CSE P 501 – Compilers

Compiler Backend Survey

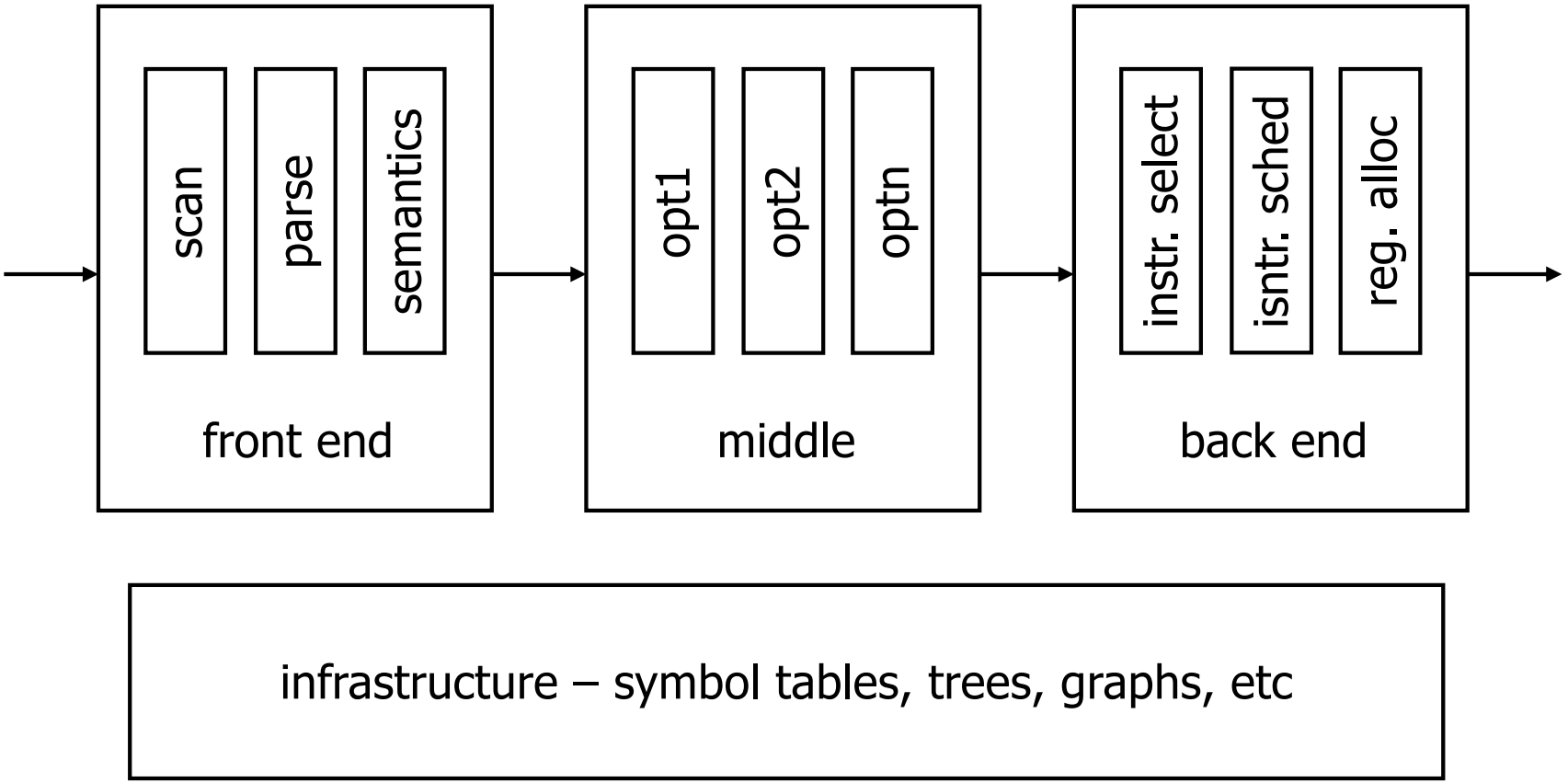
Hal Perkins

Autumn 2023

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: ~ O(n log n)
  - Instruction selection: fast or NP-Complete
  - Instruction scheduling: NP-Complete
  - Register allocation: NP-Complete

IR for Code Generation

- Assume a (very) low-level IR
  - 3 address, register-register instructions plus load/store

    r1 <- r2 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 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!) code on classic RISC architectures (basically 3-address 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 instruction scheduling pass afterwards if spill code inserted

Conventional Wisdom

- We typically lose little by solving these independently
  - But not always, of course (iterating phases on x86-64 can help because of limited registers and use of memory operands)
- Instruction selection
  - Use some form of pattern matching
  - ∞ 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

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

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:
  - CONST(i) – integer constant i
  - TEMP(t) – temporary t (i.e., register)
  - BINOP(op,e1,e2) – application of op to e1,e2
  - MEM(e) – contents of memory at address e
    - Means value when used in an expression
    - Means address when used as target of assignment
  - CALL(f,args) – apply function f to argument list args

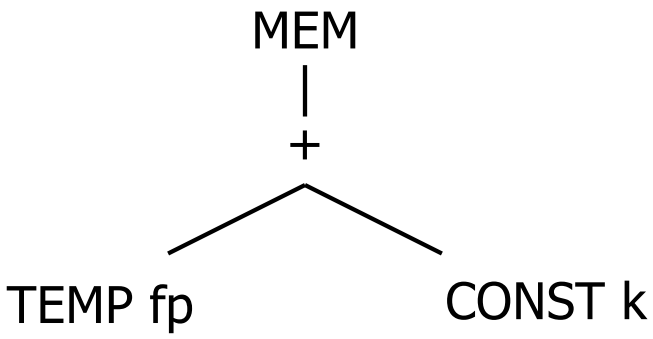
Simple Low-Level IR (2)

- 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 compex (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)

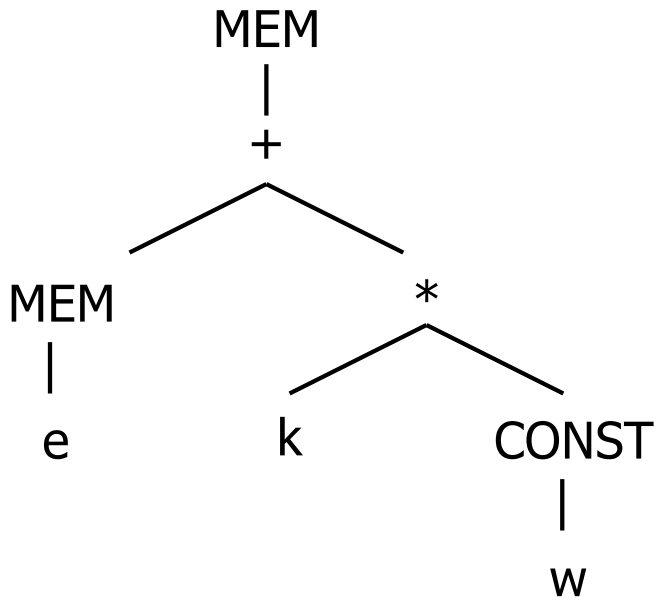
- Access a local variable at a known offset k from the frame pointer fp
  - Linear

    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. set %rax to 0 on x86-64 (did we miss any?)

```
movq  $0,%rax          salq   64,%rax
subq  %rax,%rax        shrq   64,%rax
xorq  %rax,%rax        imulq  $0,%rax
```

  - Many machine instructions do several things at once – e.g., register arithmetic and effective address calculation, e.g.,

```
movq  offset(%rbase, %rindex, scale), %rdest
```

Instruction Selection Criteria

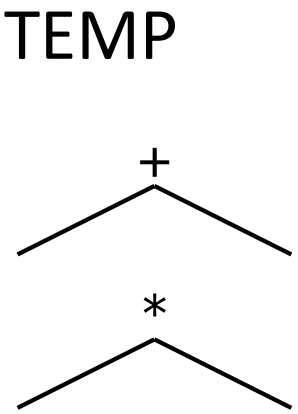
- Several possibilities
  - Fastest
  - Smallest
  - Minimize power consumption (ex: don't use a function unit if leaving it powered-down is a win)
  - Reduce memory traffic
  - etc. etc.
- Sometimes not obvious
  - e.g., if one of the function units in the processor is idle and we can select an instruction 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

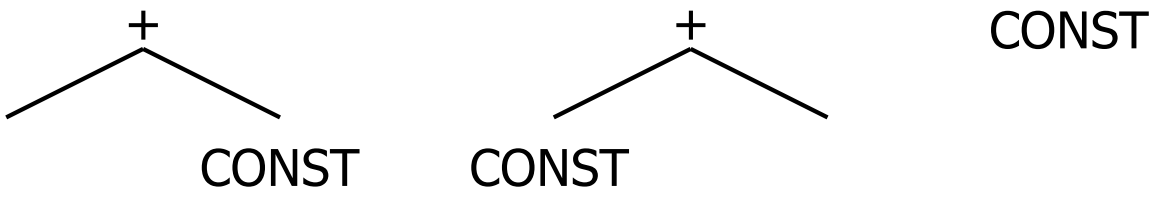
An Example Target Machine (1)

- Arithmetic Instructions
  - (unnamed) ri
  - ADD ri <- rj + rk
  - MUL ri <- rj * rk
  - SUB and DIV are similar
  - For some examples, we'll assume there is at least one register (R0) hardwired to be 0 always

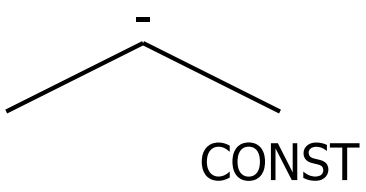


An Example Target Machine (2)

- Immediate Instructions
  - ADDI ri <- rj + c

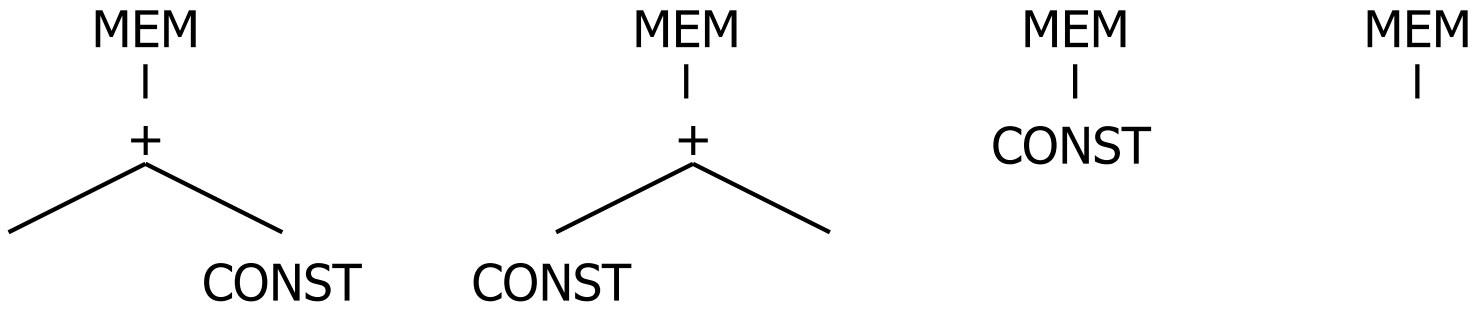


  - SUBI ri <- rj - c



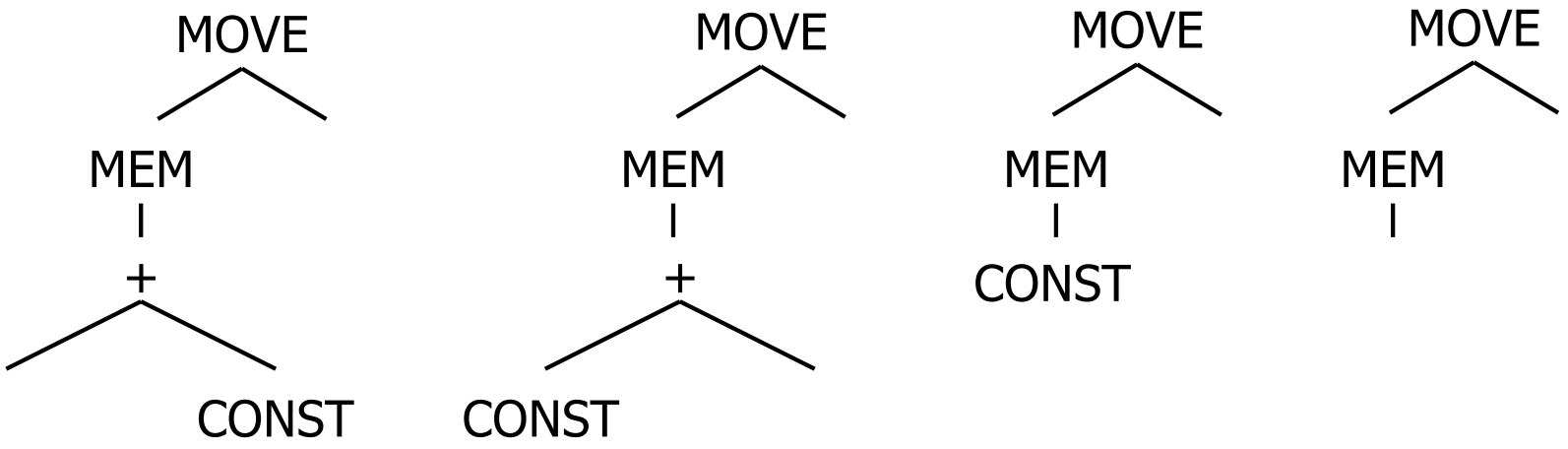
An Example Target Machine (3)

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



An Example Target Machine (4)

- Store
  - STORE M[rj + c] <- ri



Tree Pattern Matching (1)

- Goal: Tile the low-level IR tree with operation (instruction) trees
- A *tiling* is a collection of <node,op> pairs
  - node is a node in the tree
  - op is an operation tree
  - <node,op> 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 <node,op> 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

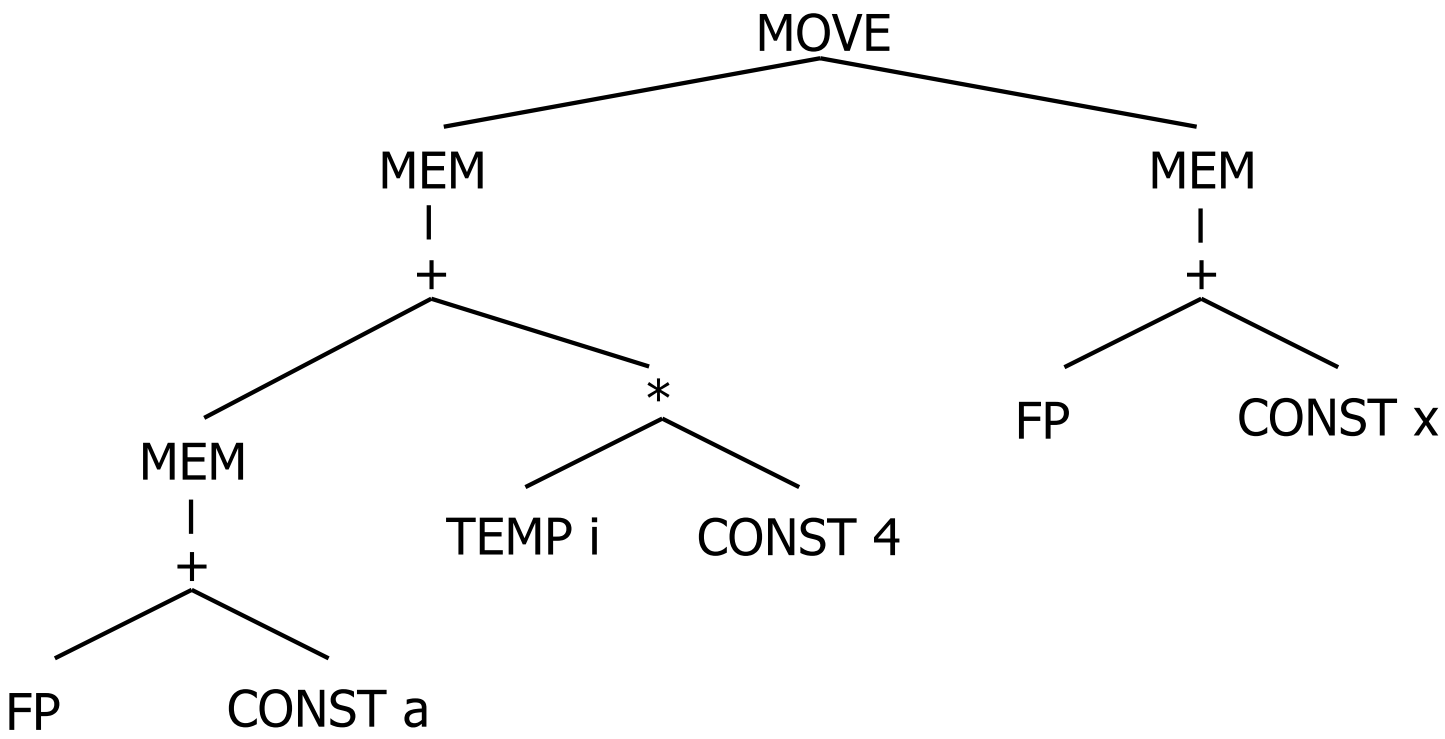
Two common algorithms

- 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

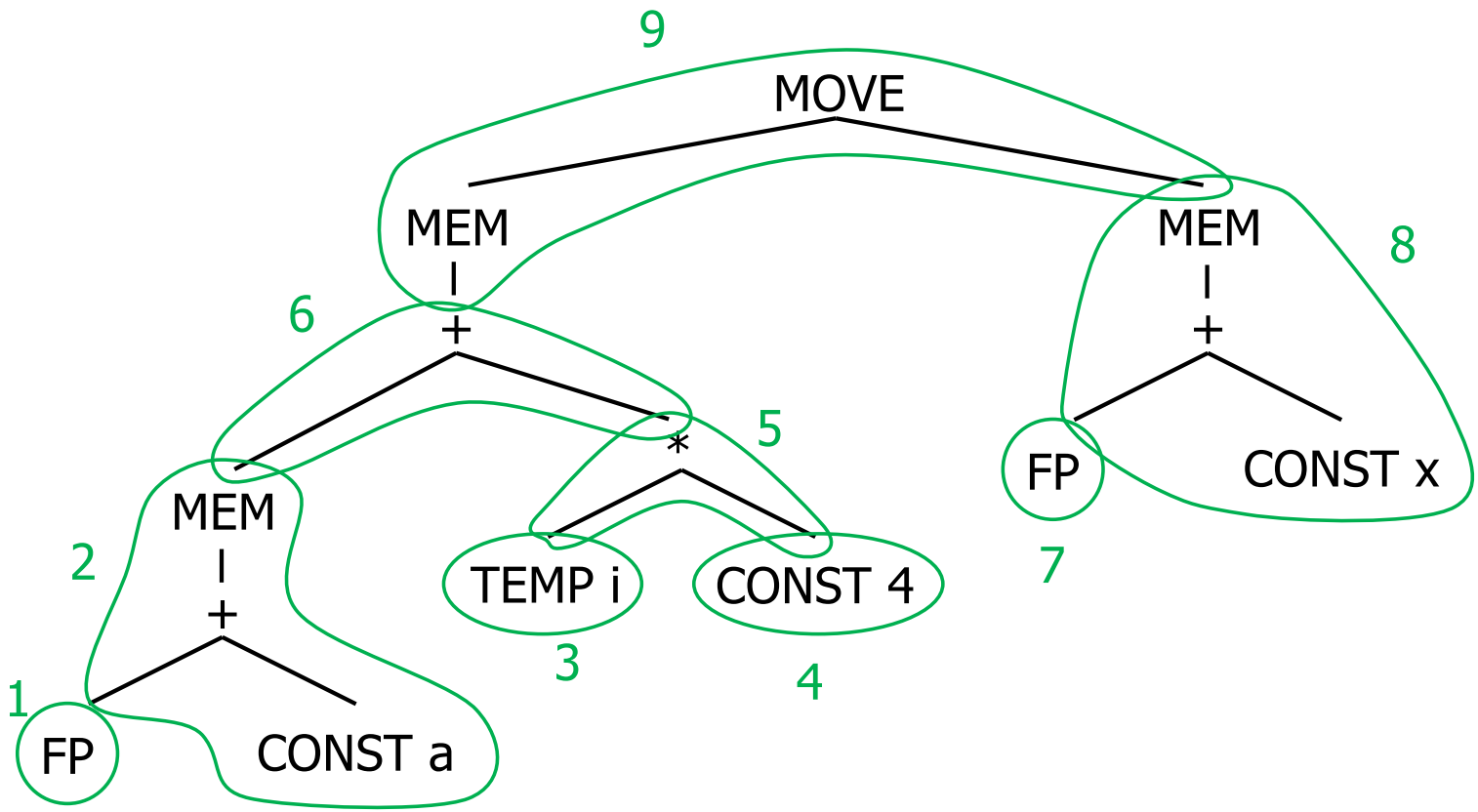
Heuristic: One instruction that "does more" is likely cheaper than several that do less

Slower, but optimal for a given cost model

Example – Tree for a[i]:=x



Example – Tree for a[i]:=x

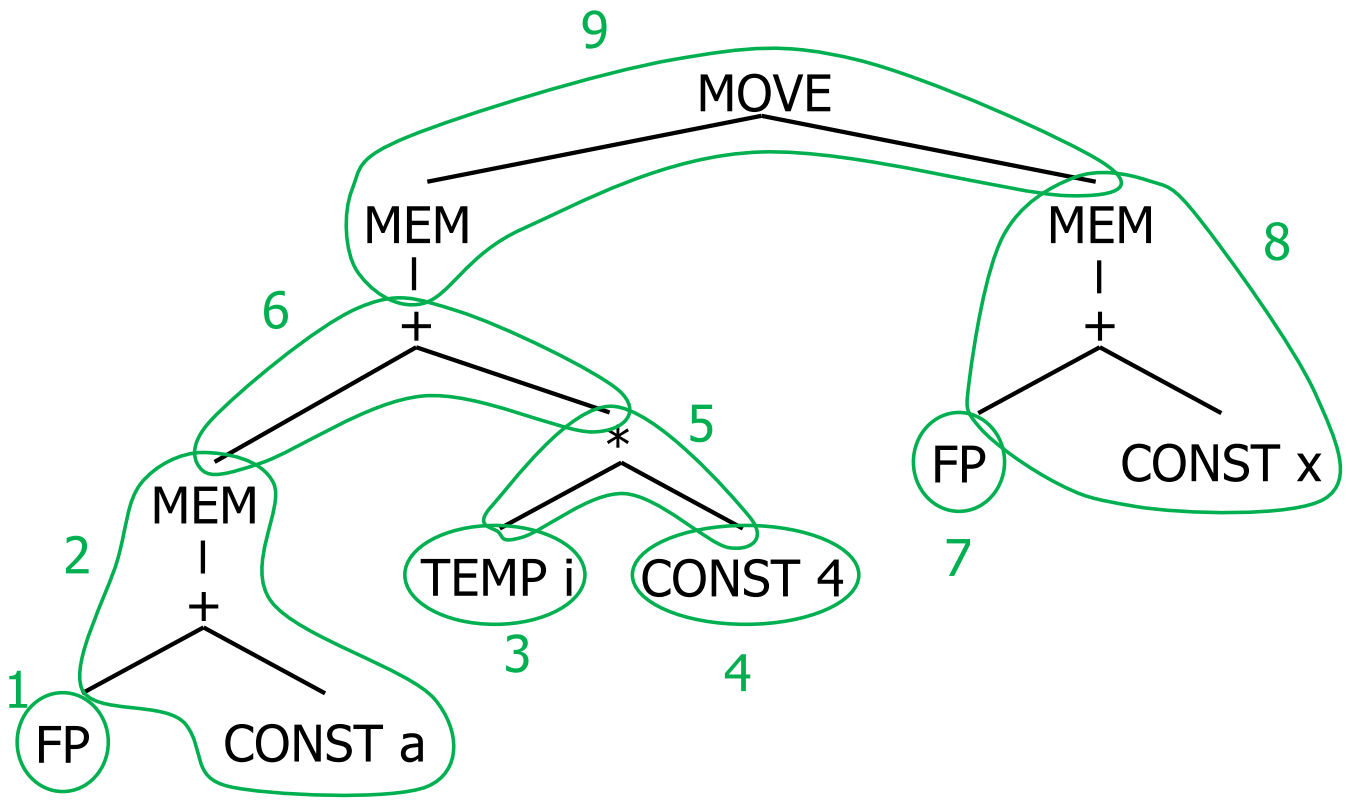


Generating Code

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

Example – Tree for a[i]:=x



2. LOAD r1 <- M[fp+a_{off}]
4. ADDI r2 <- 4 + r0
5. MUL r2 <- r2 * r_i
6. ADD r1 <- r1 + r2
8. LOAD r2 <- M[fp+x_{off}]
9. STORE M[r1+0] <- r2

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

Instruction Scheduling

- Reorder instructions to minimize execution time given instruction and operand latencies

- Assume fixed program code at this point

Some Scheduling Issues (1)

- 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 (many) cycles after load/store starts to do other useful work

Some Scheduling Issues (2)

- Branch costs vary
- Branches on some processors have delay slots
  - (relatively rare on newer processors)
- Modern processors have good heuristics to predict whether branches are taken and try to keep pipelines full, but good code from compiler makes these more effective

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

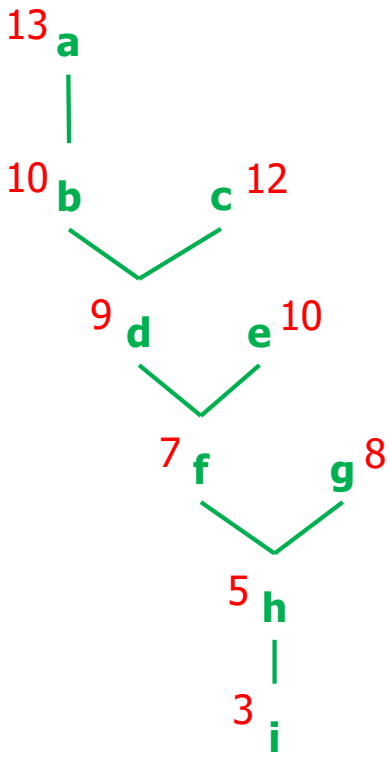
  type – kind of operation

  delay – latency until end of graph
- If node n2 uses the result of node n1, there is an edge e = (n1,n2) 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



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

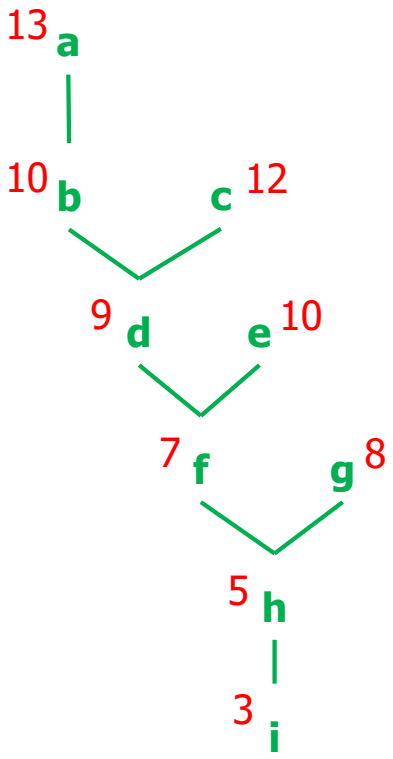
List Scheduling Algorithm

```
Cycle = 1;  Ready = leaves of P;  Active = empty;
while (Ready and/or Active are not empty)
    if (Ready is not empty)
      remove an op from Ready;
      S(op) = Cycle;
      Active = Active ∪ op;
    Cycle++;
    for each op in Active
      if (S(op) + delay(op) <= Cycle)
            remove op from Active;
            for each successor s of op in P
                  if (s is ready – i.e., all operands available)
                        add s to Ready
```

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



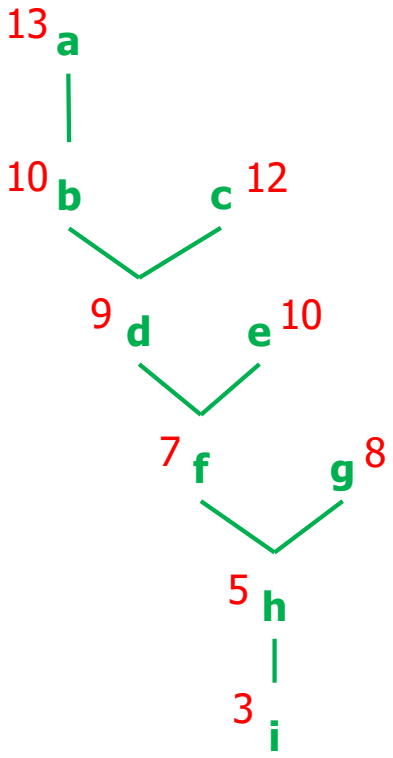
#	instr	done
1	a LOAD	4

cycle: 1
ready: a c e g
active: --

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



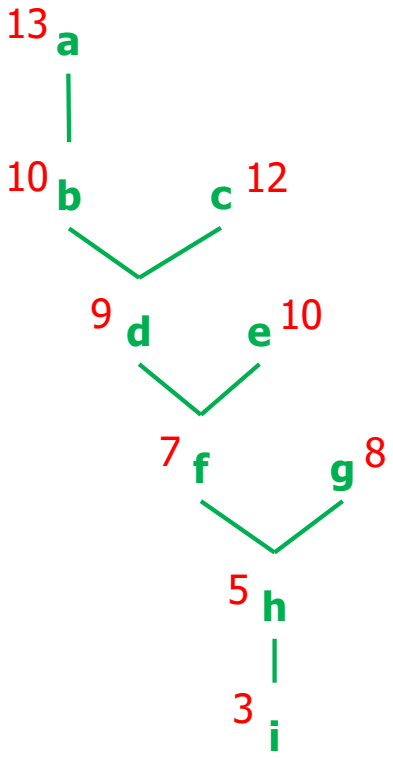
#	instr	done
1	a LOAD	4
2	c LOAD	5

cycle: ~~1~~ 2
ready: ~~a~~ c e g
active: a

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



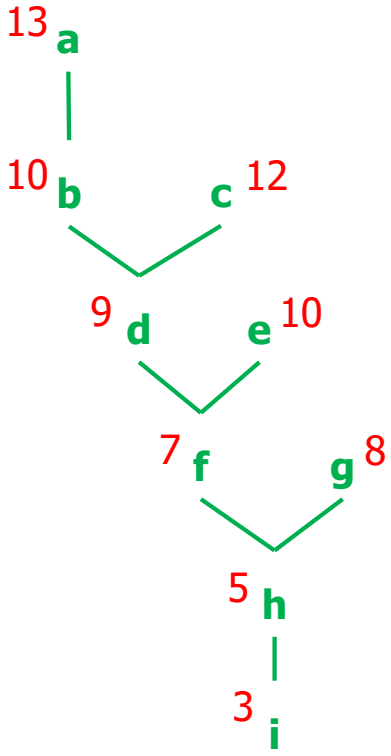
#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6

cycle: ~~1 2~~ 3
ready: ~~a c~~ e g
active: a c

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



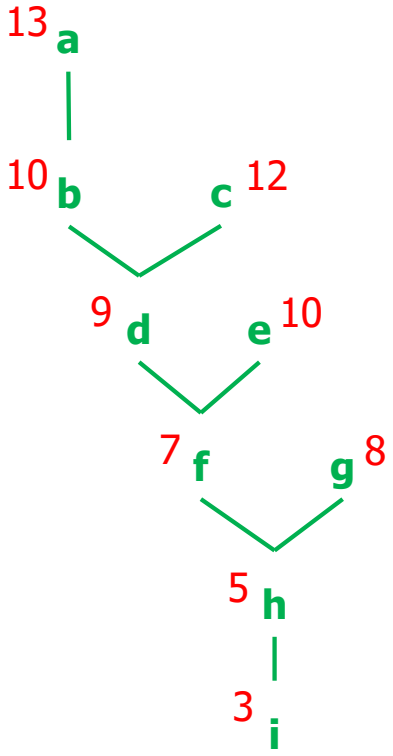
#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5

cycle: ~~1 2 3~~ 4
ready: ~~a c e~~ g b
active: ~~a~~ c e

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



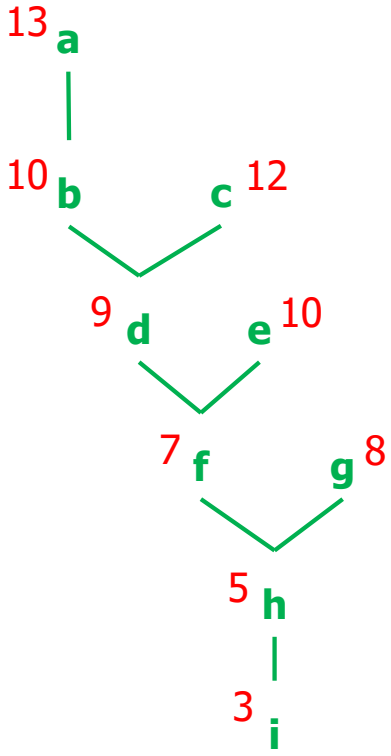
#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7

cycle: ~~1 2 3 4~~ 5
ready: ~~a c e~~ g ~~b~~ d
active: ~~a c~~ e ~~b~~

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



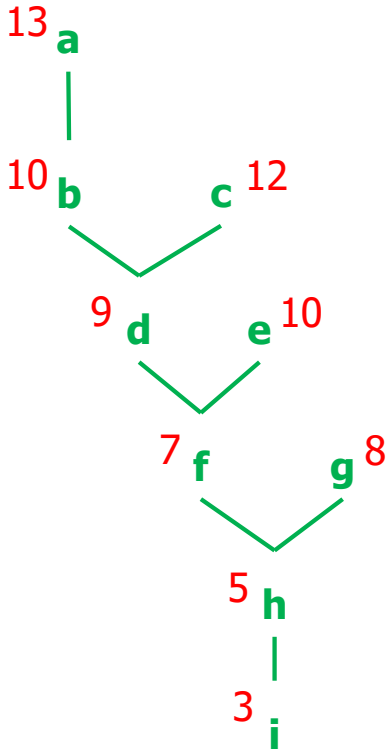
#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9

cycle: ~~1 2 3 4 5~~ 6
ready: ~~a c e~~ g ~~b d~~
active: ~~a c e b~~ d

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



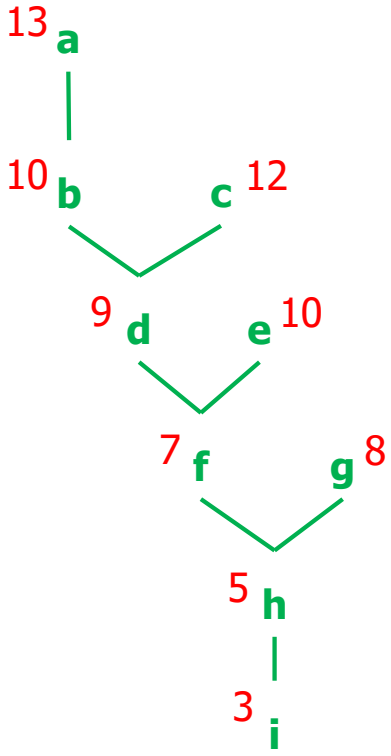
#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9
7	f MULT	9

cycle: ~~1 2 3 4 5 6~~ 7
ready: ~~a c e g b d~~ f
active: ~~a c e b d~~ g

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



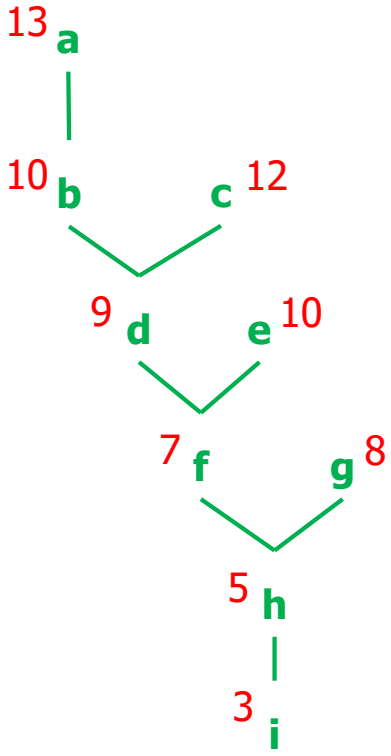
#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9
7	f MULT	9
8	---	

cycle: ~~1 2 3 4 5 6 7~~ 8
ready: ~~a c e g b d f~~
active: ~~a c e b d~~ g f

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



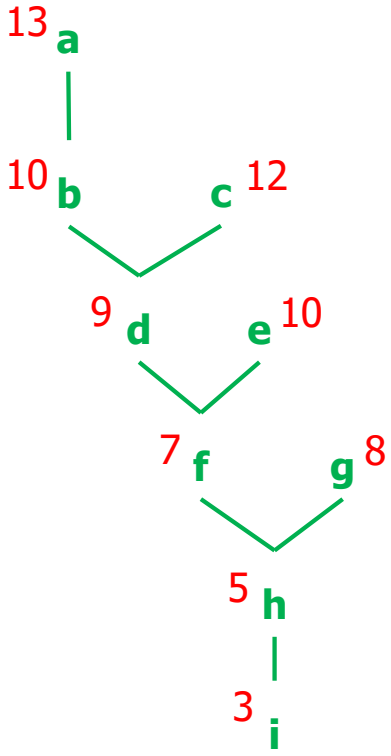
#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9
7	f MULT	9
8	---	
9	h MULT	11

cycle: ~~1 2 3 4 5 6 7 8~~ 9
ready: ~~a c e g b d f~~ h
active: ~~a c e b d g f~~

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



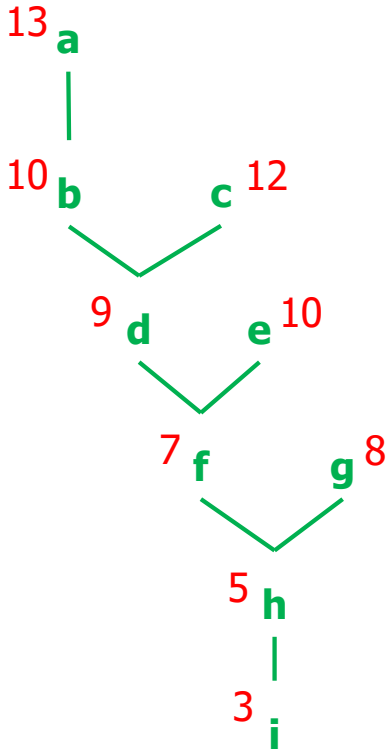
#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9
7	f MULT	9
8	---	
9	h MULT	11
10	---	

cycle: ~~1 2 3 4 5 6 7 8 9~~ 10
ready: ~~a c e g b d f h~~
active: ~~a c e b d g f~~ h

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



#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9
7	f MULT	9
8	---	
9	h MULT	11
10	---	
11	i STORE	14

cycle: ~~1 2 3 4 5 6 7 8 9 10~~ 11
ready: ~~a c e g b d f h~~ i
active: ~~a c e b d g f h~~

Forward vs Backwards

- Alternative: 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

Beyond Basic Blocks

- List scheduling dominates, but moving beyond basic blocks can improve quality of the code. Some possibilities:
  - Schedule extended basic blocks
    - Watch for exit points – limits reordering or requires compensating
  - Trace scheduling
    - Use profiling information to select regions for scheduling using traces (paths) through code
      - Optimize schedules for high-frequency paths

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

k

- Intermediate code typically assumes infinite number of registers
- Real machine has k registers available
- Goals
  - Produce correct code that uses k or fewer registers
  - Minimize added loads and stores
  - Minimize space needed for spilled values
  - Do this efficiently – O(n), O(n log n), maybe $O(n^2)$

Register Allocation

- Task
  - At each point in the code, pick the values to keep in registers
  - Insert code to move values between registers and memory
    - No additional transformations – scheduling should have done its job
      - But we will usually rerun scheduling if we insert spill code
  - Minimize inserted code, both dynamically and statically

Allocation vs Assignment

- Allocation: deciding which values to keep in registers
- Assignment: choosing specific registers for values
- Compiler must do both

Local Register Allocation

- Apply to basic blocks
- Produces decent register usage inside a block
  - But can have inefficiencies at boundaries between blocks
- Two variations: top-down, bottom-up

Top-down Local Allocation

- Principle: keep most heavily used values in registers
  - Priority = # of times register referenced in block
- If more virtual registers than physical,
  - Reserve some registers for values allocated to memory
    - Need enough to address and load two operands and store result
  - Other registers dedicated to “hot” values
    - But are tied up for entire block with particular value, even if only needed for part of the block

Bottom-up Local Allocation (1)

- Keep a list of available registers (initially all registers at beginning of block)
- Scan the code
- Allocate a register when one is needed
- Free register as soon as possible
  - In x:=y op z, free y and z if they are no longer needed before allocating x

Bottom-up Local Allocation (2)

- If no registers are free when one is needed for allocation:
  - Look at values assigned to registers – find the one not needed for longest forward stretch in the code
  - Insert code to spill the value to memory and insert code to reload it when needed later
    - If a copy already exists in memory, no need to spill

Local "bottom-up" Register Allocation, -1

```
1.  ; load v2 from memory
2.  ; load v3 from memory
3.  v1  = v2 + v3
4.  ; load v5, v6 from memory
5.  v4  = v5 - v6
6.  v7  = v2 - 29
7.  ; load v9 from memory
8.  v8  = - v9
9.  v10 = v6 * v4
10. v11 = v10 - v3
```

- Still in LIR. So lots (too many!) virtual registers required (v2, etc).
- Grey instructions (1,2,4,7) load operands from memory into virtual registers.
- We will ignore these going forward. Focus on mapping virtual to physical.

Local "bottom-up" Register Allocation, 0

1. v1 = v2 + v3
2. v4 = v5 - v6
3. v7 = v2 - 29
4. v8 = - v9
5. v10 = v6 * v4
6. v11 = v10 - v3

pReg	vReg
R1	-
R2	-
R3	-
R4	-

vReg	NextRef
v1	1
v2	1
v3	1
v4	2
v5	2
v6	2
v7	3
v8	4
v9	4
v10	5
v11	6

Local "bottom-up" Register Allocation, 1

1. v1 = v2 + v3
2. v4 = v5 - v6
3. v7 = v2 - 29
4. v8 = - v9
5. v10 = v6 * v4
6. v11 = v10 - v3

pReg	vReg
R1	v2
R2	v3
R3	v1
R4	-

R3 = R1 + R2

vReg	NextRef
v1	~~1~~ ∞
v2	~~1~~ 3
v3	~~1~~ 6
v4	2
v5	2
v6	2
v7	3
v8	4
v9	4
v10	5
v11	6

Local "bottom-up" Register Allocation, 2

1. v1 = v2 + v3
2. v4 = v5 - v6
3. v7 = v2 - 29
4. v8 = - v9
5. v10 = v6 * v4
6. v11 = v10 - v3

pReg	vReg
R1	v2
R2	~~v3~~ v4
R3	~~v1~~ v6
R4	v5

vReg	NextRef
v1	∞
v2	3
v3	6
v4	~~2~~ 5
v5	~~2~~ ∞
v6	~~2~~ 5
v7	3
v8	4
v9	4
v10	5
v11	6

```
R3 = R1 + R2
; spill R3
; spill R2? - no - still clean
R2 = R4 - R3
```

Local "bottom-up" Register Allocation, 3

1. v1 = v2 + v3
2. v4 = v5 - v6
3. v7 = v2 - 29
4. v8 = - v9
5. v10 = v6 * v4
6. v11 = v10 - v3

pReg	vReg
R1	v2
R2	v4
R3	v6
R4	~~v5~~ v7

vReg	NextRef
v1	∞
v2	~~3~~ ∞
v3	6
v4	5
v5	∞
v6	5
v7	~~3~~ ∞
v8	4
v9	4
v10	5
v11	6

```
R3 = R1 + R2
; spill R3
; spill R2? - no!
R2 = R4 - R3
; spill R4? - no!
R4 = R1 - 29
```

And so on . . .

Bottom-Up Allocator

- Invented about once per decade
  - Sheldon Best, 1955, for Fortran I
  - Laslo Belady, 1965, for analyzing paging algorithms
  - William Harrison, 1975, ECS compiler work
  - Chris Fraser, 1989, LCC compiler
  - Vincenzo Liberatore, 1997, Rutgers
- Will be reinvented again, no doubt
- Many arguments for optimality of this

Global Register Allocation by Graph Coloring

- Convert the (seemingly) infinite sequence of temporary data references, t1, t2, … 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

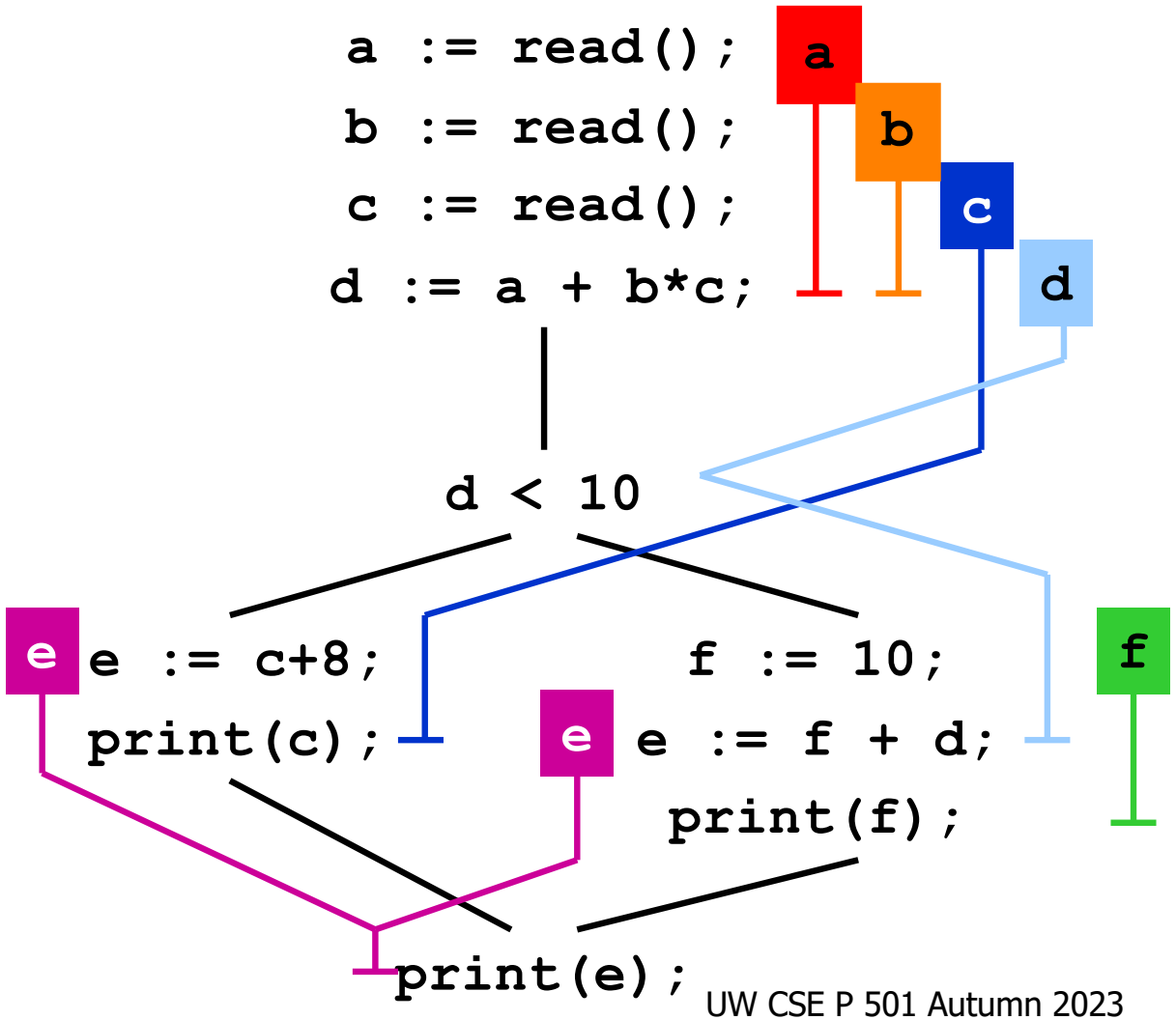
Begin With Data Flow Graph

- 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

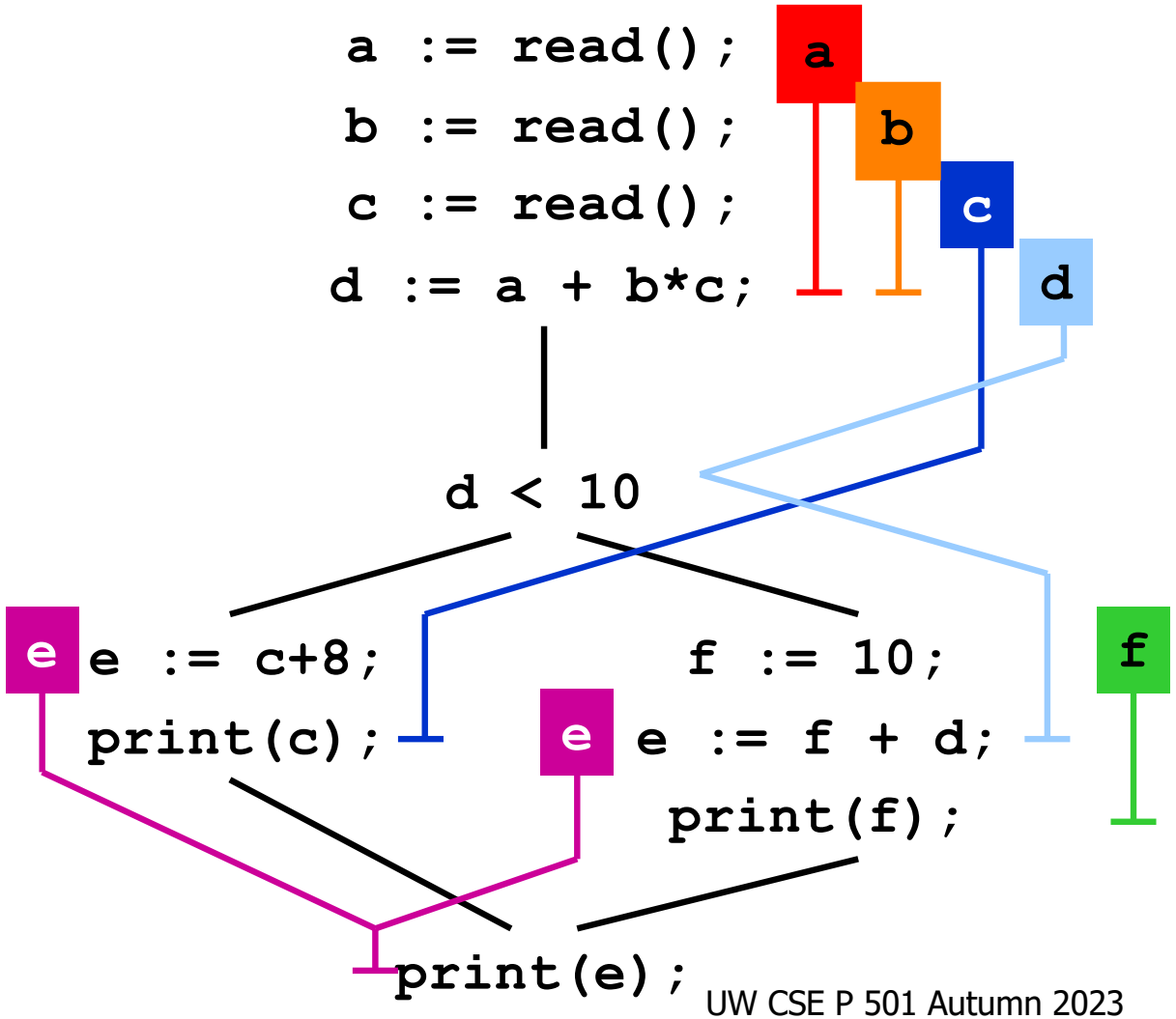
- two variables(values) interfere when their live ranges overlap

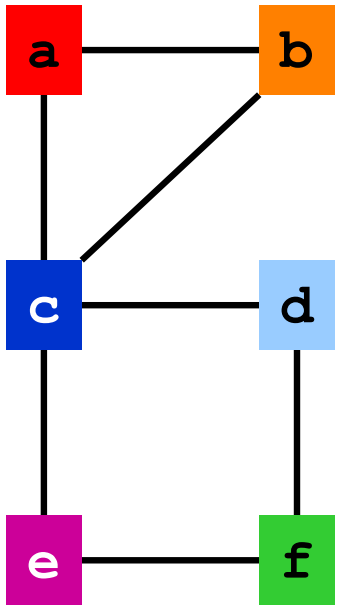
Live Variable Analysis




```
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);
```

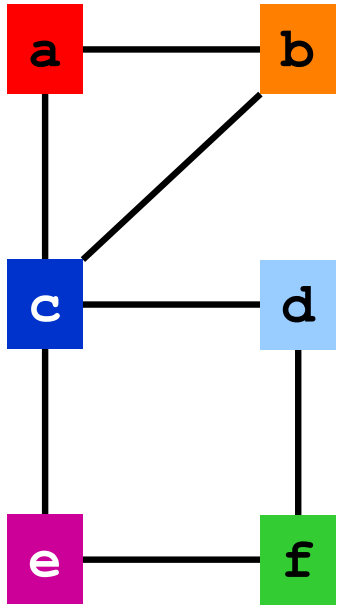
Register Interference Graph





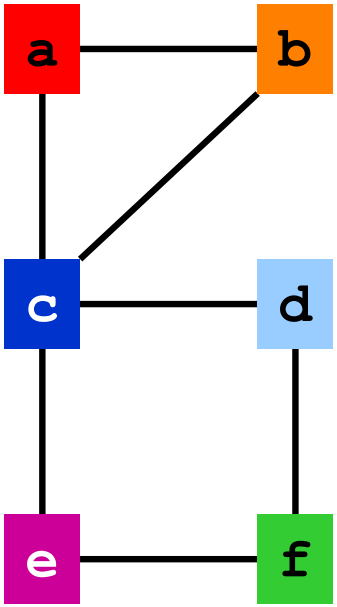
Graph Coloring

- NP complete problem

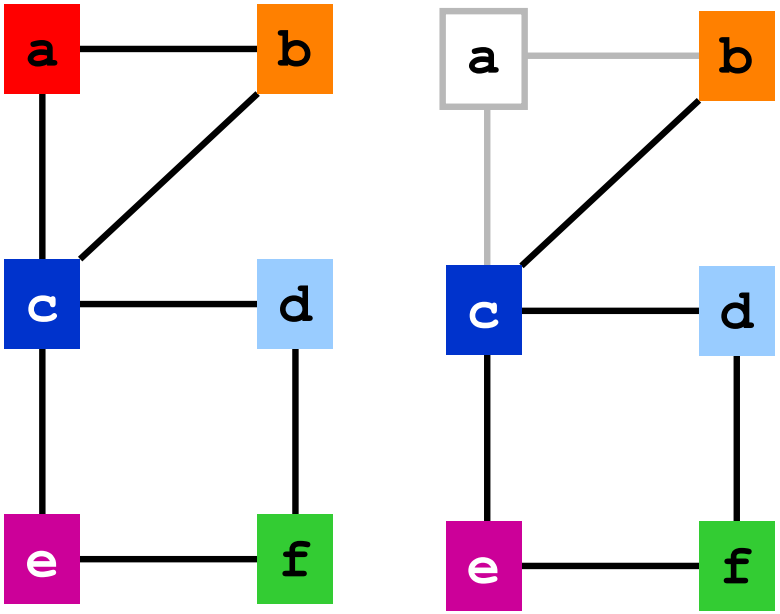


- Heuristic: color easy nodes last
  - find node N with lowest degree
  - remove N from the graph
  - color the simplified graph
  - set color of N to the first color that is not used by any of N ’s neighbors
- Basics due to Chaitin (1982), refined by Briggs (1992)

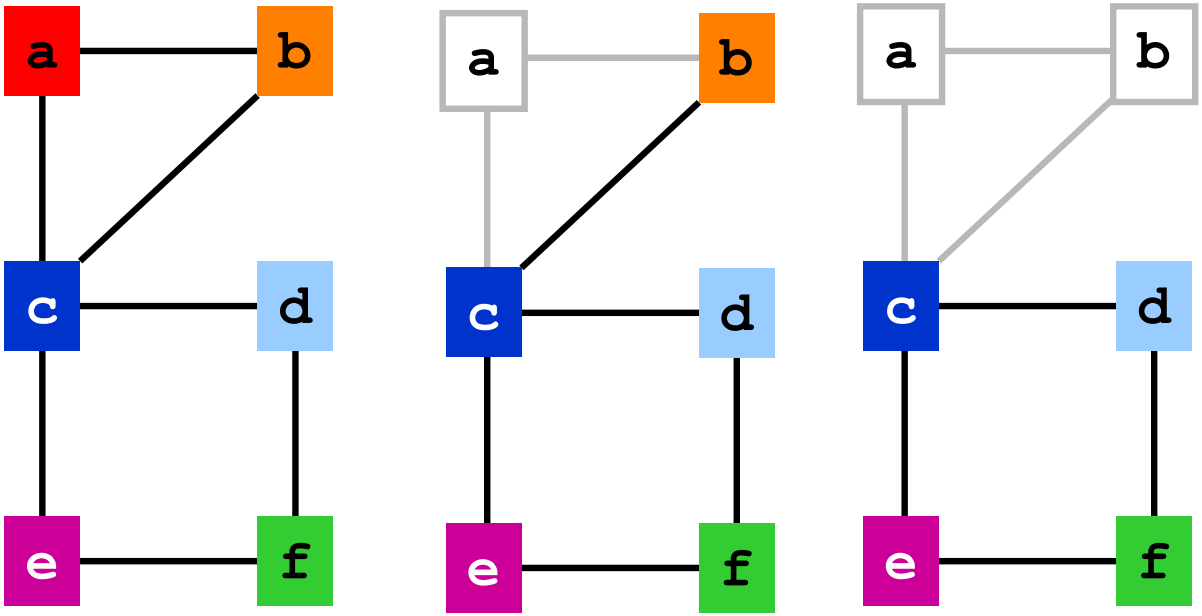
Apply Heuristic



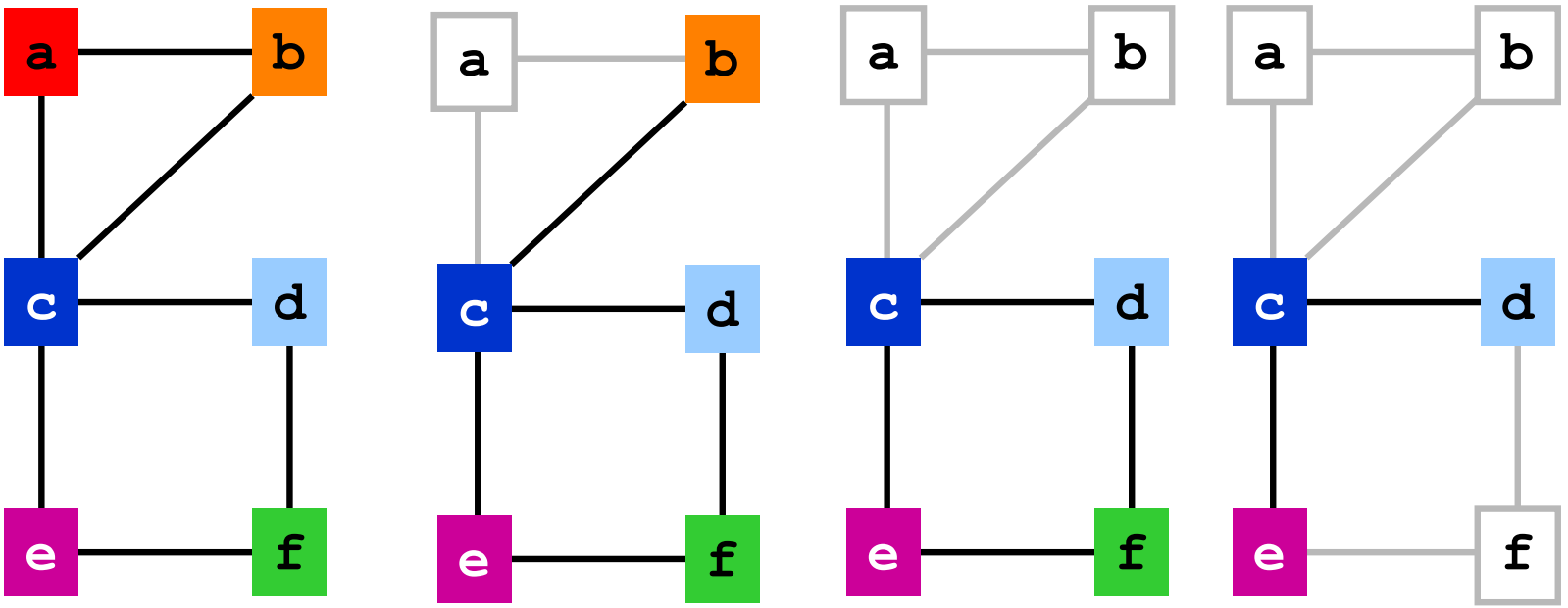
Apply Heuristic



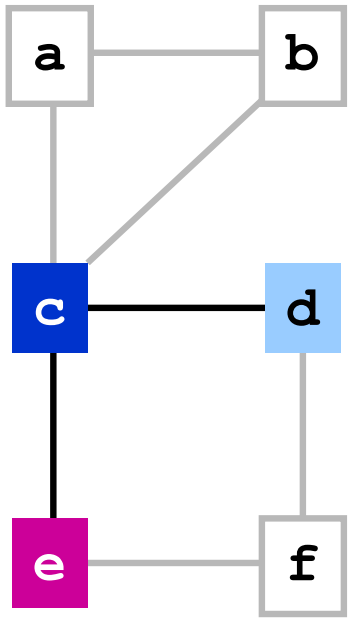
Apply Heuristic



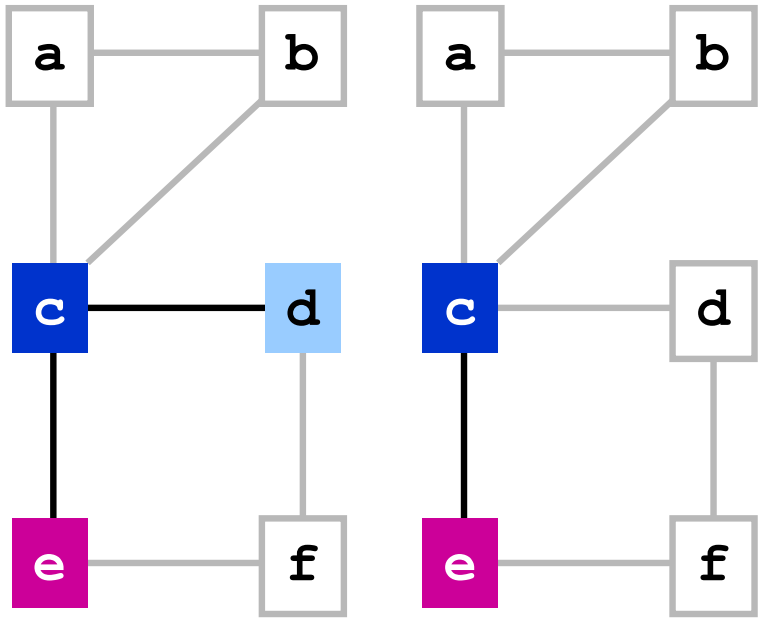
Apply Heuristic



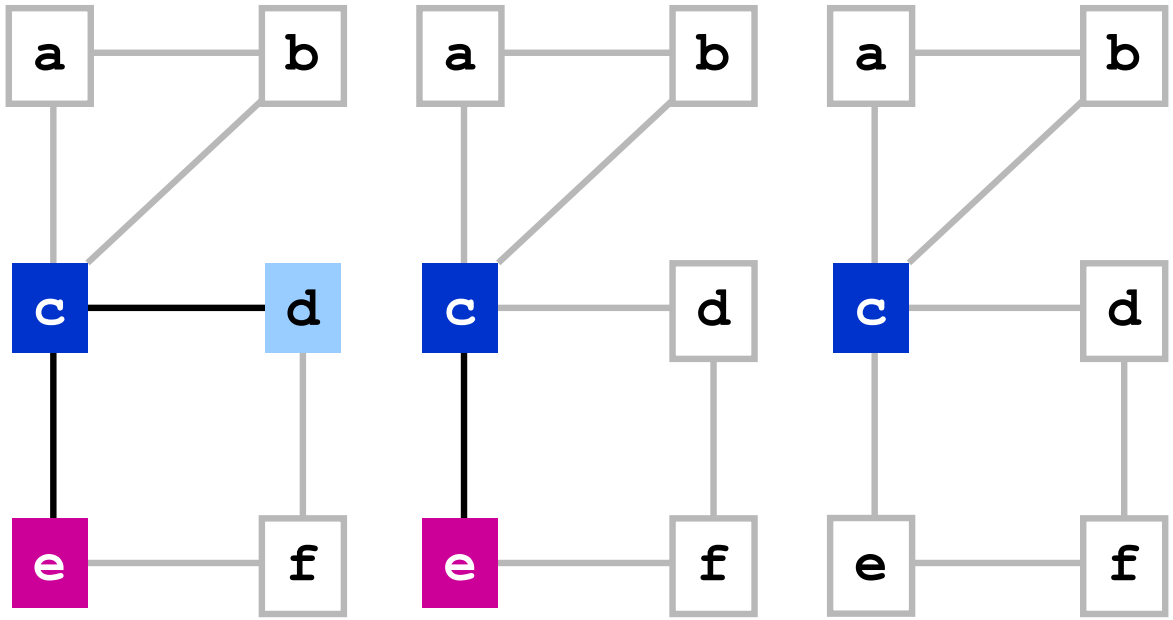
Continued



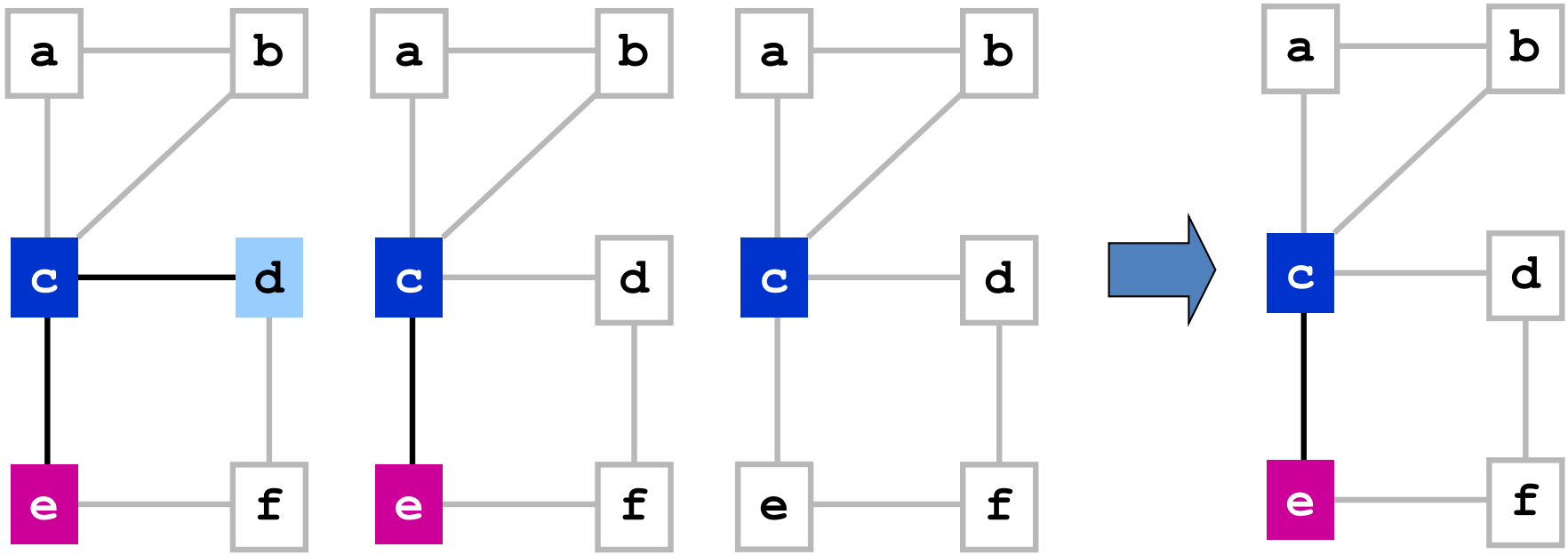
Continued



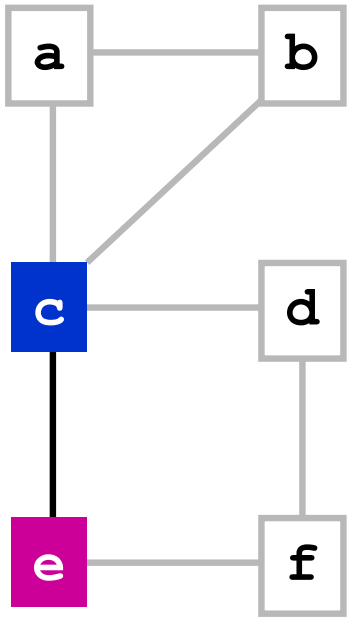
Continued



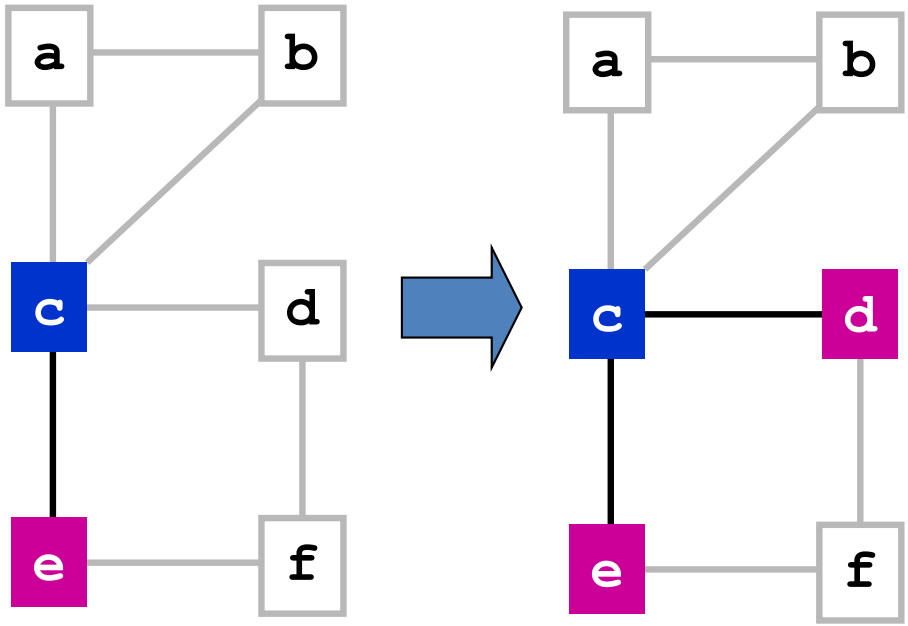
Continued



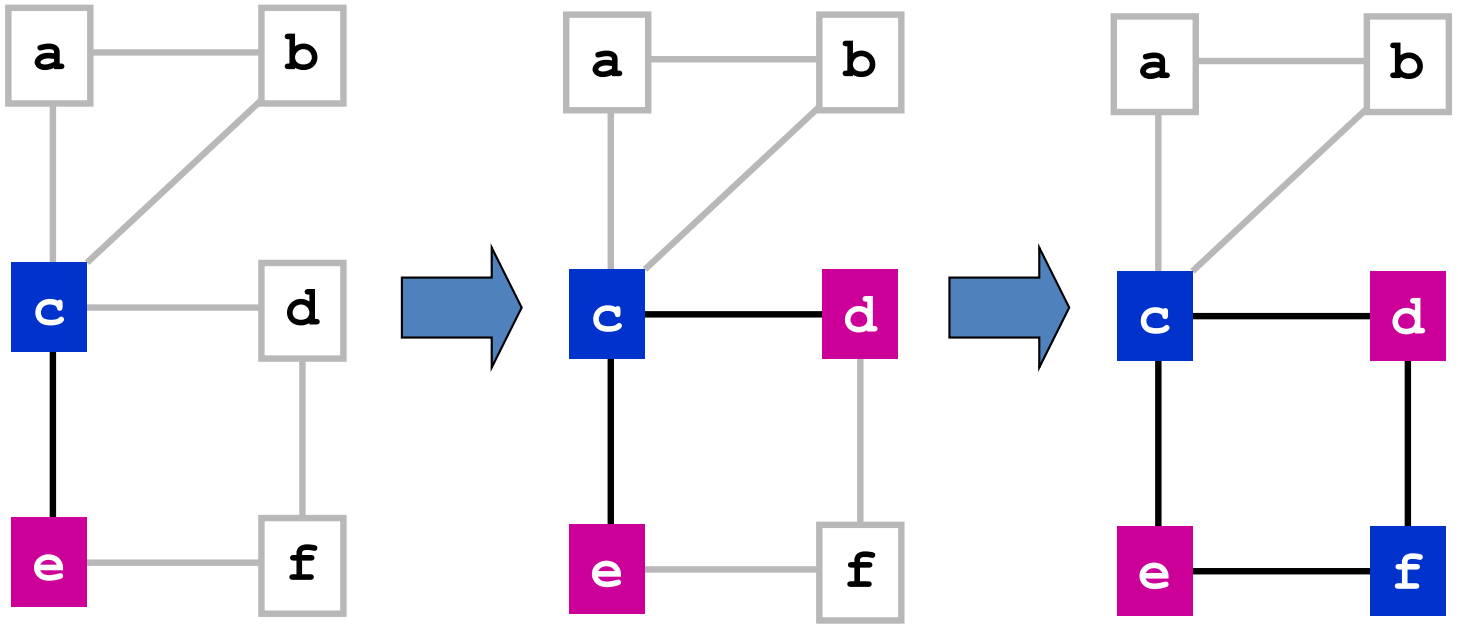
Continued



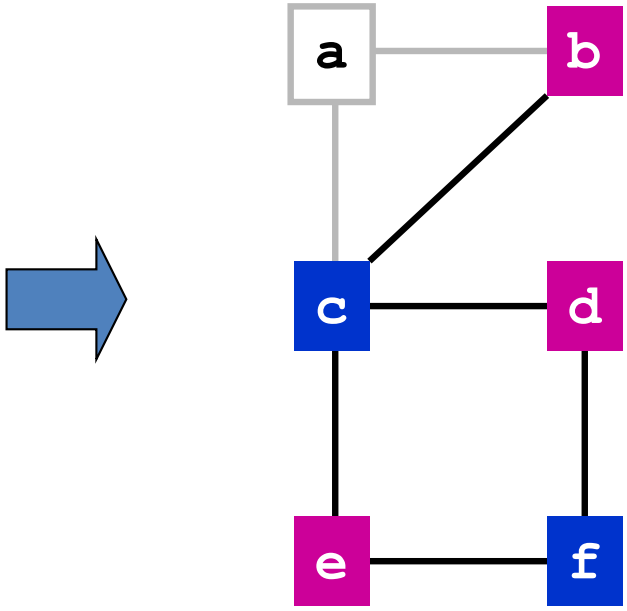
Continued



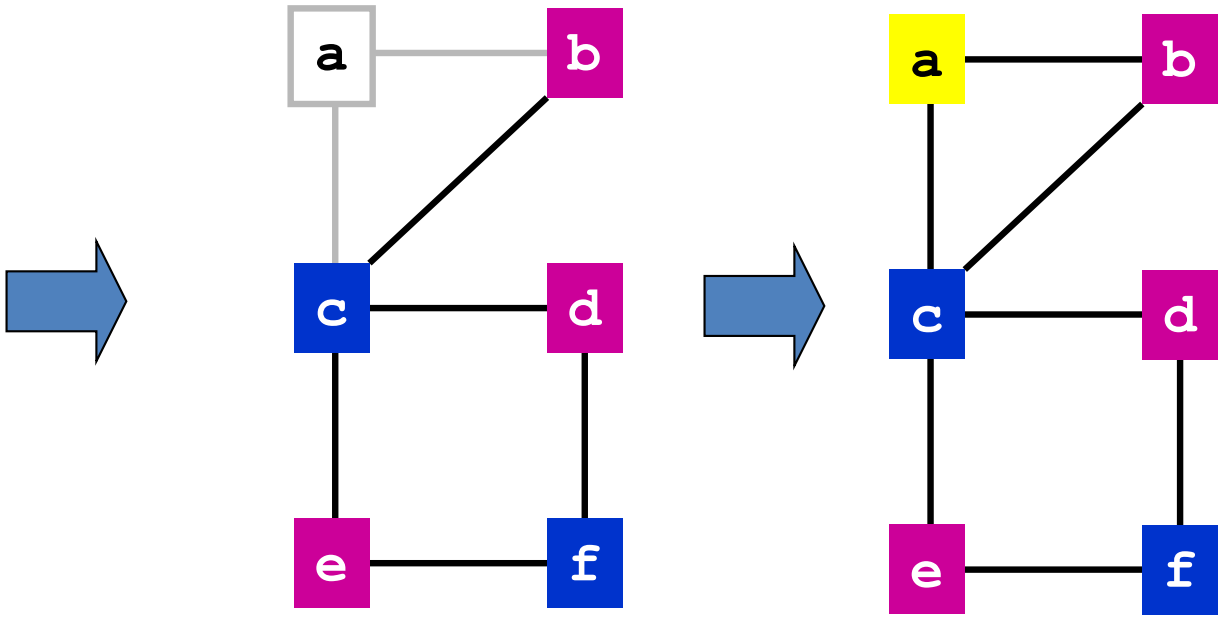
Continued



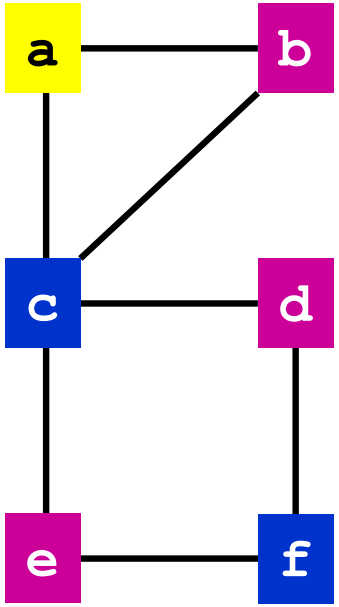
Continued



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

Live Ranges

- Real graph-coloring register allocators don't allocate temp registers – they allocate *live ranges*
- A *live range* is a set of definitions and uses that flow together
  - In a basic block is the interval between a defn and last use
  - In a CFG, similar but more complex. Result is a coherent set of definitions and uses.
  - 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
  - So we build a SSA form of the IR to construct the interference graph!

Live Ranges: Example

1. loadi … → rfp
2. loadai rfp, 0 → rw
3. loadi 2 → r2
4. loadai rfp,xoffset → rx
5. loadai rfp,yoffset → ry
6. loadai rfp,zoffset → rz
7. mult rw, r2 → rw
8. mult rw, rx → rw
9. mult rw, ry → rw
10. mult rw, rz → rw
11. storeai rw → rfp, 0

Register	Interval
rfp	[1,11]
rw	[2,7]
rw	[7,8]
rw	[8,9]
rw	[9,10]
rw	[10,11]
r2	[3,7]
rx	[4,8]
ry	[5,9]
rz	[6,10]

Coloring by Simplification

- Linear-time approximation that generally gives good results
  1. Build: Construct the interference graph
  2. Simplify: Color the graph by repeatedly simplification
  3. Spill: If simplify cannot reduce the graph completely, mark some node for spilling
  4. Select: Assign colors to nodes in the graph

1. Build

- Construct the interference graph
- Find live ranges – SSA!
  - Build SSA form of IR
  - Each SSA name is initially a singleton set
  - A Φ-function means form the union of the sets that includes those names (union-find algorithm)
  - Resulting sets represent live ranges
  - Either rewrite code to use live range names or keep a mapping between SSA names and live-range names

1. Build

- Use dataflow information to build interference graph
  - Nodes = live ranges
  - Add an edge in the graph for each pair of live ranges that overlap
    - But watch copy operations. MOV ri $\rightarrow$ rj does not create interference between ri, rj since they can be the same register if the ranges do not otherwise interfere

2. Simplify

- Heuristic: Assume we have K registers
- Find a node *m* with fewer than K neighbors
- Remove *m* from the graph. If the resulting graph can be colored, then so can the original graph (the neighbors of m have at most K-1 colors among them)
- Repeat by removing and pushing on a stack all nodes with degree less than K
  - Each simplification decreases other node degrees – may make more simplifications possible

3. Spill

- If simplify stops because all nodes have degree ≥ k, mark some node for spilling
  - This node is in memory during execution
  - ∴ Spilled node no longer interferes with remaining nodes, reducing their degree.
  - Continue by removing spilled node and push on the stack (optimistic – hope that spilled node does not interfere with remaining nodes – Briggs allocator)

3. Spill

- Spill decisions should be based on costs of spilling different values
- Issues
  - Address computation needed for spill
  - Cost of memory operation
  - Estimated execution frequency

    (e.g., inner loops account for most execution time)

4. Select

- Assign nodes to colors in the graph:
  - Start with empty graph
  - Rebuild original graph by repeatedly adding node from top of the stack
    - (When we do this, there must be a color for it if it didn't represent a potential spill – pick a different color from any adjacent node)
  - When a potential spill node is popped it may not be colorable (neighbors may have k colors already). This is an actual spill.

5. Start Over

- If Select phase cannot color some node (must be a potential spill node), add loads before each use and stores after each definition
  - Creates new temporaries with tiny live ranges
- Repeat from beginning
  - Iterate until Simplify succeeds
  - In practice a couple of iterations are enough

Coalescing Live Ranges

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

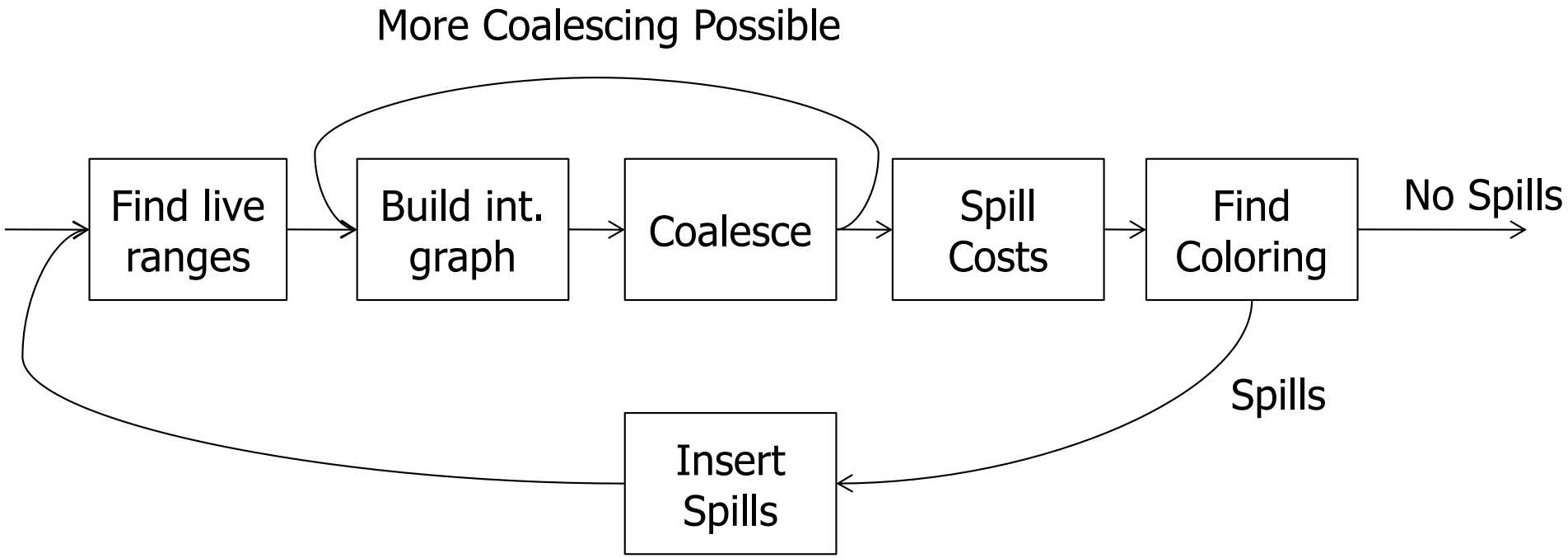
Advantages?

- 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 ri, rj
- 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!

Graph Representation

- The interference graph representation drives the time and space requirements for the allocator (and maybe the compiler)
- Not unknown to have O(5K) nodes and O(1M) edges
- Dual representation works best
  - Triangular bit matrix for efficient access to interference information
  - Vector of adjacency vectors for efficient access to node neighbors

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…