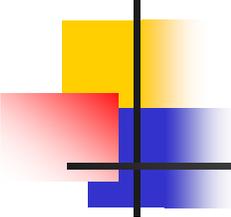


CSE 401 – Compilers

Compiler Backend Survey

Hal Perkins

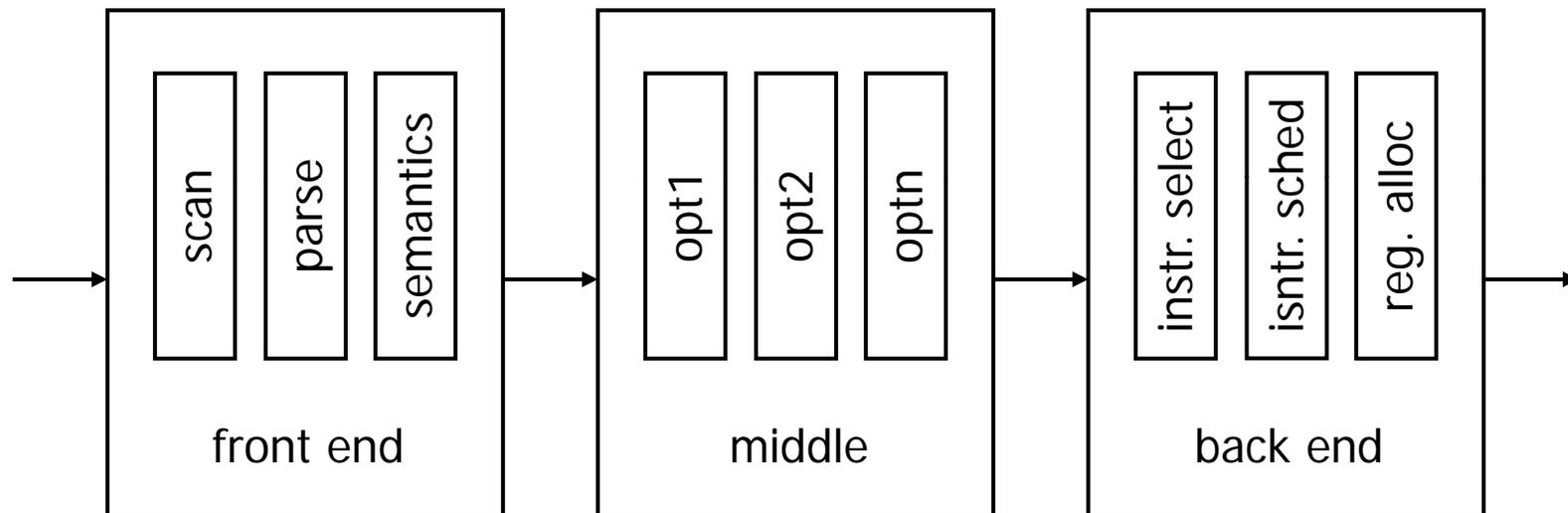
Winter 2009



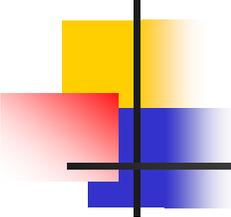
Agenda

- A survey of the major pieces of the back end of the compiler
 - 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

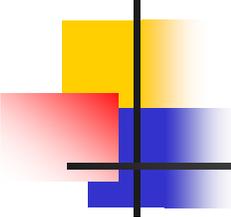


infrastructure – symbol tables, trees, graphs, etc



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



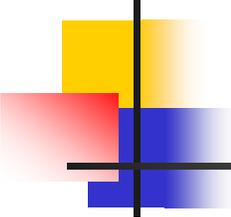
IR for Code Generation

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

Overview

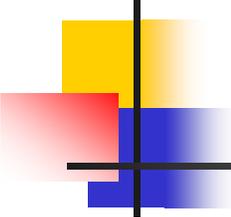
Instruction Selection

- 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.)



A Simple Low-Level IR (1)

- What's important for us is to get a feel for the level of detail involved; the specifics don't matter as much
- 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 on left side of assignment
 - $\text{CALL}(f, args)$ – application of 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

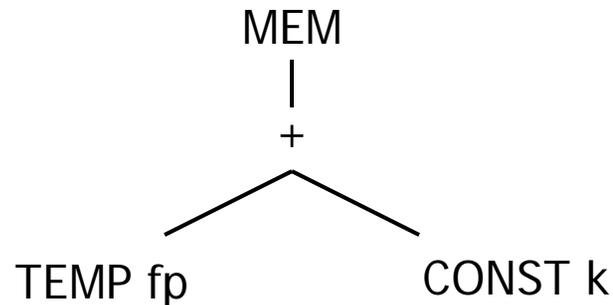
Low-Level IR Example (1)

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

- Linear

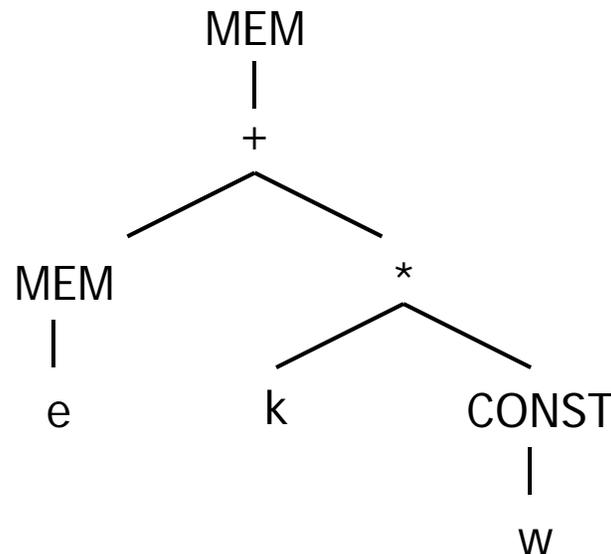
MEM(BINOP(PLUS, TEMP fp , CONST k))

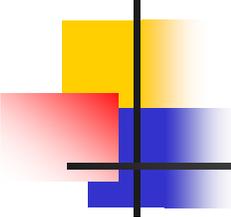
- Tree



Low-Level IR Example (2)

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

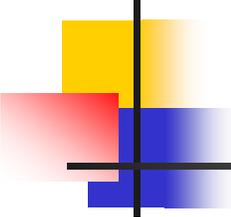




Instruction Selection Issues

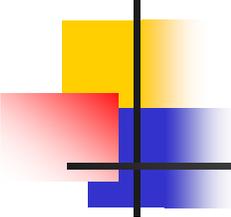
- Given the low-level IR, there are many possible code sequences that implement it correctly
 - e.g. to set `eax` to 0 on x86

```
mov  eax,0      xor  eax,eax
sub  eax,eax    imul eax,0
```
 - Many machine instructions do several things at once – e.g., register arithmetic and effective address calculation



Implementation

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



An Example Target Machine (1)

- Arithmetic Instructions

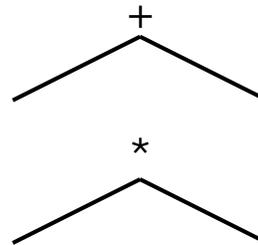
- (unnamed) r_i

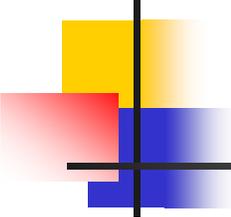
- $\text{ADD } r_i \leftarrow r_j + r_k$

- $\text{MUL } r_i \leftarrow r_j * r_k$

- SUB and DIV are similar

TEMP

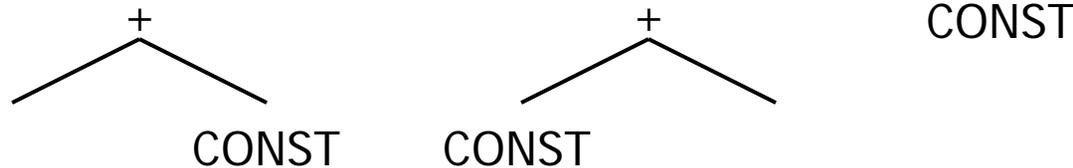




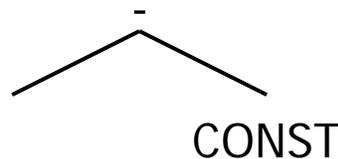
An Example Target Machine (2)

- Immediate Instructions

- ADDI $r_i \leftarrow r_j + c$



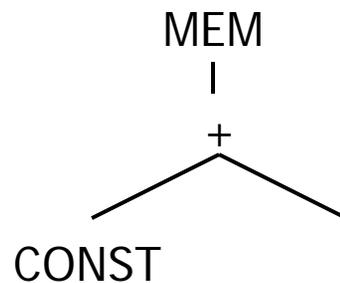
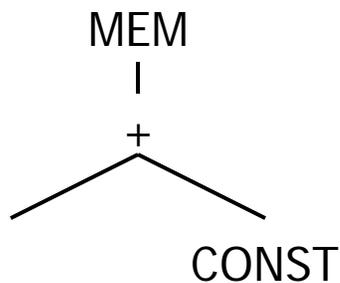
- SUBI $r_i \leftarrow r_j - c$



An Example Target Machine (3)

- Load

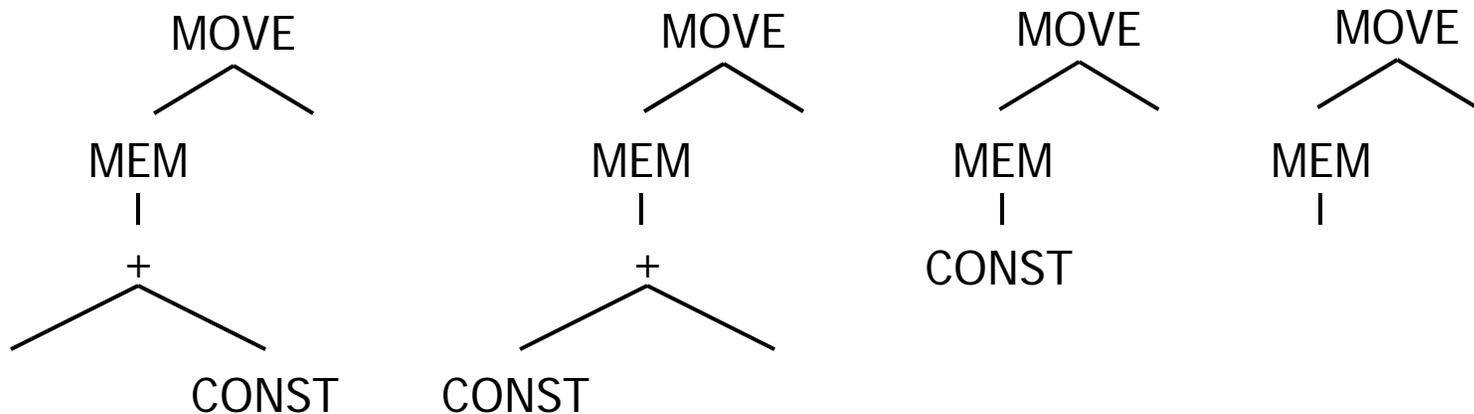
- `LOAD ri <- M[rj + c]`

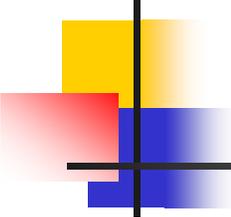


An Example Target Machine (4)

- Store

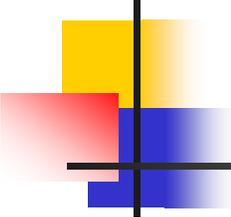
- STORE $M[rj + c] \leftarrow r_i$





Tree Pattern Matching (1)

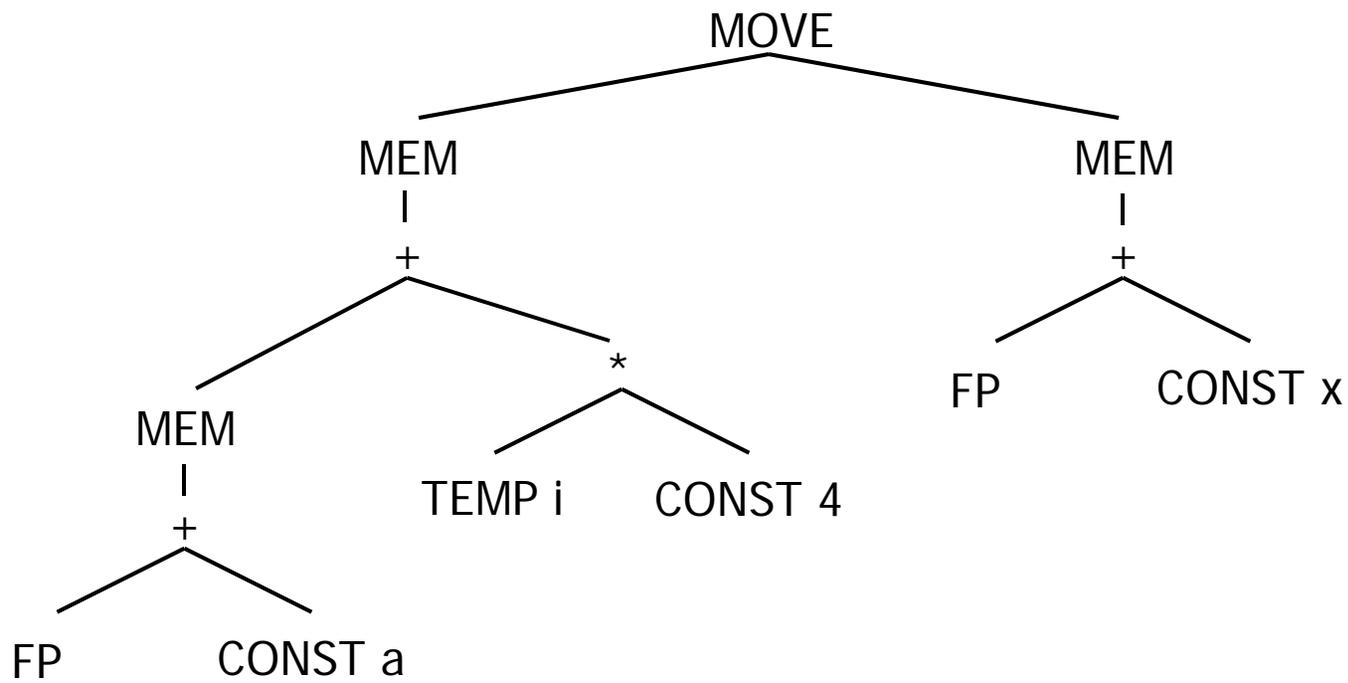
- Goal: Tile the low-level tree with operation (instruction) trees
- A *tiling* is a collection of $\langle \text{node}, \text{op} \rangle$ pairs
 - node is a node in the tree
 - op is an operation tree
 - $\langle \text{node}, \text{op} \rangle$ means that op could implement the subtree at node

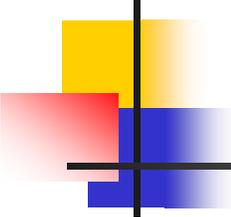


Tree Pattern Matching (2)

- 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 in 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)

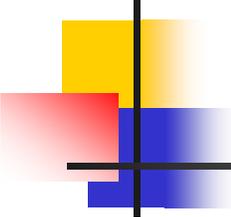
Example – Tree for $a[i] := x$





Generating Tilings

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



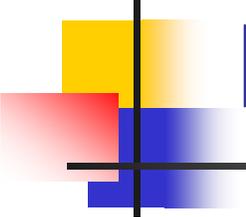
Generating Code

- Given a tiled tree, to generate code
 - Postorder treewalk; node-dependant order for children
 - Emit code sequences corresponding to tiles in order
 - Connect tiles by using same register name to tie boundaries together

Overview

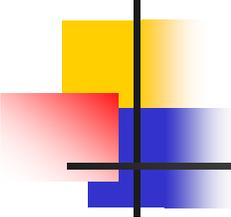
Instruction Scheduling

- Reorder instructions to minimize execution time
 - hide latencies – processor function units, memory/cache stalls
 - Originally invented for supercomputers (60s)
 - Now important everywhere
 - Even non-RISC machines, i.e., x86
 - Even if processor reorders on the fly
- Assume fixed program at this point



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

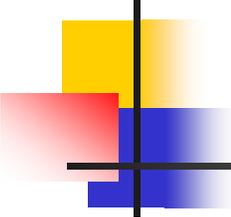
```
1 LOAD   r1 <- w
4 ADD    r1 <- r1,r1
5 LOAD   r2 <- x
8 MULT   r1 <- r1,r2
9 LOAD   r2 <- y
12 MULT  r1 <- r1,r2
13 LOAD  r2 <- z
16 MULT  r1 <- r1,r2
18 STORE w <- r1
21 r1 free
```

2 registers, 20 cycles

- Loads early

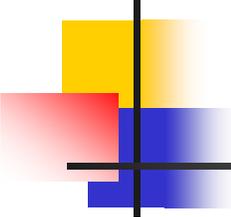
```
1 LOAD   r1 <- w
2 LOAD   r2 <- x
3 LOAD   r3 <- y
4 ADD    r1 <- r1,r1
5 MULT   r1 <- r1,r2
6 LOAD   r2 <- z
7 MULT   r1 <- r1,r3
9 MULT   r1 <- r1,r2
11 STORE w <- r1
14 r1 is free
```

3 registers, 13 cycles



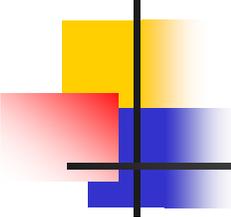
Algorithm Overview

- 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
 - At each cycle
 - Chose a ready operation and schedule it
 - Update ready queue
- Rename registers to avoid false dependencies and conflicts



Precedence Graph

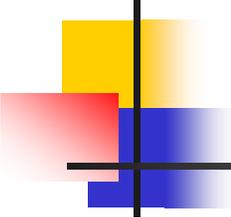
- Nodes n are operations
- Attributes of each node
 - type – kind of operation
 - delay – latency
- If node n_2 uses the result of node n_1 , there is an edge $e = (n_1, n_2)$ in the graph



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
```



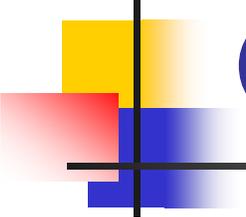
Forward vs Backwards

- Backward list scheduling
 - 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

Overview

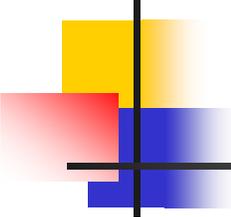
Register Allocation

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



Register Allocation by Graph Coloring

- How to convert the infinite sequence of temporary data references, t_1, t_2, \dots into finite assignment register numbers $\$8, \$9, \dots, \$25$
- Goal: Use available registers with minimum spilling
- Problem: Minimizing the number of registers is NP-complete ... it is equivalent to chromatic number--minimum colors to color nodes of graph so no edge connects same color



Begin With Data Flow Graph

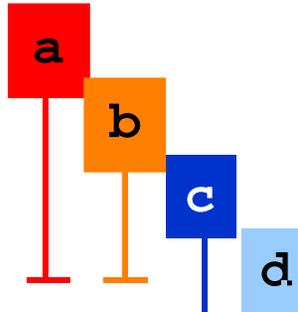
- procedure-wide register allocation
- only **live** variables require register storage

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**

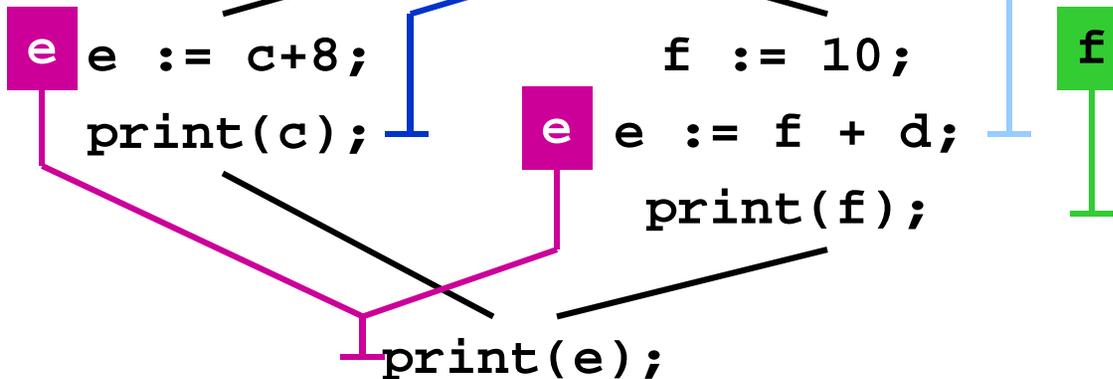
- two variables(values) interfere when their live ranges overlap

Live Variable Analysis

```
a := read();  
b := read();  
c := read();  
d := a + b*c;
```



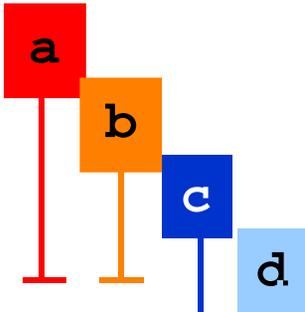
d < 10



```
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);
```

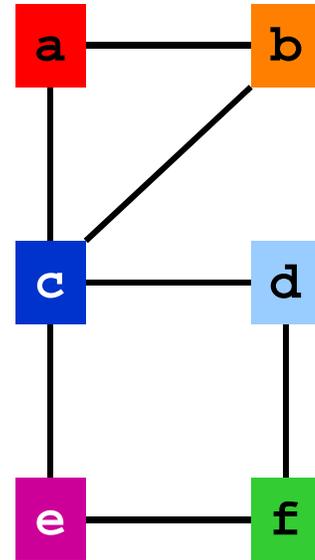
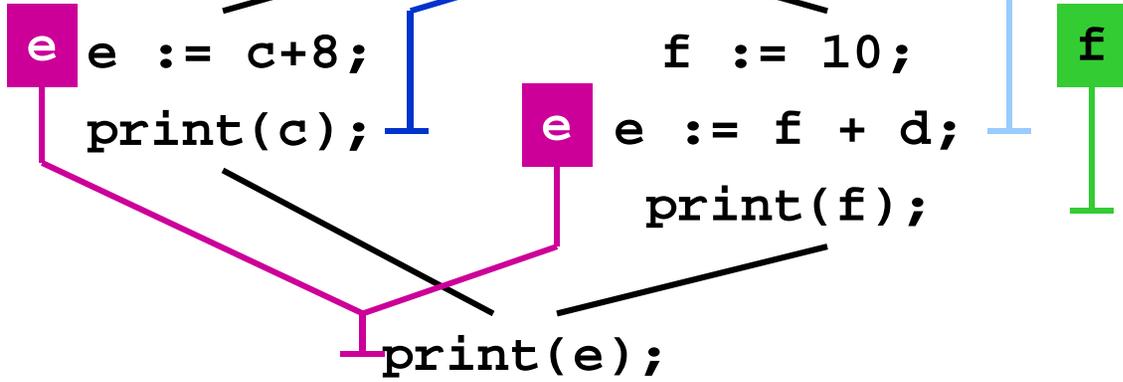
Register Interference Graph

```
a := read();  
b := read();  
c := read();  
d := a + b*c;
```



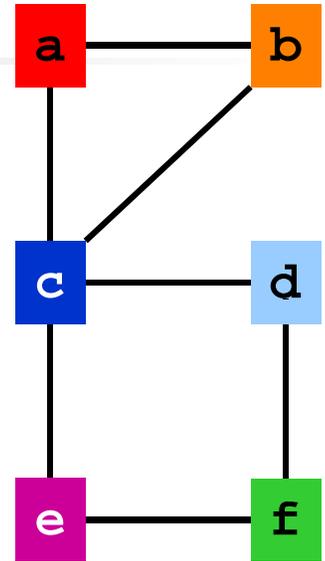
d < 10

```
e := c+8;  
print(c);  
f := 10;  
e := f + d;  
print(f);  
print(e);
```

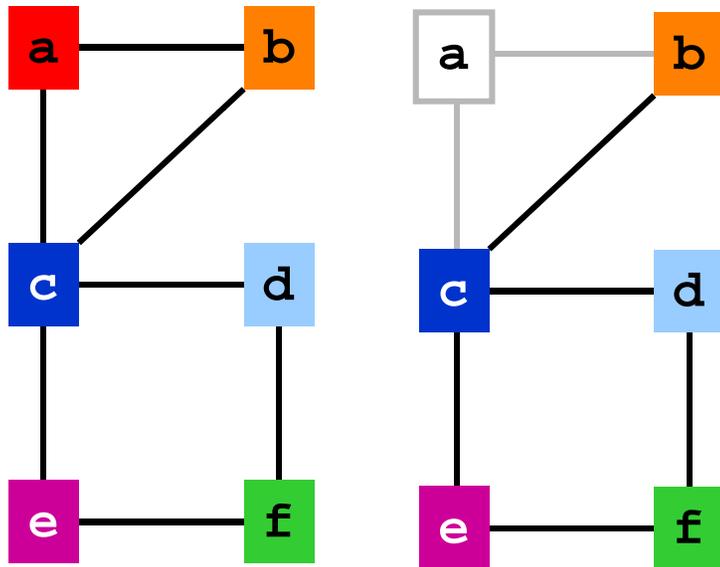


Graph Coloring

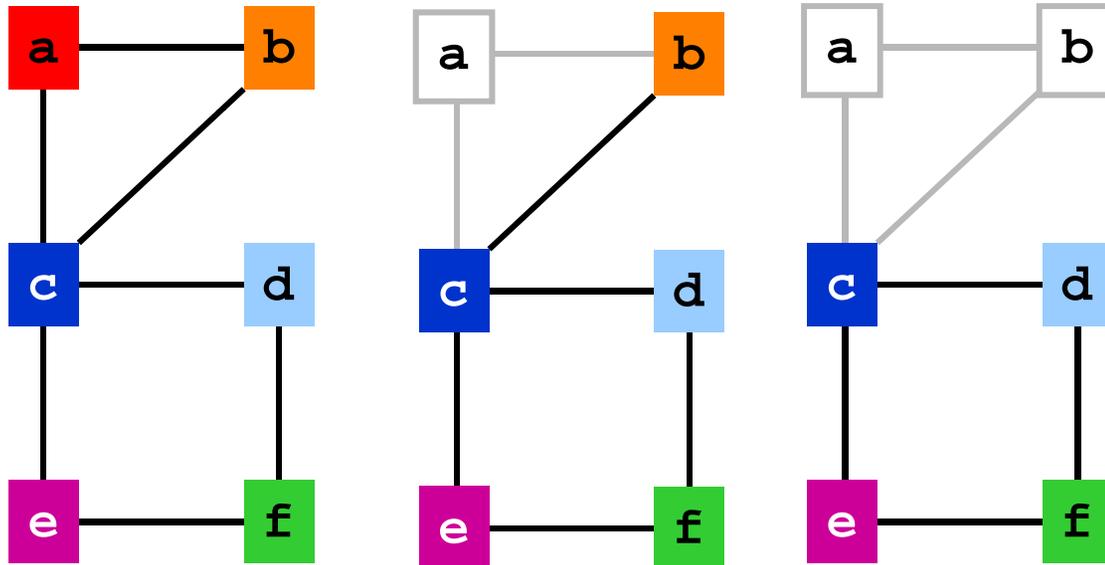
- NP complete problem
- Heuristic: color easy nodes last
 - find node N with lowest degree
 - remove N from the graph
 - color the simplified graph
 - set color of N to the first color that is not used by any of N 's neighbors
- Basics due to Chaitin (1982), refined by Briggs (1992)



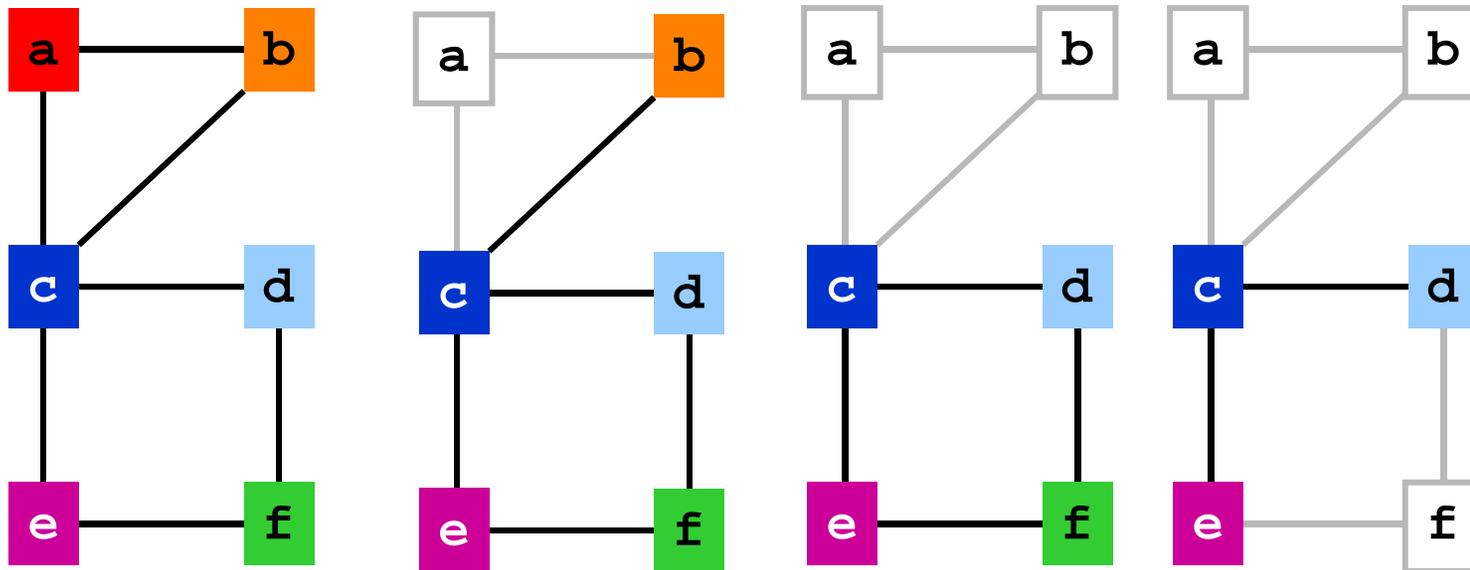
Apply Heuristic



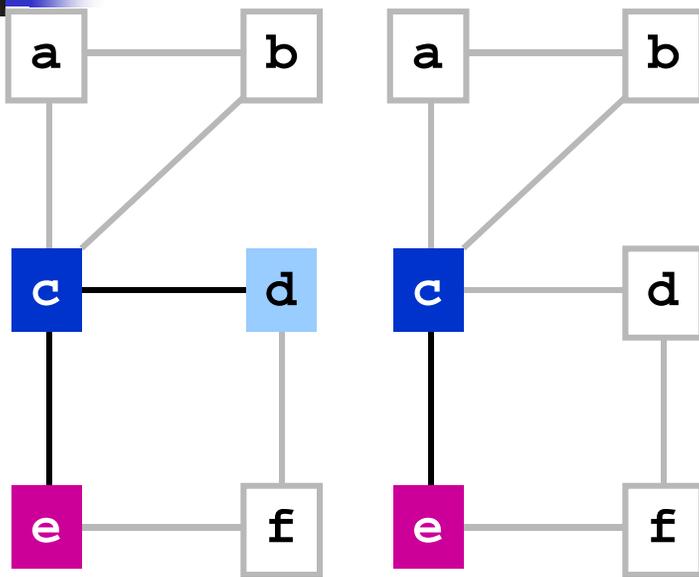
Apply Heuristic



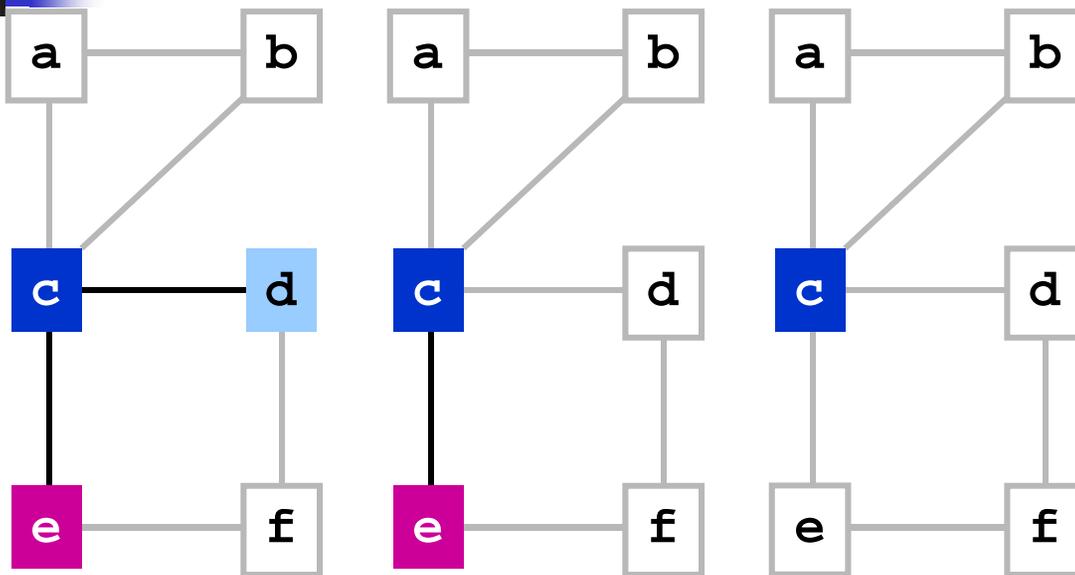
Apply Heuristic



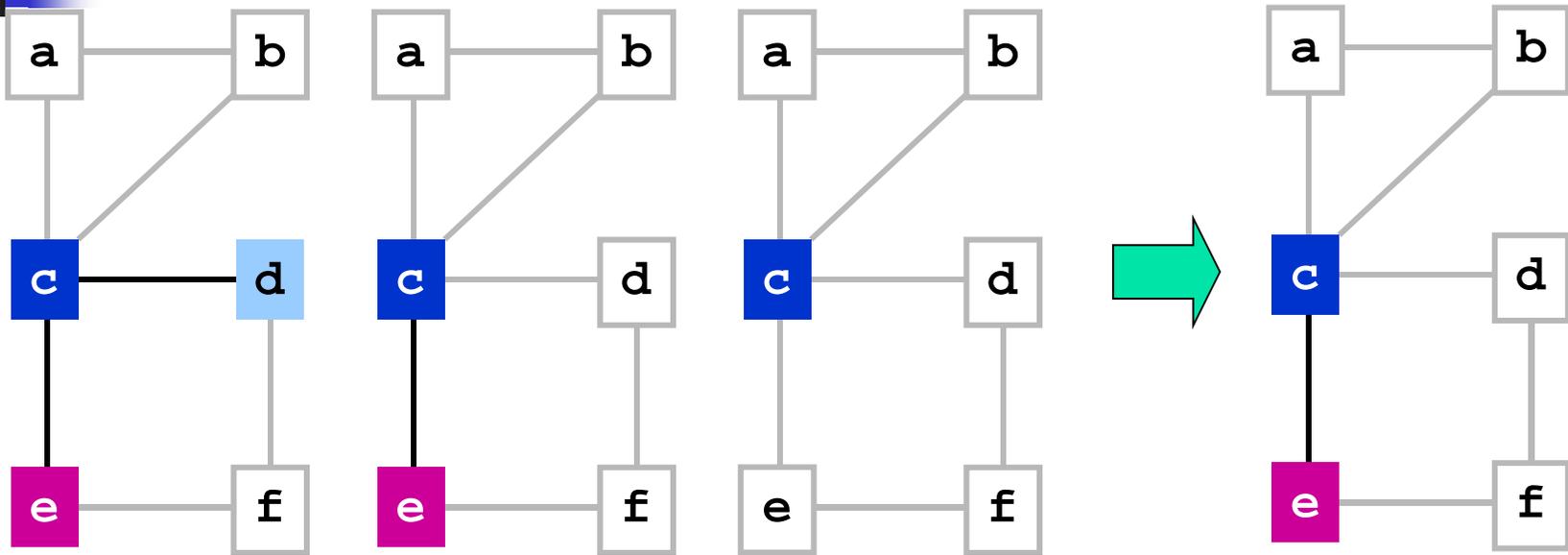
Continued



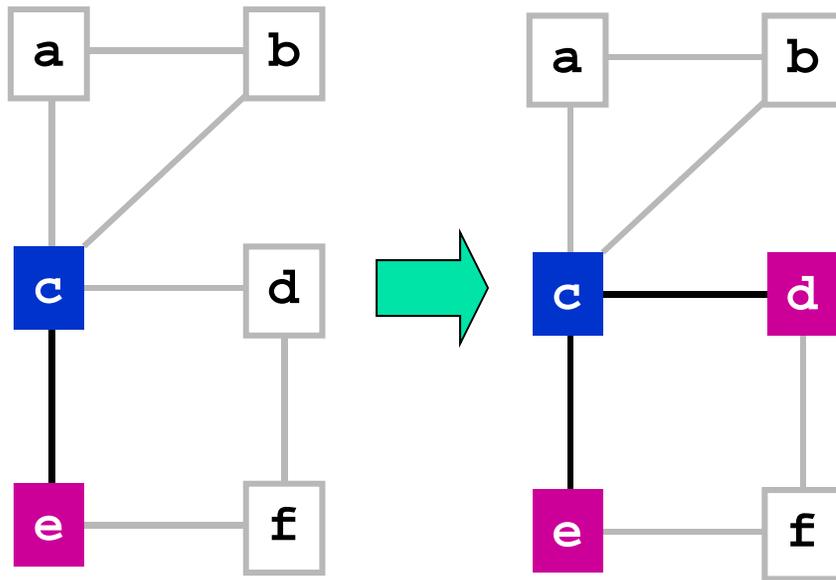
Continued



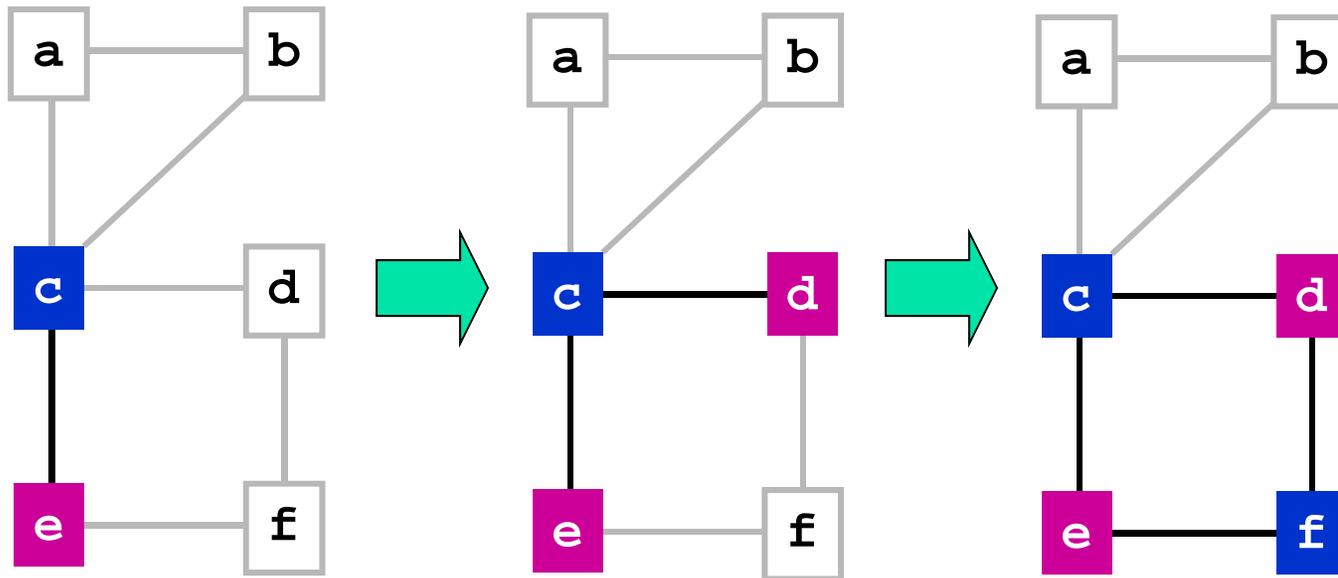
Continued



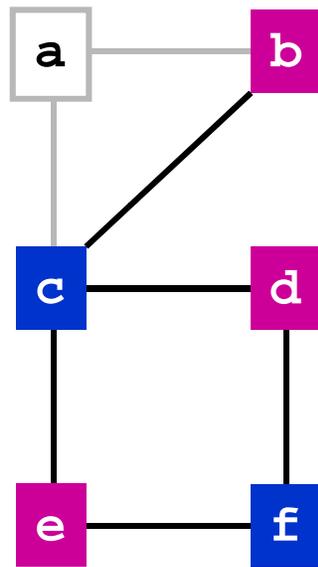
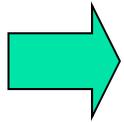
Continued



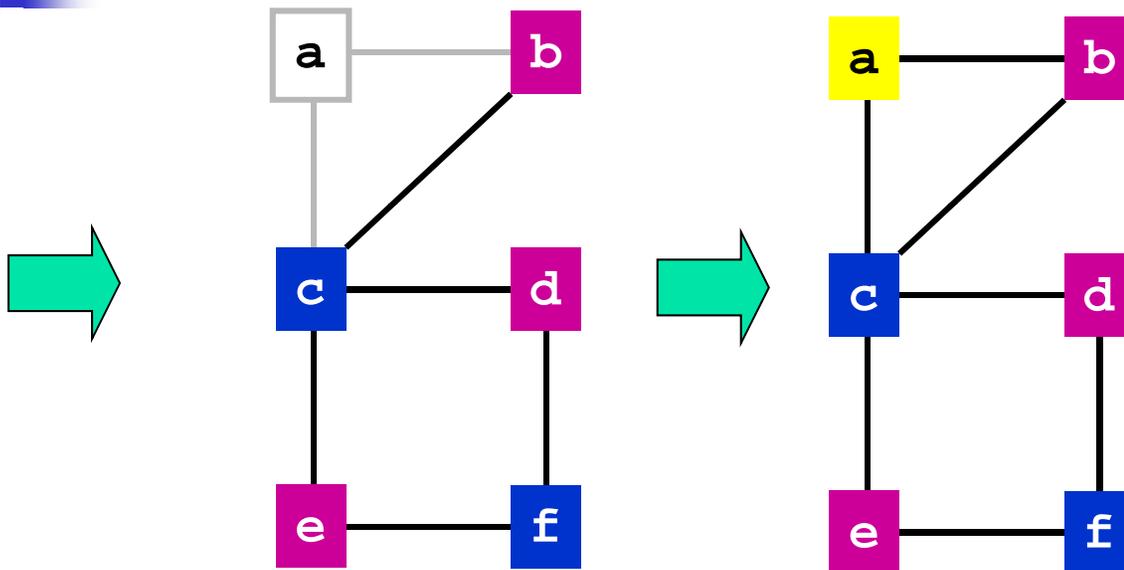
Continued



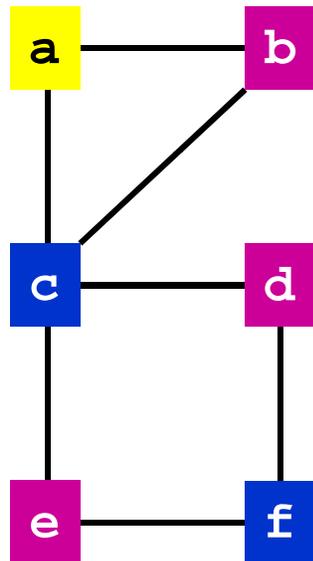
Continued



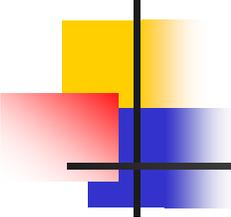
Continued



Final Assignment

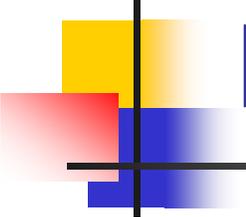


```
a := read();
b := read();
c := read();
d := a + b*c;
if (d < 10) then
    e := c+8;
    print(c);
else
    f := 10;
    e := f + d;
    print(f);
fi
print(e);
```



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

