Lecture 16

Code generation part 3

Code generation

Hyosu Kim

School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://hcslab.cau.ac.kr

hskimhello@cau.ac.kr, hskim.hello@gmail.com

Code generator

Translates an intermediate representation (e.g., three address code)

into a machine-level code (e.g., assembly code)

Goal of this stage

- Choose the appropriate machine instructions for each intermediate representation instruction
    - With the use of runtime environments

- **Efficiently allocate finite machine resources (e.g., registers, …)**

Through a better register allocation!!

Better register allocation

Goal: reduce the number of memory reads and writes with limited resources

- By using no more registers than those available

- By holding as many variables as possible in registers

  (Instead of holding them in memory)

Key idea!!

Allow to share the same register between different variables a and b

if at any point in the program at most one of a or b is a live variable

Reminder: live variable analysis

A variable is called a live variable if it holds a value that will be needed in the future

- To know whether a variable will be used in the future or not, checks the statements in a basic block in a reverse order
  1) Initially, some small set of variables are known to be live (e.g., variables will be used in the next block)

  2) Just before executing the statement **a = … b …**
     - **a is not alive** because its value will be newly overwritten
     - **b is alive** because it will be used

Three-address code	Live variables
(the value of a, b, c will be used in the future)	{a, b}
d = a + b;	
(the value of b and d will be used in the future)	{b, d}

Reminder: live variable analysis

A variable is called a live variable if it holds a value that will be needed in the future

- To know whether a variable will be used in the future or not,
  checks the statements in a basic block in a reverse order
  1) Initially, some small set of variables are known to be live

At any point in the program, at most one of a and d is a live variable!!

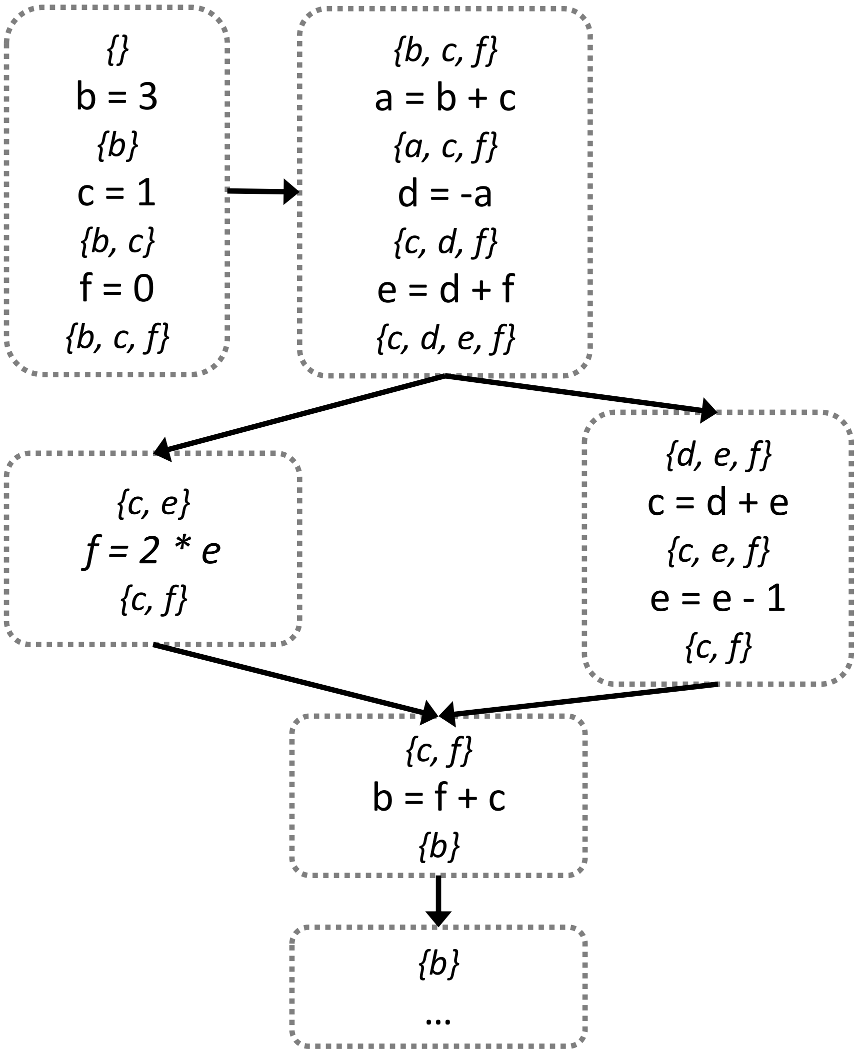
"a and d can share the same register"

- **b is alive** because it will be used

Three-address code	**Live variables**
(the value of a, b, c will be used in the future)	{**a**, b}
d = a + b;	
(the value of b and d will be used in the future)	{b, **d**}

Register allocation with live variable analysis

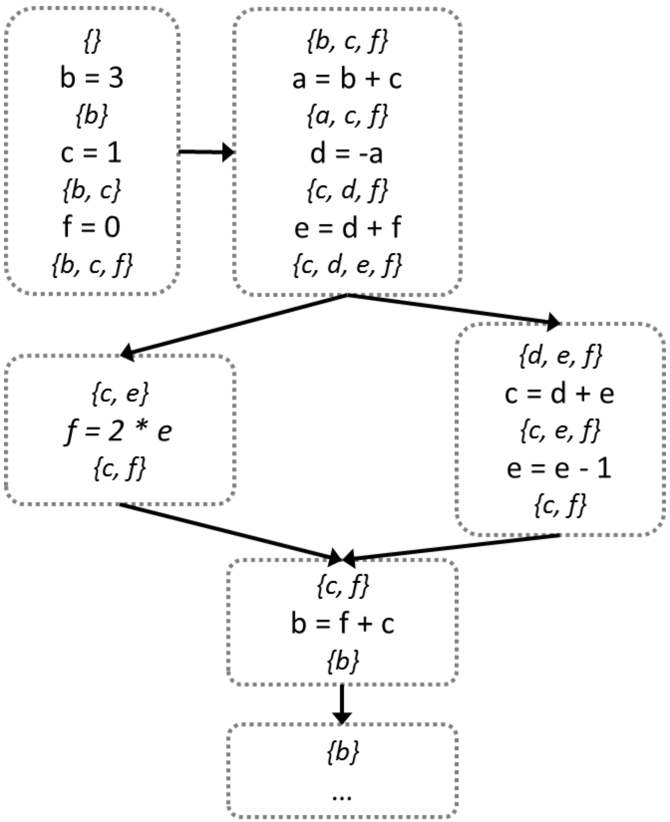
Step 1: Do live variable analysis globally

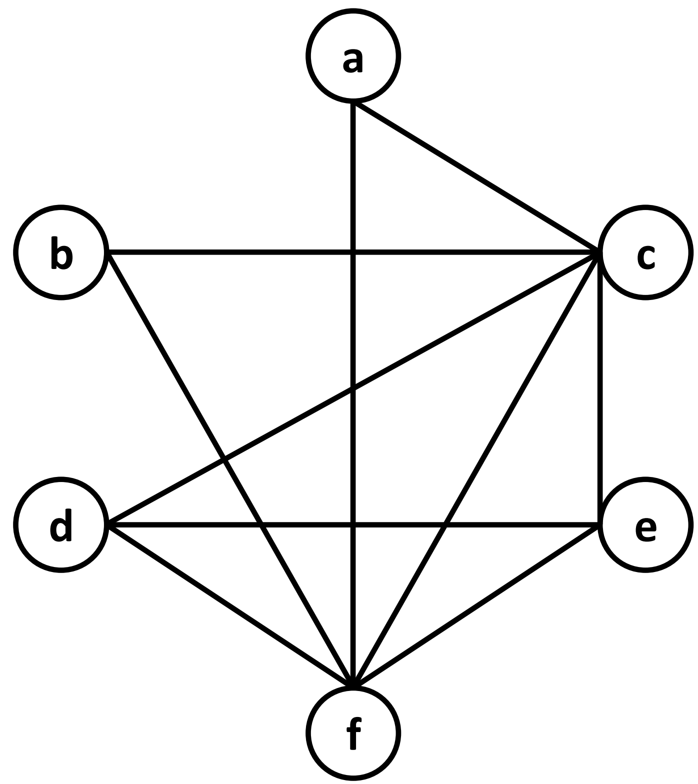


Register allocation with live variable analysis

Step 2: Construct RIG (Register Interference Graph)

- Node: each variable
- Edge between t0 and t1 represents that they are alive simultaneously at some point in the program

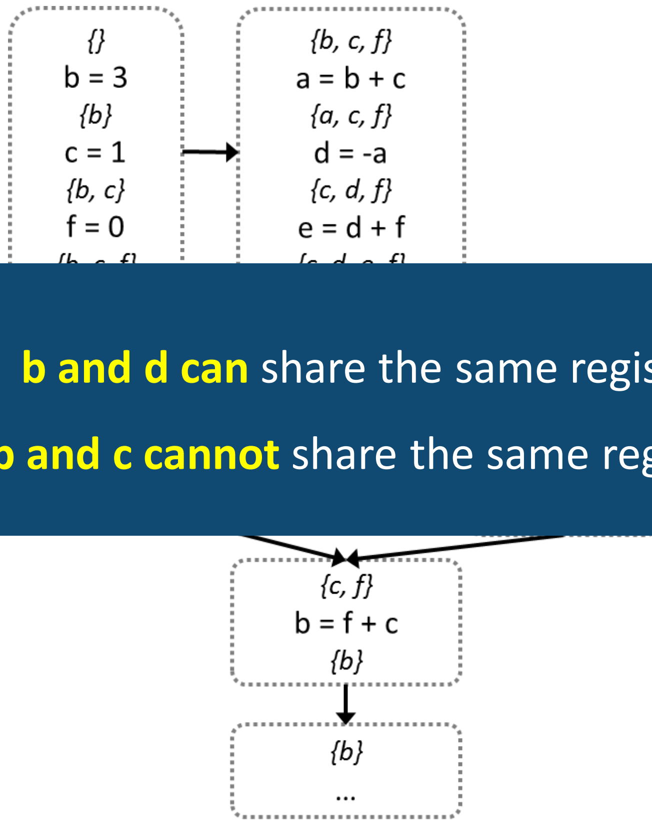




Register allocation with live variable analysis

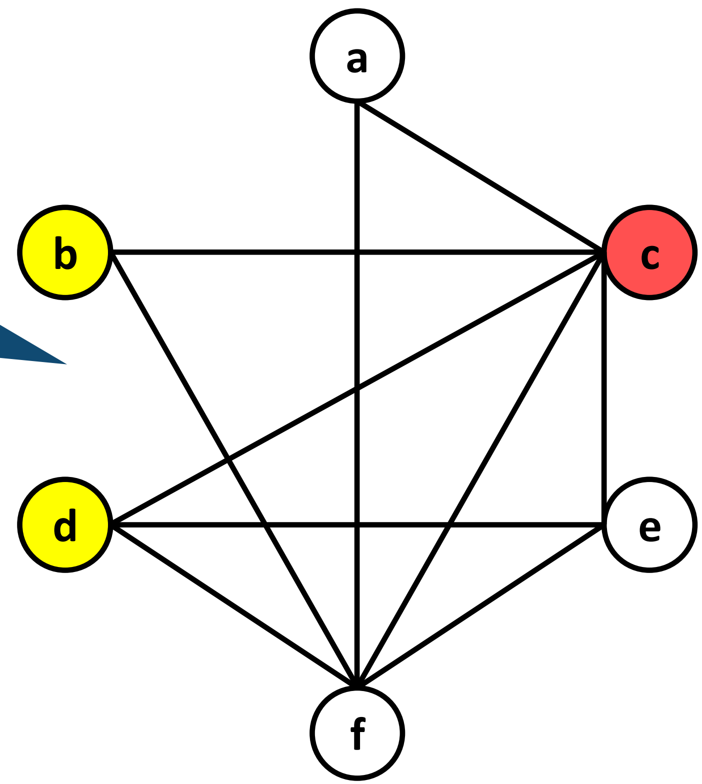
Step 2: Construct RIG (Register Interference Graph)

- Node: each variable
- Edge between t0 and t1 represents that they are alive simultaneously at some point in the program



b and d can share the same register

b and c cannot share the same register

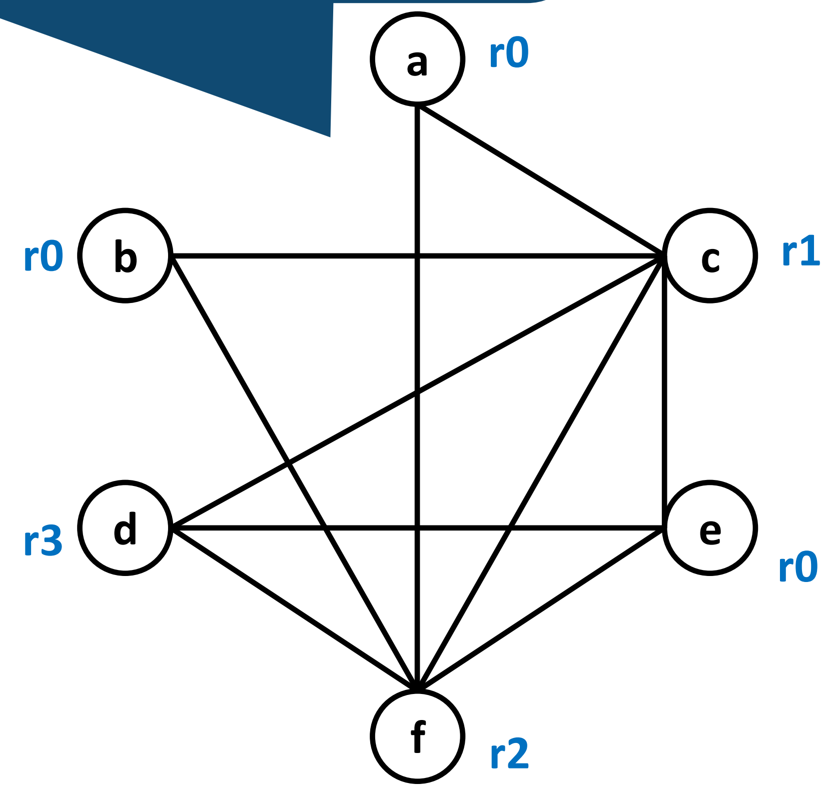


Register allocation with live variable analysis

Question (when we have **K** available registers)

Is it possible to allocate K registers

holding all the live variables in registers at any point in the program??



e.g., when K = 4

Yes, we can!!

Graph coloring = register allocation

New question (when we have **K** available colors)

Is it possible to allocate K colors

making nodes connected by an edge have different colors??

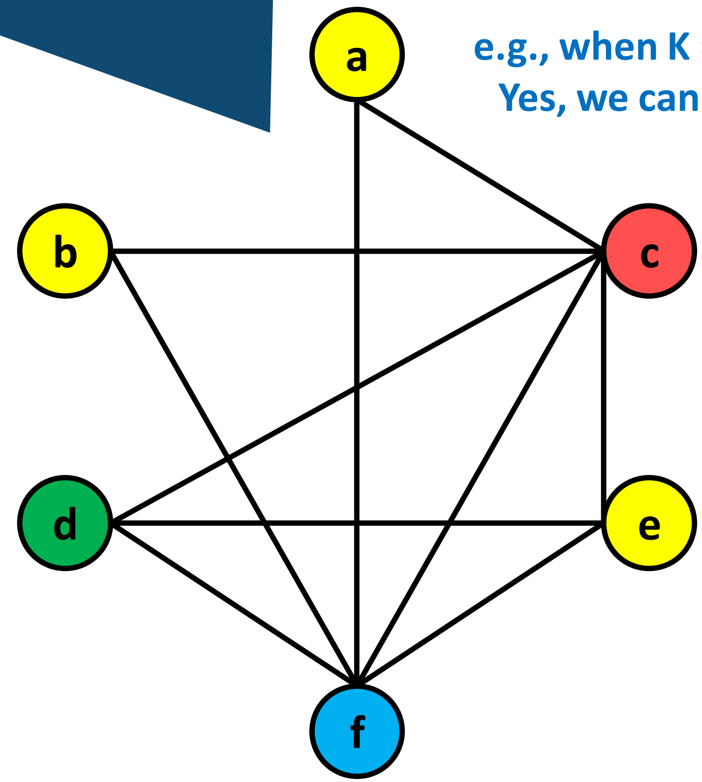
- Color = register
- If the RIG is **K-colorable**,

  then there is a register allocation that uses

  no more than K registers
- **Let's check**

  whether the given RIG is K-colorable or not

  where K is the number of available registers



e.g., when K = 4
Yes, we can!!

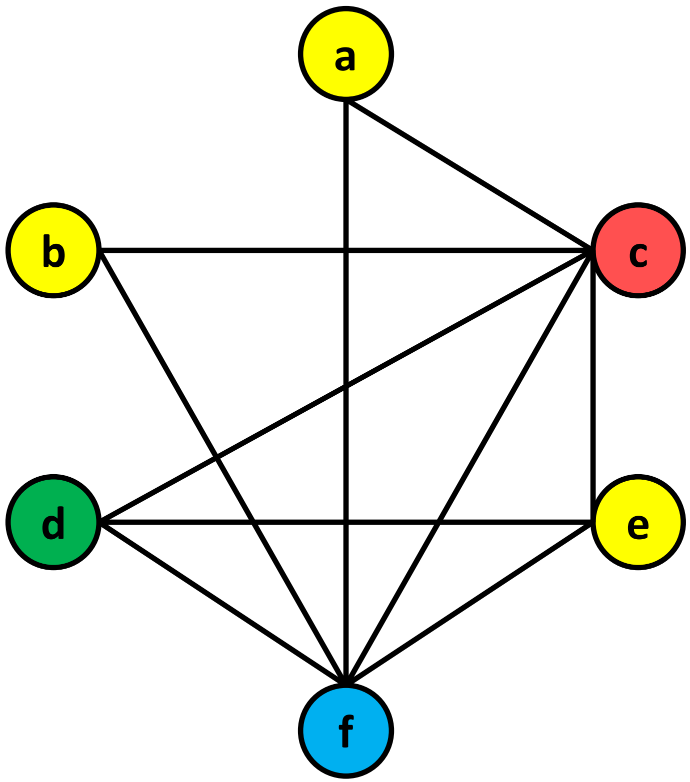
Graph coloring

But, the graph coloring problem is known as an NP-hard problem.

- No efficient algorithms are known

  The computation complexity of the most efficient one: $O(2^n n)$

  (n: the number of variables)

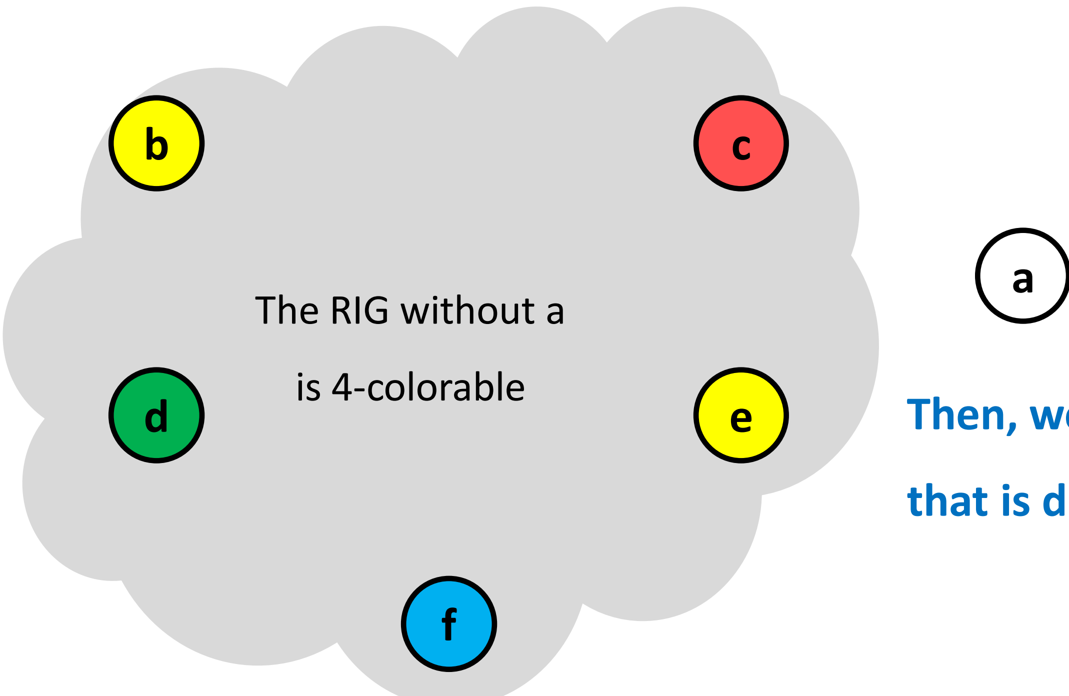


We need to use heuristics!!!

Heuristics for graph coloring

Key idea

- Eliminate a node T which has neighbors fewer than K
- If the remaining RIG is K-colorable, then the RIG with T is also K-colorable



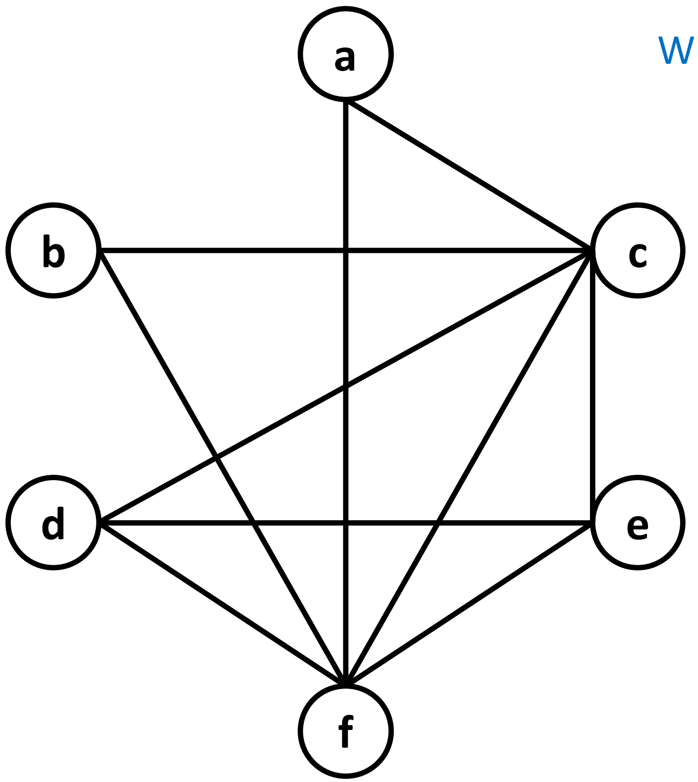
a has 3 neighbors

Then, we can allocate some color for a that is different from those of its neighbors

Heuristics for graph coloring

Implementation

- Step 1: Pick a node T with fewer than K neighbors
- Step 2: Eliminate T from the RIG and put it on a stack



When the available number of registers K = 4

[stack]: {}

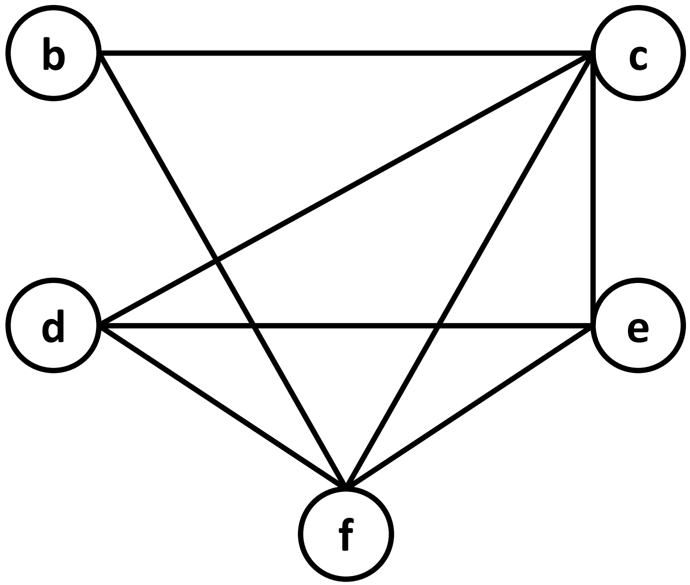
Heuristics for graph coloring

Implementation

- Step 1: Pick a node T with fewer than K neighbors
- Step 2: Eliminate T from the RIG and put it on a stack

When the available number of registers K = 4

[stack]: {a}



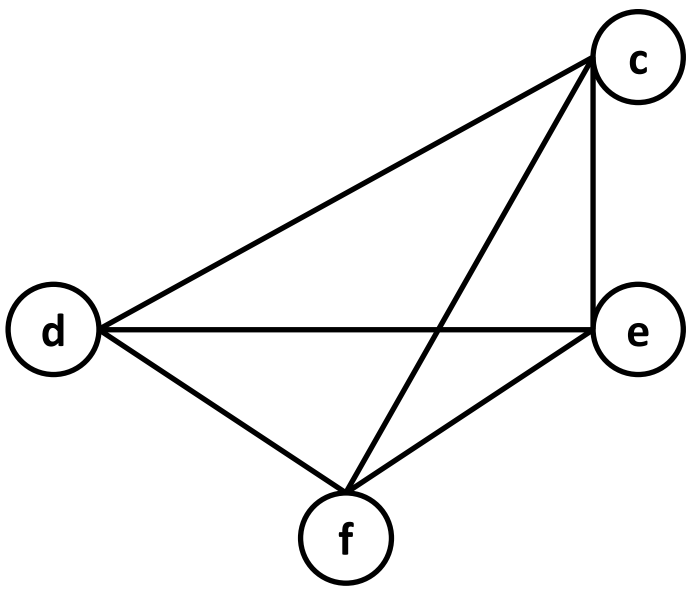
Heuristics for graph coloring

Implementation

- Step 1: Pick a node T with fewer than K neighbors
- Step 2: Eliminate T from the RIG and put it on a stack
- Step 3: Repeat step 1 ~ 2 until there is no node in the RIG

When the available number of registers K = 4

[stack]: {b, a}



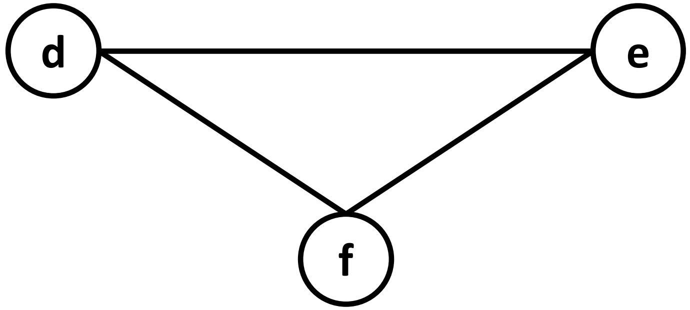
Heuristics for graph coloring

Implementation

- Step 1: Pick a node T with fewer than K neighbors
- Step 2: Eliminate T from the RIG and put it on a stack
- Step 3: Repeat step 1 ~ 2 until there is no node in the RIG

When the available number of registers K = 4

[stack]: {c, b, a}



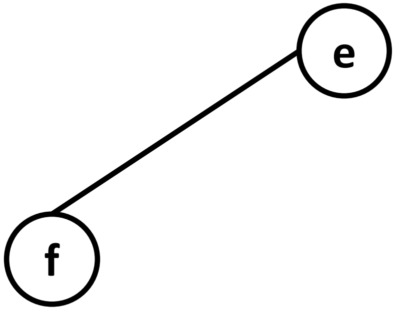
Heuristics for graph coloring

Implementation

- Step 1: Pick a node T with fewer than K neighbors
- Step 2: Eliminate T from the RIG and put it on a stack
- Step 3: Repeat step 1 ~ 2 until there is no node in the RIG

When the available number of registers K = 4

[stack]: {d, c, b, a}



Heuristics for graph coloring

Implementation

- Step 1: Pick a node T with fewer than K neighbors
- Step 2: Eliminate T from the RIG and put it on a stack
- Step 3: Repeat step 1 ~ 2 until there is no node in the RIG

When the available number of registers K = 4

[stack]: {f, e, d, c, b, a}

Heuristics for graph coloring

Implementation

- Step 4: Pick the node in the top-of-stack
- Step 5: Assign a color different from those already assigned to colored neighbors

When the available number of registers K = 4

[stack]: {e, d, c, b, a}

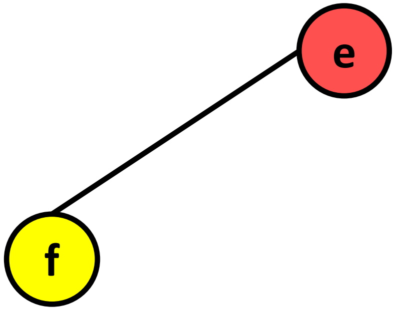
Heuristics for graph coloring

Implementation

- Step 4: Pick the node in the top-of-stack
- Step 5: Assign a color different from those already assigned to colored neighbors
- Step 6: Repeat step 4 ~ 5 until the RIG is completely restored

When the available number of registers K = 4

[stack]: {d, c, b, a}



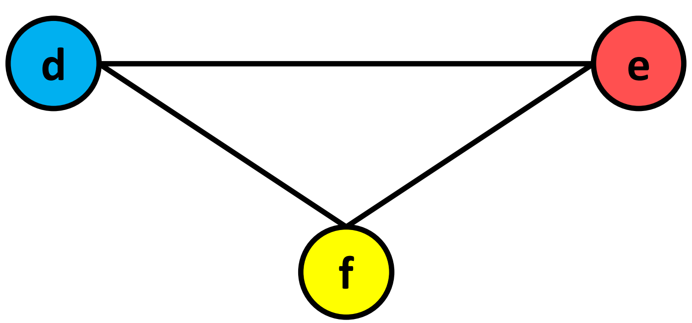
Heuristics for graph coloring

Implementation

- Step 4: Pick the node in the top-of-stack
- Step 5: Assign a color different from those already assigned to colored neighbors
- Step 6: Repeat step 4 ~ 5 until the RIG is completely restored

When the available number of registers K = 4

[stack]: {c, b, a}



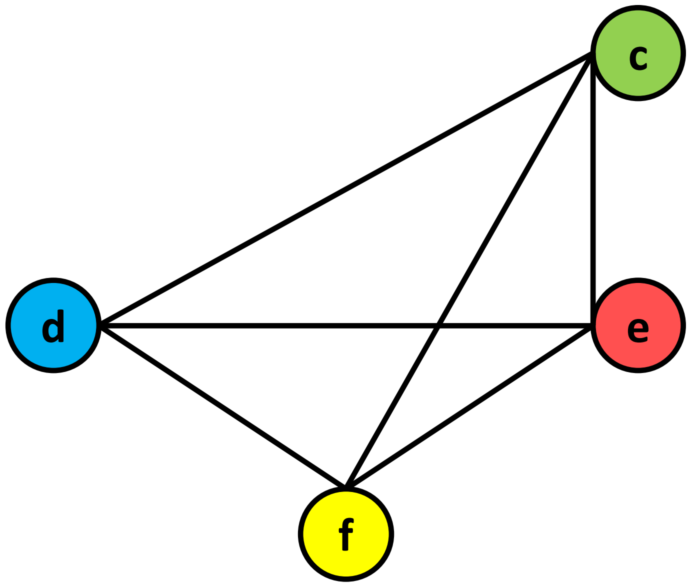
Heuristics for graph coloring

Implementation

- Step 4: Pick the node in the top-of-stack
- Step 5: Assign a color different from those already assigned to colored neighbors
- Step 6: Repeat step 4 ~ 5 until the RIG is completely restored

When the available number of registers K = 4

[stack]: {b, a}



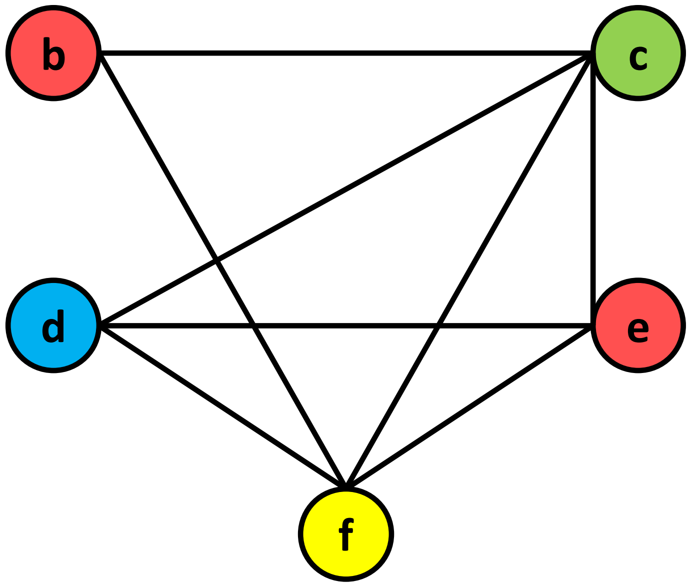
Heuristics for graph coloring

Implementation

- Step 4: Pick the node in the top-of-stack
- Step 5: Assign a color different from those already assigned to colored neighbors
- Step 6: Repeat step 4 ~ 5 until the RIG is completely restored

When the available number of registers K = 4

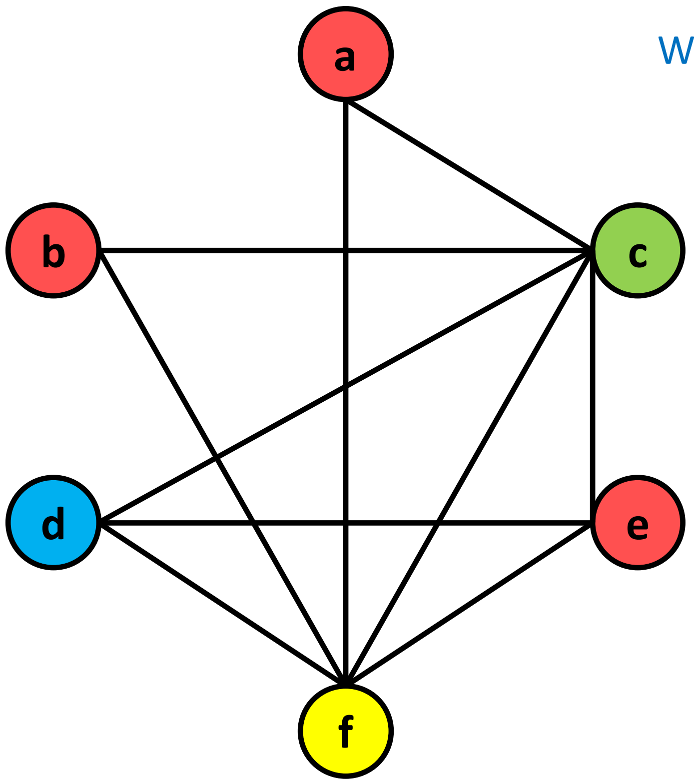
[stack]: {a}



Heuristics for graph coloring

Implementation

- Step 4: Pick the node in the top-of-stack
- Step 5: Assign a color different from those already assigned to colored neighbors
- Step 6: Repeat step 4 ~ 5 until the RIG is completely restored



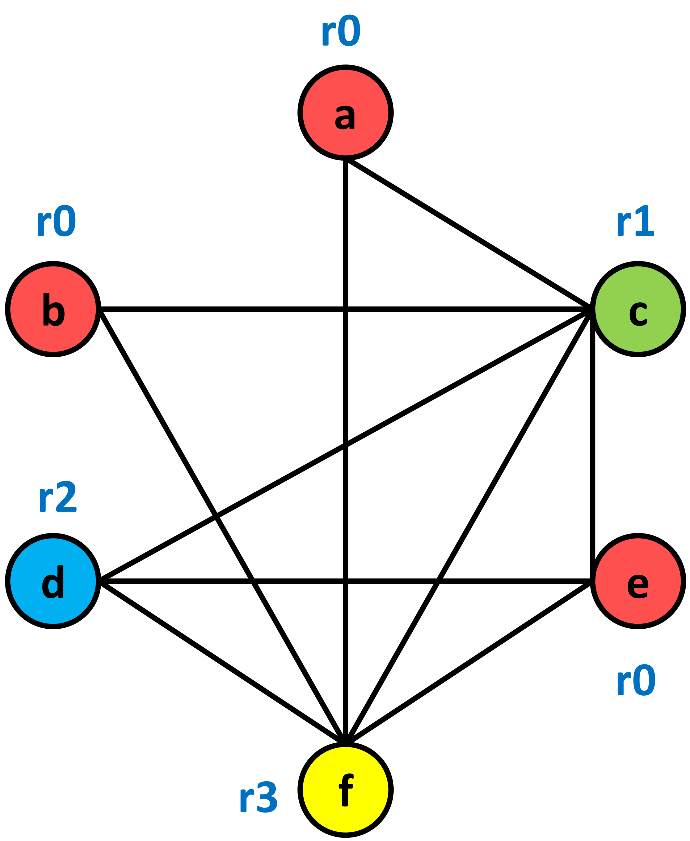
When the available number of registers K = 4

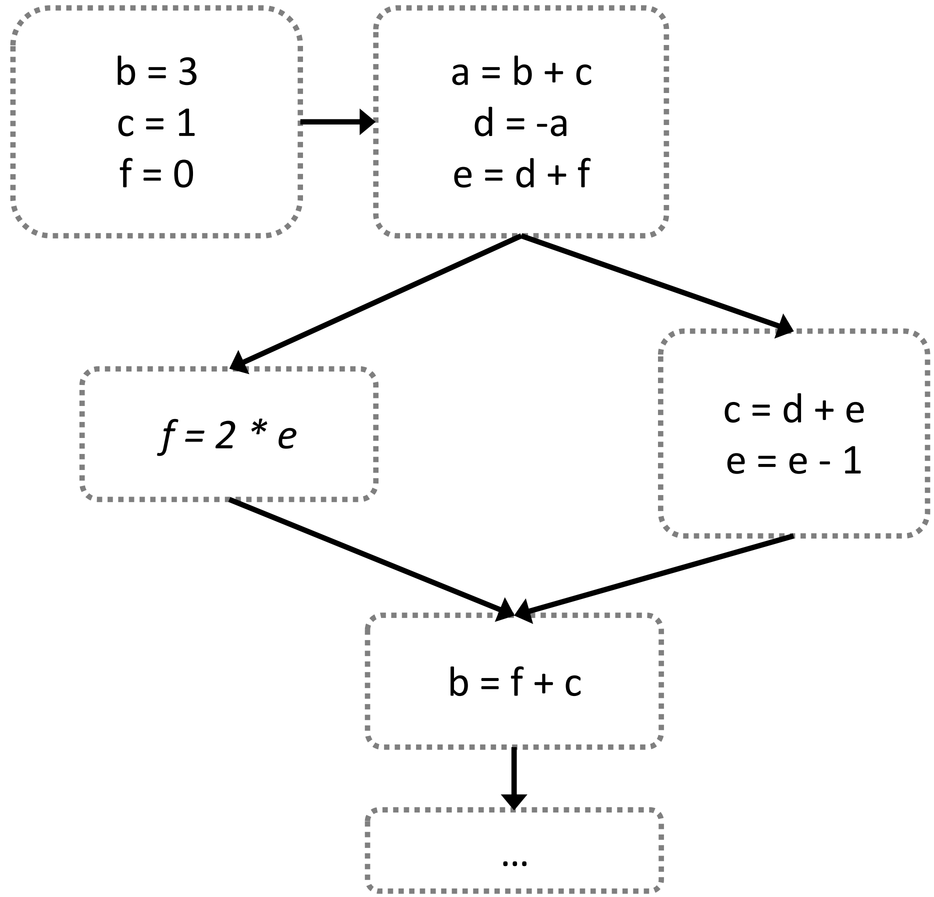
[stack]: {}

Heuristics for graph coloring

The given RIG is 4-colorable

We can avoid **unnecessary memory read / write** when we execute the given code **with only 4 registers**

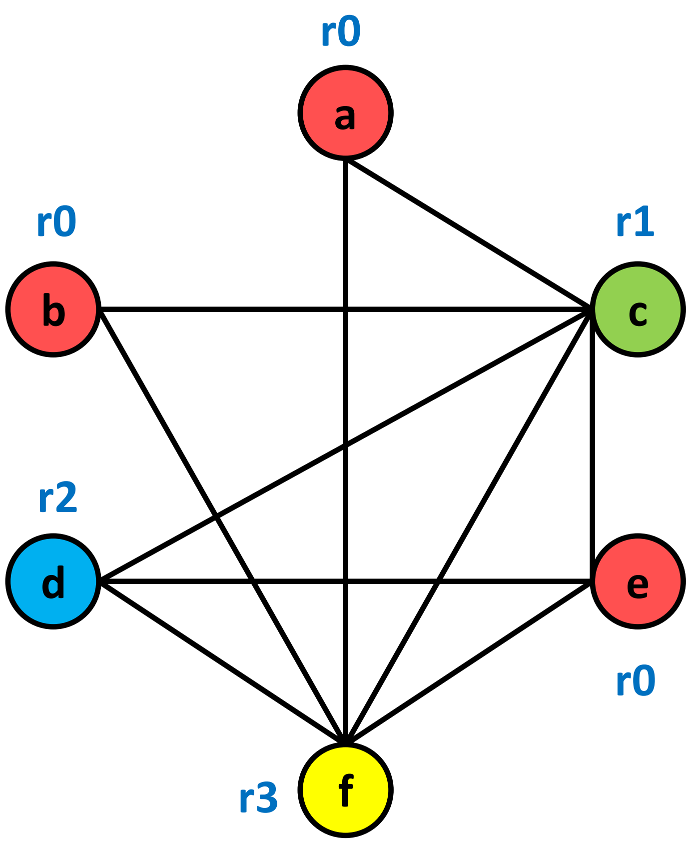


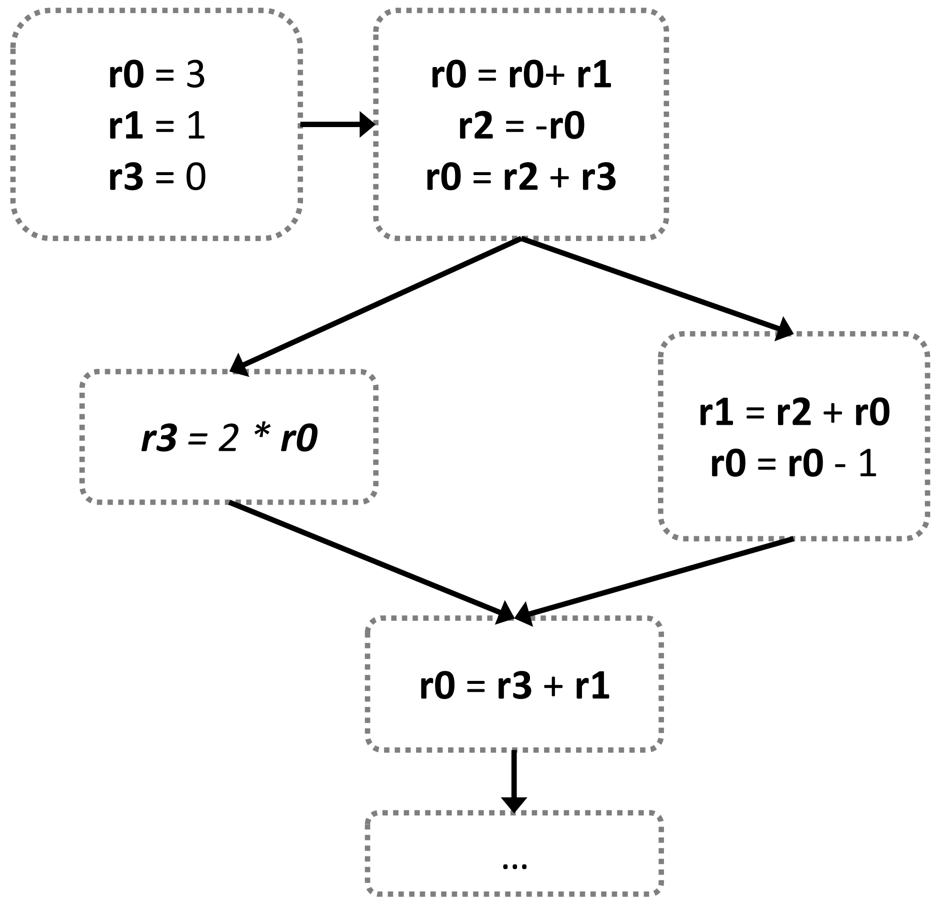


Heuristics for graph coloring

The given RIG is 4-colorable

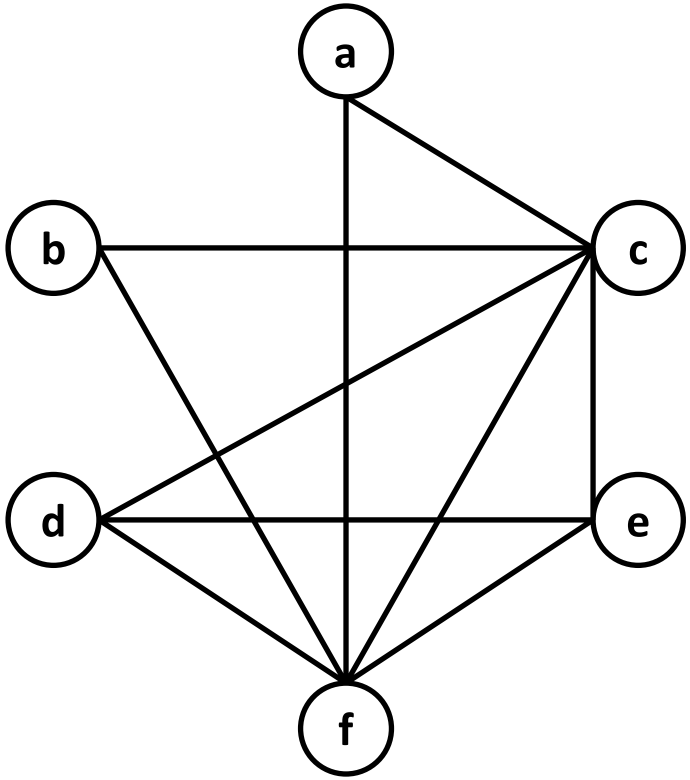
We can avoid **unnecessary memory read / write** when we execute the given code **with only 4 registers**





Heuristics for graph coloring

What will happen if K = 3???



[stack]: {}

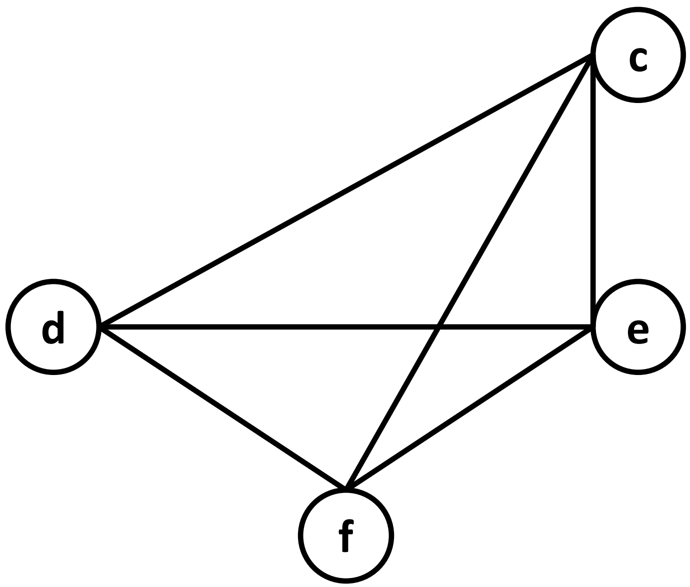
Heuristics for graph coloring

What will happen if K = 3???

There is no node with fewer than 3 neighbors

- c, d, e, and f have 3 neighbors

[stack]: {b, a}



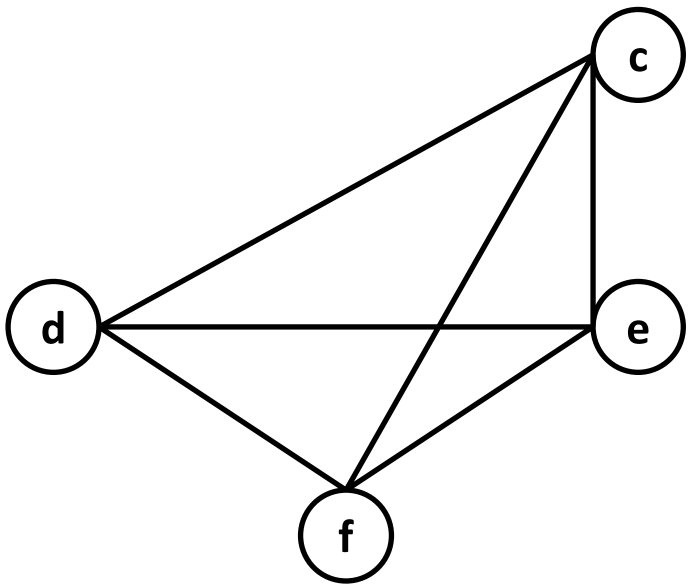
Heuristics for graph coloring

Solution: spilling

Step 1: Pick a node as a candidate for **spilling**

- A spilled variable will live in memory (not in register)

Step 2: Allocate a memory location for the variable



Example

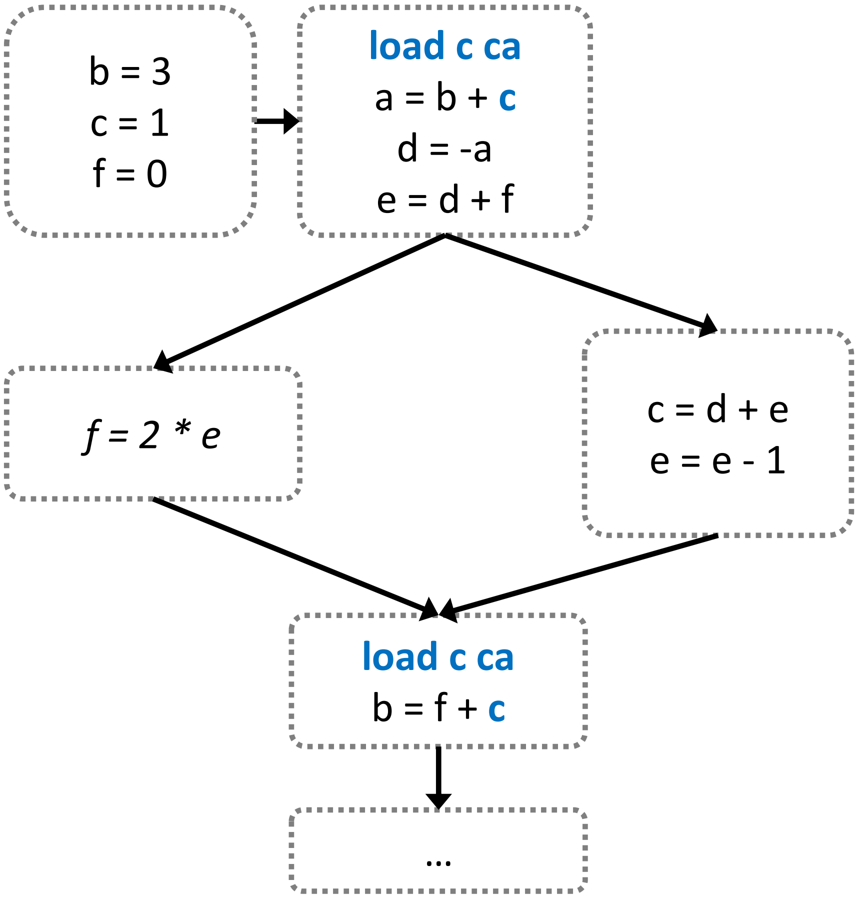
- Candidate for spilling: **c**
- The memory location of c: **ca**

Heuristics for graph coloring

Solution: **spilling**

Step 3: Rewrite the intermediate code

by inserting 1) **load c ca** before each operation that reads **c**

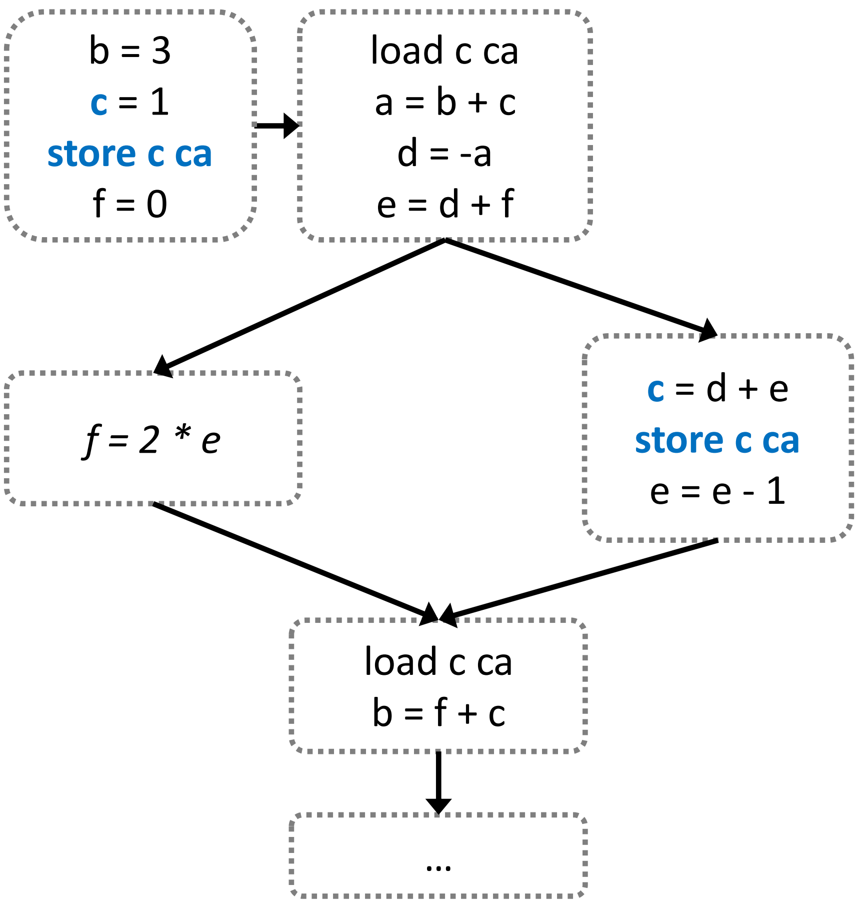


Heuristics for graph coloring

Solution: spilling

Step 3: Rewrite the intermediate code

by inserting 2) **store c ca** after each operation that writes **c**

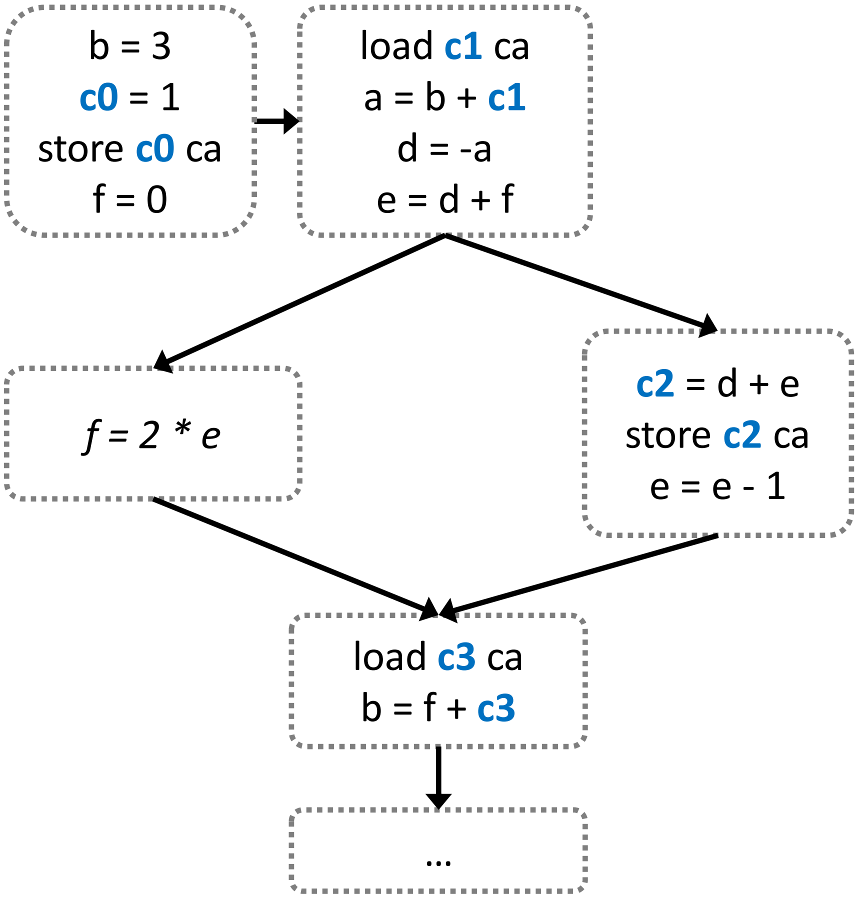


Heuristics for graph coloring

Solution: spilling

Step 3: Rewrite the intermediate code

3) by assigning different names whenever the spilled variable is used



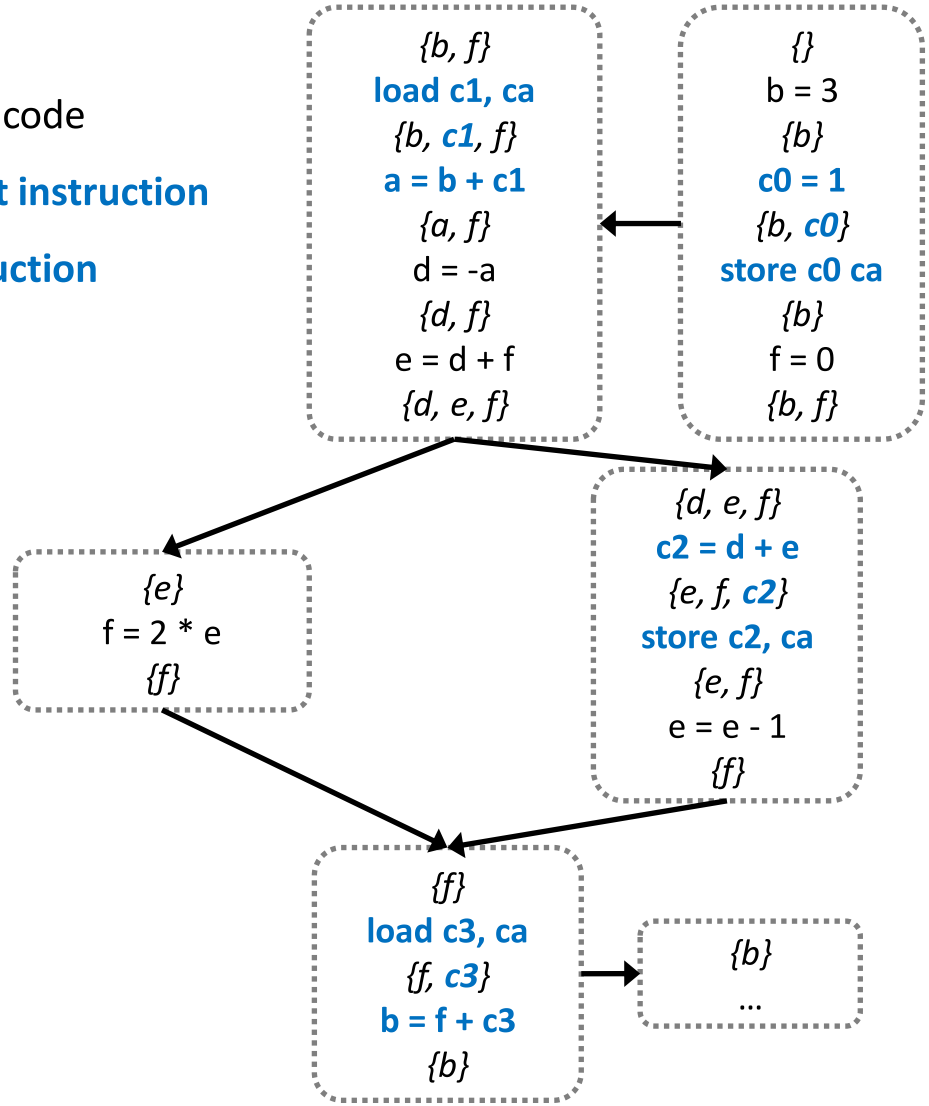
Heuristics for graph coloring

Solution: **spilling**

Step 4: Do live variable analysis with the rewritten code

- **ci** is alive only **between load ci, ca and the next instruction**

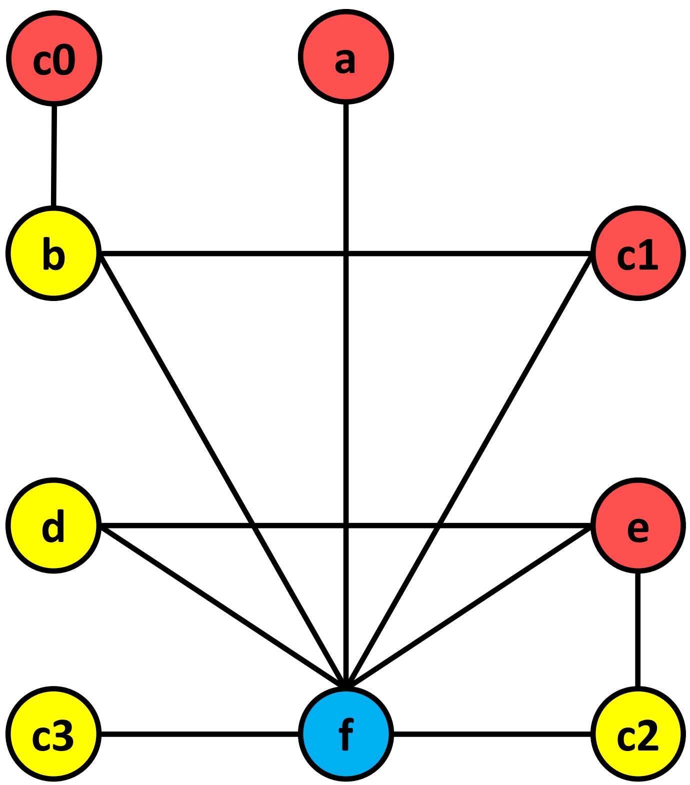
  / **between store ci, ca and the preceding instruction**

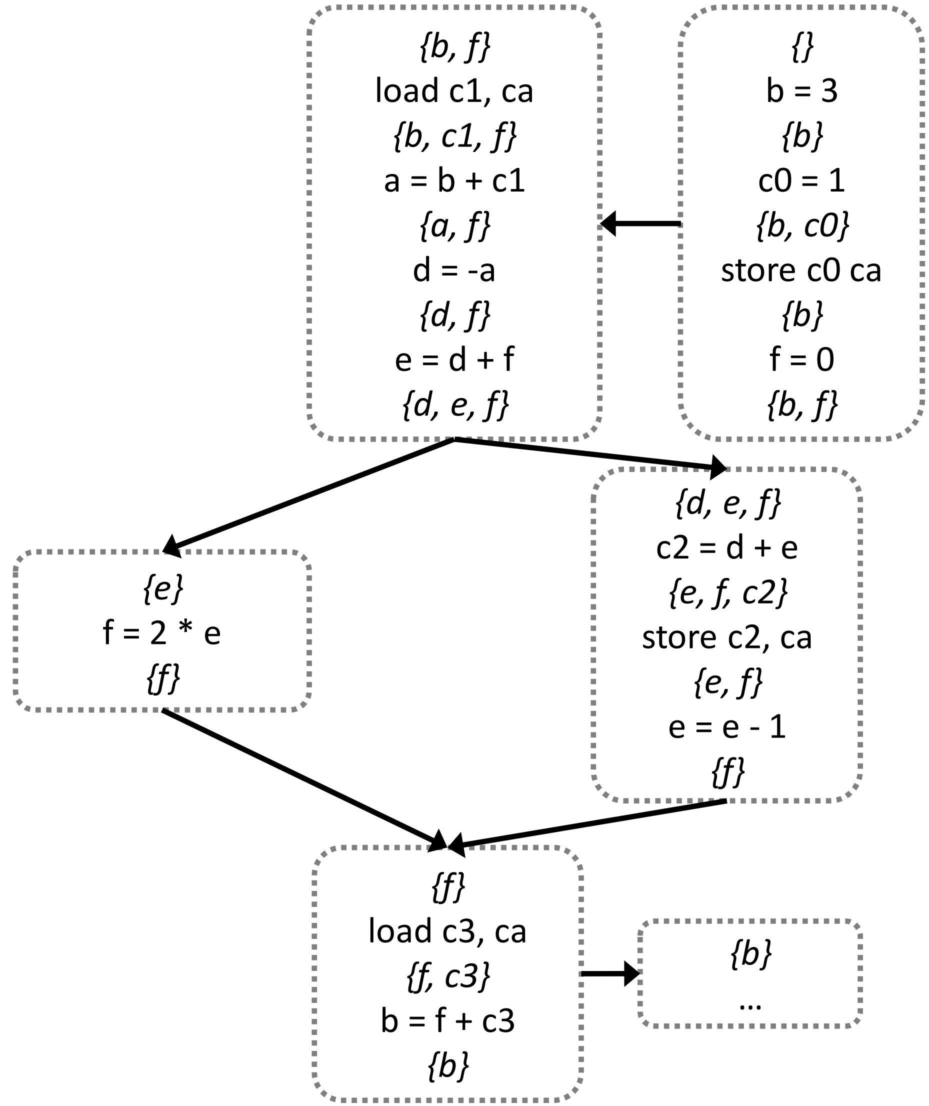


Heuristics for graph coloring

Solution: spilling

Step 5: Construct RIG and do graph coloring again

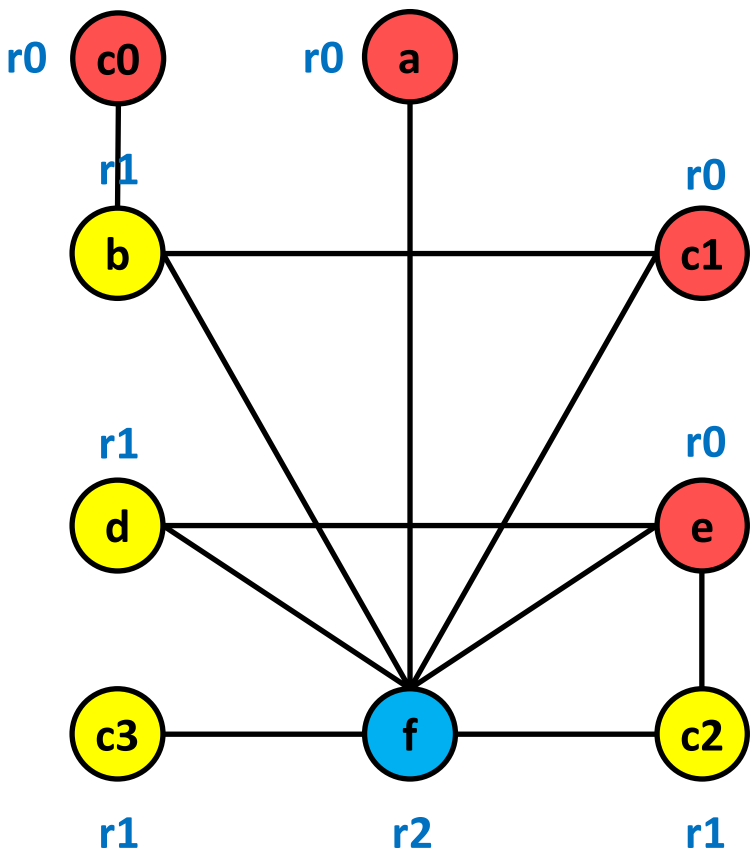


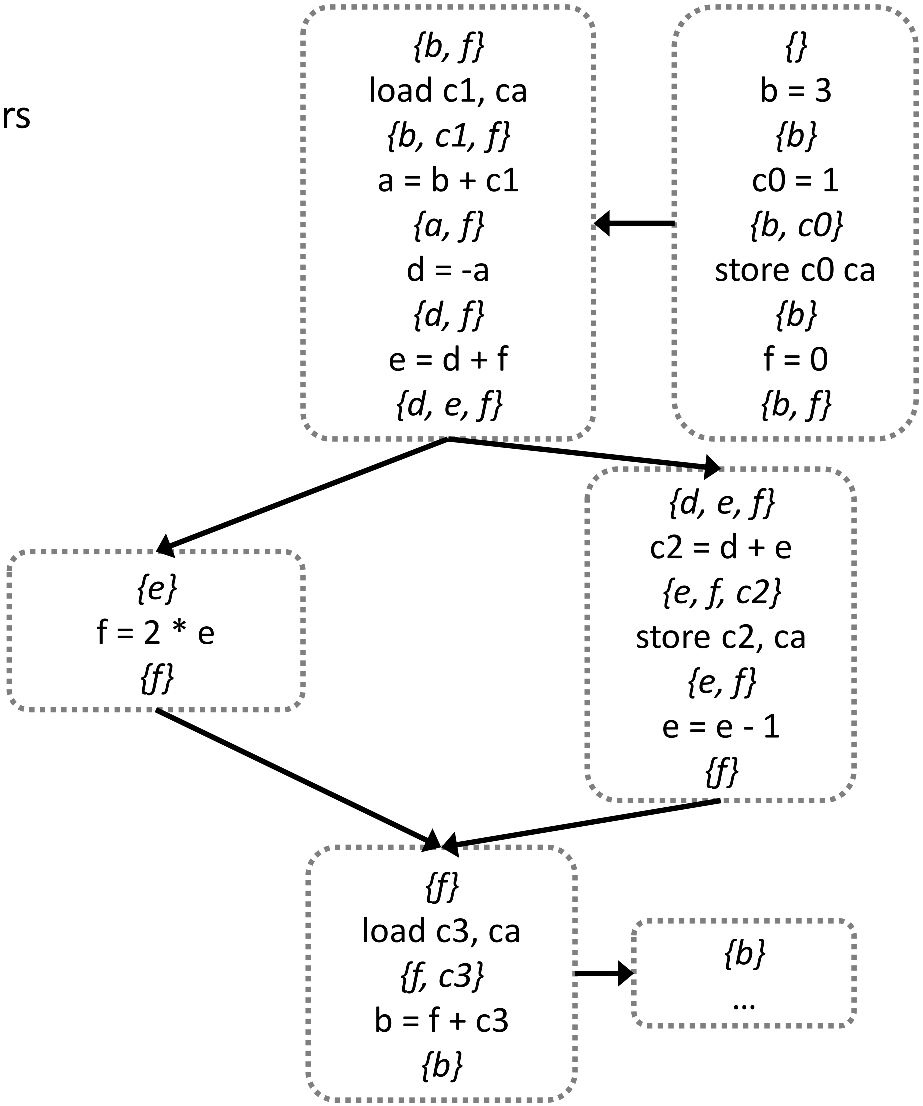


Heuristics for graph coloring

Solution: **spilling**

Step 6: Translate variables into the assigned registers





Heuristics for graph coloring

Solution: **spilling**

NOTE

- Additional spilling might be required if the RIG is not still K-colorable after spilling

- **One important problem is to decide what to spill**
  - There is no correct answer

    Any spilling decision eventually can make the RIG K-colorable
  - We can choose a candidate for spilling **in a different way depending on the situation**
    - Examples
      - Variables which have the most neighbors
      - Variables used the least frequently
      - Variables placed outside a loop

Summary: code generator

Translates an intermediate representation (e.g., three address code)

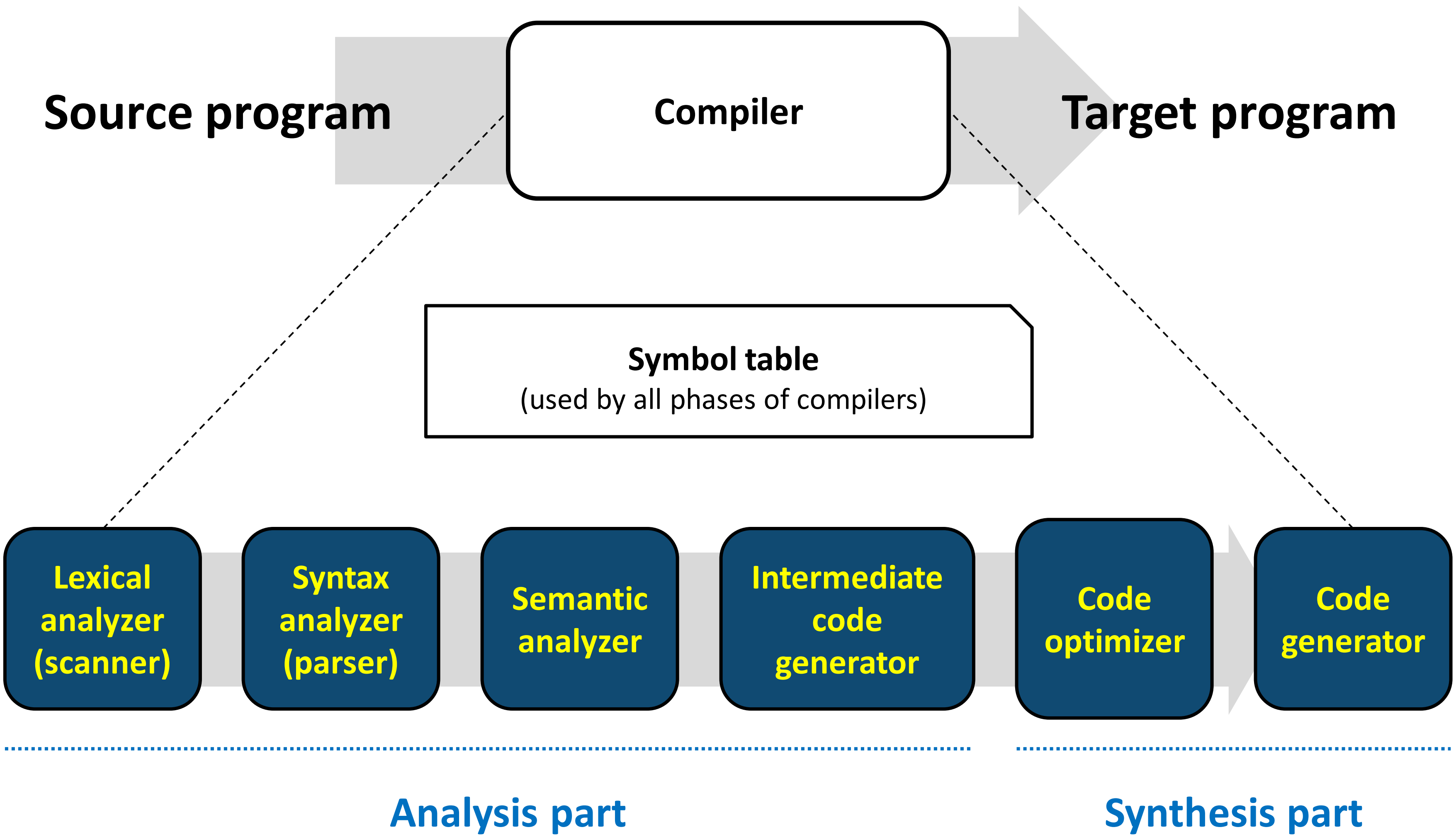
into a machine-level code (e.g., assembly code)

Goal of this stage

- **Choose the appropriate machine instructions for each intermediate representation instruction**
    - **With the use of runtime environments**

- **Efficiently allocate finite machine resources (e.g., registers, …)**

Overview



Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part