Minimizing the number of registers is NP-complete ... it is equivalent to chromatic number – minimum colors needed to color nodes of a graph so no edge connects same color

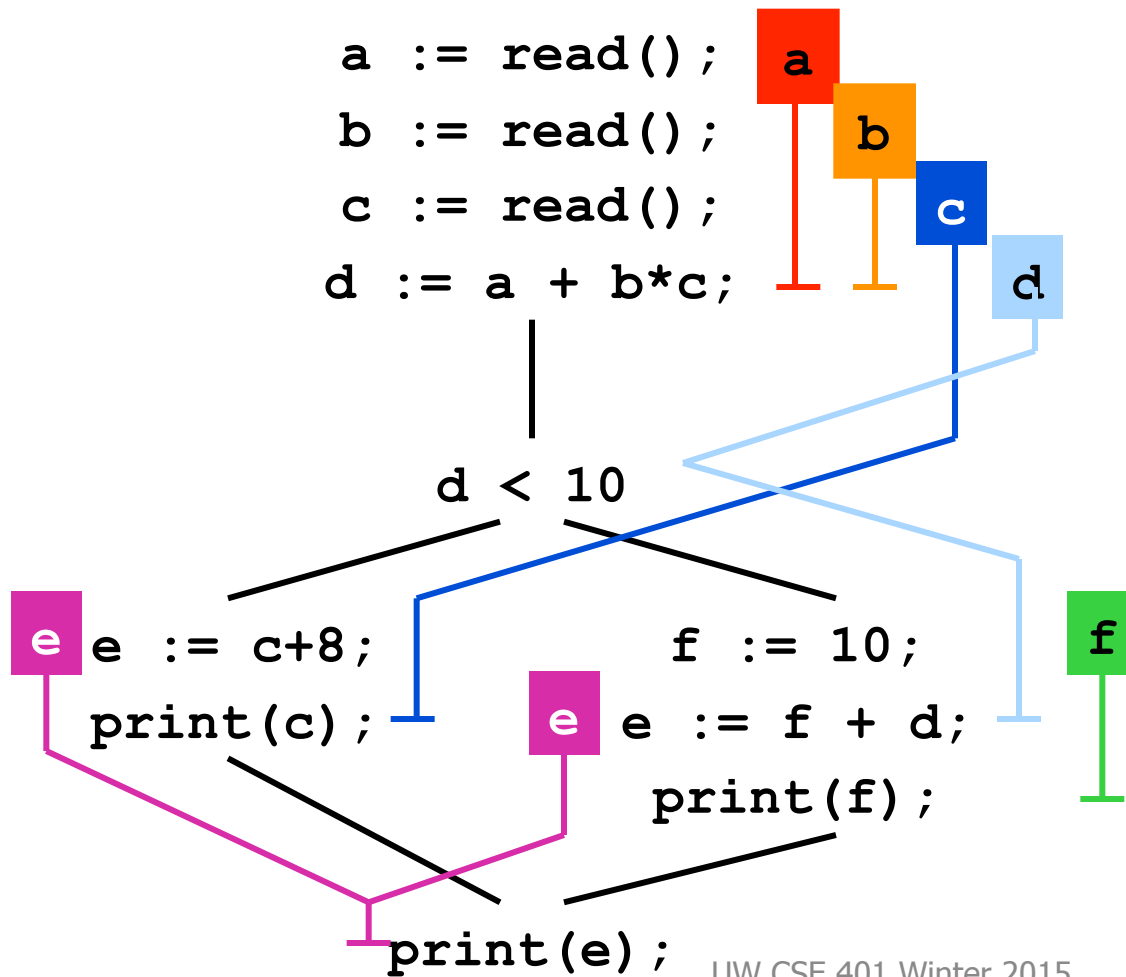
# Begin With Data Flow Graph

- procedure-wide register allocation
- only live variables require register storage

**dataflow analysis:** a variable is **live** at node N if *the value* it holds is used on some path further down the control-flow graph; otherwise it is **dead**

- two variables(values) interfere when their live ranges overlap

# Live Variable Analysis



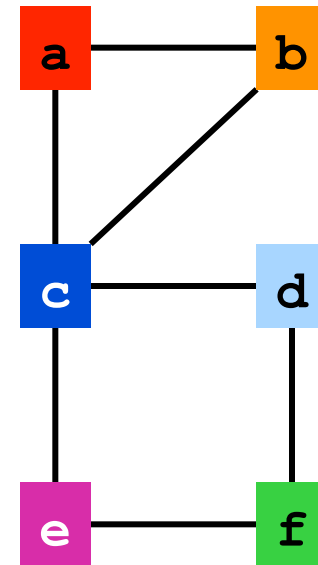
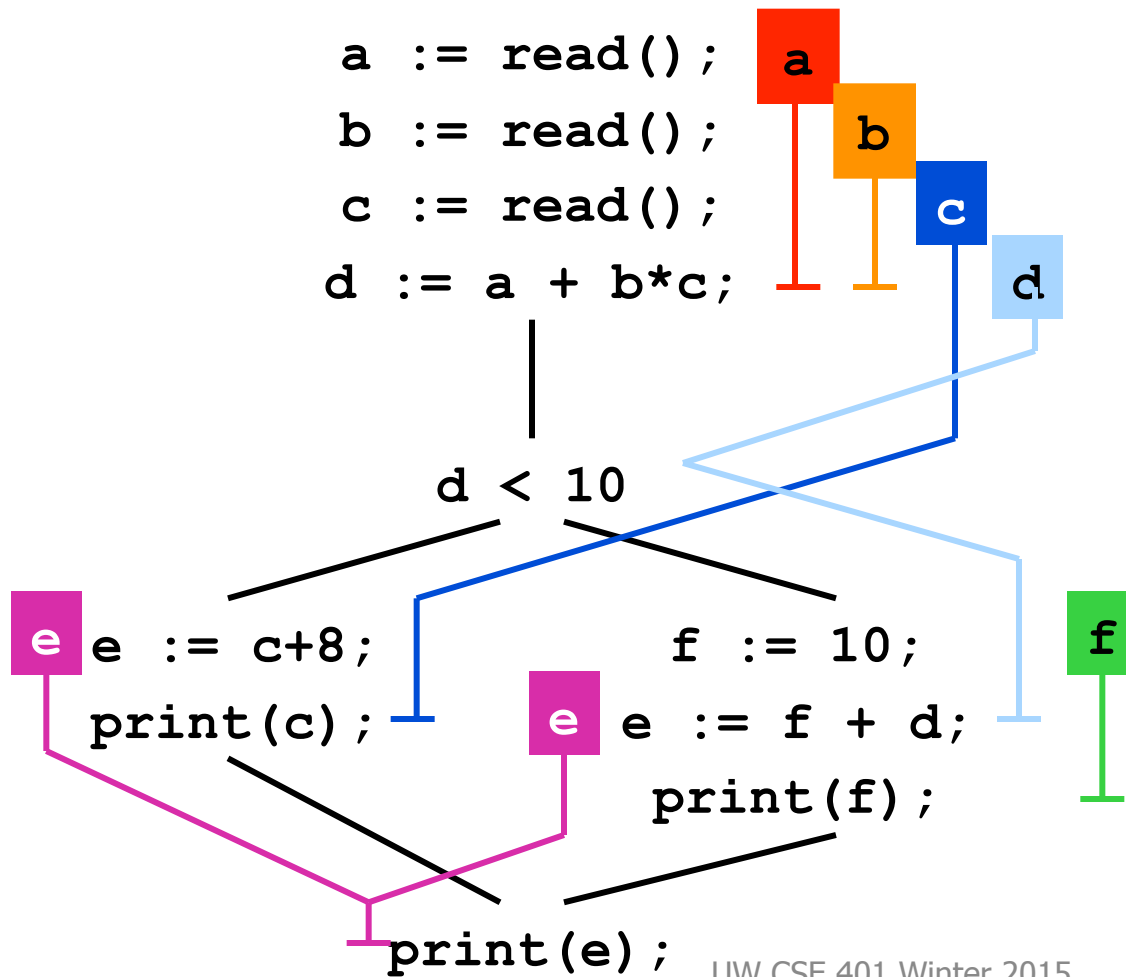
UW CSE 401 Winter 2015

```

a := read();
b := read();
c := read();
d := a + b*c;
if (d < 10 ) then
    e := c+8;
    print(c);
else
    f := 10;
    e := f + d;
    print(f);
fi
print(e);
    
```

Q-39

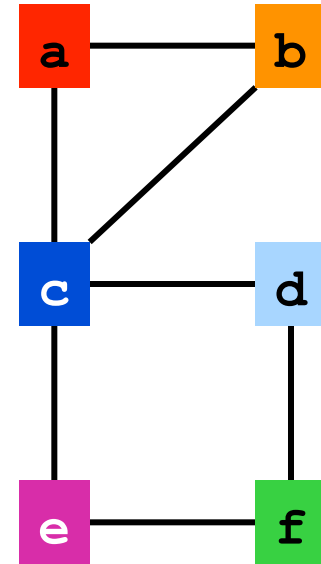
# Register Interference Graph



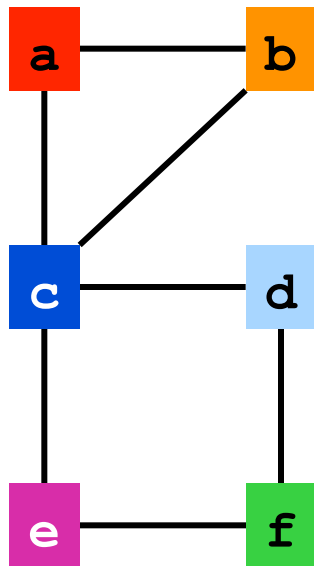


# Graph Coloring

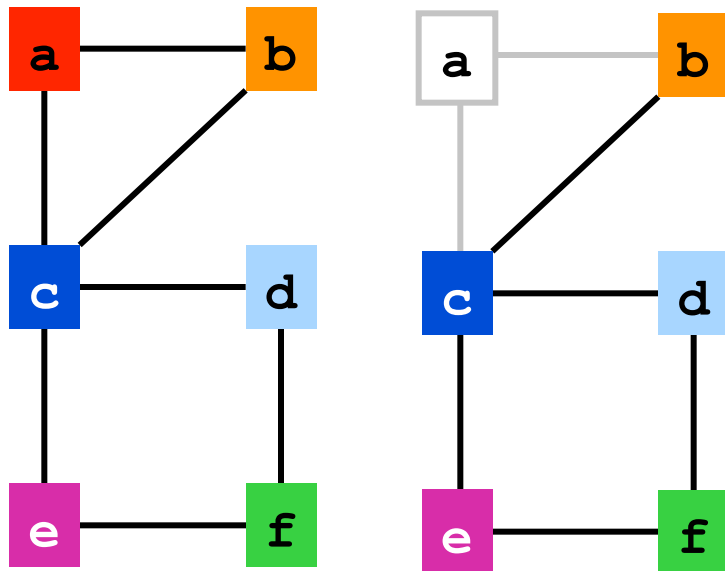
- NP complete problem
- Heuristic: color easy nodes last
  - find node N with lowest degree
  - remove N from the graph
  - color the simplified graph
  - set color of N to the first color that is not used by any of N 's neighbors
- Basics due to Chaitin (1982), refined by Briggs (1992)



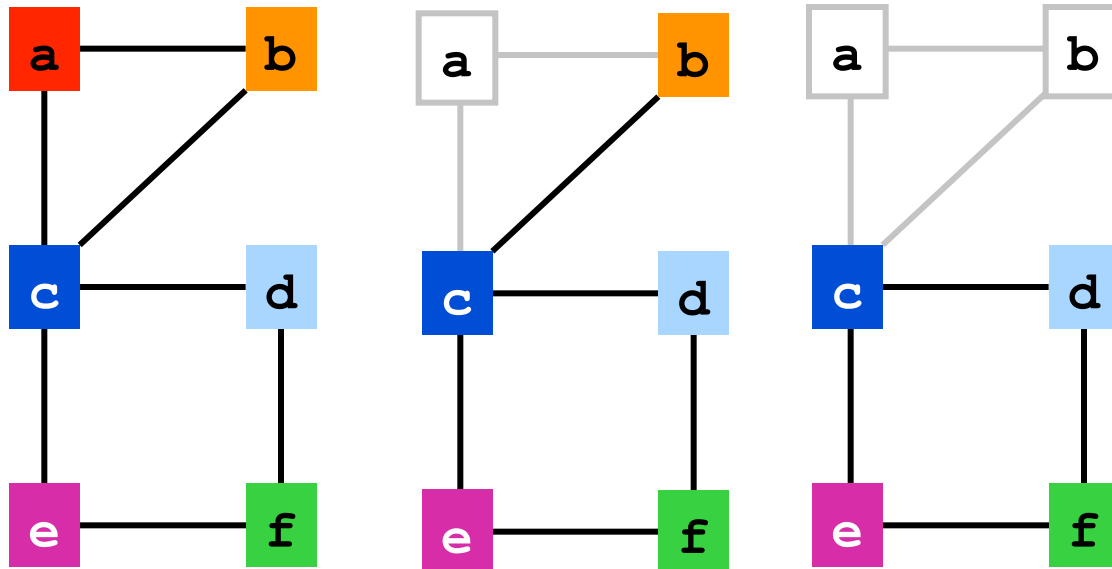
# Apply Heuristic



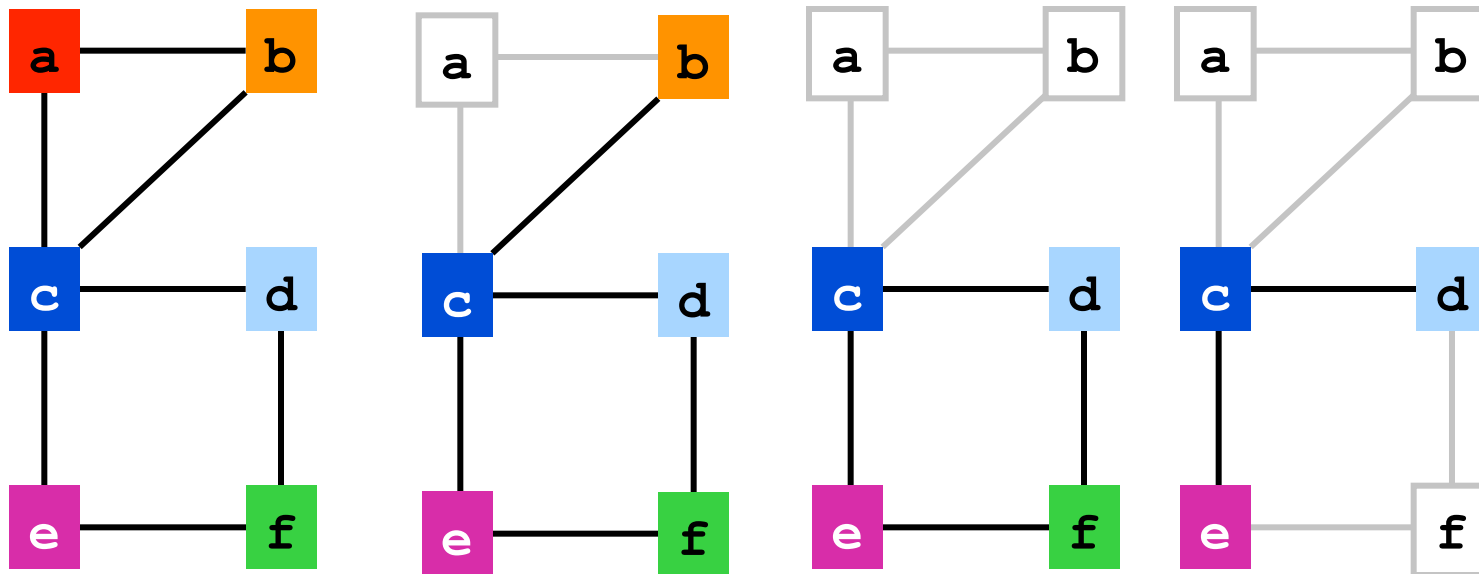
# Apply Heuristic



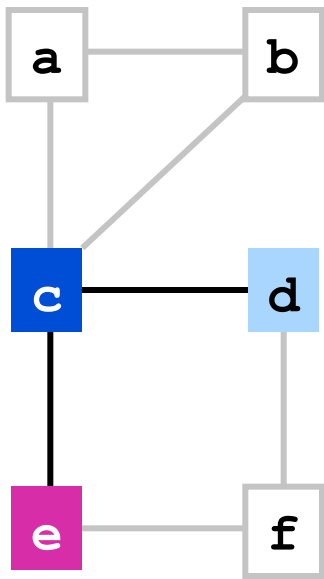
# Apply Heuristic



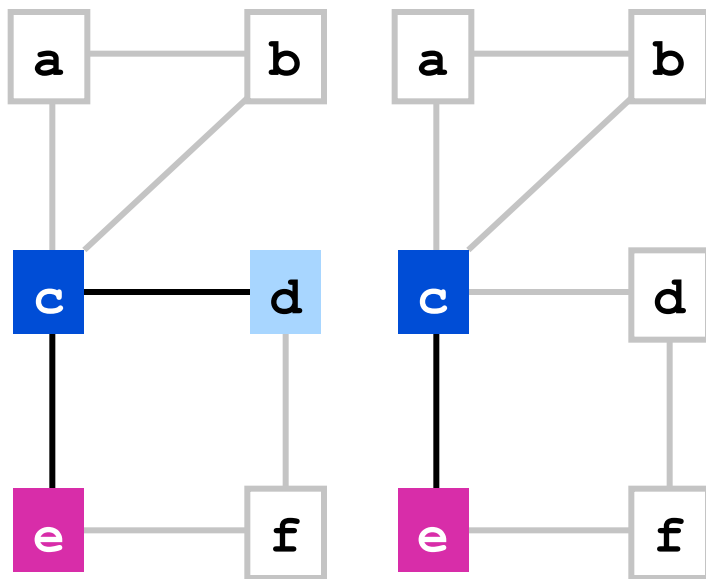
# Apply Heuristic



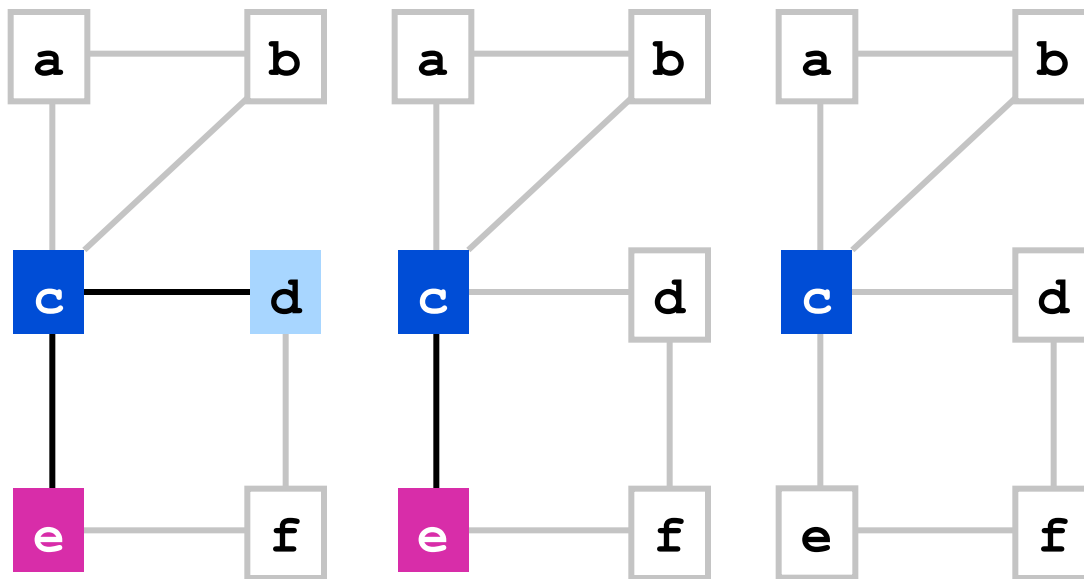
# Continued



# Continued

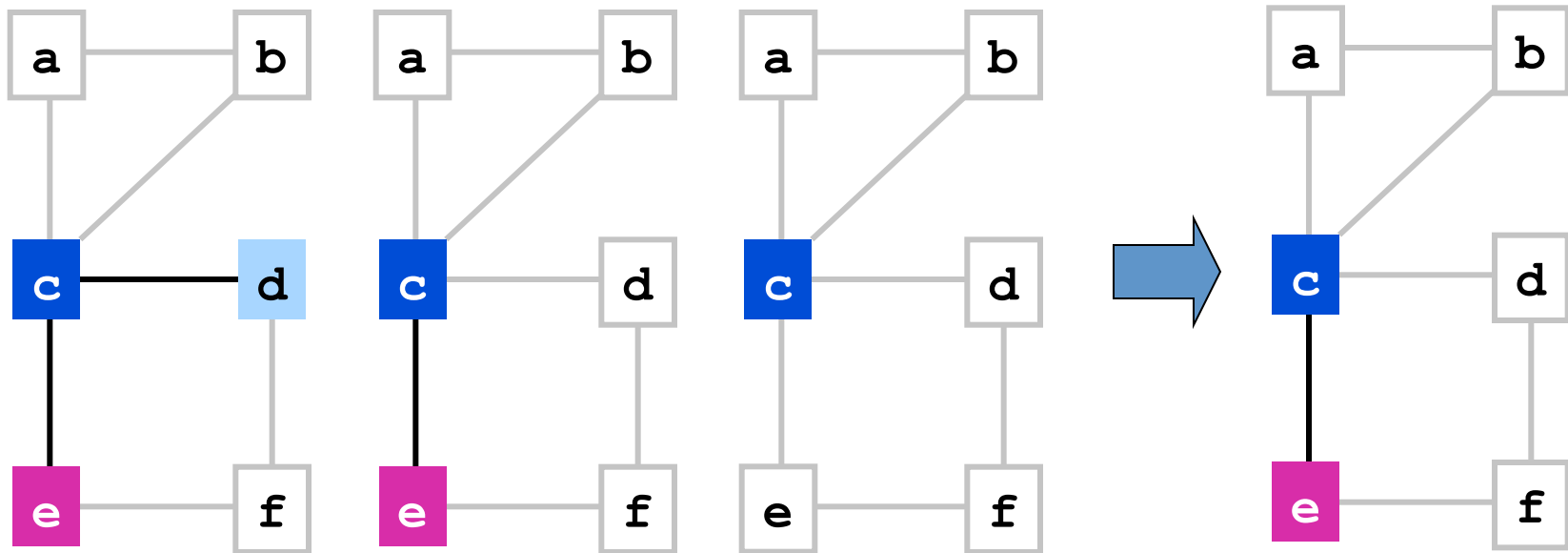


# Continued

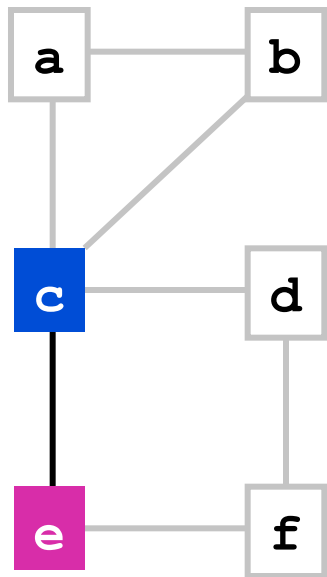




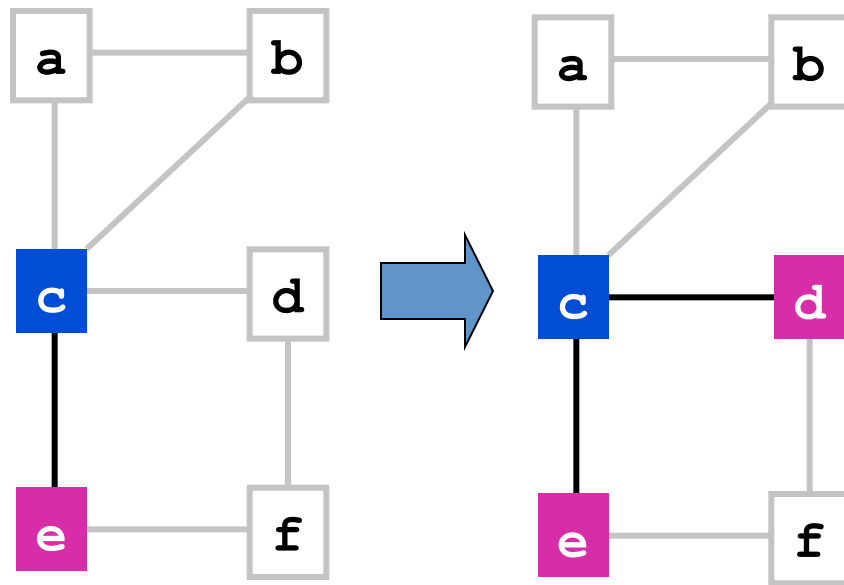
# Continued



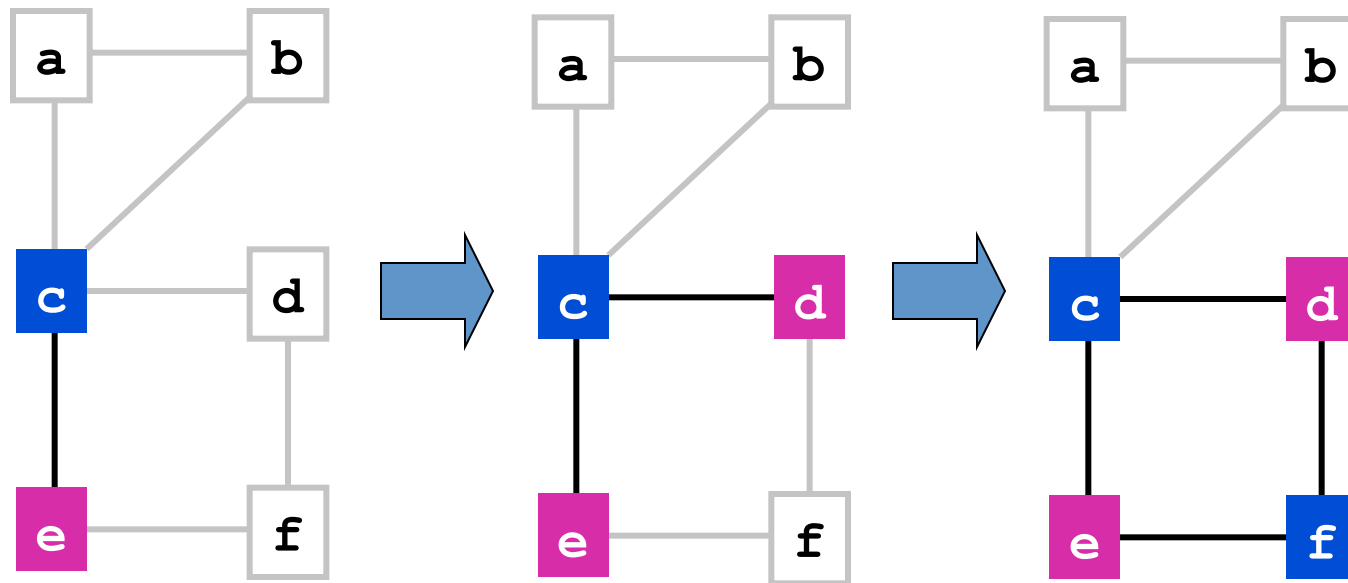
# Continued



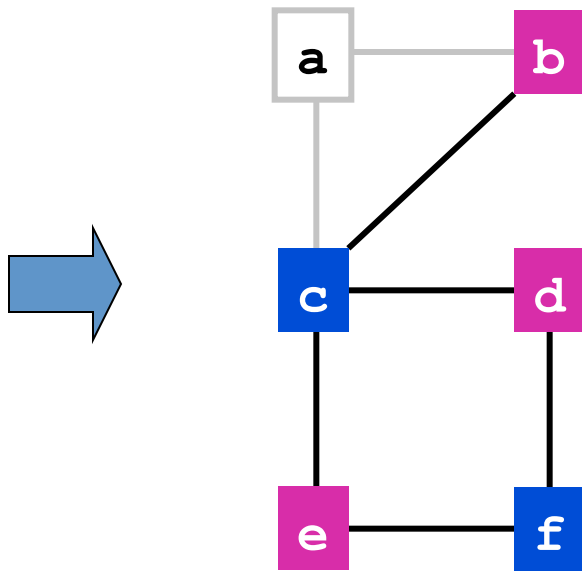
# Continued



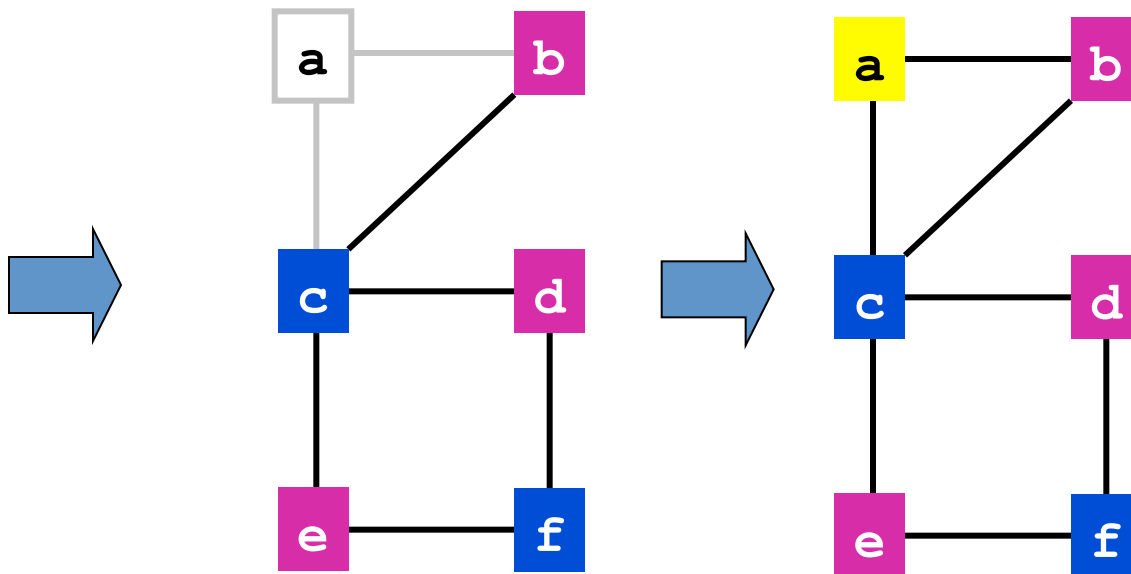
# Continued



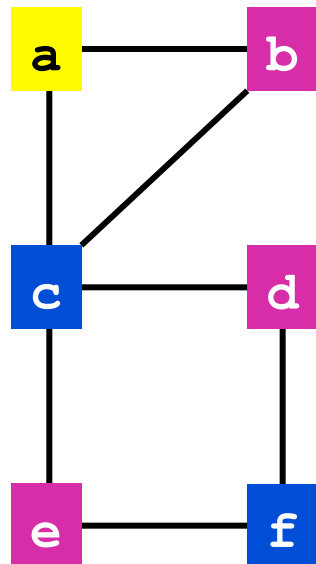
# Continued



# Continued



# Final Assignment



```
a := read();  
b := read();  
c := read();  
d := a + b*c;  
if (d < 10) then  
    e := c+8;  
    print(c);  
else  
    f := 10;  
    e := f + d;  
    print(f);  
fi  
print(e);
```

# Some Graph Coloring Issues

- May run out of registers
  - Solution: insert spill code and reallocate
- Special-purpose and dedicated registers
  - Examples: function return register, function argument registers, registers required for particular instructions
  - Solution: “pre-color” some nodes to force allocation to a particular register



# Exercise

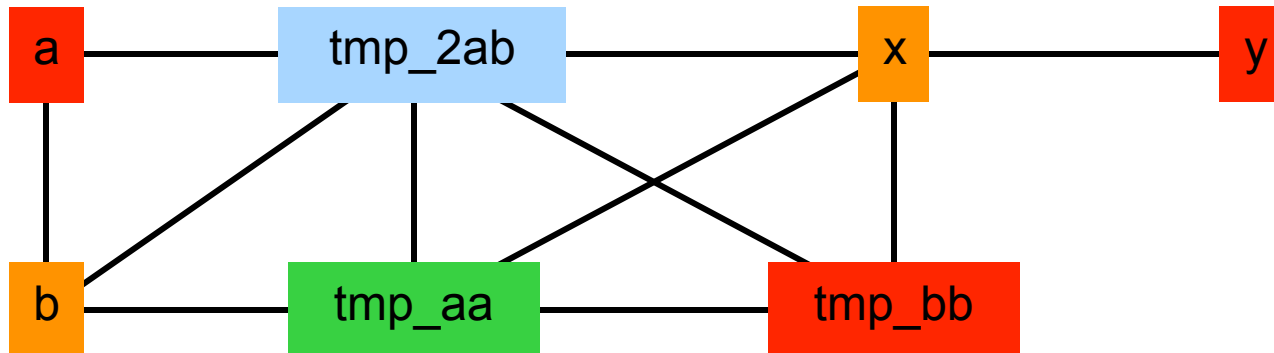
```
{  int tmp_2ab = 2*a*b;
   int tmp_aa = a*a;
   int tmp_bb = b*b;

   x := tmp_aa + tmp_2ab + tmp_bb;
   y := tmp_aa - tmp_2ab + tmp_bb;
}
```

given that  $a$  and  $b$  are live on entry and dead on exit,  
and that  $x$  and  $y$  are live on exit:

- (a) construct the register interference graph
- (b) color the graph; how many registers are needed?

# 4 Registers Needed



# Live Ranges

- Real graph-coloring register allocators don't allocate registers – they allocate *live ranges*
- A *live range* is a set of definitions and uses that flow together
  - Every definition can reach every use
  - Every use that a definition can reach is in the same live range
- Idea: disjoint uses of a variable in different parts of the program don't actually interfere so they should be in separate live ranges

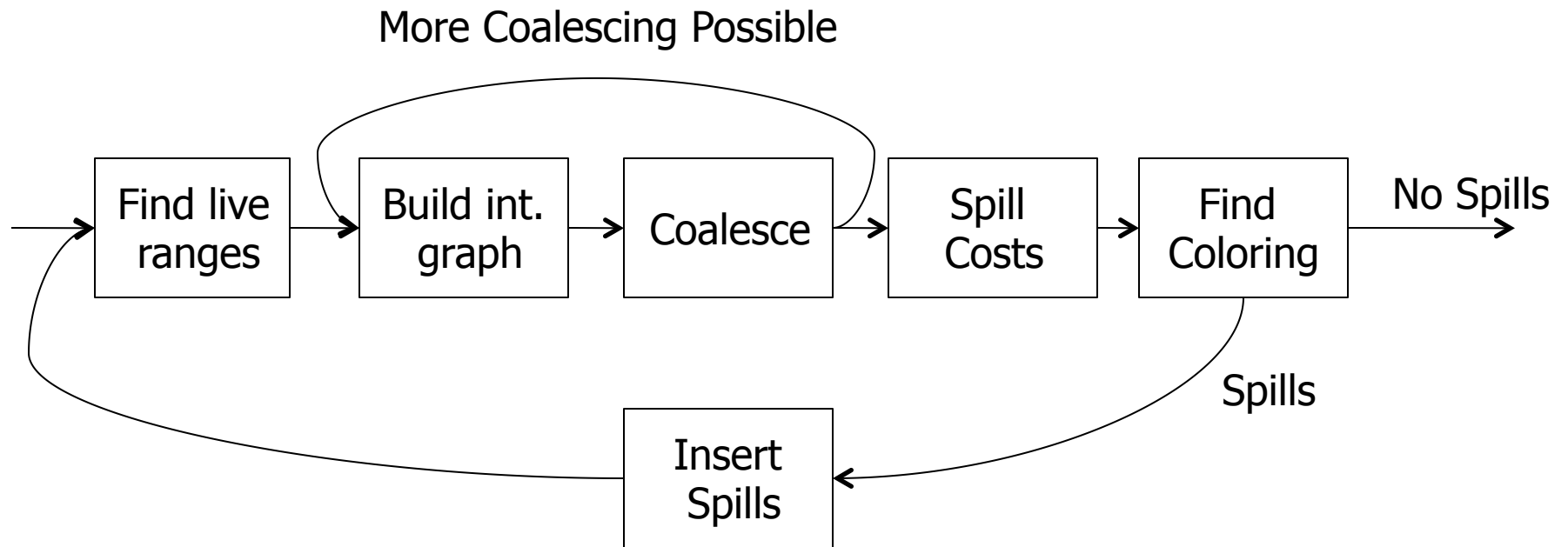
# Coalescing Live Ranges

- Idea: if two live ranges are connected by a copy operation ( $\text{MOV } r_i \rightarrow r_j$ ) but do not otherwise interfere, then the live ranges can be coalesced (combined)
  - Rewrite all references to  $r_j$  to use  $r_i$
  - Remove the copy instruction
- Then need to fix up interference graph

# Advantages?

- Makes the code smaller, faster (no copy operation)
- Shrinks set of live ranges
- Reduces the degree of any live range that interfered with both live ranges  $r_i, r_j$
- But: coalescing two live ranges can prevent coalescing of others, so ordering matters
  - Best: Coalesce most frequently executed ranges first (e.g., inner loops)
- Can have a substantial payoff – do it!

# Overall Structure



- Then you may want to iterate with additional instruction selection and scheduling passes, particularly on a complex machine where operations can have both memory or register operands (e.g., x86)

And that's it!

Modulo all the picky details, that is...