CMPSC 160
Translation of Programming Languages

Lectures 16: Code Generation: Three-Address Code + Register Allocation

Three Address Code

- Is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations.
- Each three address code instruction has at most three operands and is typically a combination of assignment and a binary operator
- In three address code, there is at most one operator on the right side of an instruction. That is no built- up arithmetic expressions are permitted Example : x + y * z
- t1 = y * z; t2 = x + t1; where t1 and t2 are compiler-generated temporary names. Temporary variables store the results at the internal nodes in the AST

Three-Address Code Instructions

- Assignments
  - `x := y`
  - `x := y op z` `op`: binary arithmetic or logical operators
  - `x := op y` `op`: unary operators (unary minus, negation, integer to float conversion)
- Branch
  - `goto L` Execute the statement with labeled L next
- Conditional Branch
  - `if x relop y goto L` `relop`: <, >, =, <=, >=, ==, !=
    - if the condition holds we execute statement labeled L next
    - if the condition does not hold we execute the statement following this statement next

Three-Address Code

```
if (x < y)
  x = 5*y + 5*y/3;
else
  y = 5;
x = x + y;
```

Variables can be represented with their locations in the symbol table

```
        if x < y goto L1
        goto L2
L1:     t1 := 5 * y
        t2 := 5 * y
        t3 := t2 / 3
        x := t1 + t3
        goto L3
L2:     y := 5
L3:     x := x + y
```

Temporaries: temporaries correspond to the internal nodes of the syntax tree

Three-Address Code vs. Stack-Based Code

- Three-Address Code:
  - Good: Statement is “self contained” in that it has the inputs, outputs, and operation all in one “instruction”
  - Bad: Requires lots of temporary variables
  - Bad: Temporary variables have to be handled explicitly

- Stack-Based Code:
  - Good: No temporaries, everything is kept on the stack
  - Good: It is easy to generate code for this
  - Bad: Requires more instructions to do the same thing

Three-Address Code

Attributes:	*E.place*: location that holds the value of expression *E* (this is the temporary variable that will hold the value of the expression) *E.code*: sequence of instructions that are generated for *E*
Procedures:	newtemp(): Returns a new temporary each time it is called gen(): Generates instruction (have to call it with appropriate arguments) lookup(id.*name*): Returns the location of id from the symbol table || denotes concatenation

Productions

$S \rightarrow \text{id} := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow (E_1)$

$E \rightarrow - E_1$

$E \rightarrow \text{id}$

$E \rightarrow \text{num}$

Three-Address Code

Attributes:	*E.place*: location that holds the value of expression *E* (this is the temporary variable that will hold the value of the expression) *E.code*: sequence of instructions that are generated for *E*
Procedures:	newtemp(): Returns a new temporary each time it is called gen(): Generates instruction (have to call it with appropriate arguments) lookup(id.*name*): Returns the location of id from the symbol table

Productions	Semantic Rules

$S \rightarrow \text{id} := E$	id.*place* ← lookup(id.*name*);
	S.code ← *E.code* || gen(id.*place* ‘:=‘ *E.place*);

$E \rightarrow E_1 + E_2$	*E.place* ← newtemp();
	E.code ← E_1.*code* || E_2.*code* || gen(*E.place* ‘:=‘ E_1.*place* ‘+’ E_2.*place*);

$E \rightarrow E_1 * E_2$	*E.place* ← newtemp();
	E.code ← E_1.*code* || E_2.*code* || gen(*E.place* ‘:=‘ E_1.*place* ‘*’ E_2.*place*);

$E \rightarrow (E_1)$	*E.code* ← E_1.*code*; *E.place* ← E_1.*place;*

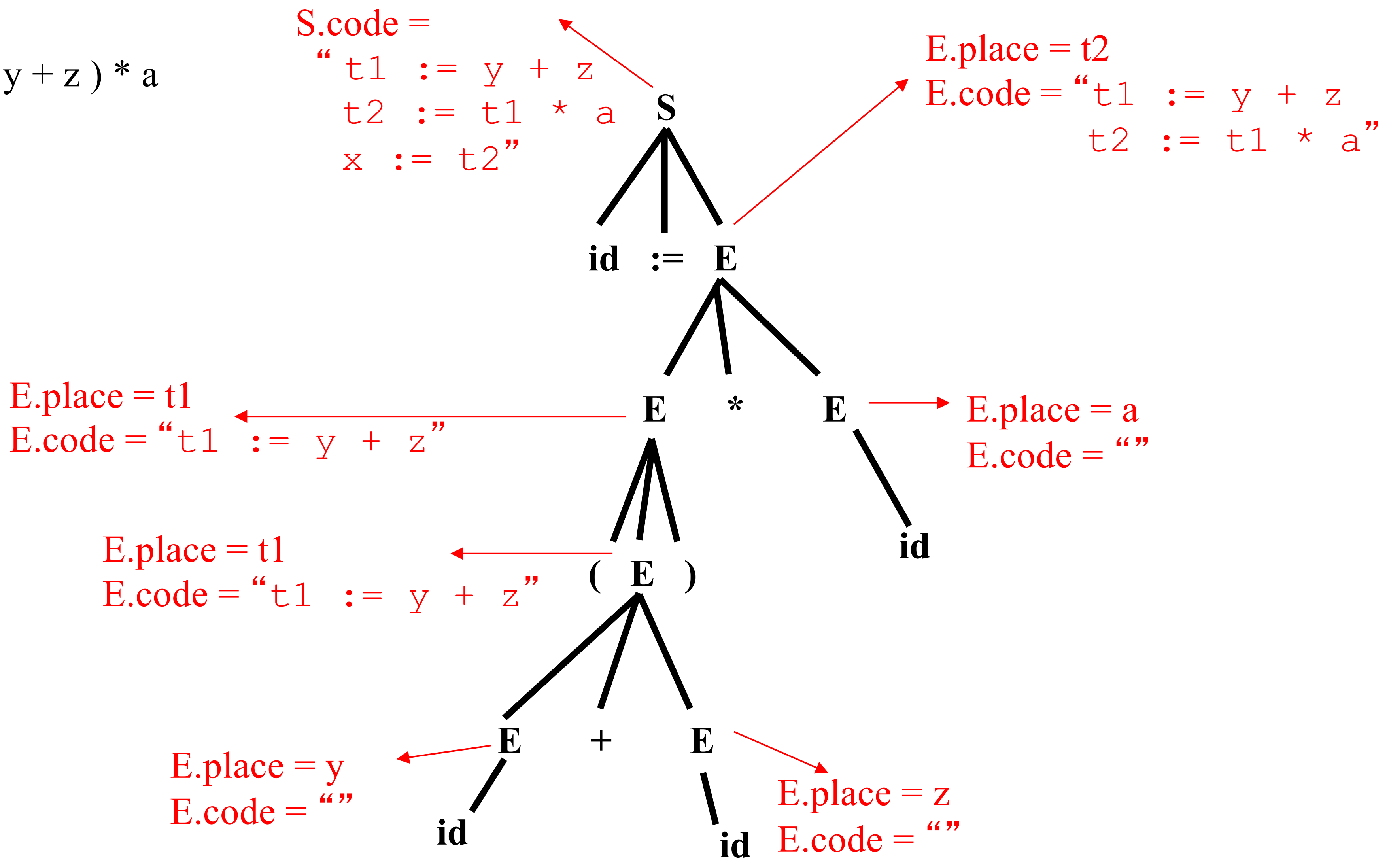
$E \rightarrow - E_1$	*E.place* ← newtemp();
	E.code ← E_1.*code* || gen(*E.place* ‘:=‘ ‘`uminus`’ E_1.*place*);

$E \rightarrow \text{id}$	*E.place* ← lookup(id.*name*); *E.code* ← ‘’ (*empty string*)

$E \rightarrow \text{num}$	*E.place* ← newtemp();
	E.code ← gen(*E.place* ‘:=‘ num.*value*);

Example

x := (y + z) * a



S.code =
“ t1 := y + z
t2 := t1 * a
x := t2”

E.place = t2
E.code = “t1 := y + z
t2 := t1 * a”

E.place = t1
E.code = “t1 := y + z”

E.place = a
E.code = “”

E.place = t1
E.code = “t1 := y + z”

E.place = y
E.code = “”

E.place = z
E.code = “”

Code Generation for Boolean Expressions

- Two approaches
  - Numerical representation
  - Implicit representation
- Numerical representation
  - Use 1 to represent true, use 0 to represent false
  - For three-address code store this result in a temporary
  - For stack machine code store this result in the stack
- Implicit representation
  - For the boolean expressions which are used in flow-of-control statements (such as if-statements, while-statements etc.) boolean expressions do not have to explicitly compute a value, they just need to branch to the right instruction
  - Generate code for boolean expressions which branch to the appropriate instruction based on the result of the boolean expression

Numerical Representation of Boolean Expressions

Input boolean expression: `x < y and a == b`

Three address code:
Instructions 100-103 are for `x < y`
Instructions 104-107 are for `a == b`

```
100     if x < y goto 103
101     t1 := 0
102     goto 104
103     t1 := 1
104     if a = b goto 107
105     t2 := 0
106     goto 108
107     t2 := 1
108     t3 := t1 and t2
```

- These are the locations of the instructions, they are not labels.
- We could generate code using labels too

Stack machine code:
Instructions 100-105 are for `x < y`
Instructions 106-111 are for `a == b`

```
100     load x
101     load y
102     if_cmplt 105
103     push 0
104     goto 106
105     push 1
106     load a
107     load b
108     if_cmpeq 111
109     push 0
110     goto 112
111     push 1
112     and
```

Implicit Representation of Boolean Expressions

These are the locations of three-address code instructions, they are not labels

Numerical representation:

```
100    if x < y goto 103
101    t1 := 0
102    goto 104
103    t1 := 1
104    if a = b goto 107
105    t2 := 0
106    goto 108
107    t2 := 1
108    t3 := t1 and t2
```

Input boolean expression:

`x < y and a == b`

Implicit representation:

```
         if x < y goto L1
         goto LFalse
L1:      if a = b goto LTrue
         goto LFalse
LTrue:

LFalse:
```

These labels will be generated later on, and will be inserted to the corresponding places

Boolean Expressions: Implicit Representation, Three-Address Code

Attributes : *E.code*: sequence of instructions that are generated for *E*
E.false: instruction to branch to if *E* evaluates to false
E.true: instruction to branch to if *E* evaluates to true
(*E.code* is synthesized whereas *E.true* and *E.false* are inherited)
id.*place*: location for id

Productions	Semantic Rules
$E \rightarrow E_1$ and E_2	$E_1.true \leftarrow$ newlabel(); $E_1.false \leftarrow E.\ false$; (*short-circuiting*) $E_2.true \leftarrow E.\ true$; $E_2.false \leftarrow E.\ false$; $E.code \leftarrow E_1.code \,\|\|\,$ gen($E_1.true$ ':') $\|\|\, E_2.code$;
$E \rightarrow E_1$ or E_2	$E_1.true \leftarrow E.true$; (*short-circuiting*) $E_1.false \leftarrow$ newlabel(); $E_2.true \leftarrow E.\ true$; $E_2.false \leftarrow E.\ false$; $E.code \leftarrow E_1.code \,\|\|\,$ gen($E_1.false$ ':') $\|\|\, E_2.code$;

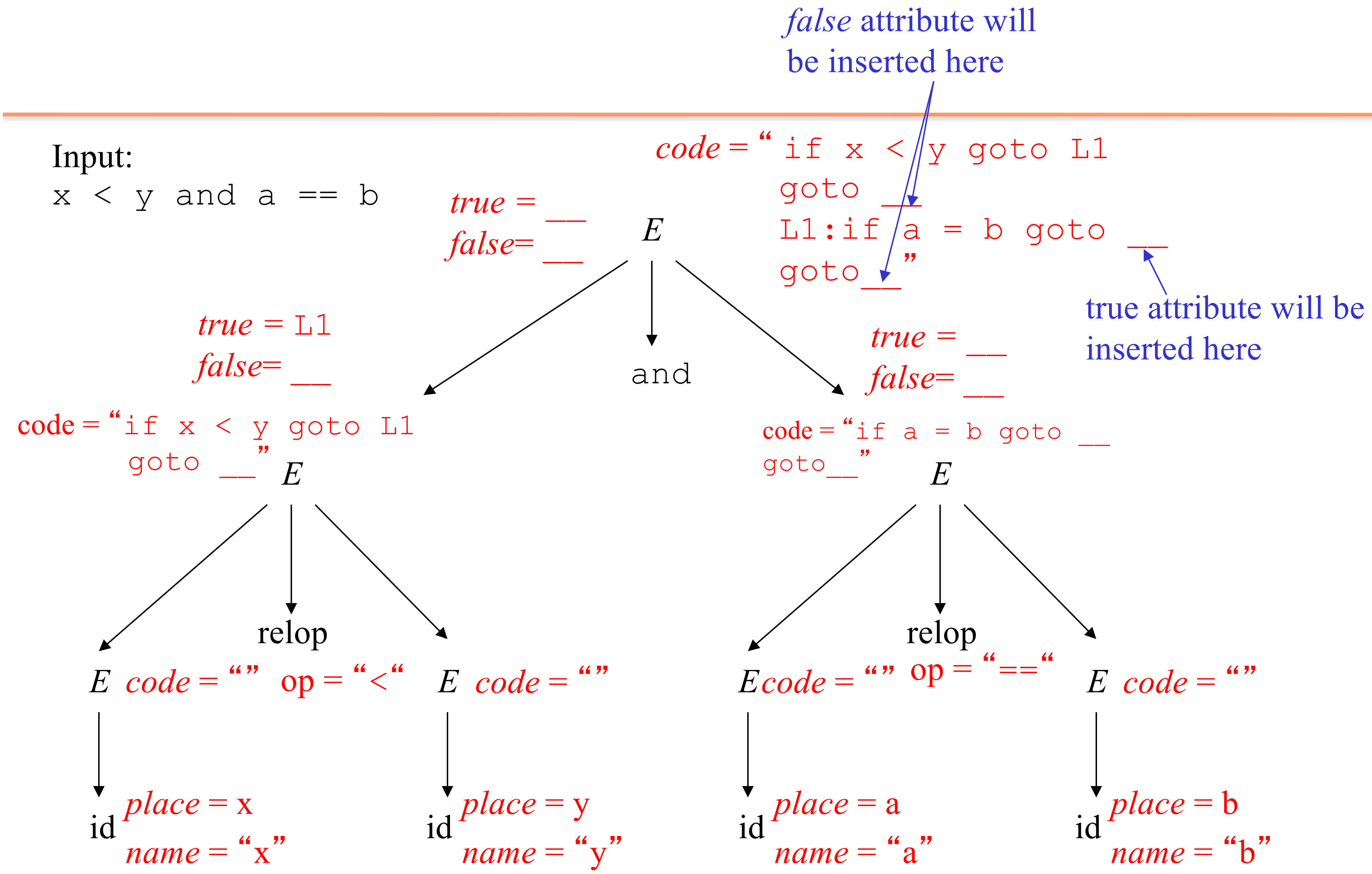
Boolean Expressions: Implicit Representation, Three-Address Code (continued)

Attributes : *E.code*: sequence of instructions that are generated for *E*
E.false: instruction to branch to if *E* evaluates to false
E.true: instruction to branch to if *E* evaluates to true
id.*place*: location for id

Productions	Semantic Rules
$E \rightarrow$ not E_1	$E_1.true \leftarrow E.false$; $E_1.false \leftarrow E.\ true$; $E.code \leftarrow E_1.code$;
$E \rightarrow E_1$ relop E_2	$E.code \leftarrow E_1$.code || E_2.code || gen(‘`if`’ $E_1.place$ relop.*op* $E_2.place$ ‘`goto`’ $E.true$) || gen(‘`goto`’ $E.false$);
$E \rightarrow$ true	gen(`` `goto’ `` $E.true$);
$E \rightarrow$ false	gen(`` `goto’ `` $E.false$);

Three-Address Code, Implicit Representation

Input:
`x < y and a == b`



false attribute will be inserted here

code = "
```
if x < y goto L1
goto __
L1:if a = b goto __
goto__
```
"

true attribute will be inserted here

true = __
false= __
E

and

true = L1
false= __

code = "`if x < y goto L1`
`goto __`"
E

true = __
false= __

code = "`if a = b goto __`
`goto__`"
E

relop

E *code* = ""

op = "<"

E *code* = ""

relop

E *code* = ""

op = "=="

E *code* = ""

id *place* = x
name = "x"

id *place* = y
name = "y"

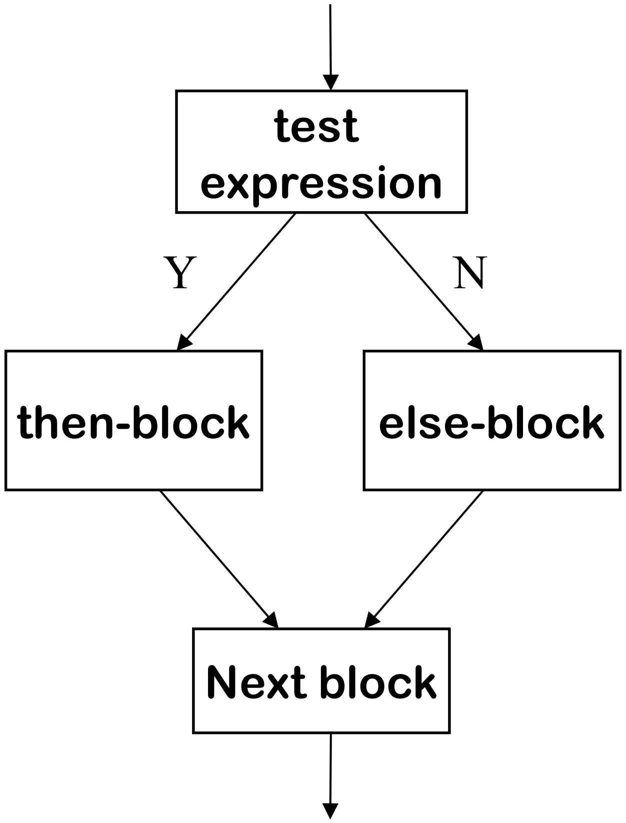
id *place* = a
name = "a"

id *place* = b
name = "b"

Flow-of-Control Statements

If-then-else

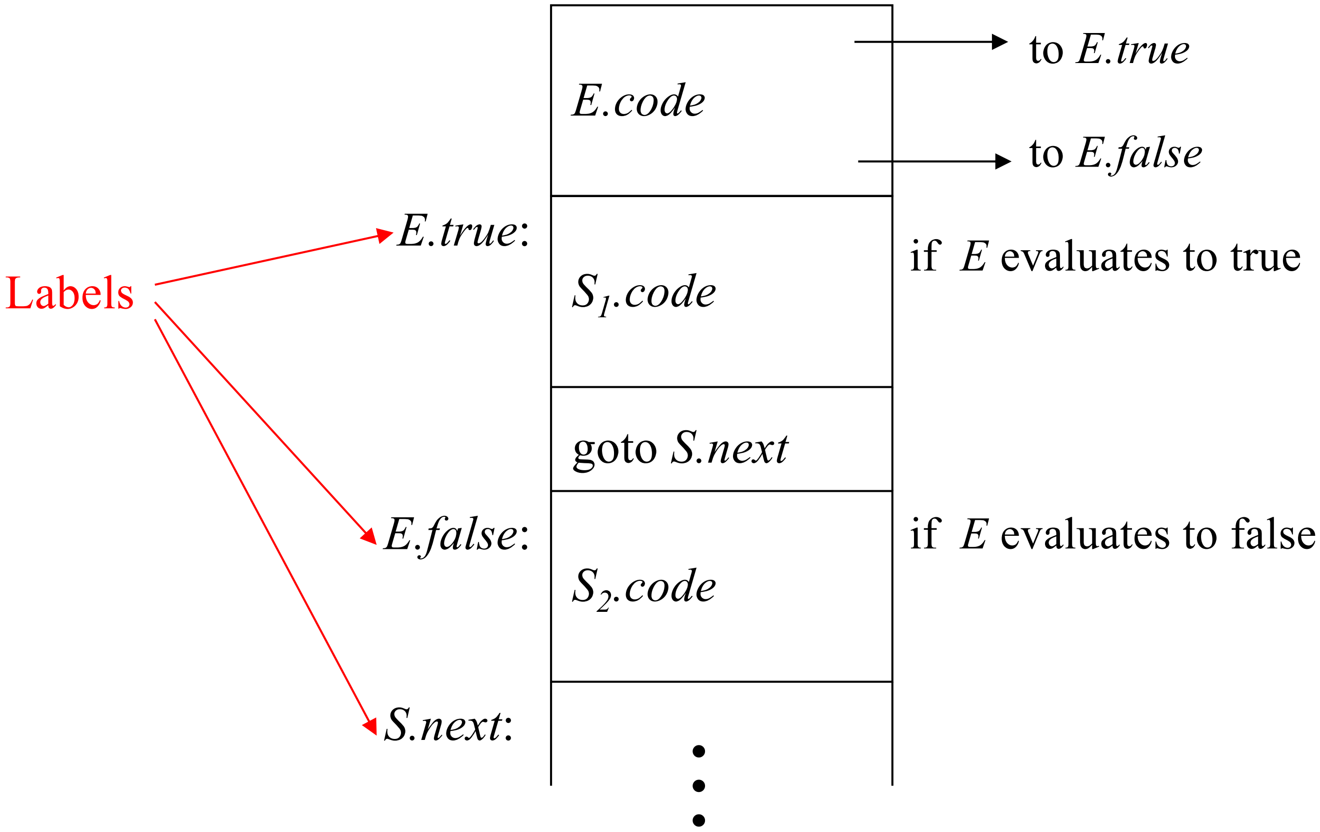
- Branch based on the result of boolean test expression



Flow-of-Control Statements: Code Structure

We have to decide on the code layout for the code for flow-of-control

$$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$$



Flow-of-Control Statements, Three-Address Code, Assuming Implicit Representation for Boolean Expressions

Attributes : *S.code*: sequence of instructions that are generated for *S*
S.next: label of the instruction that will be executed immediately after *S*
(S.next is an inherited attribute)

Productions	Semantic Rules
$S \rightarrow$ if E then S_1 else S_2	$E.true \leftarrow$ newlabel(); $E.false \leftarrow$ newlabel(); $S_1.next \leftarrow S.\ next$; $S_2.next \leftarrow S.\ next$; $S.code \leftarrow E.code$ || gen($E.true$ ':') || $S_1.code$ || gen('`goto`' $S.next$) || gen($E.false$ ':') || $S_2.code$;

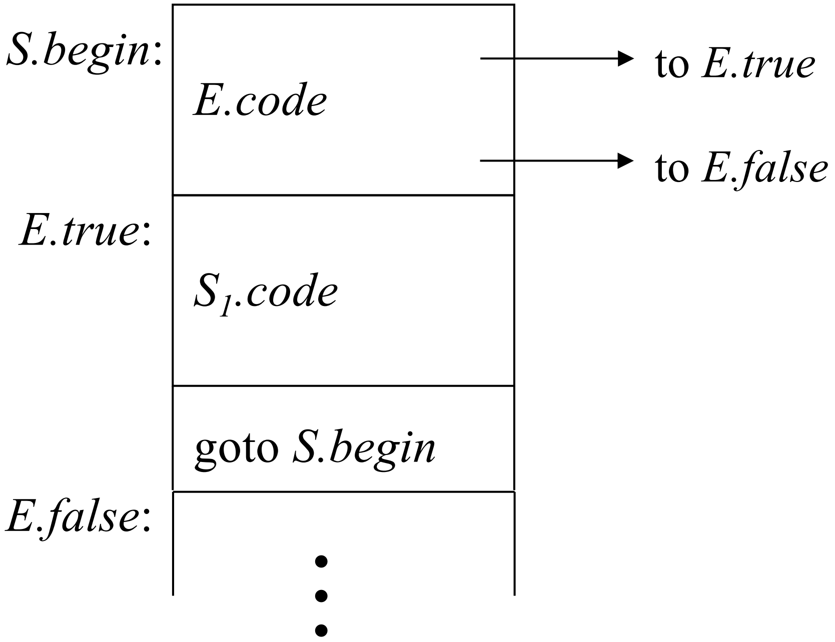
ASSUMPTION: the code generated for boolean expression *E* will branch to *E.true* or *E.false* label based on the result of the expression

Flow-of-Control Statements: Code Structure

Two different layouts for while statements:

The algorithms I give in the following slides use this layout:

$S \rightarrow$ while E do S_1



S.begin:

E.code

to *E.true*

to *E.false*

E.true:

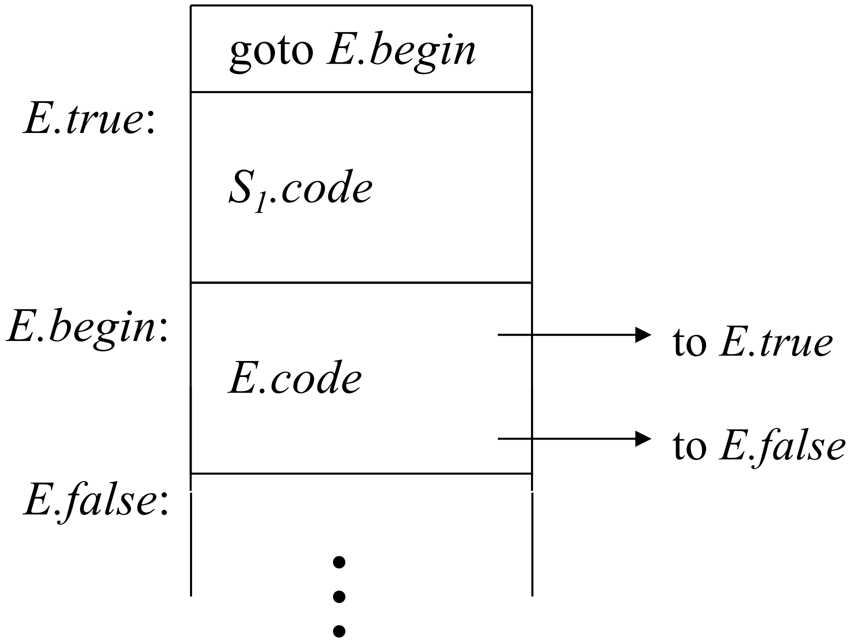
S_1*.code*

goto *S.begin*

E.false:

This layout places *E.code* after S_1*.code* :

$S \rightarrow$ while E do S_1



goto *E.begin*

E.true:

S_1*.code*

E.begin:

E.code

to *E.true*

to *E.false*

E.false:

Flow-of-Control Statements, Three-Address Code, Assuming Implicit Representation for Boolean Expressions

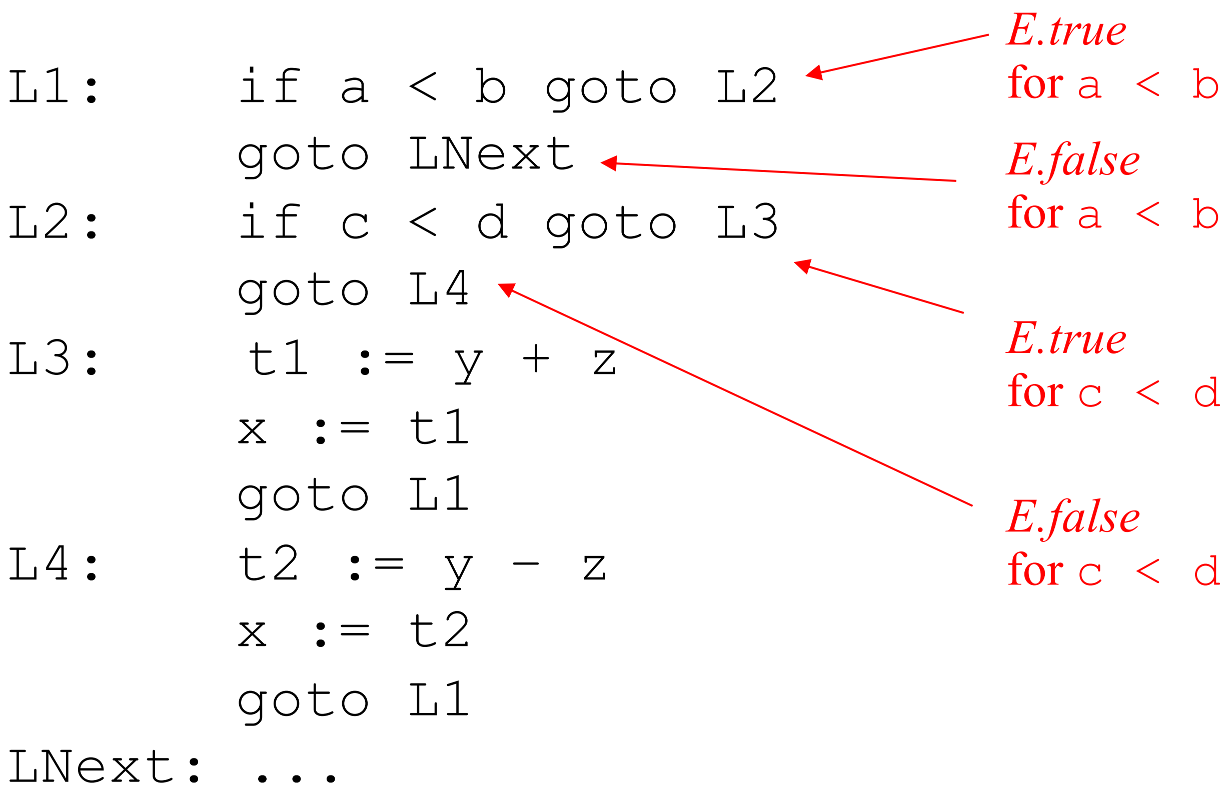
Attributes : *S.code*: sequence of instructions that are generated for *S*
S.next: label of the instruction that will be executed immediately after *S*
(S.next is an inherited attribute)

Productions	Semantic Rules
$S \rightarrow$ while E do S_1	$S.begin \leftarrow \text{newlabel}();$ $E.true \leftarrow \text{newlabel}();$ $E.false \leftarrow S.\ next;$ $S_1.next \leftarrow S.\ begin;$ $S.code \leftarrow \text{gen}(S.begin\ \text{‘:’}) \Vert E.code \Vert \text{gen}(E.true\ \text{‘:’}) \Vert S_1.code$ $\Vert \text{gen}(\text{‘}\texttt{goto}\text{’}\ S.begin);$
$S \rightarrow S_1 ; S_2$	$S_1.next \leftarrow \text{newlabel}();$ $S_2.next \leftarrow S.next;$ $S.code \leftarrow S_1.code \Vert \text{gen}(S_1.next\ \text{‘:’}) \Vert S_2.code$

Example

Input code fragment:

```
while (a < b)  {
    if  (c < d)
        x = y + z;
    else
        x = y - z
}
```



```
L1:     if a < b goto L2
        goto LNext
L2:     if c < d goto L3
        goto L4
L3:      t1  := y + z
        x  := t1
        goto L1
L4:     t2  := y - z
        x  := t2
        goto L1
LNext:  ...
```

E.true for a < b

E.false for a < b

E.true for c < d

E.false for c < d

Register Allocation

- Want to replace variables with some fixed set of registers
- First: need to know which variables are live after each instruction
  - Two simultaneously live variables cannot be allocated to the same register

Interference graph

- Nodes of the graph = variables
- Edges connect variables that interfere with one another
- Nodes will be assigned a color corresponding to the register assigned to the variable
- Two colors can’t be next to one another in the graph

Interference graph

Instructions	Live vars
b = a + 2	
c = b * b	
b = c + 1	
return b * a	

Interference graph

Instructions	Live vars
b = a + 2	
c = b * b	
b = c + 1	
	b,a
return b * a	

Interference graph

Instructions	Live vars
b = a + 2	
c = b * b	
	a,c
b = c + 1	
	b,a
return b * a	

Interference graph

Instructions	Live vars
b = a + 2	
	b,a
c = b * b	
	a,c
b = c + 1	
	b,a
return b * a	

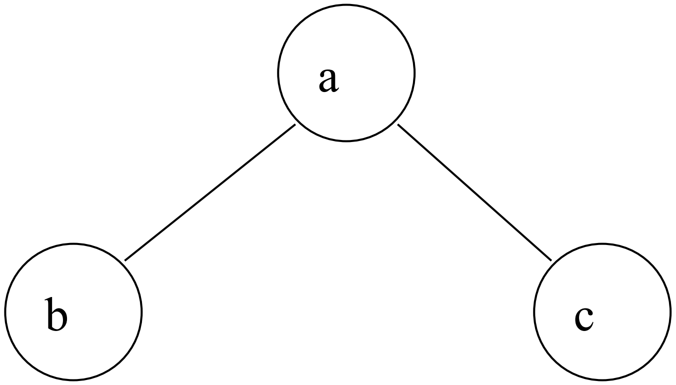
Interference graph

Instructions	Live vars
	a
b = a + 2	
	b,a
c = b * b	
	a,c
b = c + 1	
	b,a
return b * a	

Interference graph

Instructions	Live vars
	a
b = a + 2	
	a,b
c = b * b	
	a,c
b = c + 1	
	a,b
return b * a	

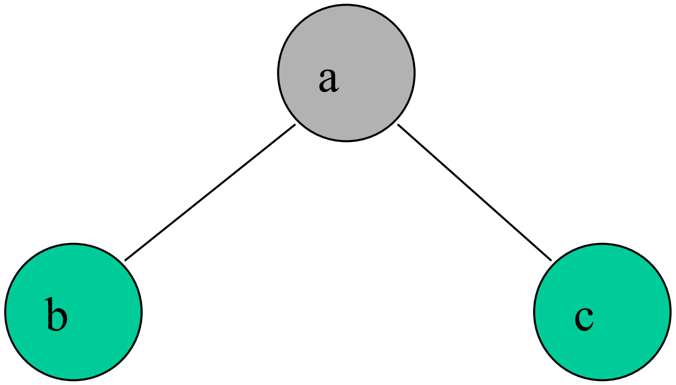
color	register
🟩	eax
▨	ebx



Interference graph

Instructions	Live vars
	a
b = a + 2	
	a,b
c = b * b	
	a,c
b = c + 1	
	a,b
return b * a	

color	register
🟩	eax
⬜	ebx



Graph coloring

- Questions:
  - Can we efficiently find a coloring of the graph whenever possible?
  - Can we efficiently find the optimum coloring of the graph?
  - What do we do when there aren’t enough colors (registers) to color the graph?

Coloring a graph

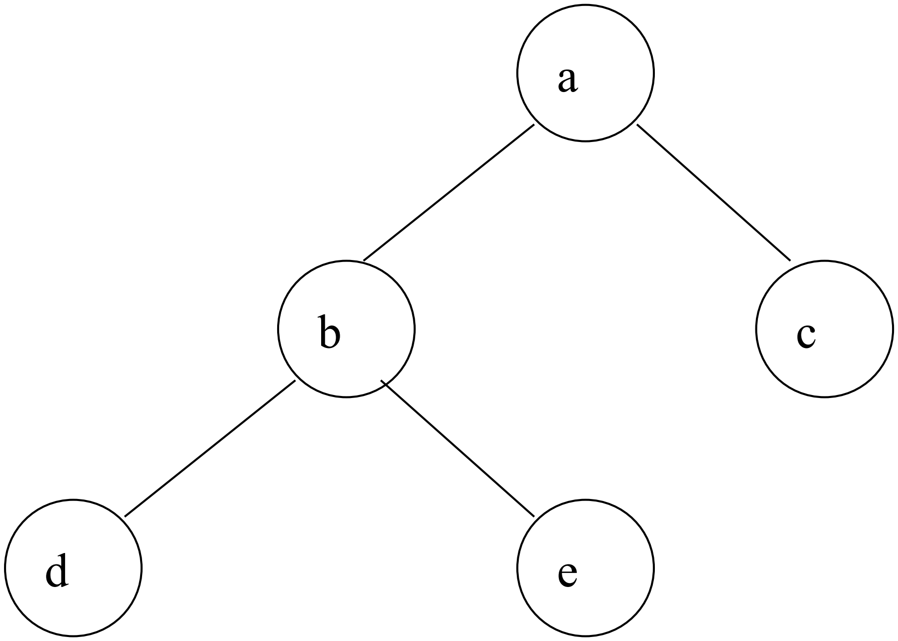
- Kempe's algorithm [1879] for finding a K-coloring of a graph
- Assume K=3
- Step 1 (simplify): find a node with at most K-1 edges and cut it out of the graph. (Remember this node on a stack for later stages.)

Coloring a graph

- Once a coloring is found for the simpler graph, we can always color the node we saved on the stack
- Step 2 (color): when the simplified subgraph has been colored, add back the node on the top of the stack and assign it a color not taken by one of the adjacent nodes

Coloring

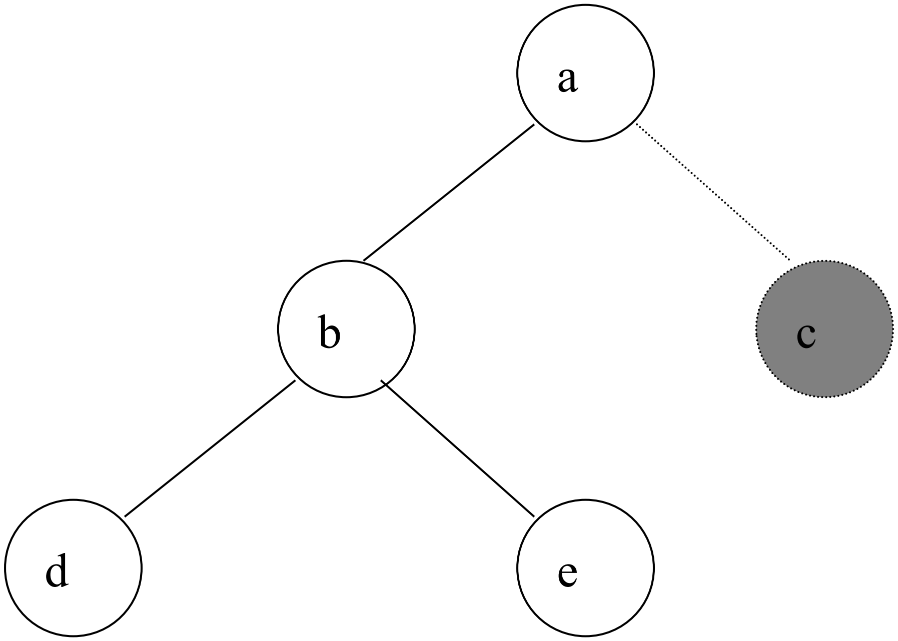
color	register
🟩	eax
▨	ebx



stack:

Coloring

color	register
🟩	eax
▨	ebx

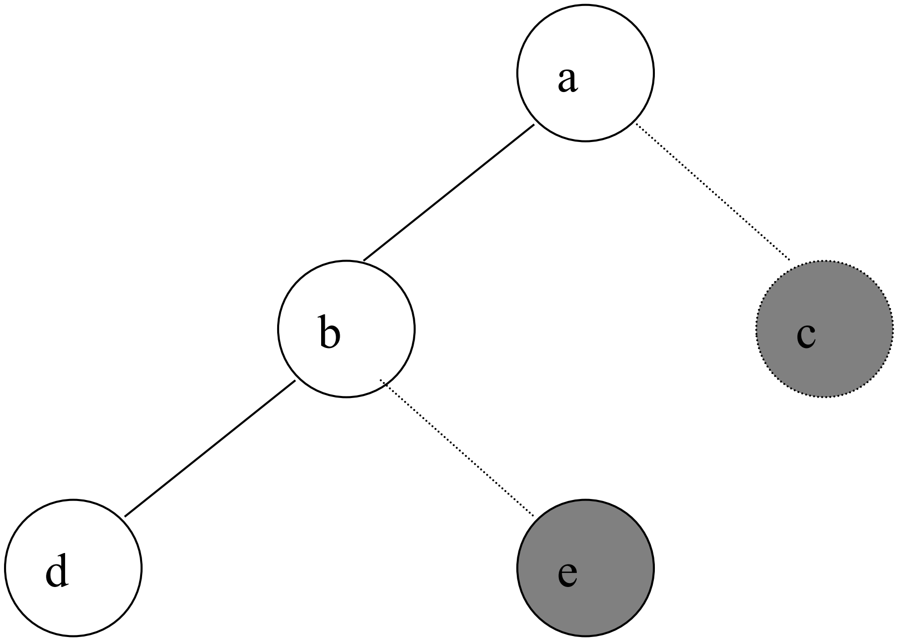


stack:

c

Coloring

color	register
🟩	eax
▨	ebx

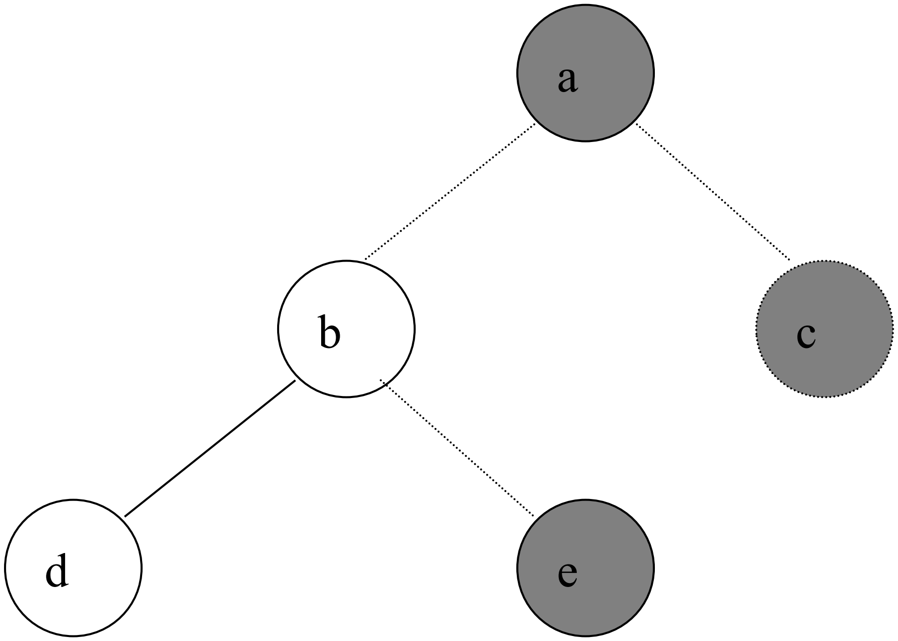


stack:

e
c

Coloring

color	register
🟩	eax
▨	ebx

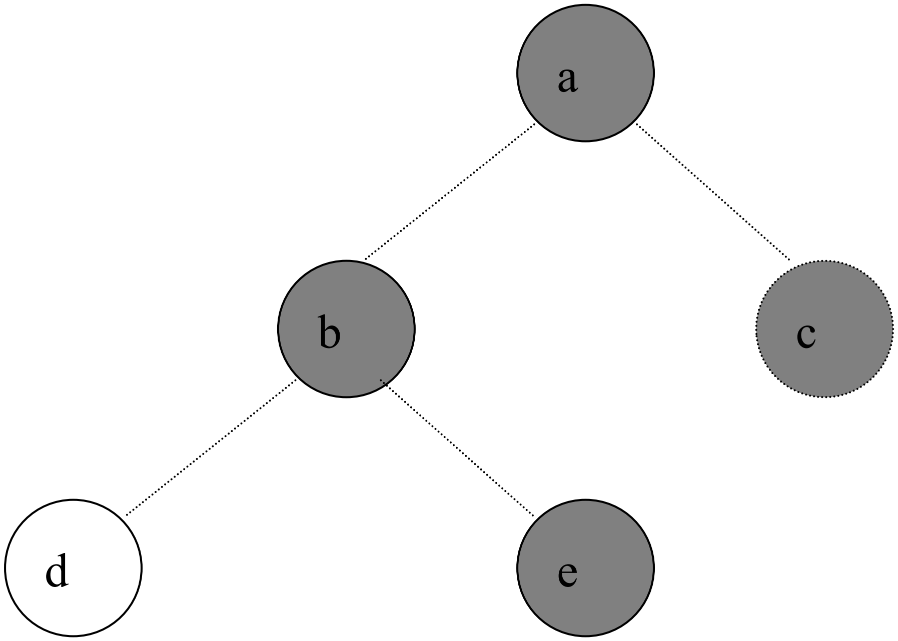


stack:

a
e
c

Coloring

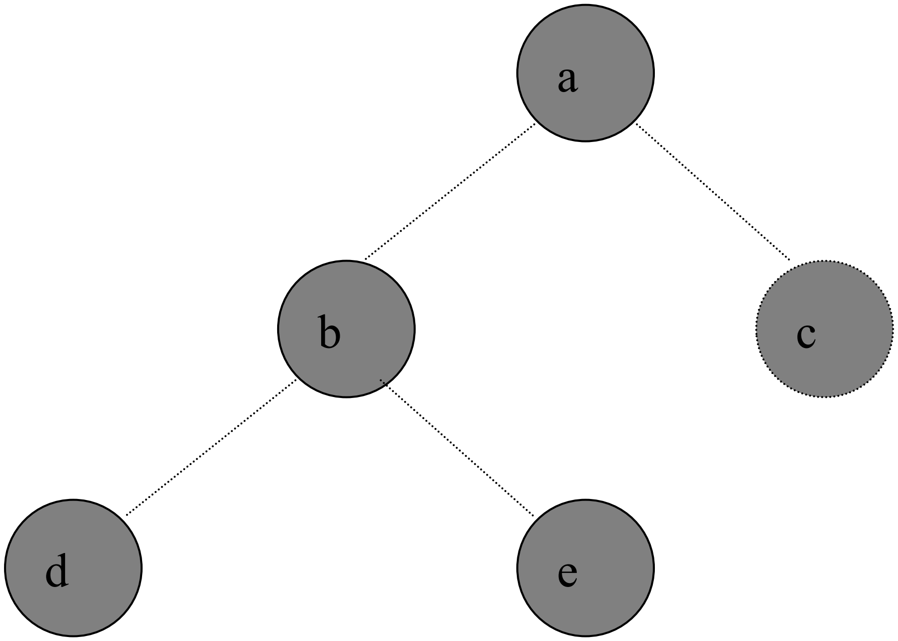
color	register
🟩	eax
◼	ebx



stack:
b
a
e
c

Coloring

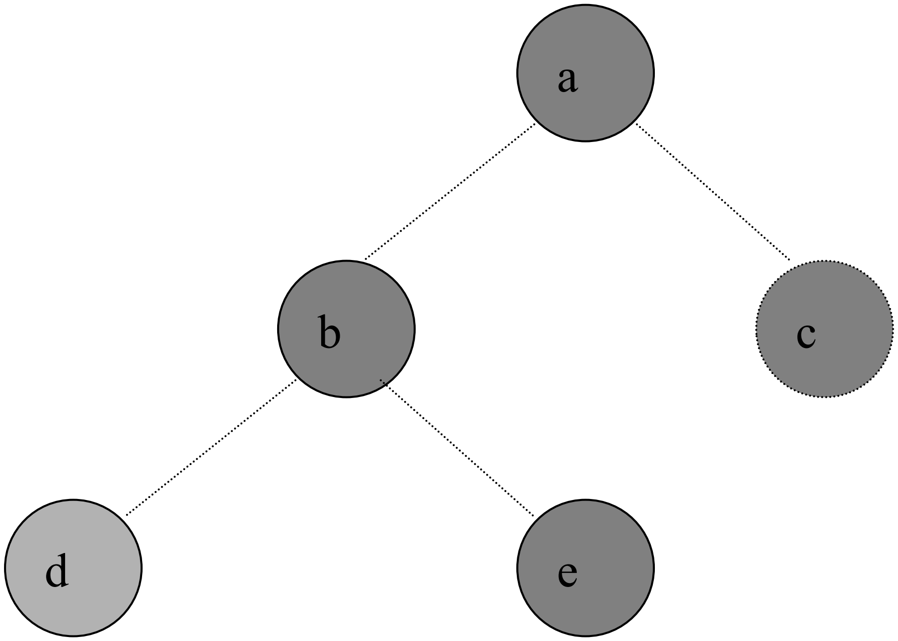
color	register
🟩	eax
▨	ebx



stack:
d
b
a
e
c

Coloring

color	register
🟩	eax
▨	ebx

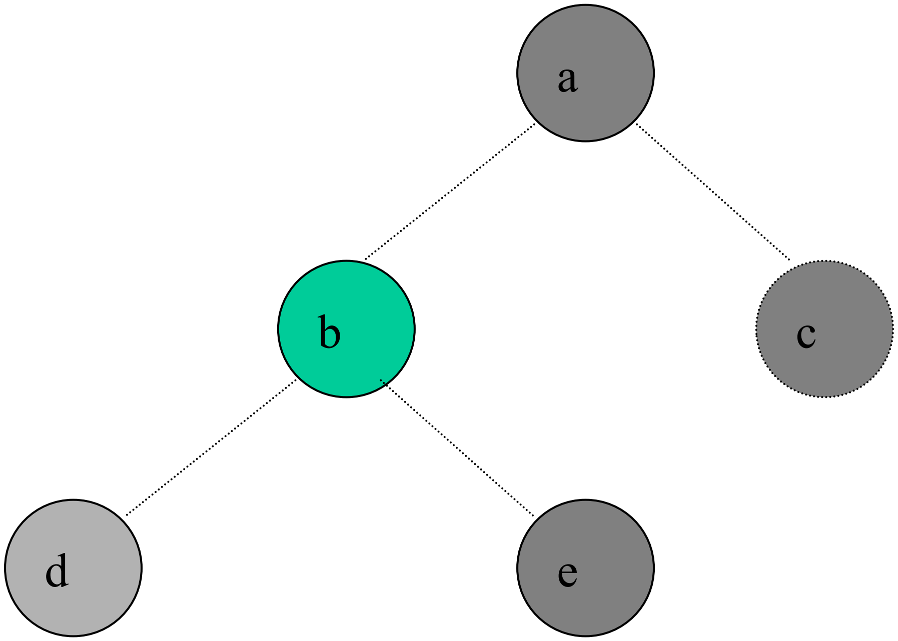


stack:

b
a
e
c

Coloring

color	register
🟩	eax
▨	ebx

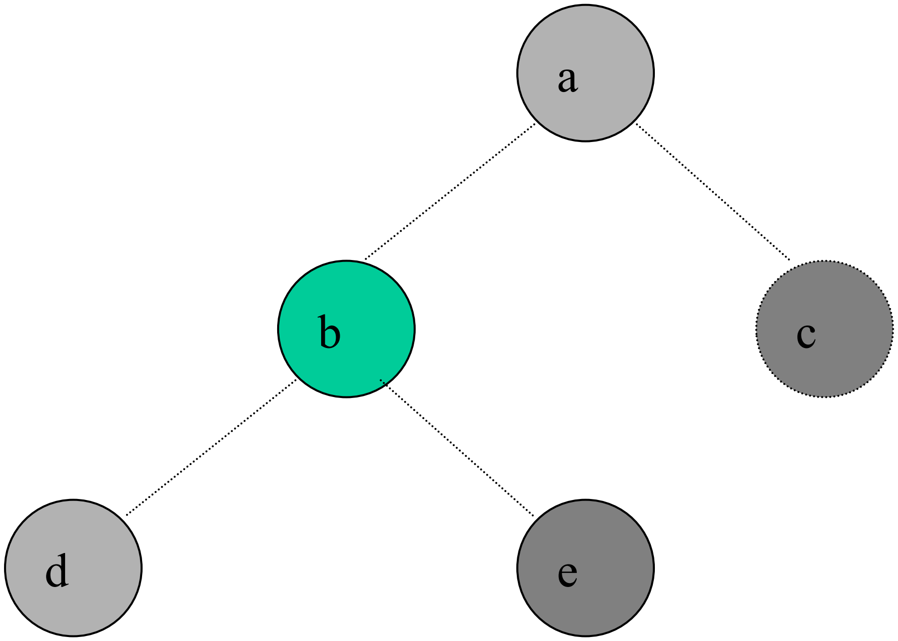


stack:

a
e
c

Coloring

color	register
🟩	eax
▨	ebx

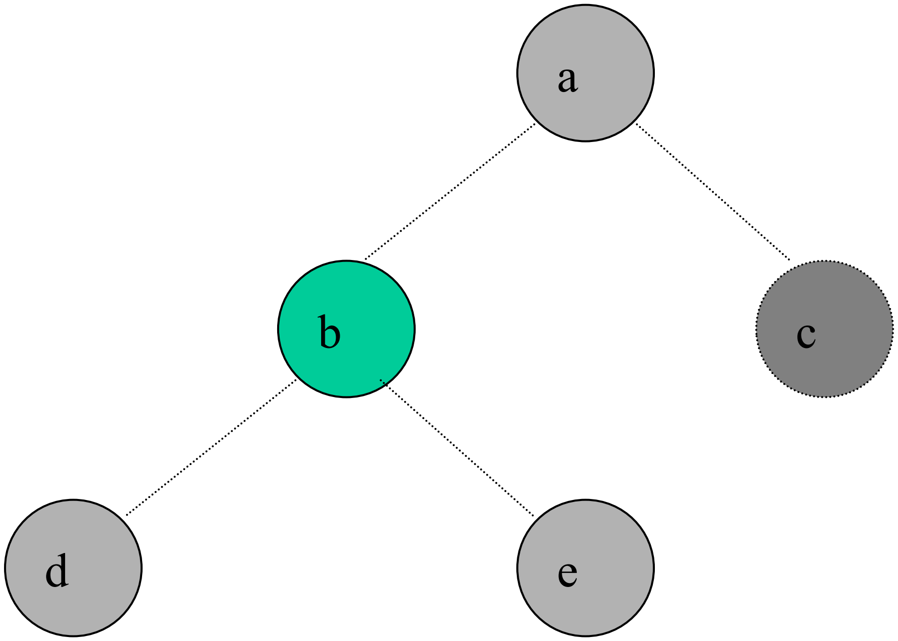


stack:

e
c

Coloring

color	register
🟩	eax
▨	ebx

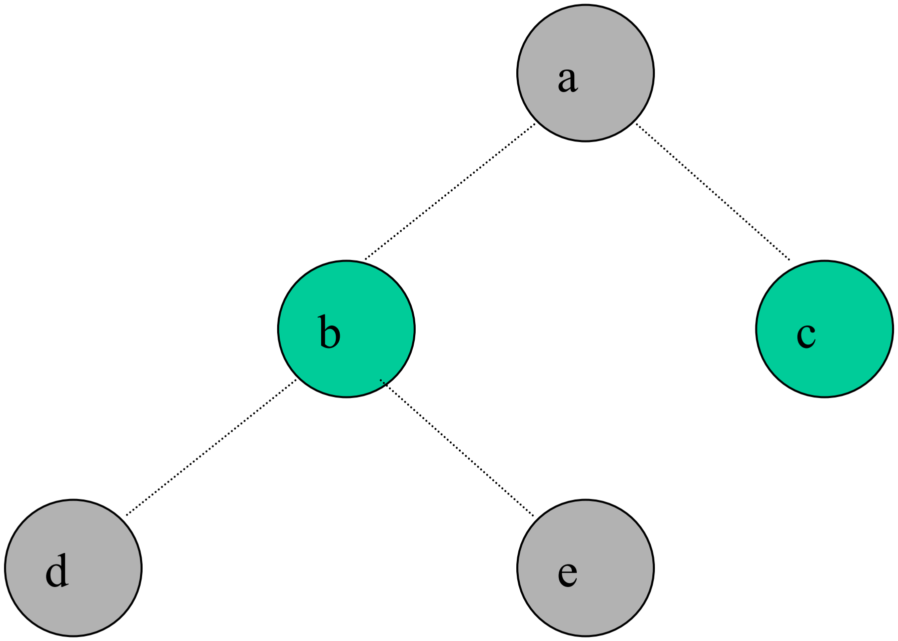


stack:

c

Coloring

color	register
🟩	eax
▩	ebx



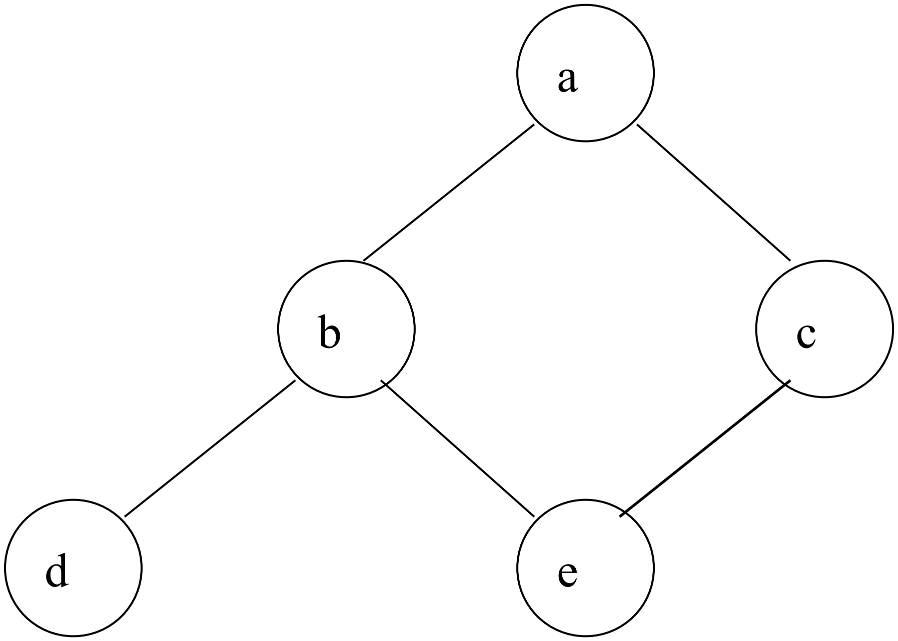
stack:

Failure

- If the graph cannot be colored, it will eventually be simplified to graph in which every node has at least K neighbors
- Sometimes, the graph is still K-colorable!
- Finding a K-coloring in all situations is an NP-complete problem
  - We will have to approximate to make register allocators fast enough

Coloring

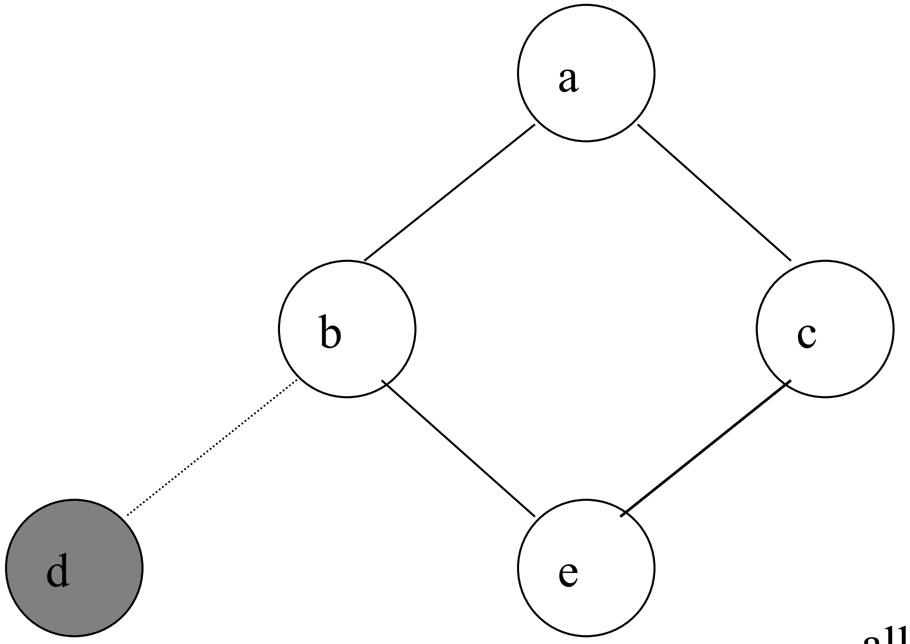
color	register
🟩	eax
▨	ebx



stack:

Coloring

color	register
🟩	eax
▨	ebx

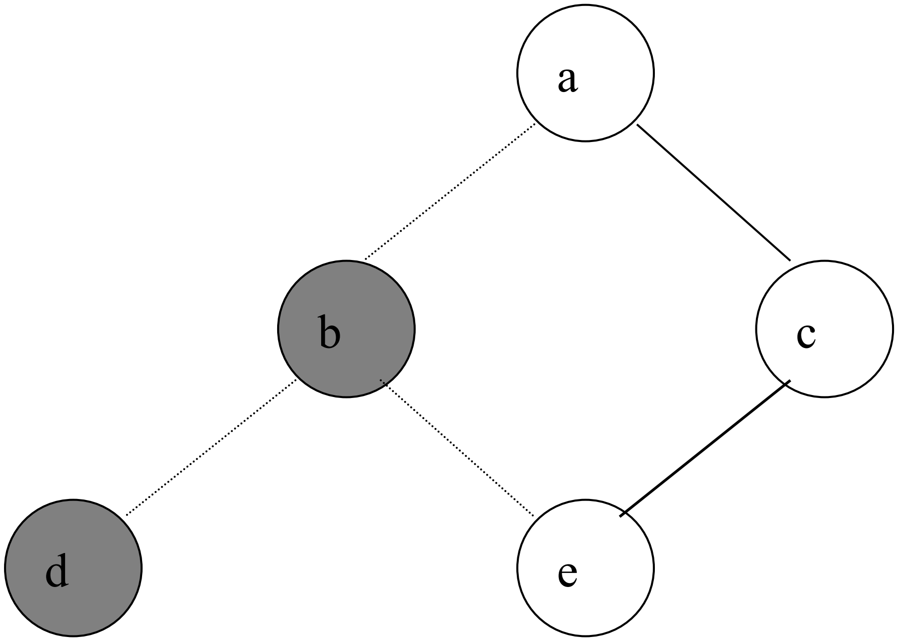


stack:
d

all nodes have
2 neighbours!

Coloring

color	register
🟩	eax
▒	ebx

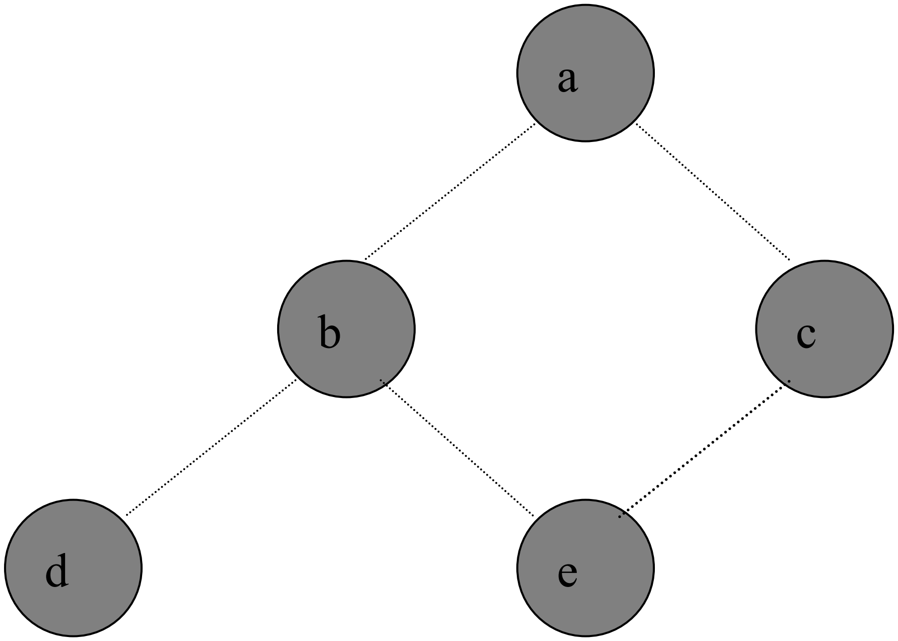


stack:

b
d

Coloring

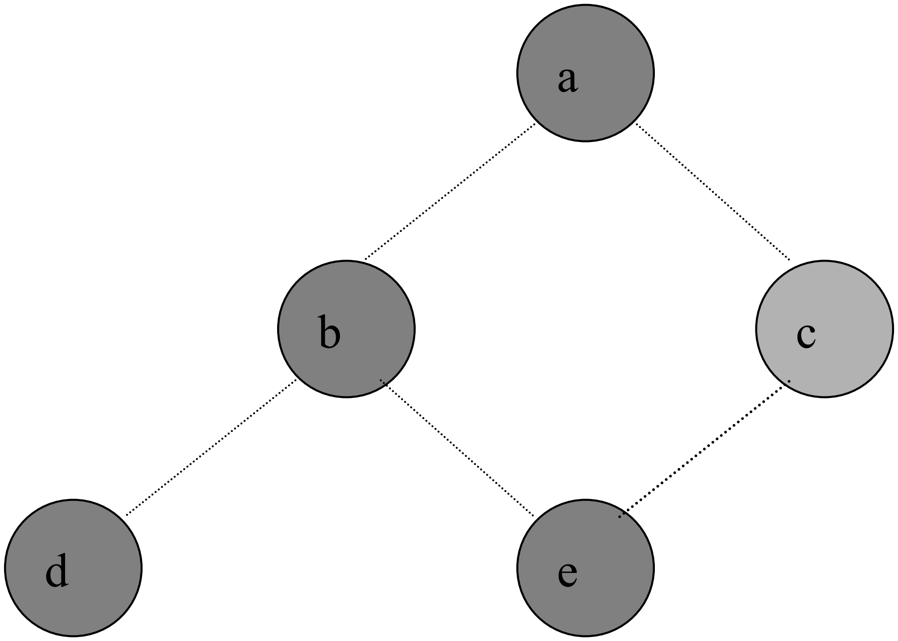
color	register
🟩	eax
▨	ebx



stack:
c
e
a
b
d

Coloring

color	register
🟩	eax
▨	ebx

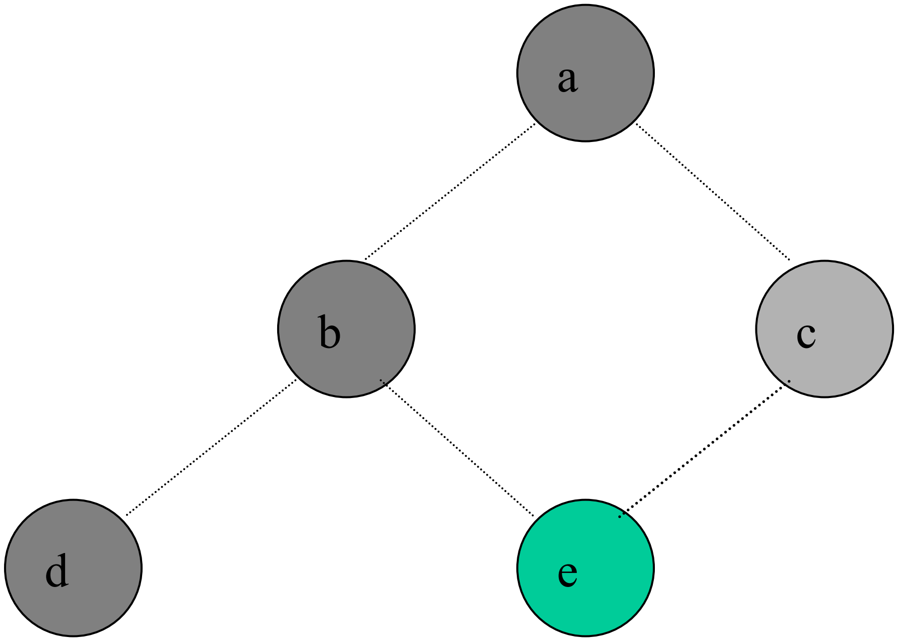


stack:

e
a
b
d

Coloring

color	register
🟩	eax
▨	ebx

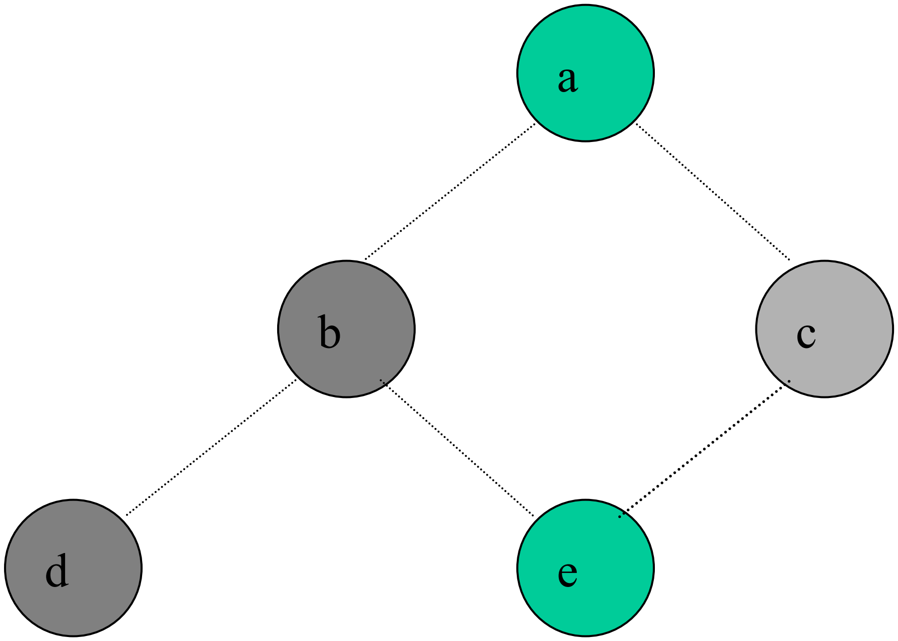


stack:

a
b
d

Coloring

color	register
🟩	eax
▨	ebx

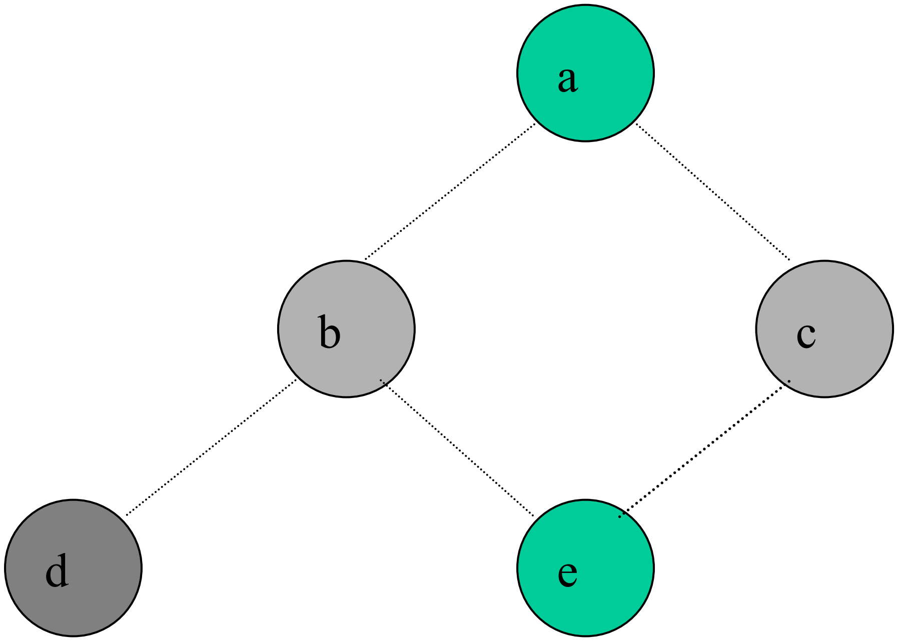


stack:

b
d

Coloring

color	register
🟩	eax
▨	ebx

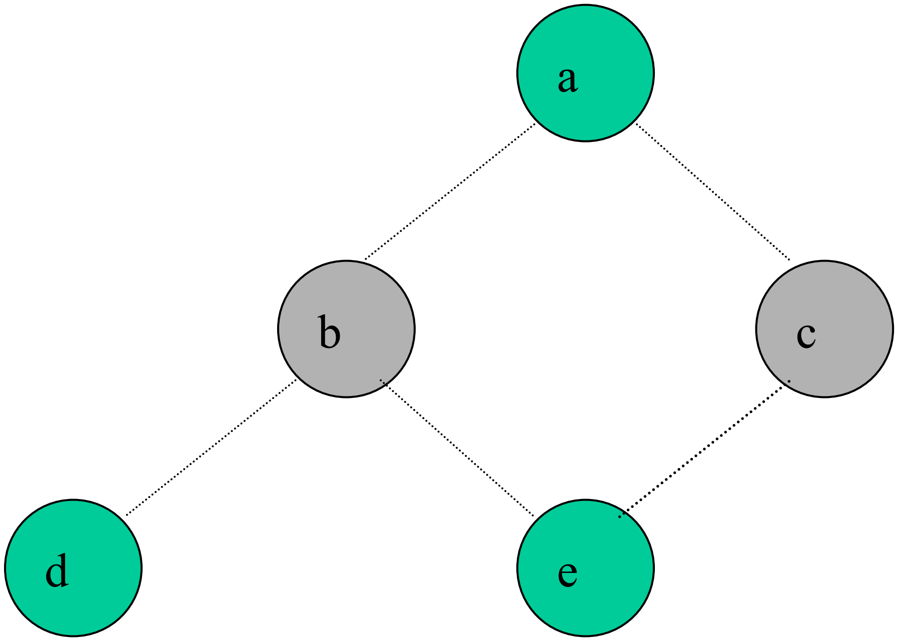


stack:

d

Coloring

color	register
🟩	eax
▨	ebx



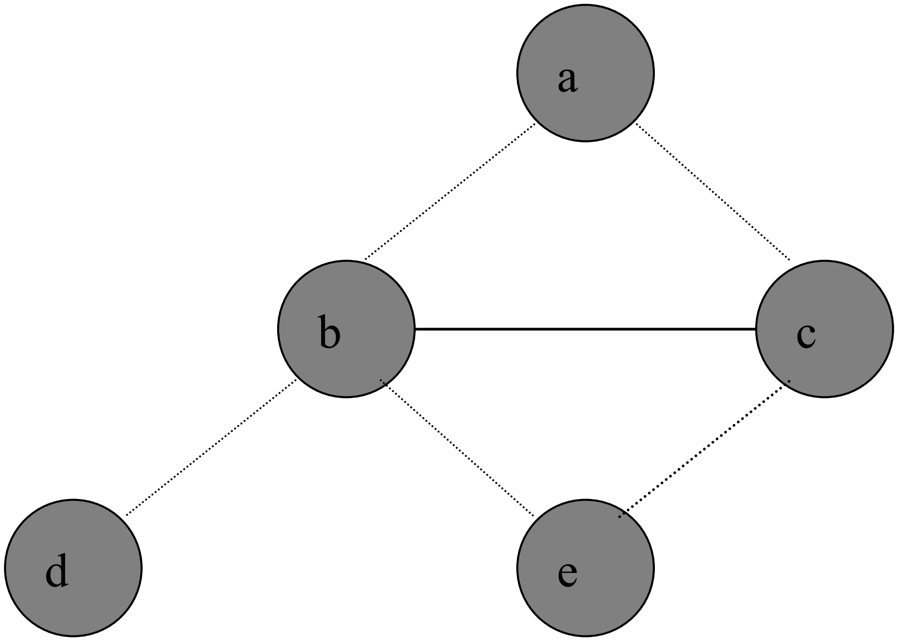
stack:

We got lucky!

Coloring

color	register
🟩	eax
▨	ebx

Some graphs can’t be colored in K colors:



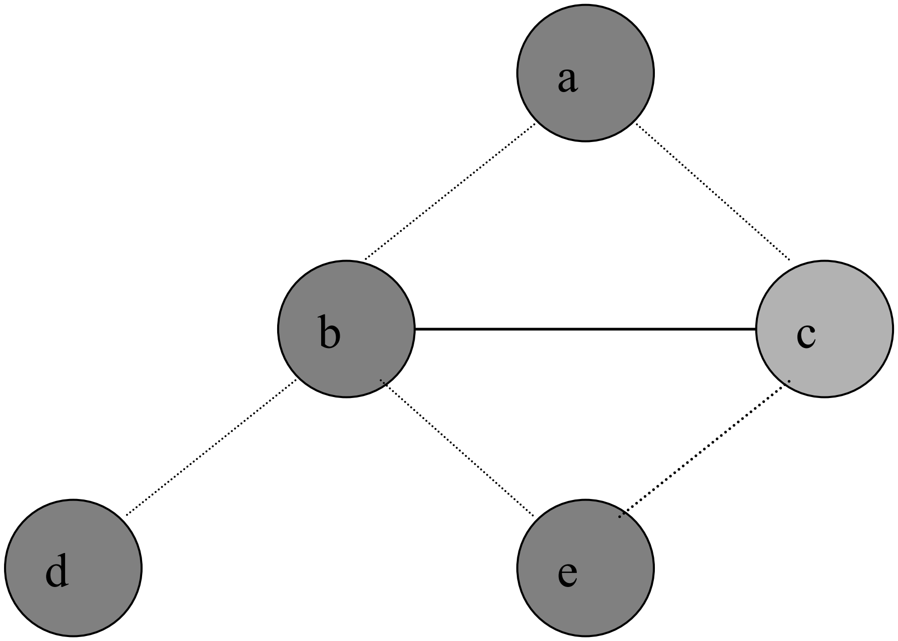
stack:

c
b
e
a
d

Coloring

color	register
🟩	eax
⬜	ebx

Some graphs can't be colored
in K colors:



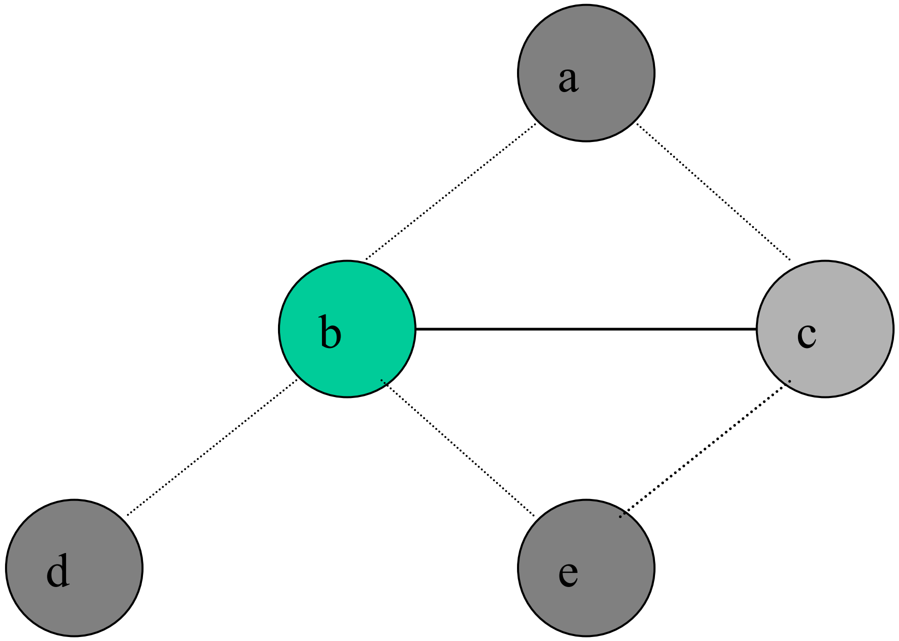
stack:

b
e
a
d

Coloring

color	register
🟩	eax
⬜	ebx

Some graphs can’t be colored in K colors:



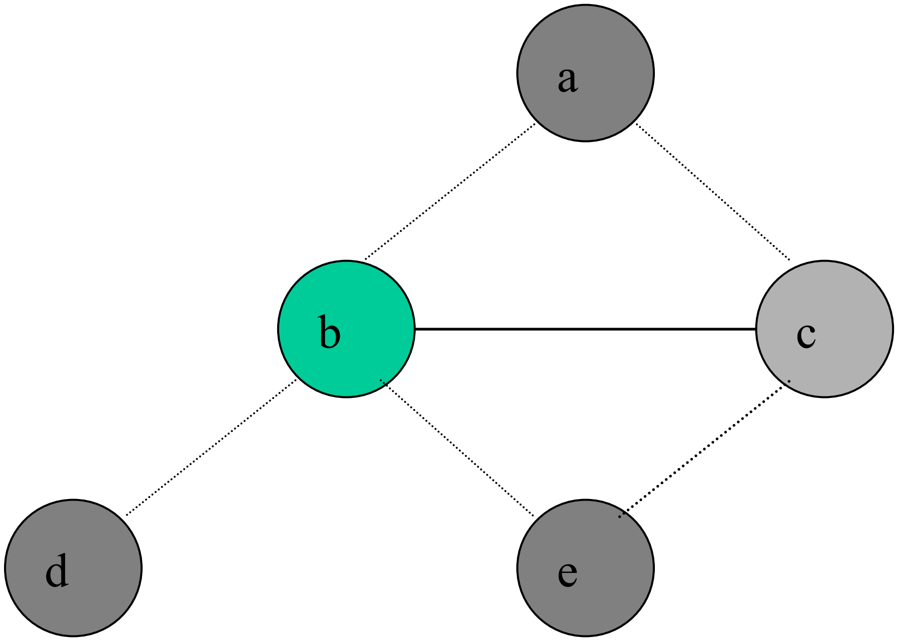
stack:

e
a
d

Coloring

color	register
🟩	eax
⬜	ebx

Some graphs can’t be colored in K colors:



stack:

e
a
d

no colors left for e!

Spilling

- Step 3 (spilling): once all nodes have K or more neighbors, pick a node for spilling
  - Storage on the stack
  - Rewrite code introducing a new temporary; rerun liveness analysis and register allocation
- There are many heuristics that can be used to pick a node
  - not in an inner loop

Rewriting code

- Consider: add t1 t2
  - Suppose t2 is a selected for spilling and assigned to stack location [ebp-4]
  - Invented new temporary t35 for just this instruction and rewrite:
    - mov t35, [ebp – 4]; add t1, t35
  - Advantage: t35 has a very short live range and is much less likely to interfere.
  - Rerun the algorithm; fewer variables will spill

Precolored Nodes

- Some variables are pre-assigned to registers
  - Eg: mul on x86/pentium
    - uses eax; defines eax, edx
  - Eg: call on x86/pentium
    - Defines caller-save registers eax, ecx, edx
- Treat these registers as special temporaries; before beginning, add them to the graph with their colors

Precolored Nodes

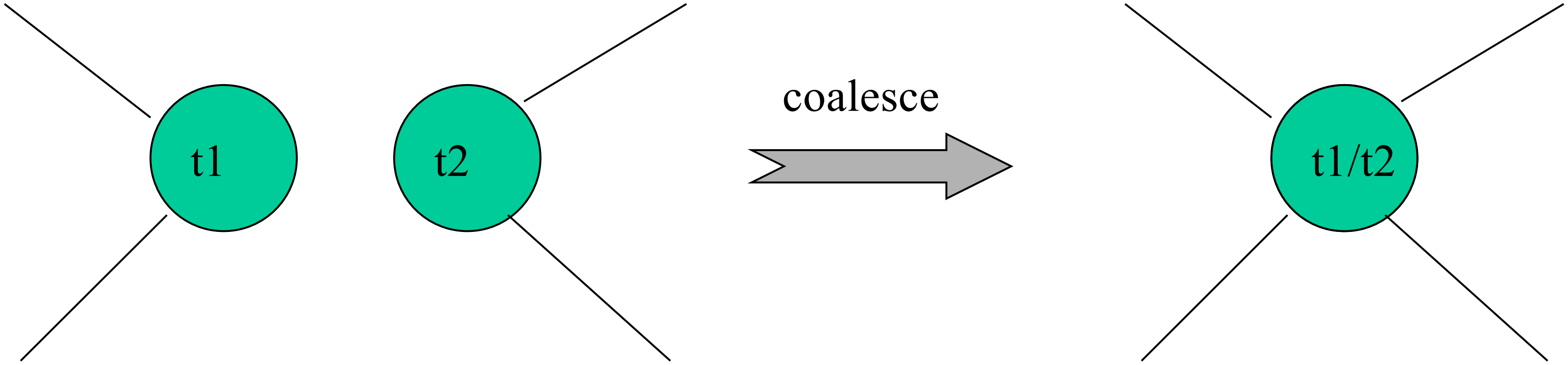
- Can’t simplify a graph by removing a precolored node
- Precolored nodes are the starting point of the coloring process
- Once simplified down to colored nodes start adding back the other nodes as before

Optimizing Moves

- Code generation produces a lot of extra move instructions
  - mov t1, t2
  - If we can assign t1 and t2 to the same register, we do not have to execute the mov
  - Idea: if t1 and t2 are not connected in the interference graph, we coalesce into a single variable

Coalescing

- Problem: coalescing can increase the number of interference edges and make a graph uncolorable



- Solution 1 (Briggs): avoid creation of high-degree (>= K) nodes
- Solution 2 (George): a can be coalesced with b if every neighbour t of a:
  - already interferes with b, or
  - has low-degree (< K)

Summary

- Register allocation has three major parts
  - Liveness analysis
  - Graph coloring
  - Program transformation (move coalescing and spilling)