Accessing Variables

How can we generate code for "x"?

a := x + y

The variable may be in a register:

ADD ...,R_x,...

Another register (perhaps the same?)

The variable may be in a static memory location:

SET x,R_w
LD [R_w],R_x
ADD ...,R_x,...

A "work" register

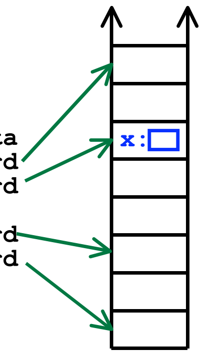
The variable may be a local variable:

LD [%fp+offset$_x$],R_x
ADD ...,R_x,...

The variable may be a non-local variable:

SET display4,R_w
LD [R_w],R_w
LD [R_w+offset$_x$],R_x
ADD ...,R_x,...

```
          .data
display5: .word
display4: .word
   •••
display1: .word
display0: .word
```



Register Strategies

Option 1

Keep every variable in memory at all times.
Use 1 or 2 "work" registers during code generation.
Code Generator #1:
Every statement in isolation.
Variables in memory between each IR instruction.

Option 2

Keep all variables in memory between basic blocks.
Generate code for each basic block in isolation.
Within the basic block, use registers to hold values.
At the end of the basic block,
store all (LIVE) variables back to memory.

Register Strategies

Option 3

Divide registers into groups
- Work registers (e.g., R0-R4)
- Variable storage (e.g., R5-R23)
- Other (e.g., %fp, not available for variable storage)

Look at the entire flow graph.
Select some variables that are used "a lot."
Put them in registers for the entire routine.
  Beginning of routine: generate a "LOAD"
  End of routine: generate a "STORE"
Remaining variables: Keep in memory
  Generate LOADs and STOREs when they are accessed.

Problems:
- Non-local access?
  Other routines expect variable to be in memory.
  Identify variables that are used only locally.
  Generate STOREs before each CALL
- How to select variables? *Register Class / Pragmas*
  A variable used only 2 times... but within a loop!

Terminology

"Local Register Allocation"

Register allocation is done for the entire basic block.
... But each basic block is done independently.

"Global Register Allocation"

Register allocation is done for the entire flow graph / routine.
... But each routine is done independently.

Option 2 + 3

Set aside some registers for
  "Local Allocation" -- within the basic block
  "Global Allocation" -- for the entire routine

Global Register Allocation

Option 4

"Global Register Allocation"
Look at the entire flow graph.
  Look at variable lifetimes
  Identify "Live Ranges"
  Map each Live Range into a register
Graph-Coloring Algorithm

The Register-To-Register Model

(Approach used in textbook)
Assume an infinite number of registers
  "Virtual Registers"
During code generation... an inexhaustable supply.

Example: a := (x + y) * z;

`LOAD`	`[%fp+offset`$_x$`],R`$_x$
`LOAD`	`[%fp+offset`$_y$`],R`$_y$
`ADD`	`R`$_x$`,R`$_y$`,R`$_{tmp1}$
`LOAD`	`[%fp+offset`$_z$`],R`$_z$
`MULT`	`R`$_{tmp1}$`,R`$_z$`,R`$_{tmp2}$
`STORE`	`R`$_{tmp2}$`,[%fp+offset`$_a$`]`

The Register Allocation Phase

After target code generation
A limited supply of *"Physical Registers"*
Must select which Virtual Regs will go into Physical Regs.
When there are not enough Physical Regs...
  Generate `STORE`s and `LOAD`s

"Spill" instructions

The Memory-To-Memory Model

Don't worry about registers.
Keep all variables in memory.
During code generation... an inexhaustable supply of *temporary variables*.

Example: **a := (x + y) * z;**

```
ADD     x,y,tmp1
MULT    tmp1,z,tmp2
STORE   tmp2,a
```

(This is the approach taken in our compiler.)

The Register Allocation Phase

After target code generation
A limited supply of ***"Physical Registers"***
Must select which variables will go into Physical Registers.
Some variables are selected for ***"promotion"*** to registers.
Must generate **LOAD**s and **STORE**s.

The approaches are similar.

Unambiguous Memory References

Consider this source code:

```
...
x := a + b;
...

...
c := x + 4;
```

Can we keep "x" in a register?

Unambiguous Memory References

Consider this source code:

```
var p: ptr to integer;
p := &x;
...
x := a + b;
...
*p := ...;
...
c := x + 4;
```

Can we keep "x" in a register?

Unambiguous Memory References

Consider this source code:

```
var p: ptr to integer;
p := &x;
...
x := a + b;
...
*p := ...;
...
c := x + 4;
```

Can we keep "x" in a register?

"Unambiguous Memory Reference"

Given a variable "x"...

Is there any other way to get at that memory location?

If so...

Must keep the value in memory.

If not...

Okay to keep the value in a register.

Register Allocation Algorithm

- Assume the code uses "Virtual Registers"
- Look at each basic block in isolation (a "local" approach)
- Identify all the virtual registers used in the basic block.
- Assign a "priority" to each virtual register.
  - Run through the instructions.
  - Count the number of times the virtual register is used.
- Assume that we have K physical registers available.
  Identify the K virtual registers with the highest priority
  Assign each to one of the physical registers.
- Run through the instructions and replace all uses of virt. registers.
  If the virtual register has been assigned to a phys register, use that.
  Otherwise, generate LOADs and STOREs as necessary.
  - Must set aside a couple of "work" registers for this.
    - Move the variable into a work register.
    - Use it.
    - Store it back in memory.

Global Register Allocation

Assign a variable to a register.
Keep it in register at all times, across Basic Block boundaries.

Problem: Non-Local Accesses

Call a subroutine?
It uses registers in its own ways.
[Will save any registers it modifies.]
It expects non-local variables to be stored in their frames, buried in the stack.

Solution #1:

Save all variables back to memory whenever a call is made.

```
store   r3,[fp-4]
store   r4,[fp-8]
store   r5,[fp-12]
...
call    foo
load    [fp-4],r3
load    [fp-8],r4
load    [fp-12],r5
...
```

Global Register Allocation

Assign a variable to a register.
Keep it in register at all times, across Basic Block boundaries.

Problem: Non-Local Accesses

Call a subroutine?
It uses registers in its own ways.
[Will save any registers it modifies.]
It expects non-local variables to be
stored in their frames, buried in the stack.

Solution #2:

Identify which variables are...
- accessed only locally
- accessed non-locally

Keep only "only locally accessed" variables in registers.
Approximation:
Keep track of compiler-generated temporaries
Keep only these in registers
But the real benefit is to keep heavily used variables in registers!

Global Alloation for Loops

Idea:

Identify Loops
Use Global Register Allocation for the duration of the loop

- **Identify the basic blocks in a loop.**
- **Before going into the loop...**
  Move variables into registers
- **Within the loop...**
  Just use the registers.

Issues:

- **How many registers for global register allocation?**
  How many registers for "working storage"?
- **Which variables to put into registers?**
  Choose "heavily used" variables
- **Nested Loops ["inner loop" / "outer loop"]**
  Do the inner loops first.
- **How to identify loops?**

Register Allocation via Graph Coloring

Goal:

Keep all variables in registers
Keep each variable in a different register
...unless they are never LIVE simultaneously!

Example: "x" and "y"

Both LIVE at some point in the code?
⇒ Must put into different registers

Never LIVE at the same time?
⇒ May keep them in the same register!

Register Allocation via Graph Coloring

Assumptions:

Memory-to-memory model
Lots of variables
Assign each to a register

Register-to-register model
Lots of (virtual) registers
Assign each Virtual Register to a Physical Register

(We'll discuss the first.)

Register Allocation via Graph Coloring

Step 1:

Construct the **"Interference Graph"**

This occurs iff "x" and "y" are LIVE at the same time.

Nodes: Variables

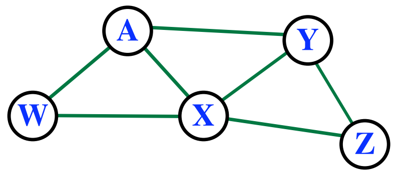
Undirected Edges:

If variable "x" is LIVE at the point where variable "y" is defined...

Add an edge between "x" and "y."

Intuition:

The edge means that "x" and "y" must go into different regs.



Register Allocation via Graph Coloring

Coloring a Graph

Given: A graph with undirected edges

Goal: Assign a color to each node

Such that:

Adjacent nodes have different colors.

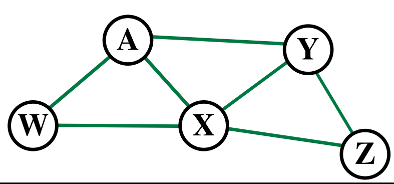
A "coloring" of the graph.

Coloring the Graph with K Colors

Assume we have only K different colors.

A "K-Coloring" of the graph.

Example:
Find a 3-Coloring



Register Allocation via Graph Coloring

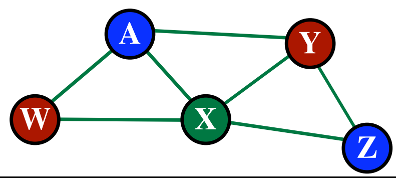
Coloring a Graph

Given: A graph with undirected edges
Goal: Assign a color to each node
Such that:
Adjacent nodes have different colors.
A "coloring" of the graph.

Coloring the Graph with K Colors

Assume we have only K different colors.
A "K-Coloring" of the graph.

Example:
Find a 3-Coloring



Register Allocation via Graph Coloring

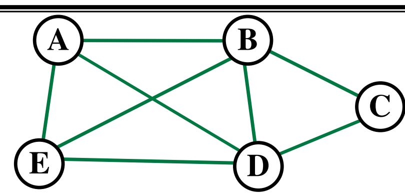
Coloring a Graph

Given: A graph with undirected edges
Goal: Assign a color to each node
Such that:
Adjacent nodes have different colors.
A "coloring" of the graph.

Coloring the Graph with K Colors

Assume we have only K different colors.
A "K-Coloring" of the graph.

Example:
Find a 3-Coloring



Register Allocation via Graph Coloring

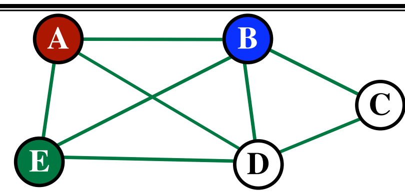
Coloring a Graph

Given: A graph with undirected edges
Goal: Assign a color to each node
Such that:
Adjacent nodes have different colors.
A "coloring" of the graph.

Coloring the Graph with K Colors

Assume we have only K different colors.
A "K-Coloring" of the graph.

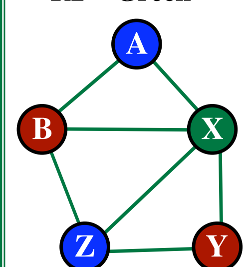
Example:
Find a 3-Coloring
Not Possible!
(but it is 4-Colorable)



Register Allocation via Graph Coloring

Assume we have K physical registers.
Find a K-Coloring of the graph.

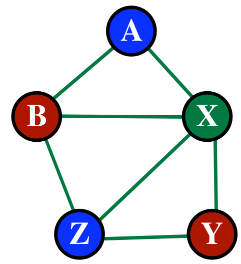
Example: K=3
R0 = Blue
R1 = Red
R2 = Green



Register Allocation via Graph Coloring

Assume we have K physical registers.
Find a K-Coloring of the graph.
The coloring is an assignment
of variables to registers.

Example: K=3
R0 = Blue
R1 = Red
R2 = Green



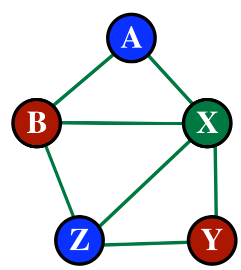
A: R0
B: R1
X: R2
Y: R1
Z: R0

Register Allocation via Graph Coloring

Assume we have K physical registers.
Find a K-Coloring of the graph.
The coloring is an assignment
of variables to registers.

Can we color a graph using only K colors?
("Is the graph K-Colorable?")

Example: K=3
R0 = Blue
R1 = Red
R2 = Green



A: R0
B: R1
X: R2
Y: R1
Z: R0

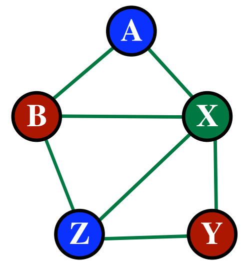
Register Allocation via Graph Coloring

Assume we have K physical registers.
Find a K-Coloring of the graph.
The coloring is an assignment
of variables to registers.

Can we color a graph using only K colors?
("Is the graph K-Colorable?")

Unfortunately
Answering this question is NP-Complete
≈ Exponentially hard

Example: **K=3**
R0 = Blue
R1 = Red
R2 = Green



A: R0
B: R1
X: R2
Y: R1
Z: R0

Register Allocation via Graph Coloring

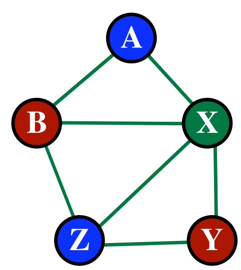
Assume we have K physical registers.
Find a K-Coloring of the graph.
The coloring is an assignment
of variables to registers.

Can we color a graph using only K colors?
("Is the graph K-Colorable?")

Unfortunately
Answering this question is NP-Complete
≈ Exponentially hard

Fortunately
We have a good heuristic algorithm
Gregory Chaitin
Finds a K-Coloring (usually)
If problems (i.e., not enough registers)
Generate "spill" instructions and keep going.

Example: **K=3**
R0 = Blue
R1 = Red
R2 = Green



A: R0
B: R1
X: R2
Y: R1
Z: R0

The Graph Coloring Algorithm

REPEAT
Find a node with fewer than K neighbors.
Eliminate that node (and its edges).
(If you can find a K-Coloring for the smaller graph, then all you have to do is add back this node and give it a color that is different from the colors of its neighbors.)
UNTIL all nodes have been eliminated

The Graph Coloring Algorithm

REPEAT
Find a node with fewer than K neighbors.
Eliminate that node (and its edges).
(If you can find a K-Coloring for the smaller graph, then all you have to do is add back this node and give it a color that is different from the colors of its neighbors.)
UNTIL all nodes have been eliminated

Remember the order of elimination.

Add back the nodes in reverse order,
assigning colors as you go.

The Graph Coloring Algorithm

REPEAT
    Find a node with fewer than K neighbors.
    Eliminate that node (and its edges).
        (If you can find a K-Coloring for the smaller graph, then all you have to do is add back this node and give it a color that is different from the colors of its neighbors.)
UNTIL all nodes have been eliminated

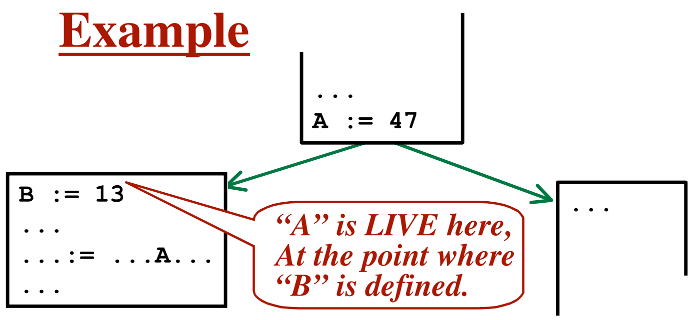
Remember the order of elimination.

Add back the nodes in reverse order,
    assigning colors as you go.

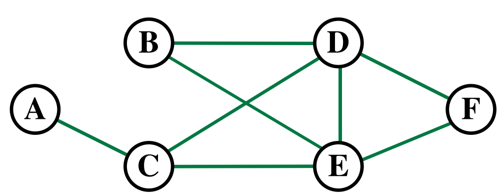
No nodes with fewer than K-neighbors?

This algorithm fails to find a K-Coloring.
    ...even though one may exist!
Will need to generate LOAD and STORE
    instructions for this variable.

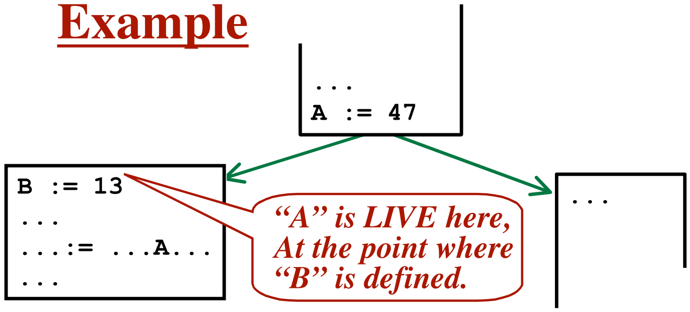
Example



"A" and "B" are simultaneously LIVE.
    Need to add an edge.

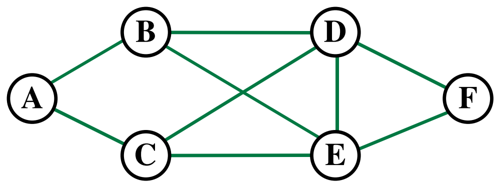


Example

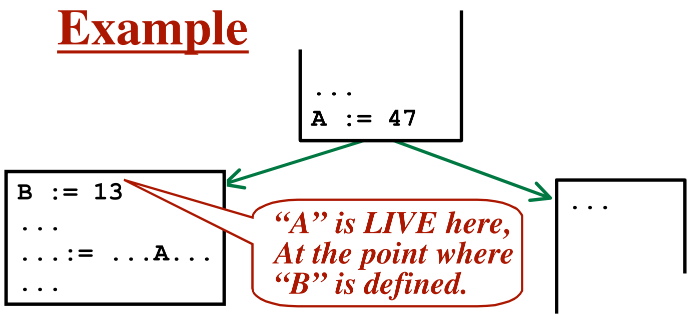


"A" and "B" are simultaneously LIVE.

Need to add an edge.



Example



"A" and "B" are simultaneously LIVE.

Need to add an edge.

Assume K=3 registers are available.

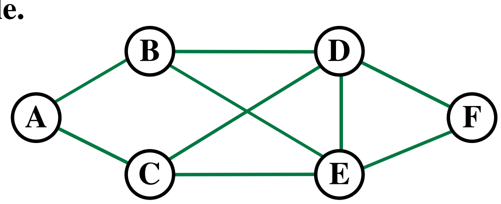
R0 = Blue

R1 = Red

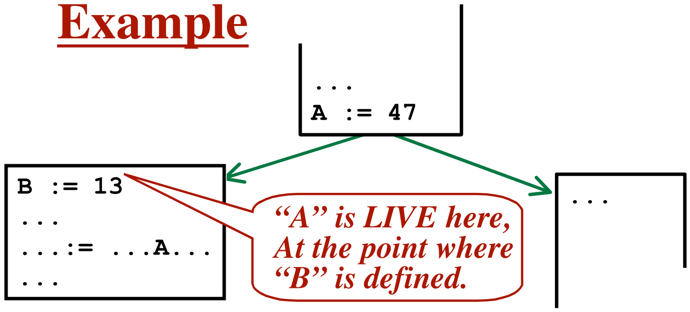
R2 = Green

Order Eliminated:

Reverse Order:

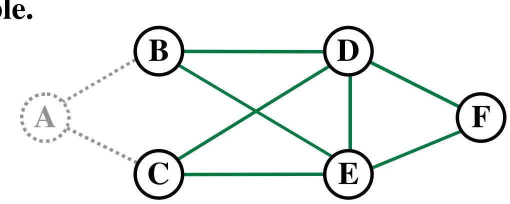


Example



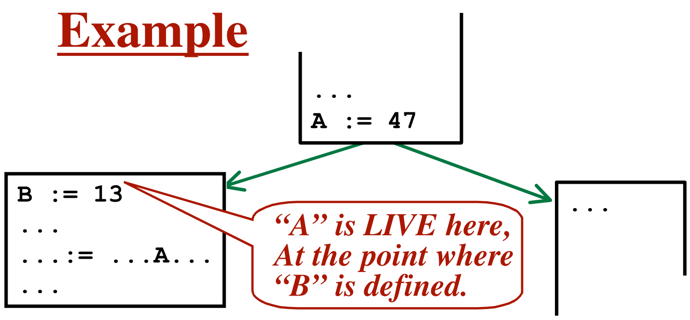
"A" and "B" are simultaneously LIVE.
    Need to add an edge.

Assume K=3 registers are available.
    R0 = Blue
    R1 = Red
    R2 = Green



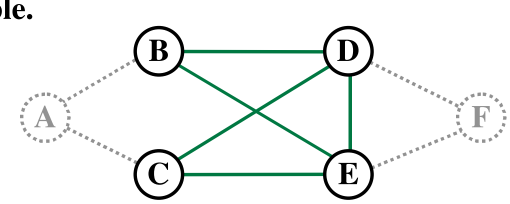
Order Eliminated: A
Reverse Order:

Example



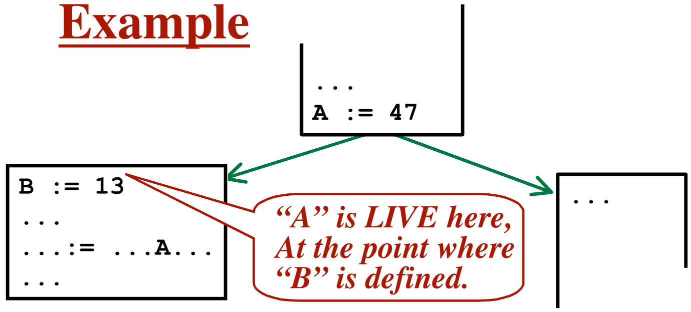
"A" and "B" are simultaneously LIVE.
    Need to add an edge.

Assume K=3 registers are available.
    R0 = Blue
    R1 = Red
    R2 = Green



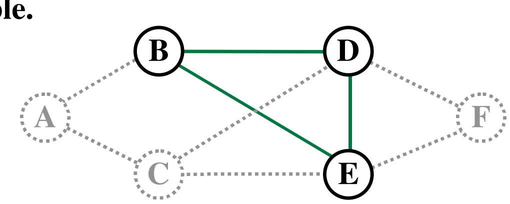
Order Eliminated: AF
Reverse Order:

Example



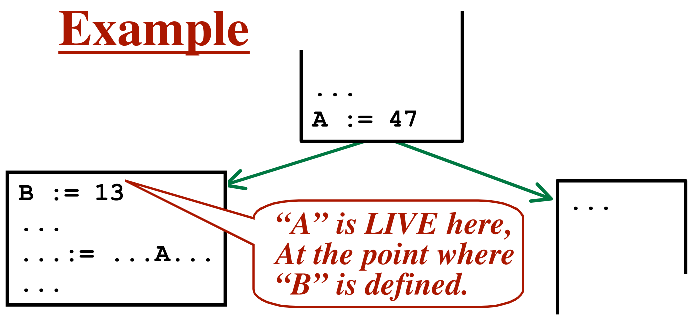
"A" and "B" are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



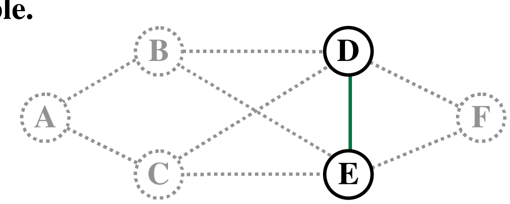
Order Eliminated: AFC
Reverse Order:

Example



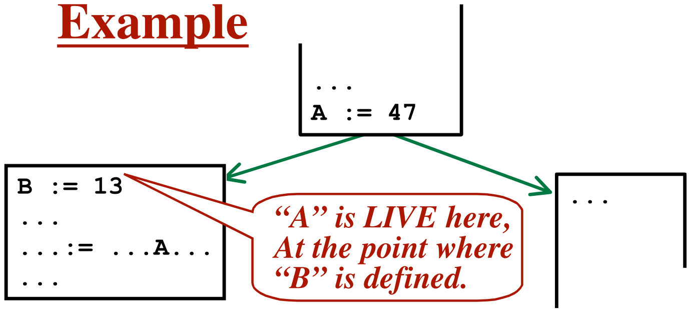
"A" and "B" are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



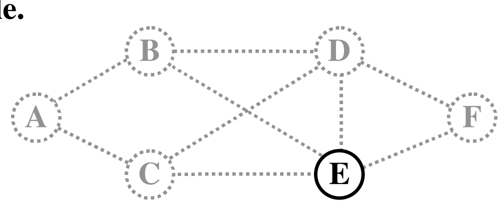
Order Eliminated: AFCB
Reverse Order:

Example



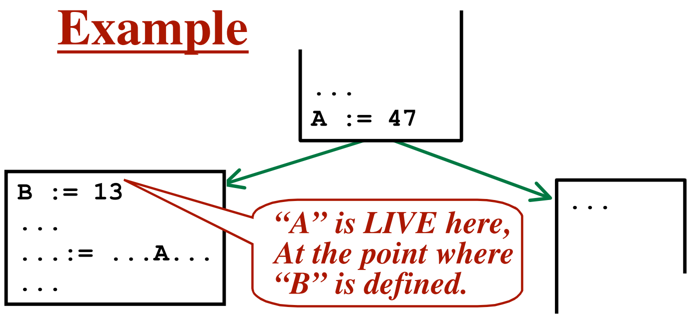
“A” and “B” are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



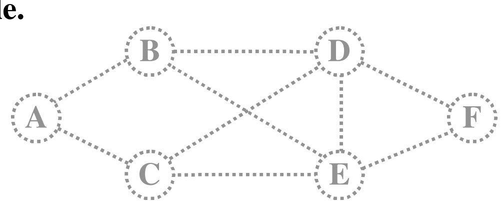
Order Eliminated: AFCBD
Reverse Order:

Example



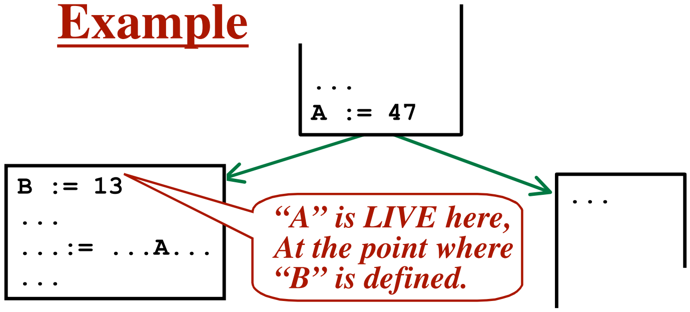
“A” and “B” are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



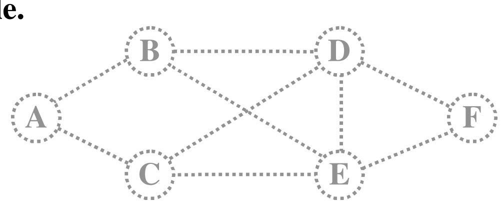
Order Eliminated: AFCBDE
Reverse Order:

Example



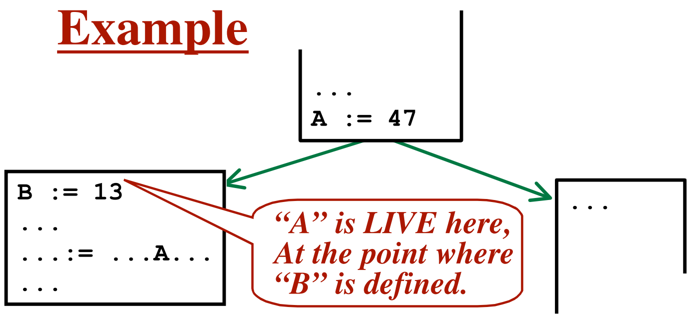
"A" and "B" are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



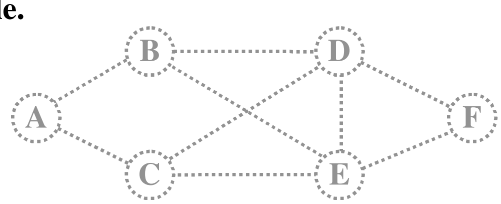
Order Eliminated: AFCBDE
Reverse Order: EDBCFA

Example



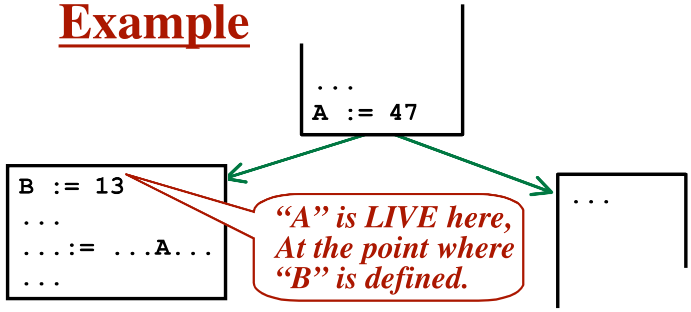
"A" and "B" are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



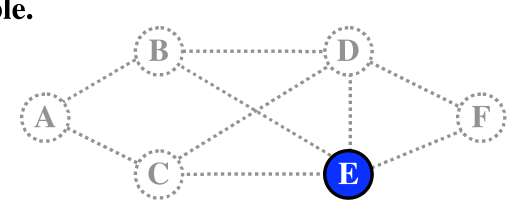
Order Eliminated: AFCBDE
Reverse Order: EDBCFA
Add Nodes Back, assigning colors.

Example



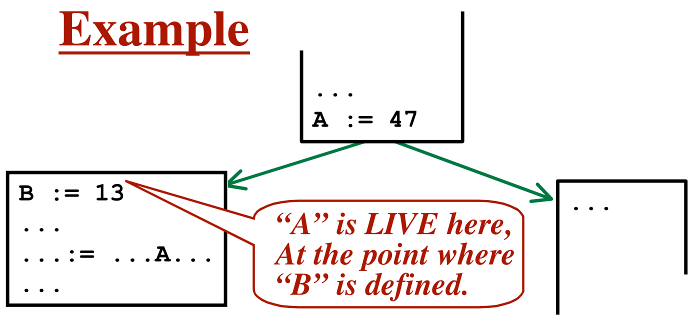
"A" and "B" are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



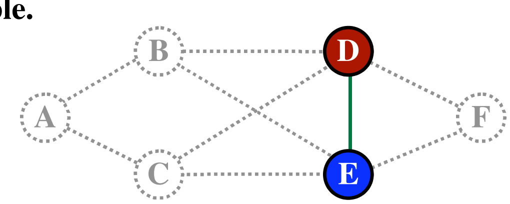
Order Eliminated: AFCBDE
Reverse Order: EDBCFA
Add Nodes Back, assigning colors.

Example



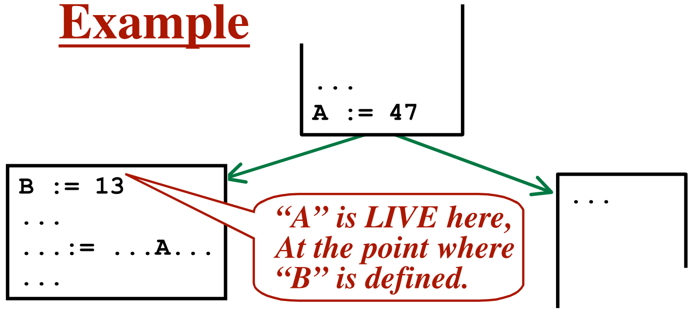
"A" and "B" are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



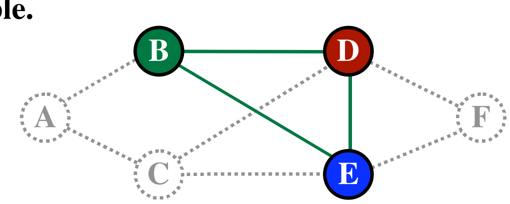
Order Eliminated: AFCBDE
Reverse Order: EDBCFA
Add Nodes Back, assigning colors.

Example



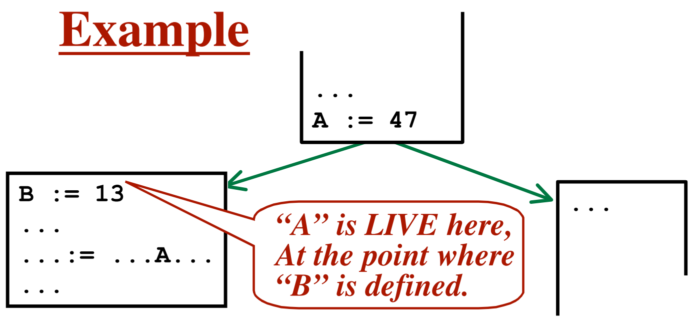
"A" and "B" are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



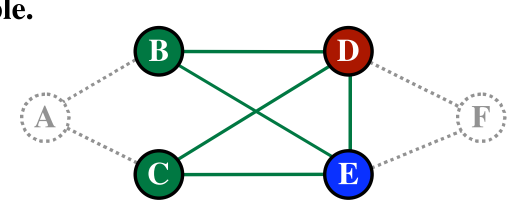
Order Eliminated: AFCBDE
Reverse Order: EDBCFA
Add Nodes Back, assigning colors.

Example



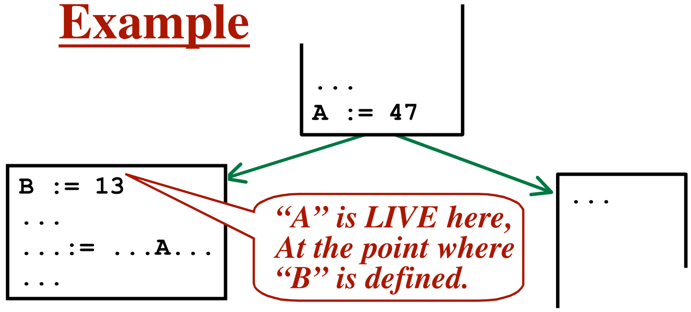
"A" and "B" are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



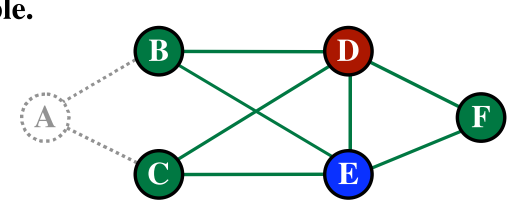
Order Eliminated: AFCBDE
Reverse Order: EDBCFA
Add Nodes Back, assigning colors.

Example



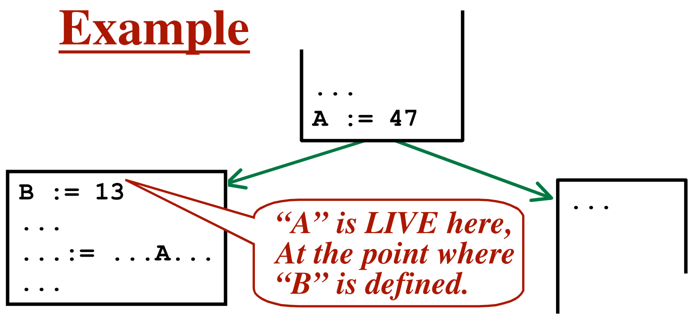
“A” and “B” are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



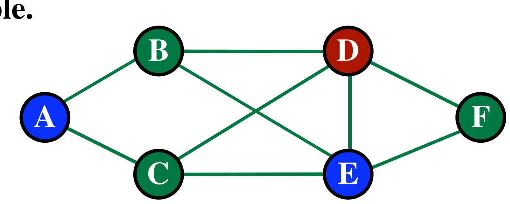
Order Eliminated: AFCBDE
Reverse Order: EDBCFA
Add Nodes Back, assigning colors.

Example



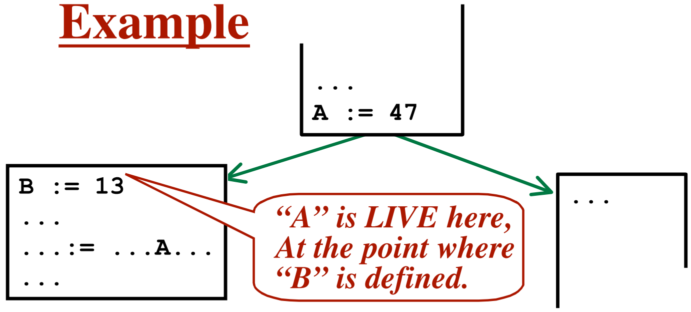
“A” and “B” are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



Order Eliminated: AFCBDE
Reverse Order: EDBCFA
Add Nodes Back, assigning colors.

Example

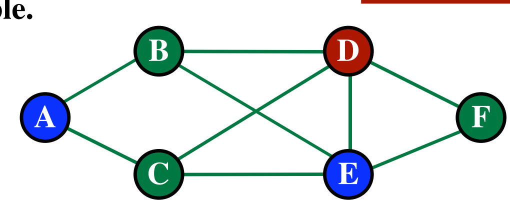


Register Assignment:

A:	R0
B:	R2
C:	R2
D:	R1
E:	R0
F:	R2

"A" and "B" are simultaneously LIVE.
Need to add an edge.

Assume K=3 registers are available.
R0 = Blue
R1 = Red
R2 = Green



Order Eliminated: AFCBDE
Reverse Order: EDBCFA
Add Nodes Back, assigning colors.