Code generation and optimization

Greta Yorsh

PA1-PA3

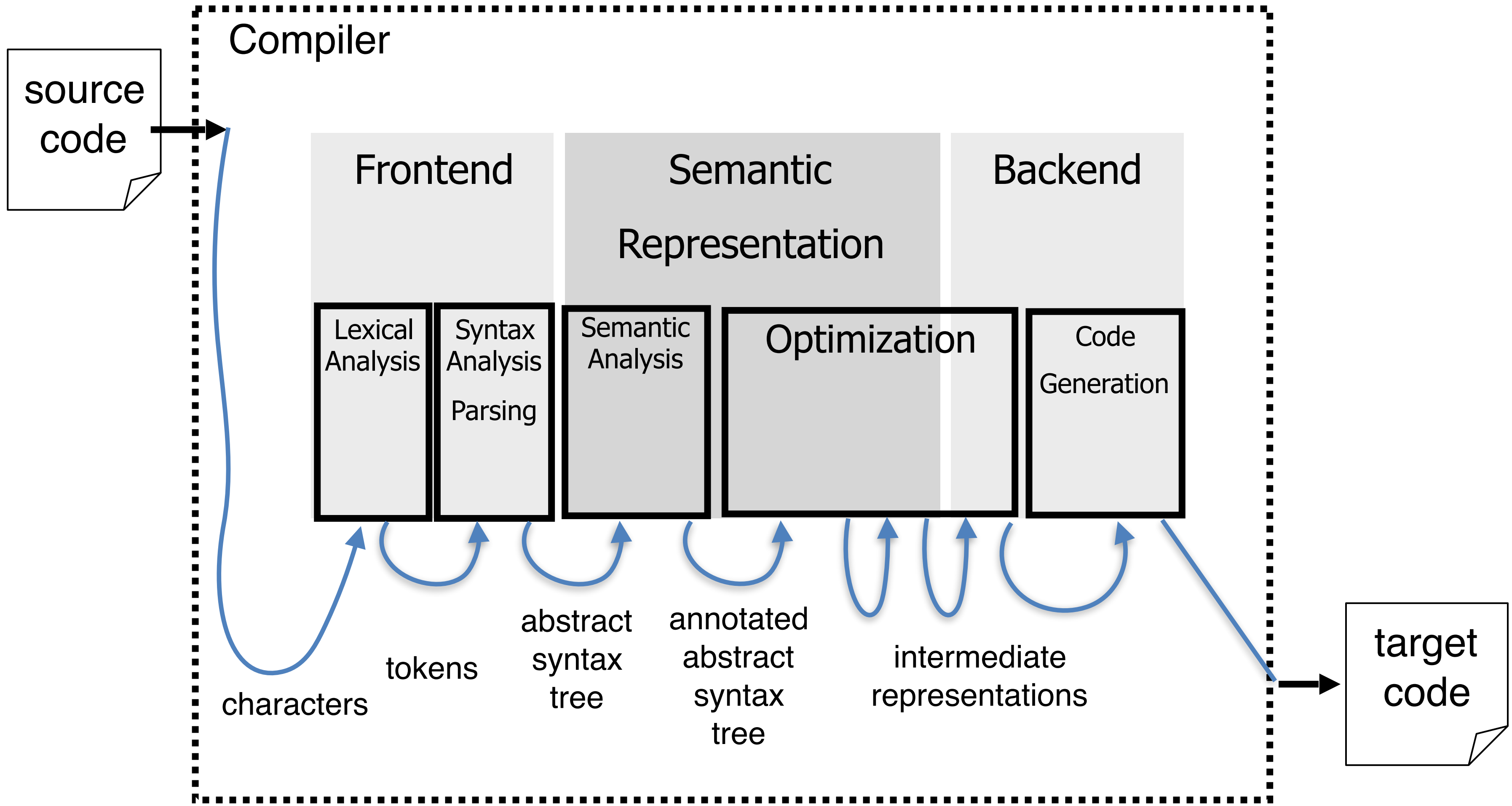
- My apologies for the delays
- Final deadline: midnight Sunday, 8 April
- Marking: best 2 out of 3 for pa1-pa3

- PA1 and PA2 code review: end of this week
- PA3
  - very small
  - you need to know this material for the exam

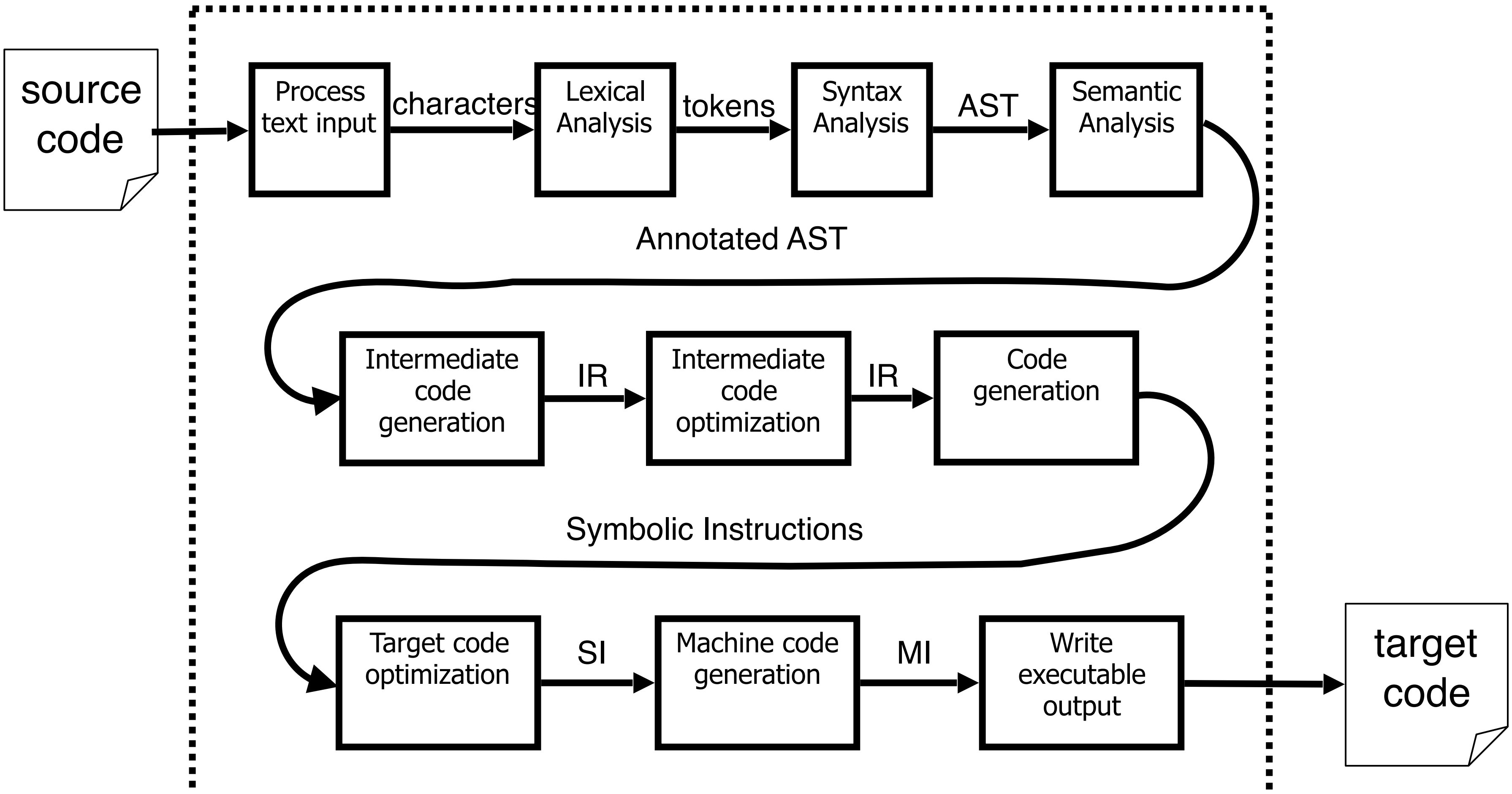
Outline

- Code generation for expressions
  - ✓ simple stack machine
  - ✓ stack machine with accumulator
  - weighted register allocation
  - graph coloring: introduction
- How to prepare for the exam?

Anatomy of a modern compiler



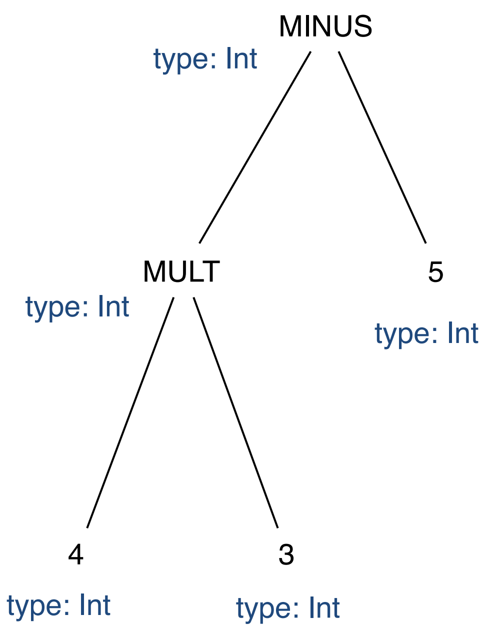
The real anatomy of a modern compiler



Recap: stack machine with accumulator

- For each expression e, function cgen(e) generates MIPS code that
  - computes the value of e in the accumulator $a0
  - preserves the contents of the stack
- Implement cgen(e) as a traversal of AST
- For a binary operation e1 op e2,
  - after computing e1, push the accumulator on the stack
  - the result of e2 is in the accumulator before op
  - after the operation, pop one value off the stack

Example: stack machine

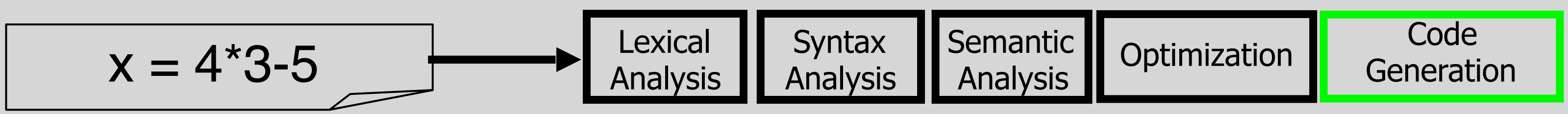


Intermediate representation: stack machine

```
acc := 4
push
acc := 3
acc := top * acc
pop
push
acc := 5
acc := top - acc
pop
```

MIPS Assembly

```
li     $a0 4
sw     $a0 0($sp)
addiu  $sp $sp -4
li     $a0 3
lw     $t1 4($sp)
mul    $a0 $a0 $t1
addiu  $sp $sp 4
sw     $a0 0($sp)
addiu  $sp $sp -4
li     $a0 5
lw     $t1 0($sp)
sub    $a0 $t1 a0
add    $sp $sp 4
```



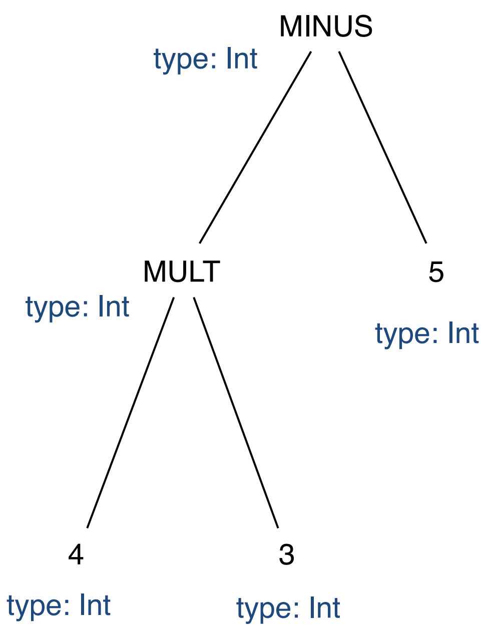
In the real world….

- Production compilers do different things
- Emphasis is on keeping values in registers
- Intermediate results are laid out on the stack, not pushed and popped from the stack

LIR vs. Assembly

	LIR	**Assembly**
Register number	**Unlimited**	**Limited**
Function calls	Implicit	Runtime stack
Instruction set	Abstract	Concrete
Types	Basic and user defined	Limited basic types

Example: registers

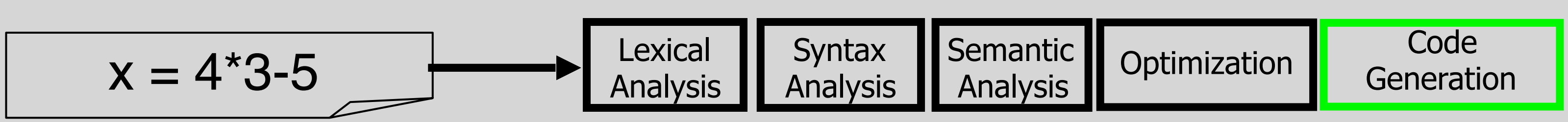


Intermediate representation: three address code

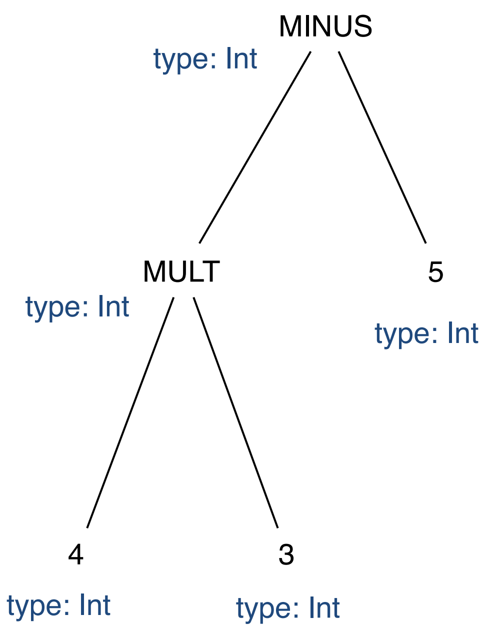
```
t0 := 4
t1 :=  t0 * 3
a0 := t1 - 5
```

MIPS Assembly

```
li  $t0 4
mul $t1 $t0 3
sub $a0 $t1 5
```



Example: simple optimization

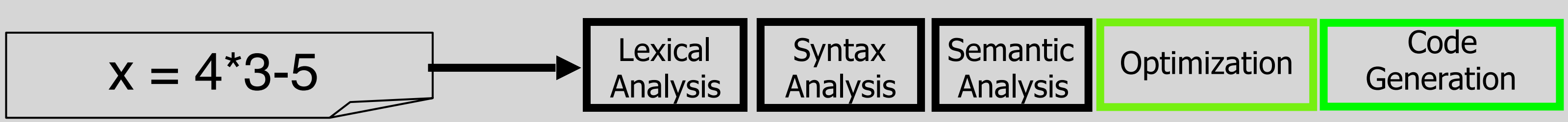


Intermediate representation: after constant propagation

a0 := 7

MIPS Assembly

```
li $a0 7
```



CODE GENERATION FOR EXPRESSIONS

Instruction selection: example

- Which instructions to use for each expression?

x + (2 + 3)

```
lw $t0 x

li $t1 2

li $t2 3

add $t1 $t1 $t2

add $t0 $t0 $t1
```

```
lw $t0 x

li $t1 2

addi $t1 $t1 3

add $t0 $t0 $t1
```

Register allocation

- What computational results are kept in registers?

- Possibly not enough registers for all values
- Some instructions have restrictions on registers they can access

Recap: Registers

- One register stores a single word (4 bytes)
- Number of registers is limited
- Very fast access, even compared to cached memory
- Typical uses of registers
  - operands of instructions
  - store temporary results
  - loop indexes
  - store administrative info
    - $sp for top of the runtime stack
    - $a0 used to return values

Simple register allocation

- Assumptions
  - enough registers
  - all registers are general-purpose
  - we have all registers available for our use
  - ignore registers allocated for stack management
- Formals and locals not given registers (always spilled on stack)

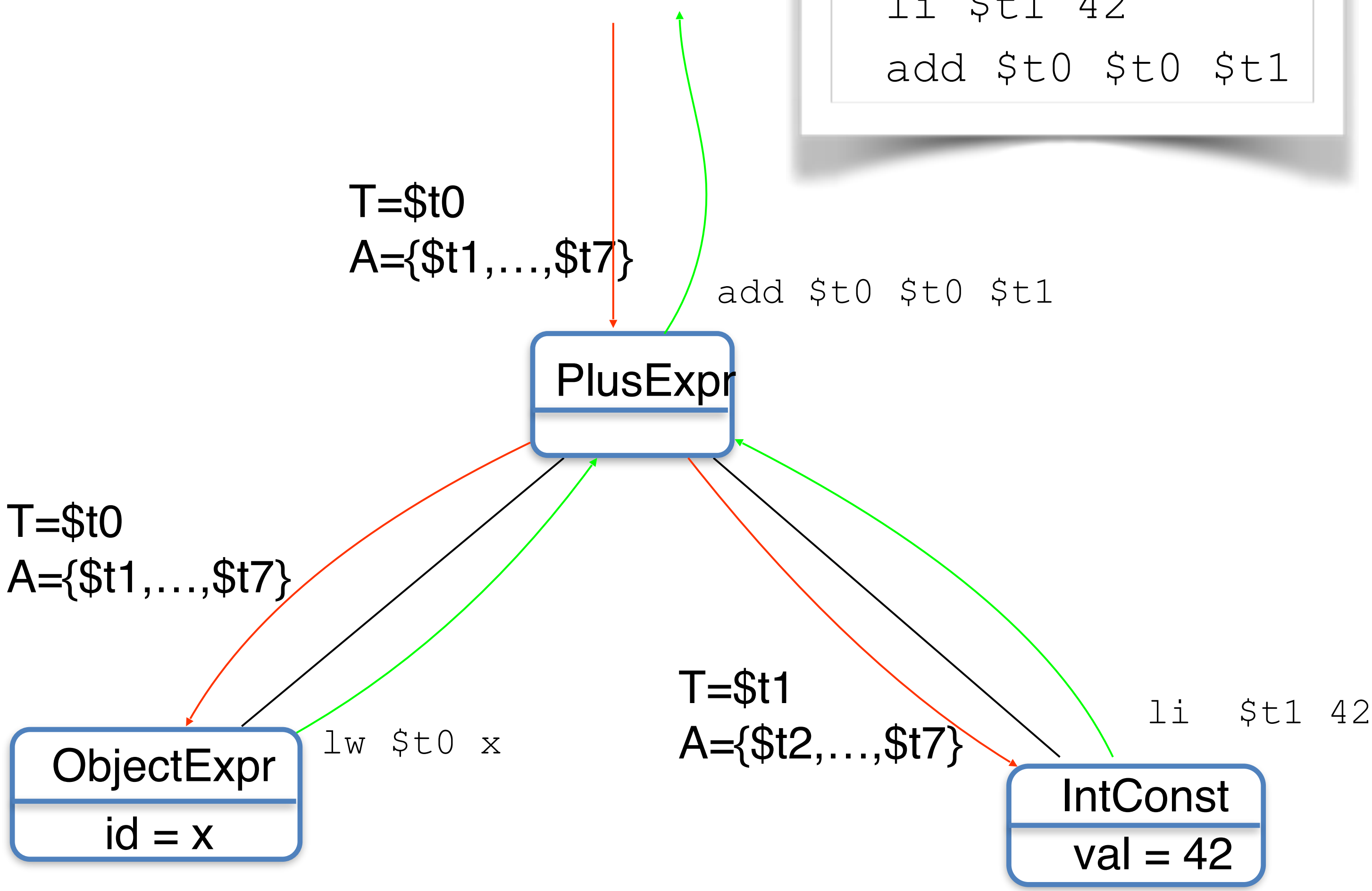
Simple register allocation

- AST traversal
- Top-down: assign registers to subtrees
  - T is target register designated for storing result
  - A is a set of auxiliary registers for temporaries
- Bottom-up: use code templates for AST nodes
  - cgen(node, T, A)

Simple register allocation

cgen(x + 42,T,A) =

```
lw $t0 x
li $t1 42
add $t0 $t0 $t1
```



Simple register allocation

- cgen(x + 42,T,A)

T=\$t0
A={\$t1,…,\$t7}

```
add $t0 $t0 $t1
```

- cgen(x,T,A)

T=\$t0
A={\$t1,…,\$t7}

```
lw $t0 x
```

- cgen(42,T,A)

T=\$t1
A={\$t2,…,\$t7}

```
li $t1 42
```

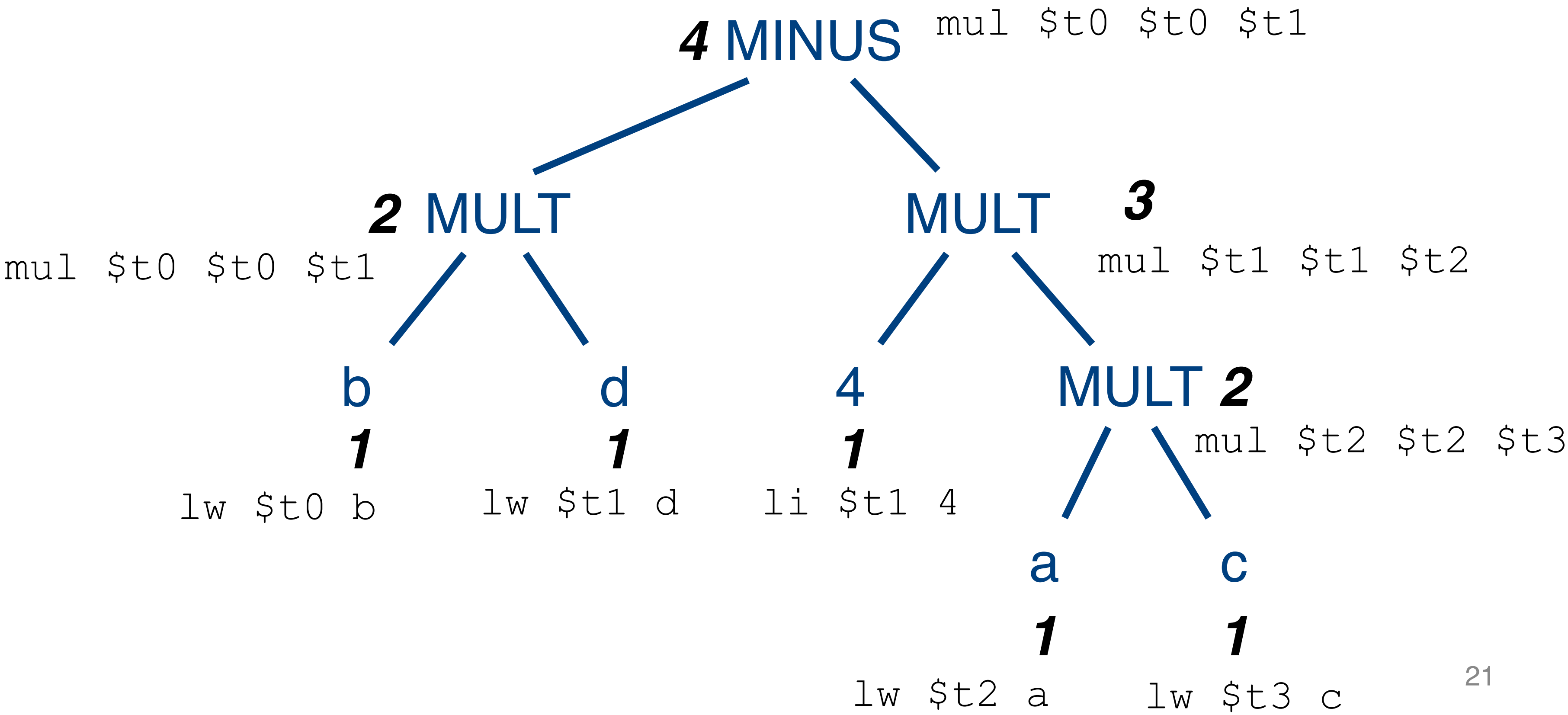
Simple register allocation

- Leaf nodes
  - emit code using target register
  - no auxiliaries required
- Internal nodes
  - process first child, store result in target register t
  - process second child
  - target is now occupied by first result
  - allocate a new target register t’
    from available set for result of second child
  - apply node operation on t and t’
  - store result in target register
  - all initially available register now available again
  - result of internal node stored in target

Simple register allocation

b*d-4*a*c

4 temporaries



Weighted register allocation

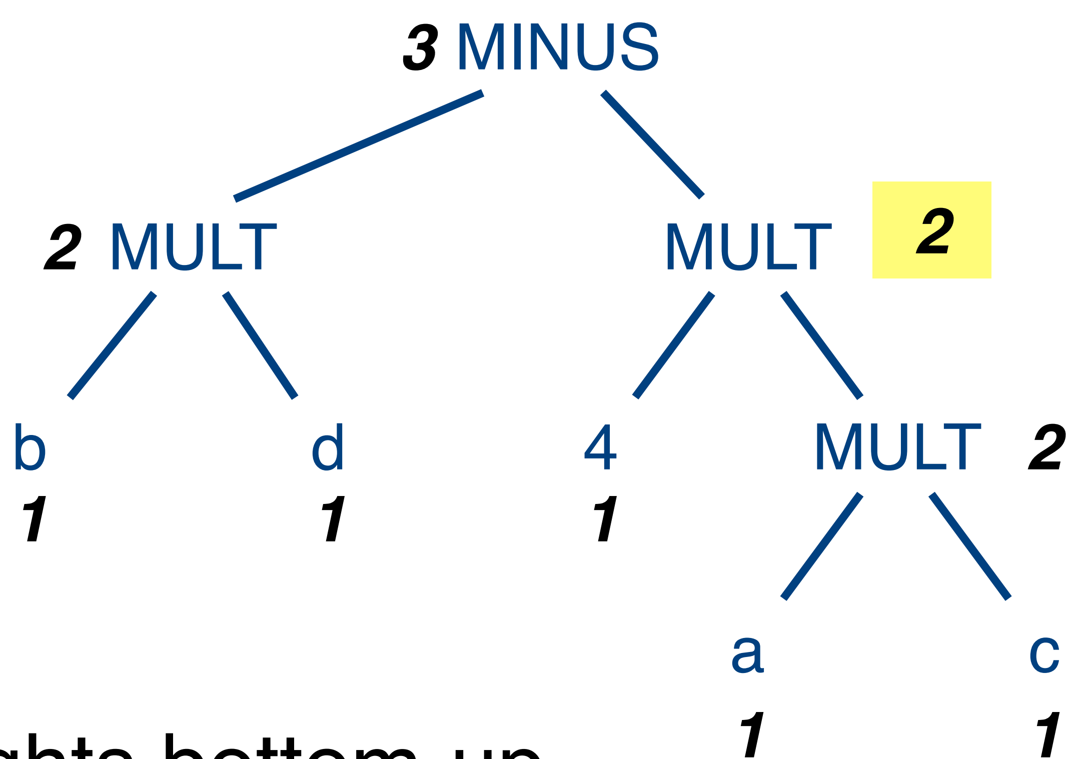
- Bottom-up: label each node with its weight
  - **weight** is the minimal number of temporaries needed
  - if w(e1) > w(e2) then w(e1+e2) := w(e1)
  - if w(e1) < w(e2) then w(e1+e2) = w(e2)
  - if w(e1) = w(e2) then w(e1+e2) = w(e1) + 1

- Top-Down: generate code for **heavier subtree first**

Weighted register allocation

b*d-4*a*c

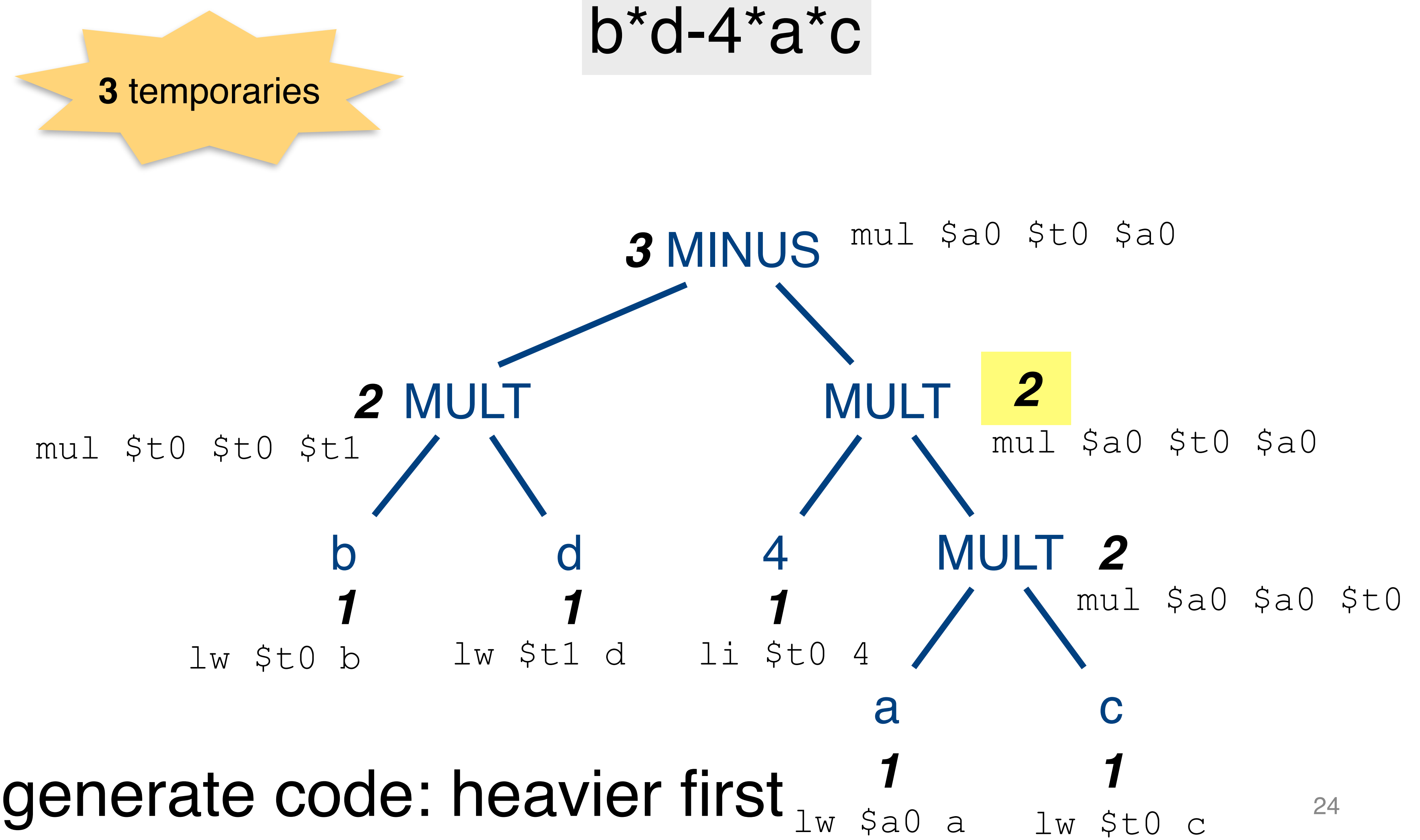
3 temporaries



assign weights bottom-up

Weighted register allocation

b*d-4*a*c



generate code: heavier first

Weighted register allocation

- Re-ordering subtree computations
- Register reuse
- Optimal under certain conditions
  - uniform instruction cost
  - “symbolic” trees
- Correct for subexpressions without side-effects

- What if we need more registers than available?

Simple **spilling** method

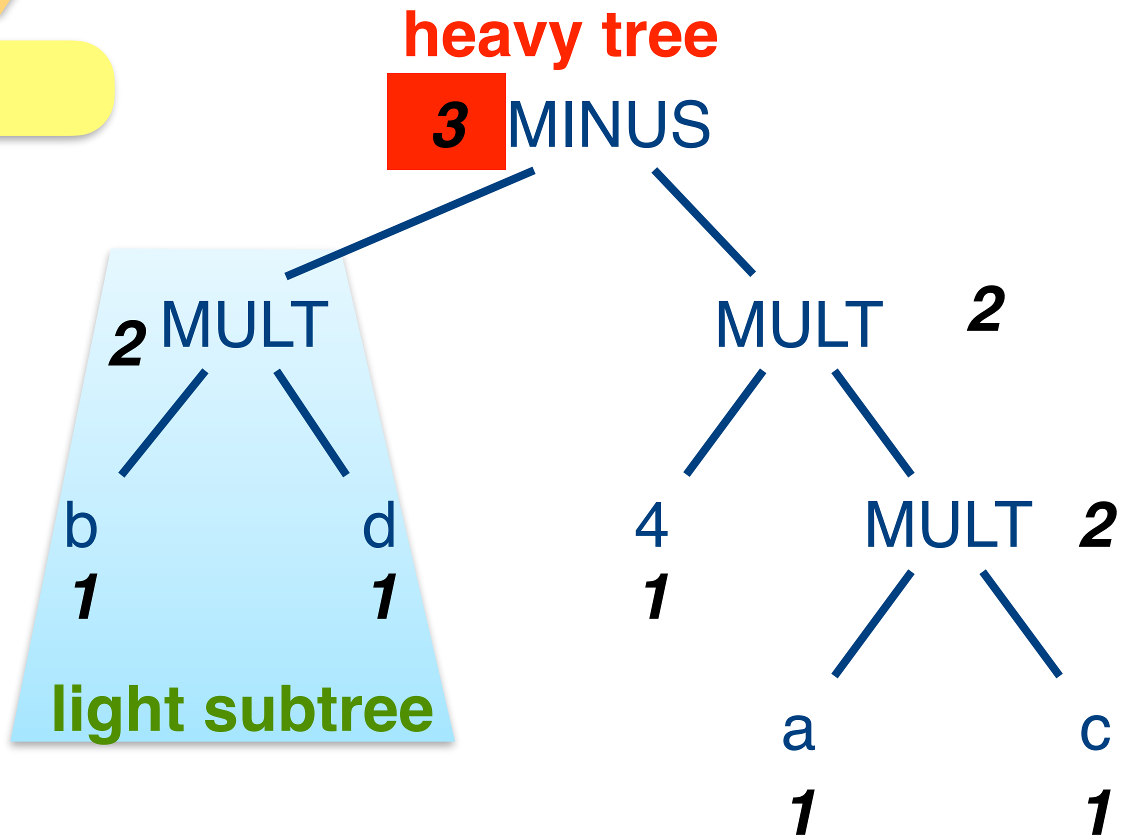
- A **heavy** tree contains a heavy subtree whose dependents are **light**
- Heavy tree needs more registers than available
- Generate code for the light tree
- Spill the result to memory and replace subtree by temporary
- Generate code for the resultant tree

Example: simple spilling

b*d-4*a*c

3 temporaries

2 registers



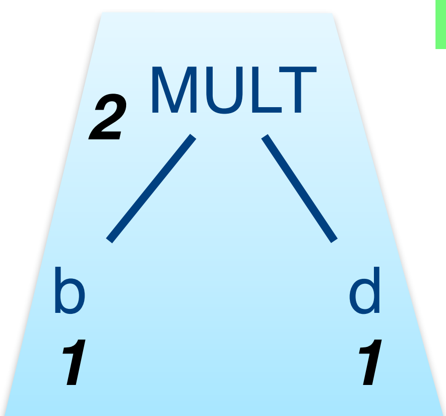
Example: simple spilling

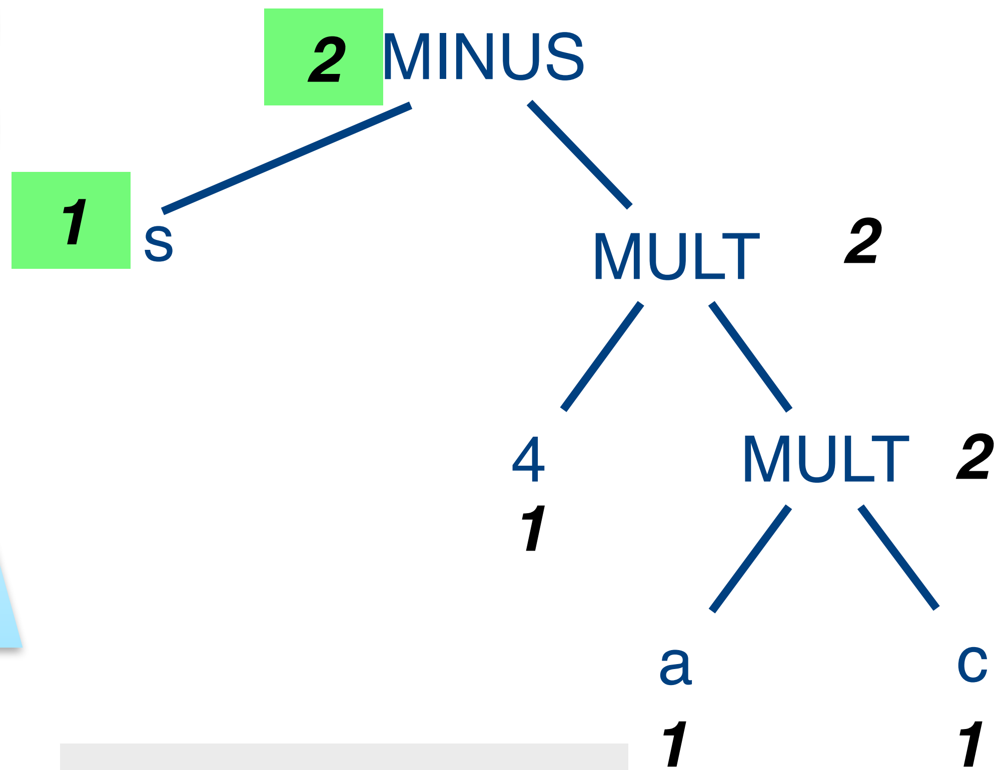
b*d-4*a*c

3 temporaries

2 registers

1 spill





s := b * d

x := s – 4 * a * c

Generalization

- More than two arguments for operators
- Function calls
- Register/memory operations
- Multiple effected registers
- Handle non-uniform instruction costs

- Share registers between different expressions
- Keep frequently accessed values in registers
  - example: loop counters

Global code generation

- Construct
    - **dependency graph**
    - **control flow graph**
    - **liveness** information for each variable
    - **interference graph** for variables

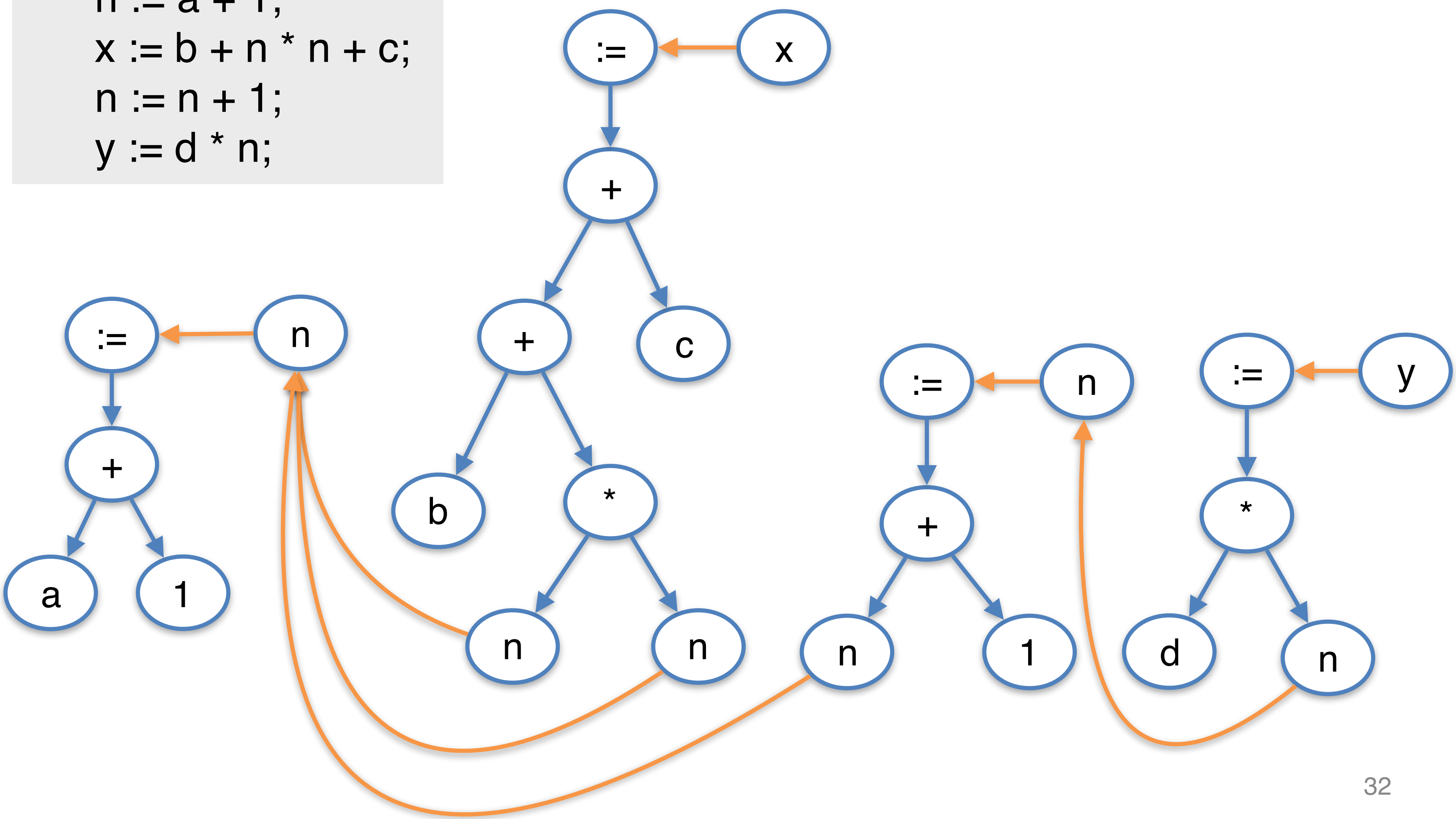
- Code selection: linearizations of dependencies
- Register allocation: interference graph coloring

Dependency graph

- Nodes are variable occurrences
- Edge between **use** and **definition** of a variable
  - for example, x := y + z
    - defines x
    - uses y and z
- Variable: definition vs declaration
  - definition is an assignment of a value to the variable
  - declaration is an association of a scope/type with a variable

Examples: dependency graph

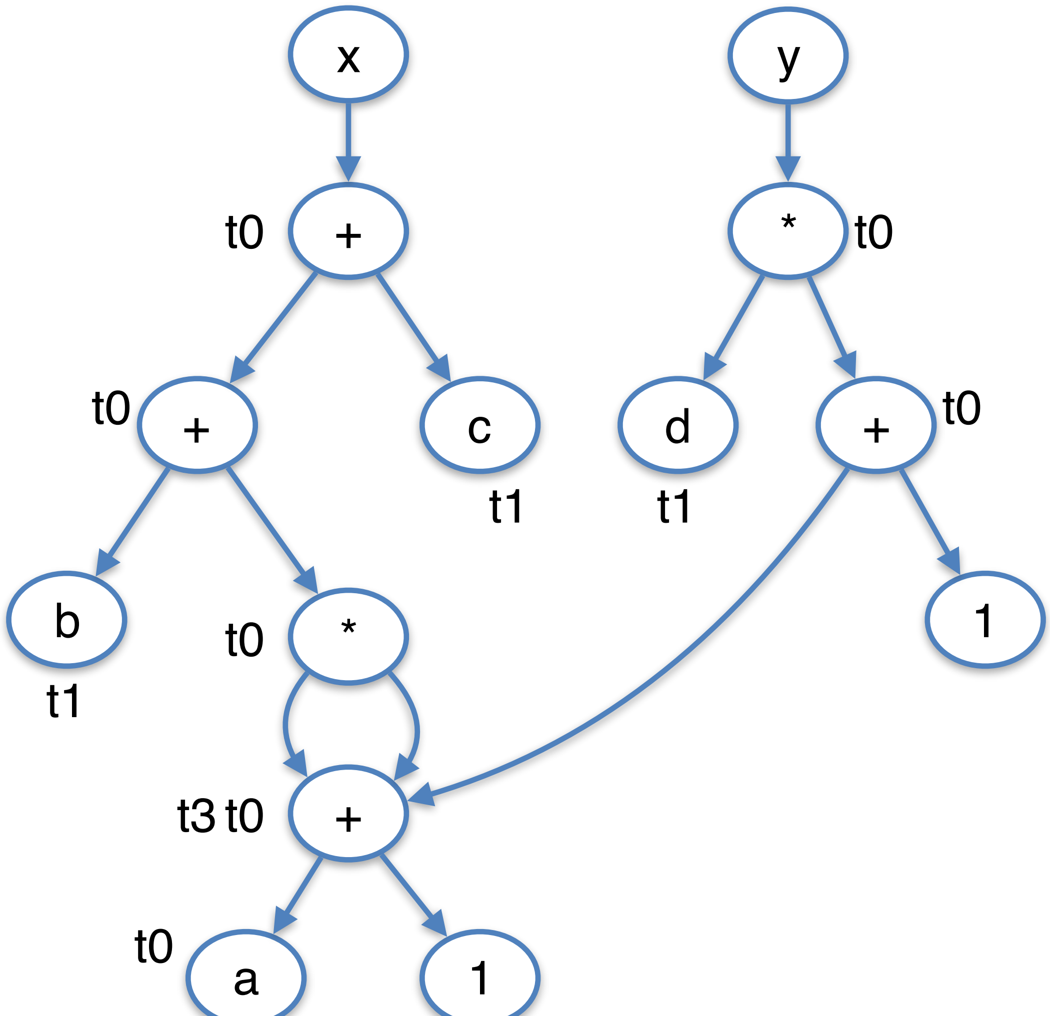
```
n := a + 1;
x := b + n * n + c;
n := n + 1;
y := d * n;
```



Examples: dependency graph

n := a + 1;
x := b + n * n + c;
n := n + 1;
y := d * n;

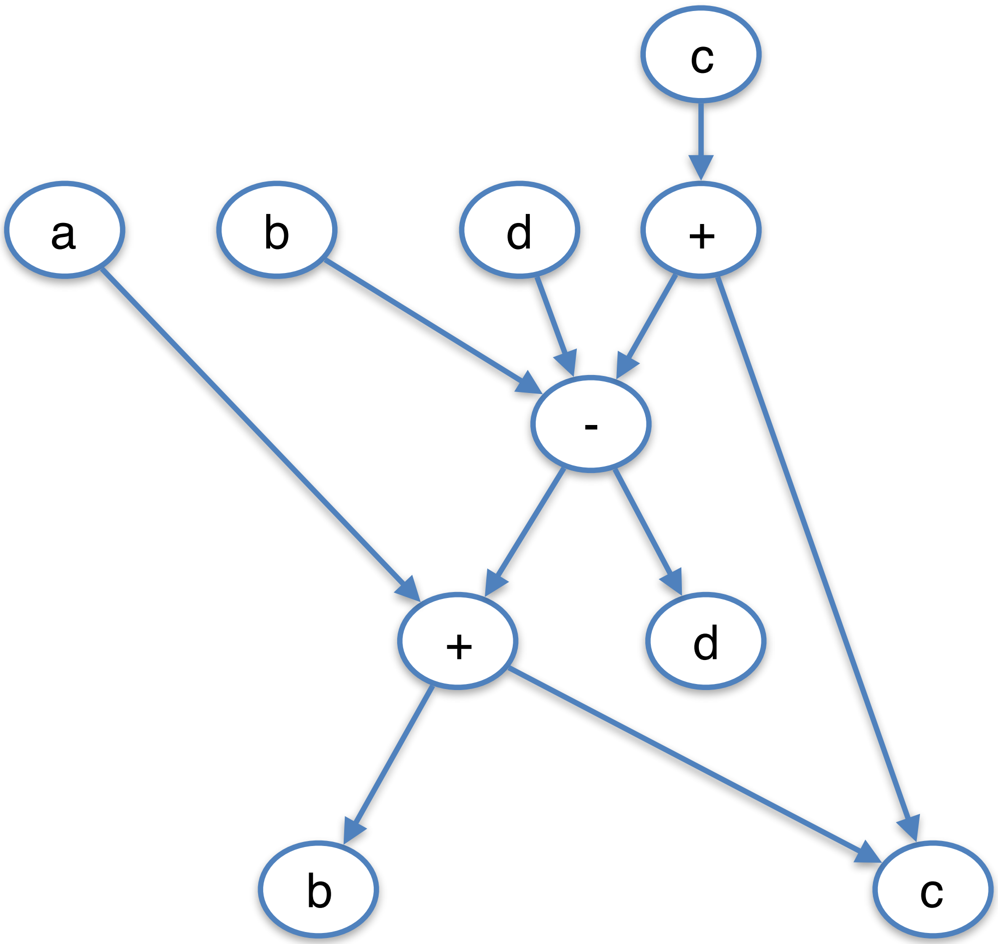
```
lw      t0  a
addi    t0  t0  1
move    t3  t0
mul     t0  t0  t0
lw      t1  b
add     t0  t1  t0
lw      t1  c
add     t0  t0  t1
sw      t0  x
addi    t0  t3  1
lw      t1  d
mul     t0  t1  t0
sw      t0  y
```



Examples: dependency graph

a = b + c
b = **a - d**
c = b + c
d = **a - d**

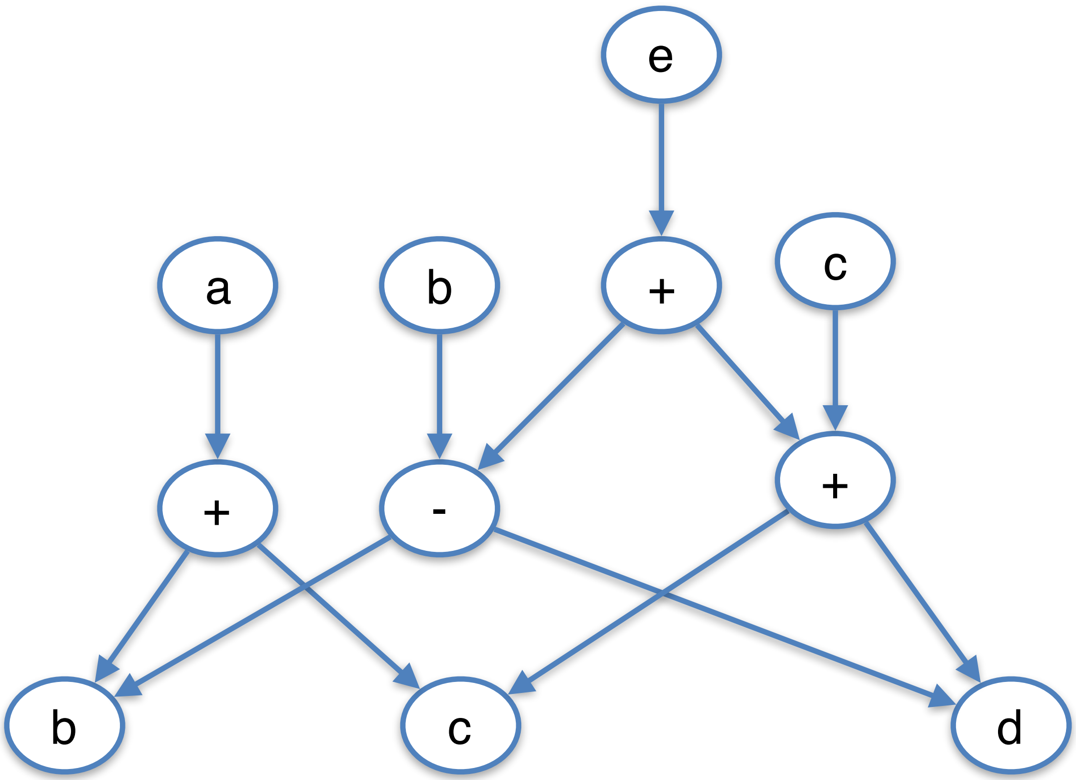
```
lw    t0 b
lw    t1 c
add   t0 t0 t1
sw    t0 a
lw    t2 d
sub   t0 t0 t2
sw    t0 b
sw    t0 d
add   t0 t0 t1
sw    t0 c
```



Examples: dependency graph

a = b + c
b = b - d
c = c + d
e = b + c

```
lw    t0  b
lw    t1  c
add   t2  t0  t1
sw    t1  a
lw    t3  d
sub   t0  t0  t2
sw    t0  b
sw    t0  d
add   t0  t0  t1
sw    t0  c
```



Dependencies

- Fundamental concept in programming languages
  - control dependencies
  - data dependencies
- Example: loop parallelization
  respect dependencies between loop iterations

```
#pragma omp parallel for
for (i=0; i<n; i++)
      a[i] = b[i]
```

✓

```
#pragma omp parallel for
for (i=1; i<n; i++)
      a[i] = a[i-1]
```

?

Basic Block

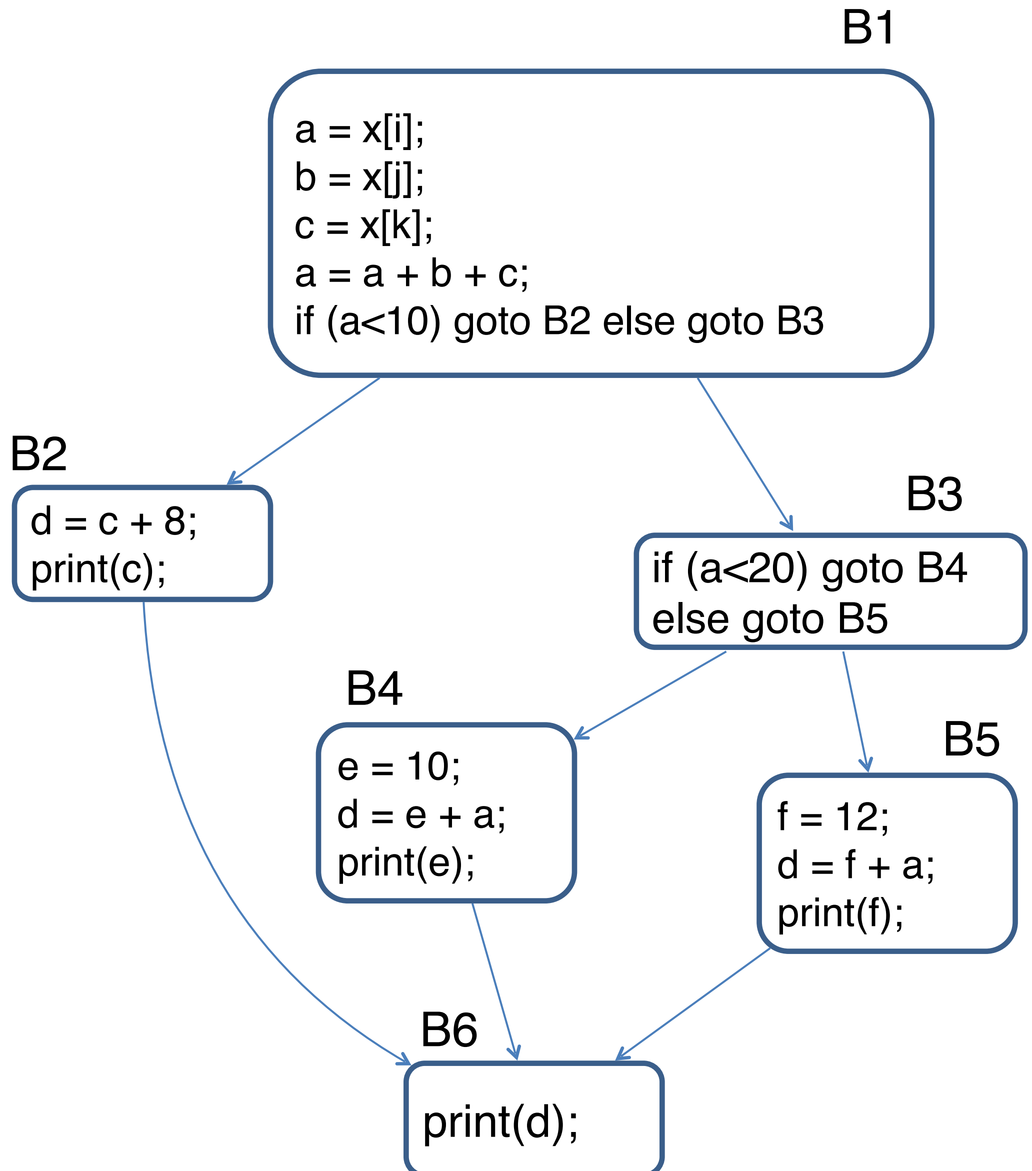
- A sequence of instructions
- Single entry
  - to first instruction
  - no jumps to the middle of the block
- Single exit
  - last instruction
  - no jumps (explicit/implicit) out of the block in the middle
- Code executes as a sequence from first instruction to last instruction without any jumps

Control flow graph (CFG)

- Nodes are basic blocks
- Edge from basic block B1 to block B2 when the last statement of B1 may jump to B2

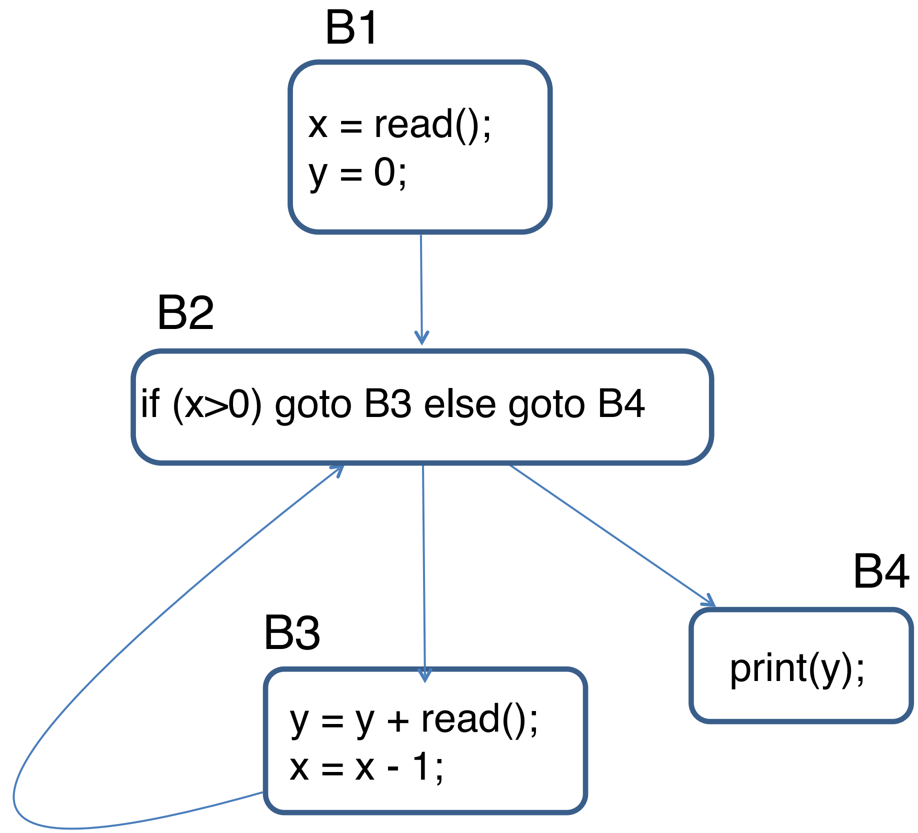
Example: CFG for conditionals

```
B1:  a = x[i];
     b = x[j];
     c = x[k];
     a = a + b + c;
     if (a<10) {
B2:      d = c + 8;
         print(c);
     } else
B3:  if (a<2o) {
B4:      e = 10;
         d = e + a;
         print(e);
     } else {
B5:      f = 12;
         d = f + a;
         print(f);
     }
B6:  print(d);
```



Example: CFG for loop

```
B1: x = read();
    y = 0;
B2: while (x > 0) {
B3:     y = y + read();
        x = x - 1;
    }
B4: print(y);
```



Variable liveness

- A variable x is **live** at program point L
if the value that x has at point L is used later in an execution
- Backwards analysis

	x is dead, y dead, z dead
y = 42	
	x is dead, y live, z dead
z = 73	
	x is dead, y live, z live
x = y + z	
	x is live, y dead, z dead
print(x);	
	x is dead, y dead, z dead

print(x)
uses x

Example: liveness

	{a}
b = a + 2	
	{b, a}
c = b * b	
	{a, c}
b = c + 1	
	{b, a}
return b * a	

Example: liveness

can we change the order between assignment to y and z?

	{ }
x = 1	
	{x}
y = x + 3	
	{x}
z = x * 3	
	{x, z}
x = x * z	

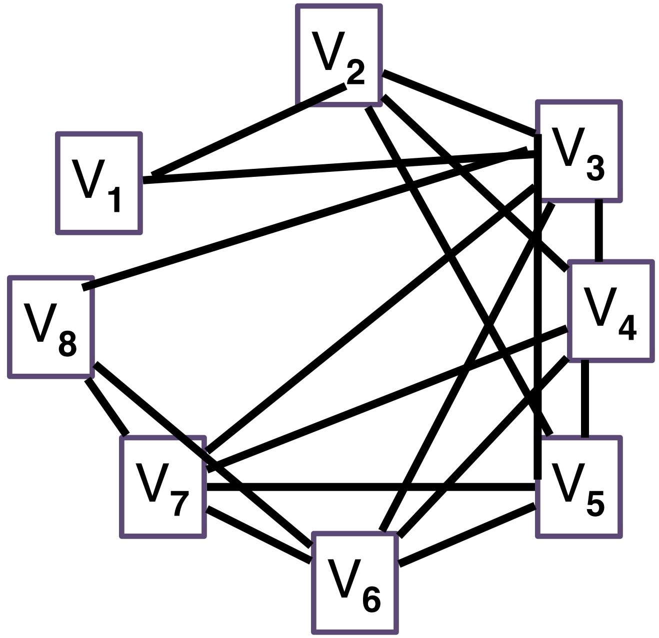
Interference graph

- Node for every variable
- Edges between variables that **interfere**

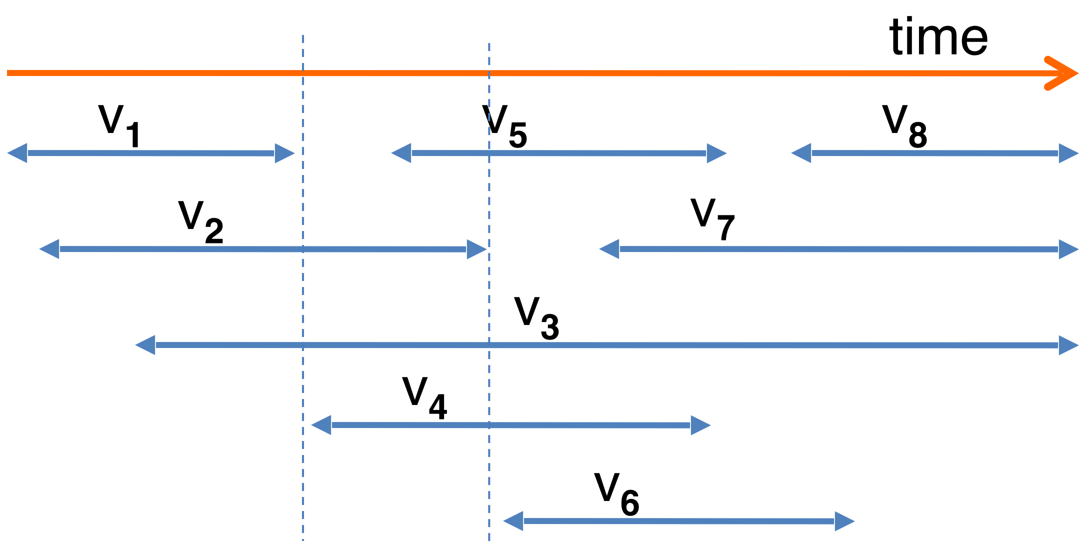
- Variables **interfere** if they are **live at the same time**
- Variables **do not interfere** if they have **disjoint live ranges**

Examples: interference graph

Interference graph

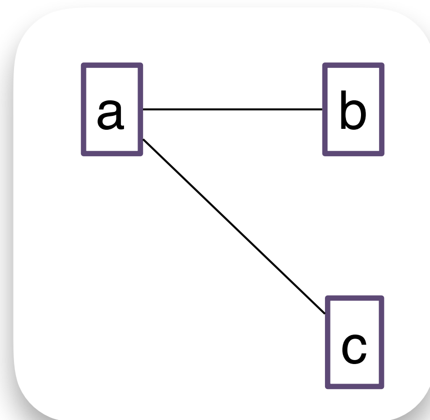


Variable liveness



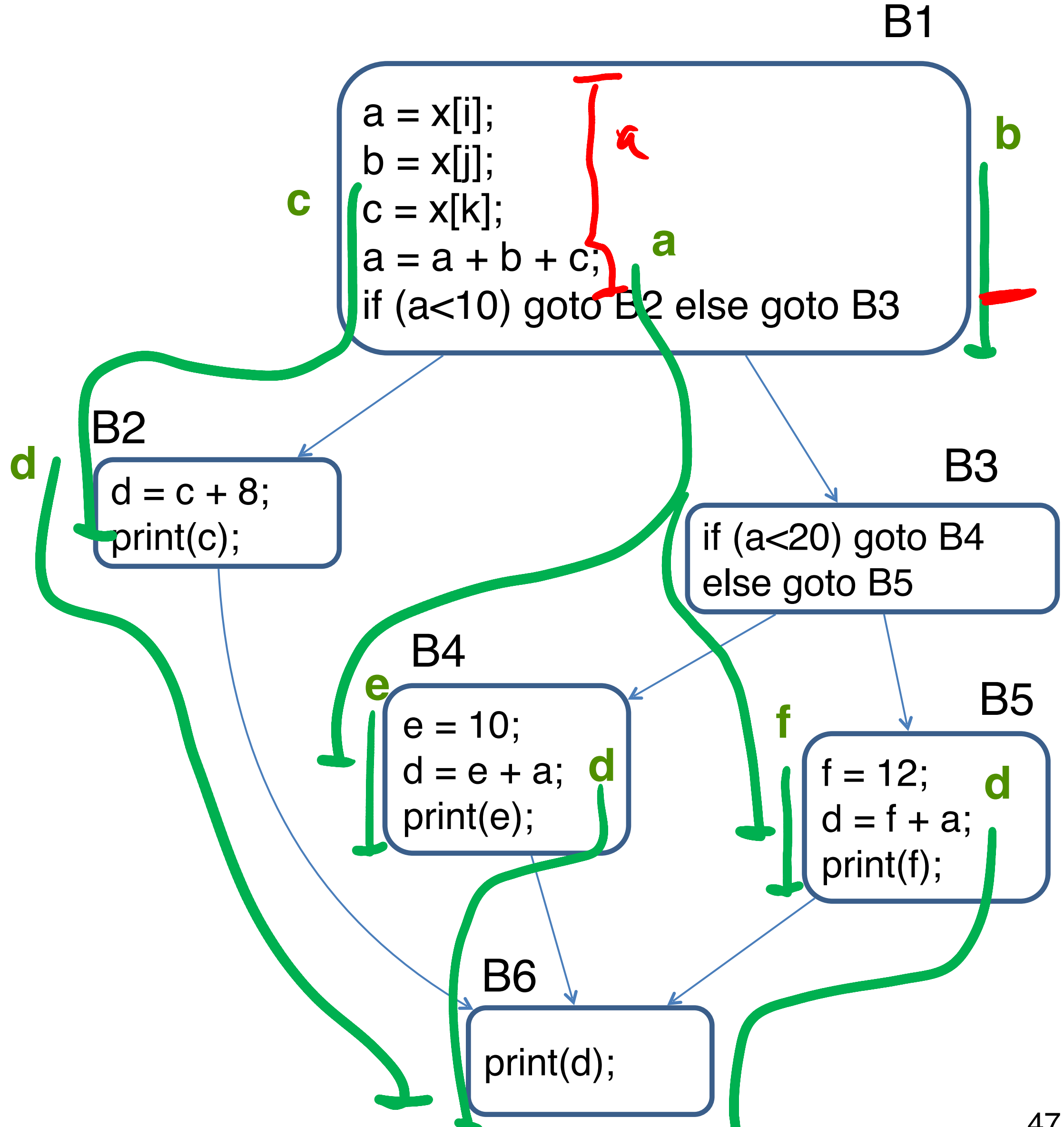
Examples: interference graph

```
b = a + 2

c = b * b

b = c + 1

return b * a
```

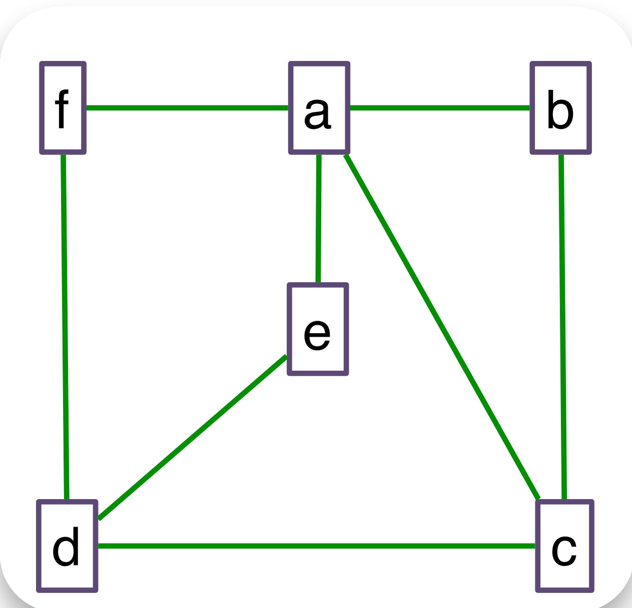


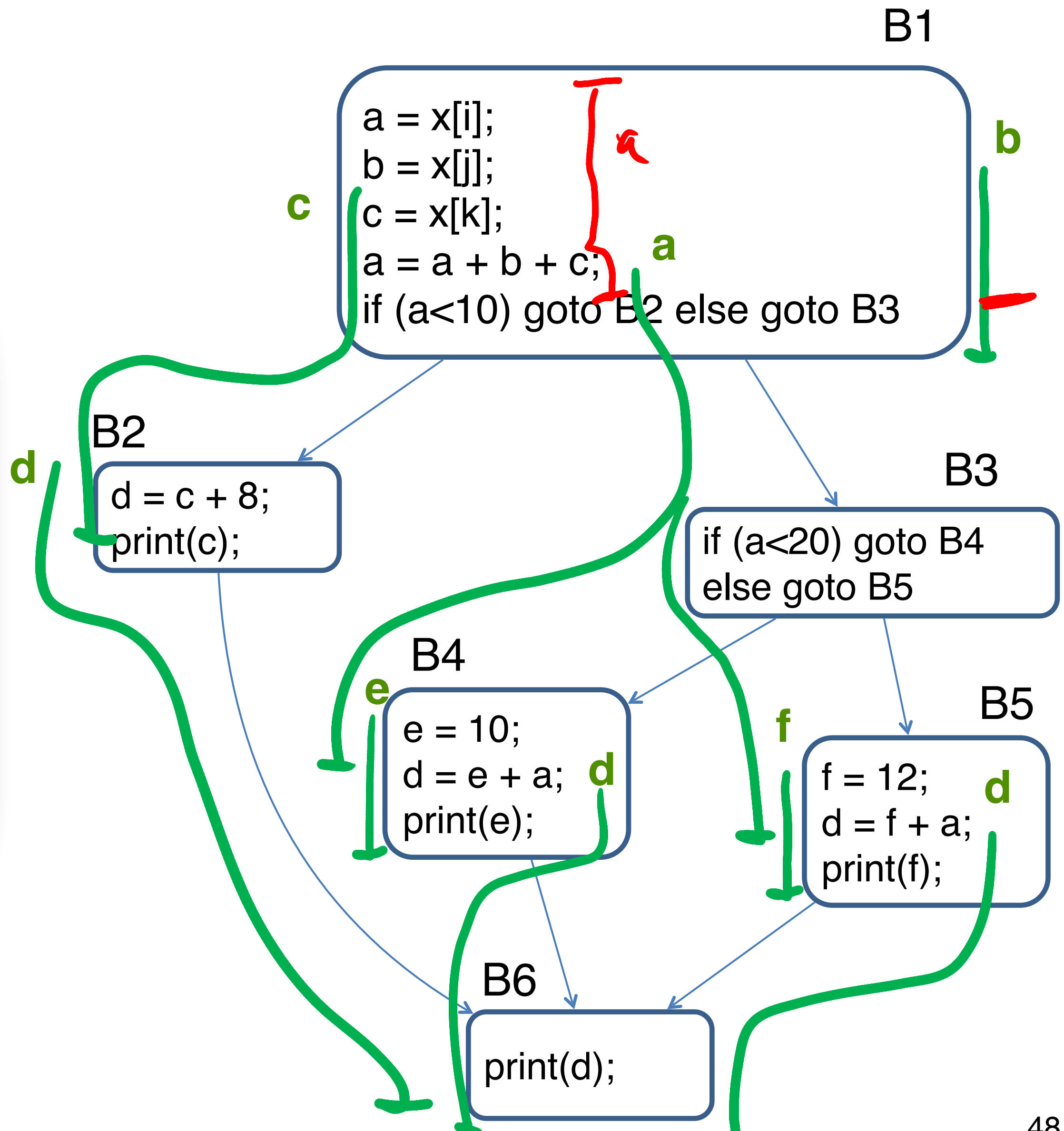
Example: variable liveness

```
B1:  a = x[i];
     b = x[j];
     c = x[k];
     a = a + b + c;
     if (a<10) {
B2:     d = c + 8;
        print(c);
     } else
B3:  if (a<2o) {
B4:     e = 10;
        d = e + a;
        print(e);
     } else {
B5:     f = 12;
        d = f + a;
        print(f);
     }
B6:  print(d);
```



Example: variable liveness





Register Allocation by Graph Coloring

- Reduction from register allocation to graph coloring
- Coloring of an **interference** graph
- Number of colors = number of registers
- Color of a node corresponds to the register assigned to the variable
- Variables that do not interfere with each other can be assigned the same register

Graph coloring

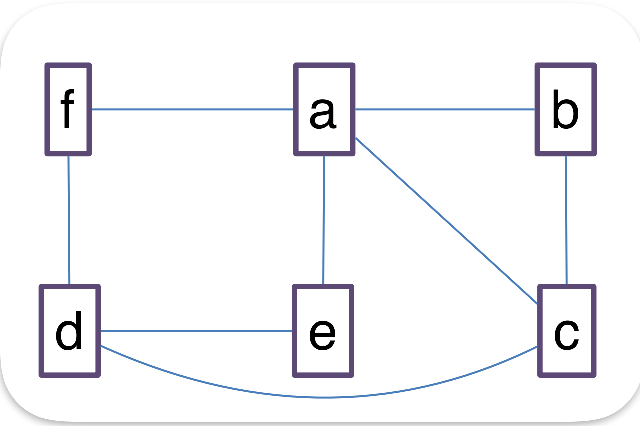
- Classical problem
- How to color all nodes of a graph
  using the **smallest** possible number of colors
  such that **no two adjacent nodes have the same color**

- NP-complete
- There are pretty good heuristic approaches

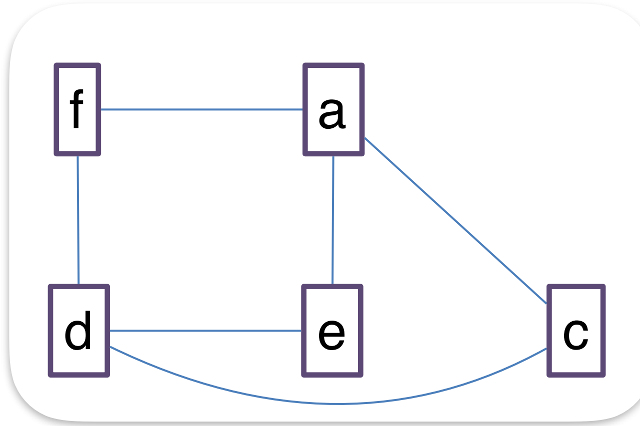
Heuristic graph coloring

- Easiest nodes to color are nodes with the lowest degree
  - **degree** of a node is the **number of neighbors**
  - **lowest degree** means **fewer conflicts**
- Color nodes one by one, coloring the easiest nodes last
- Algorithm at high-level
  - find the least connected node
  - remove least connected node from the graph
  - color the reduced graph recursively
  - re-attach the least connected node

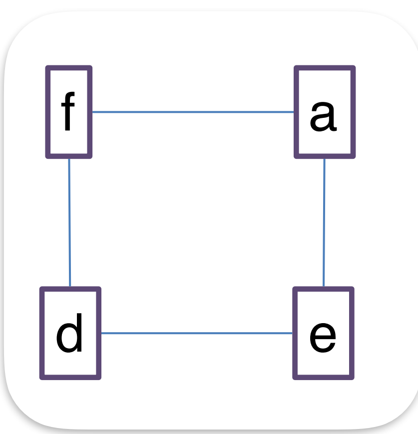
Example: heuristic graph coloring



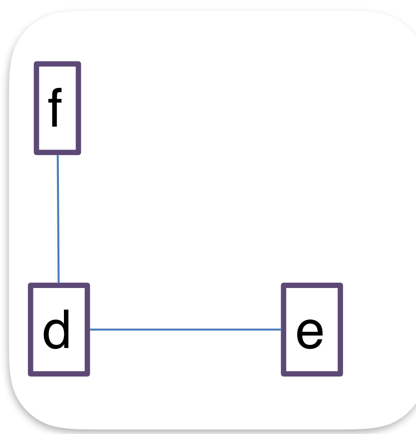
stack: ε



stack: b

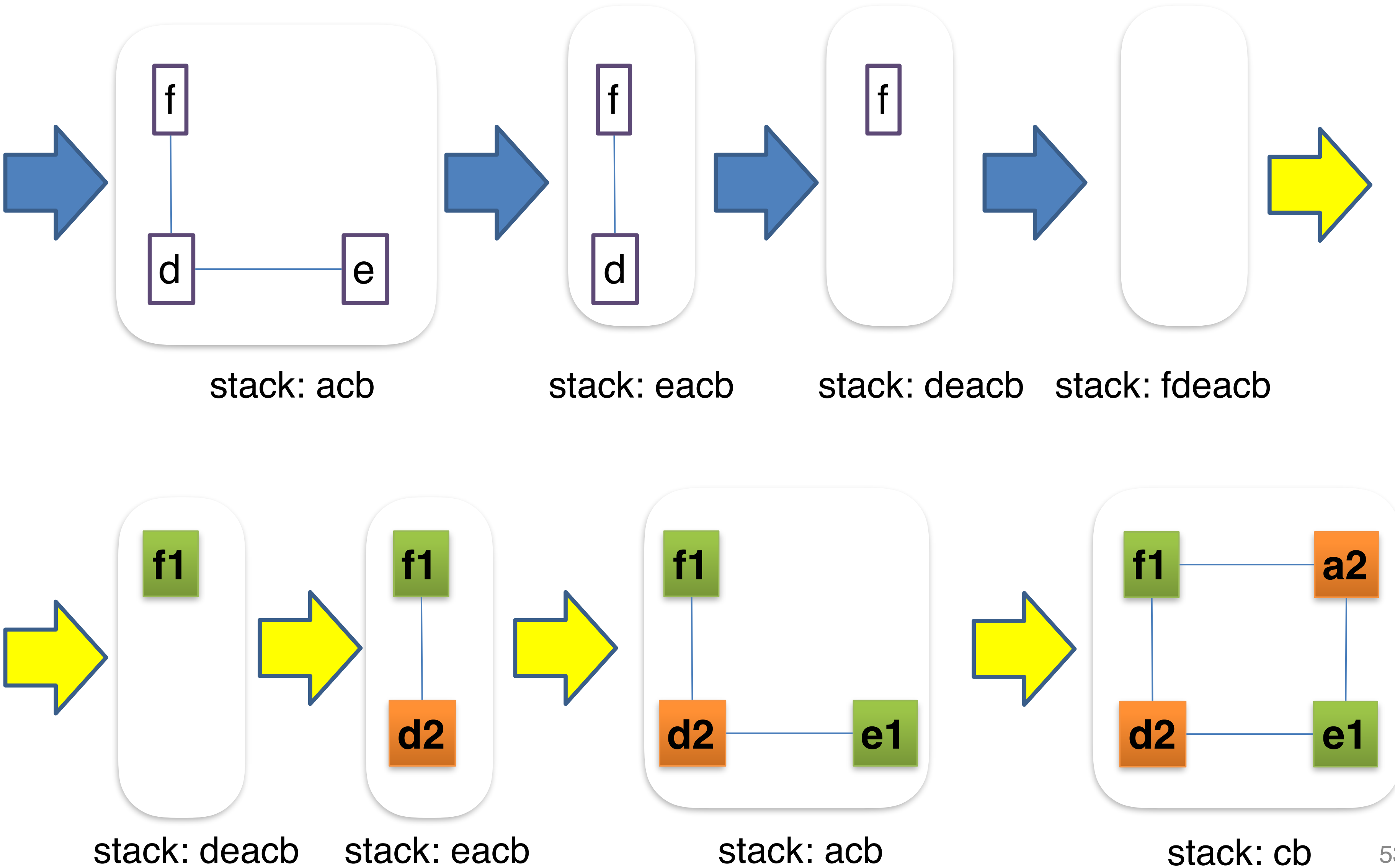


stack: cb

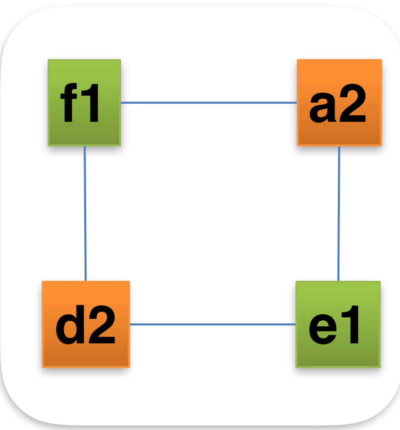


stack: acb

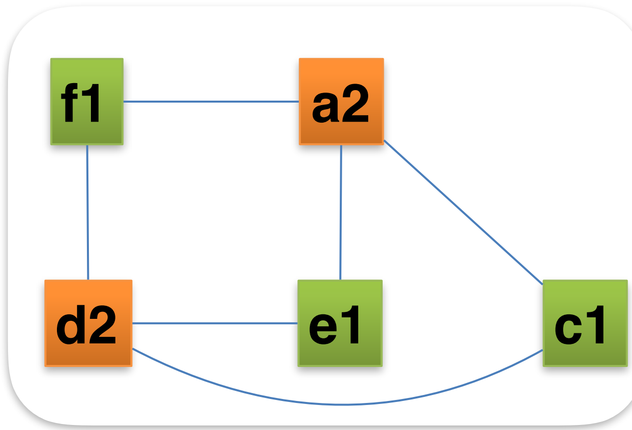
Example: heuristic graph coloring



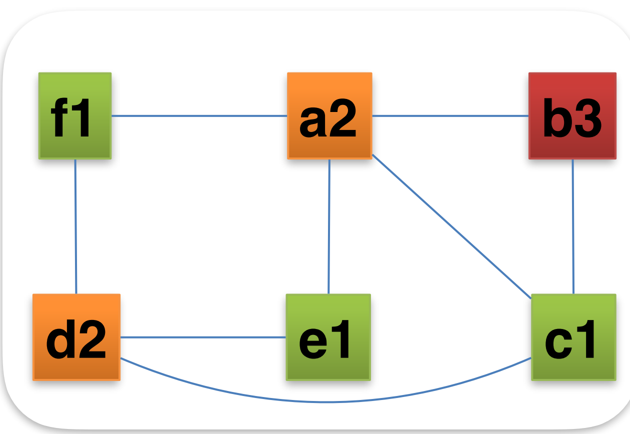
Example: heuristic graph coloring



stack: cb



stack: b



stack: ε

Result: 3 registers for 6 variables

Can we do with 2 registers?

Heuristic graph coloring

- Sources of **non-determinism** in the algorithm
  - choosing which of the (possibly many) nodes of lowest degree should be detached
  - choosing which of the available colors to use

Spilling

- If every node has at least K neighbors, the graph cannot be colored with K colors
- Which node to spill?
  - try to pick node not used much, not in inner loop
- How to spill?
  - rewrite code to spill, recompute liveness, and try to color again
- **Precolored nodes** for registeres with designated uses
  - infinite degree: cannot be spilled or coaleased

Coalescing

- Eliminate register-to-register moves
  - move r1 r2
  - r1 and r2 do not interfere
  - merge nodes r1 and r2 and unify the sets of neighbors
- Might fail to color the graph (why?)
- Conservative: merge nodes if the resulting node has **fewer than K neighbors with degree K** (in the resulting graph)

Why **not** graph coloring?

- Interference graph is too expensive to build
- Flexibility is more important than efficiency
  - spill code placement
  - aliases and overlapping register classes

Summary: code generation and optimizaton

- Multiple passes: register allocation, instruction selection, instruction scheduling,…
- Well-defined goals and clear specification for each pass
- Reduction to known problems/algorithms
- Easier to write, maintain, prove correctness, achieve optimality
- In reality…
    - correctness depends on (implicit) invariants between phases
    - non-monotonic effect on performance: one pass counteracts another
    - missed optimization opportunities
    - peephole optimizations

Superoptimization

- Exhaustive search in the space of (small) programs for finding optimal code sequences
  - often counterintuitive results, not what a human would write
  - generated code can be very efficient
  - generate/test paradigm

```
; n in register %ax

   cwd              ; convert to double word:
                    ;   (%dx,%ax) = (extend_sign(%ax), %ax)
   negw %ax         ; negate: (%ax,cf) = (-%ax,%ax != 0)
   adcw %dx,%dx     ; add with carry: %dx = %dx + %dx + cf

; sign(n) in %dx
```

Compiler correctness

- Generated code correctly implements the source code

- Concerns with correctness of translation
- Different from code correctness
- Compilers do not guarantee to generate “correct code”
- For example, consider a program that throws a NullPointerException at runtime

Compiler design goals

- Correctness: generated code correctly implements the source code
- Metrics for generated code
  - performance/speed
  - size
  - power consumption/energy efficiency
  - security/reliability
  - easy to debugging
  - portable
- Metrics for compilers
  - fast/efficient compilation
  - good error reporting

Optimizations

- “Optimal code” is out of reach
  - many problems are undecidable or too expensive
  - use approximation and/or heuristics
  - optimizations must guarantee correctness of compiler
  - should (mostly) improve code
- Majority of compilation time is spent in optimizations
- Leverage compile-time information to save work at runtime (precompute)

Example optimizations

- Loop optimizations: hoisting, unrolling
- Peephole optimizations
- Constant propagation
- Dead code elimination
- Instruction selection: convert IR to machine instructions
- Instruction scheduling: reorder instructions
- Register allocation: assign variables to memory locations
  - optimal register assignment is NP-Complete
  - in practice, known heuristics perform well
- Modern architectures include challenging features
  - multicore
  - memory hierarchies

Compiler construction tools

- Parts of the compiler are automatically generated from specification
  - simplify compiler construction
  - less error prone
  - more flexible
  - use of pre-canned tailored code
  - use of dirty programming tricks
  - reuse of specification

Compiler construction tools

- Lexical analysis generators
  - lex, flex, jflex, antlr
- Parser generators
  - yacc, bison, java_cup, antlr
- Syntax-directed translators
- Dataflow analysis engines

Summary

- Compiler is a **program** that translates code from **source** language to **target** language
- Compilers play a central role
  - bridge from high-level programming languages to machines
  - many useful techniques
  - many useful tools (e.g., lexer/parser generators)
- Compiler vs Interpreter
- Just-In-Time compilation
- Time of events: compiler, linker, loader, runtime
- Bootstrapping a compiler
- Compiler constructed from modular phases

Meanwhile, in the real world

- **new compilers** for **new languages**
- new compilers for old languages
  - e.g., Java->JavaScript
- new uses of compiler technology
- …

HOW TO PREPARE FOR THE EXAM?

How to prepare for the exam?

- Make sure you understand the material covered in
  - slides on QM+ for lectures and tutorials
  - exercises and their solutions
  - programming assignments
  - cool reference manual
- Solve the **mock exam** and compare to model answers
- Revision lecture: Tuesday, 24 April Arts Two 3.16
- Solve questions from past exams
- How to get more help?
  - use the forum on qm+
  - office hours: email me to arrange

What is **not** for the exam?

- Bottom-up parsing algorithms
- Tail recursion
- Register allocation via graph coloring
- Static vs dynamic scoping
- Cool operational semantics rules [section 13 of Cool Reference Manual]

Structure of the exam

- Answer ALL FOUR questions

Question 1 [30 marks] Parsing

Question 2 [20 marks] Classify events

Question 3 [20 marks] Concepts in compilers

Question 4 [30 marks] Modify Cool Compiler

Examples: modify cool compiler

- Add new control flow constructs
  - **repeat** expr **until** expr **teaper**
  - **for** i ← [0..n] **do** expr **rof**
- Add data types
  - records
  - arrays
- Change OO features
  - inheritance
  - interfaces / abstract / virtual

THANK YOU!