Compiler Optimization

Compiler optimizations transform code

- Code optimization transforms code to equivalent code
  - ... but with better performance
- The code transformation can involve either
  - **Replacing** code with more efficient code
  - **Deleting** redundant code
  - **Moving** code to a position where it is more efficient
  - **Inserting** new code to improve performance

The four categories of code transformations

- Replacing code (e.g. **strength reduction**)

  A=2*a; ≡ A=a«1;

- Deleting code (e.g. **dead code elimination**)

  A=2; A=y; ≡ A=y;

- Moving code (e.g. **loop invariant code motion**)

```
for (i = 0; i < 100; i++) { sum += i + x * y; }
≡
t = x * y;
for (i = 0; i < 100; i++) { sum += i + t; }
```

- Inserting code (e.g. **data prefetching**)

```
for (p = head; p != NULL; p = p->next)
{ /* do work on node p */ }
≡
for (p = head; p != NULL; p = p->next)
{ prefetch(p->next); /* do work on node p */ }
```

Compiler optimization categories according to range

- How much code does the compiler view while optimizing?
  - The wider the view, the more powerful the optimization

- Axis 1: optimize across control flow?
  - **Local optimization**: optimizes only within straight line code
  - **Global optimization**: optimizes across control flow (if,for,...)

- Axis 2: optimize across function calls?
  - **Intra-procedural optimization**: only within function
  - **Inter-procedural optimization**: across function calls

- The two axes are orthogonal (any combination is possible)

Local vs. Global Constant Propagation

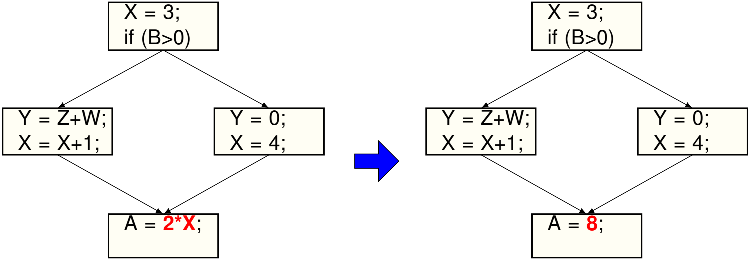
- **Constant propagation**
  - ➢ Optimization: if **x= y op z** and y and z are constants then compute at compile time and replace
- Local Constant Propagation

```
X = 3;
X = X+1;
A = X*2;
```

➡

```
X = 3;
X = X+1;
A = 8;
```

- Global Constant Propagation



Intra- vs. Inter-procedural Constant Propagation

- Intra-procedural Constant Propagation

```
X = 3;
X = X+1;
A = X*2;
```

→

```
X = 3;
X = X+1;
A = 8;
```

- Inter-procedural Constant Propagation

```
X = 3;
foo(X);

void foo(int arg) {
  arg = arg+1;
  A = arg*2;
}
```

→

```
X = 3;
foo(X);

void foo(int arg) {
  arg = arg+1;
  A = 8;
}
```

  - Assuming all other calls to foo always pass in constant 3

Control Flow Analysis

Basic Block

- A function body is composed of one or more **basic blocks.**
- **Basic block**: a maximal sequence of instructions that
  - Has no jumps into the block other than the first instruction
  - Has no jumps out of the block other than the last instruction
- That means:
  - No instruction other than the first is a jump target
  - No instruction other than the last is a jump or branch
- Either all instructions in basic block execute or none
  - Smallest unit of execution in control flow analysis
  - Hence the descriptor "basic" in the name

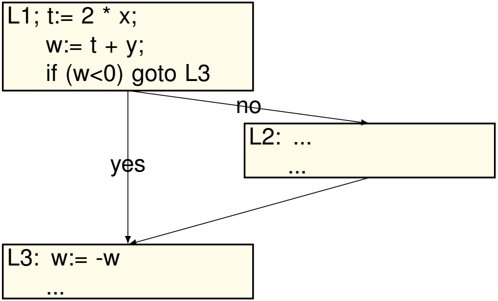
Control Flow Graph

- A **Control Flow Graph (CFG)** is a directed graph in which
  - Nodes are basic blocks
  - Edges represent flows of execution between basic blocks

- CFGs are widely used to represent a program for analysis
- CFGs are especially essential for global optimizations

Control Flow Graph Example

```
L1; t:= 2 * x;
    w:= t + y;
    if (w<0) goto L3
L2:  ...
    ...
L3:  w:= -w
    ...
```

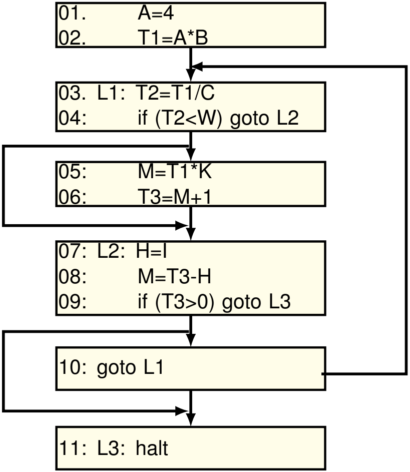


Construction of CFG

- Step 1: partition code into basic blocks
  - Identify **leader** instructions, where a leader is either:
    - the first instruction of a program, or
    - the target of any jump/branch, or
    - an instruction immediately following a jump/branch
  - Create a basic block out of each leader instruction
  - Expand basic block by adding subsequent instructions (Stopping when the next leader instruction is encountered)
- Step 2: add edge between two basic blocks B1 and B2 if
  - there exist a jump/branch from B1 to B2, or
  - B2 follows B1, and B1 does not end with jump/branch

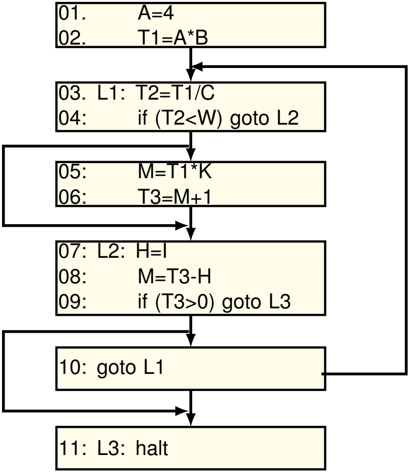
Example

```
01.        A=4
02.        T1=A*B
03. L1: T2=T1/C
04:        if (T2<W) goto L2
05:        M=T1*K
06:        T3=M+1
07: L2: H=I
08:        M=T3-H
09:        if (T3>0) goto L3
10: goto L1
11: L3: halt
```



Example

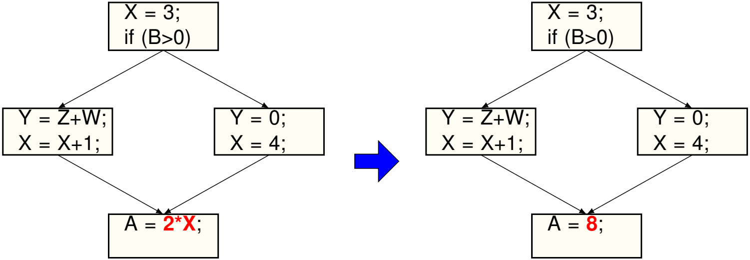
```
01.        A=4
02.        T1=A*B
03.  L1:  T2=T1/C
04:        if (T2<W) goto L2
05:        M=T1*K
06:        T3=M+1
07:  L2:  H=I
08:        M=T3-H
09:        if (T3>0) goto L3
10:  goto L1
11:  L3:  halt
```



Data Flow Analysis

Global Optimizations

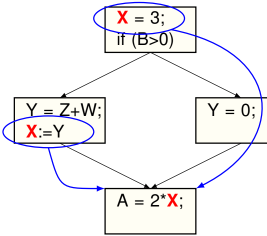
- Extend optimizations across control flows, i.e. CFG
- Like in this example of Global Constant Propagation (GCP):



- How do we know it is OK to globally propagate constants?

Correctness criteria for GCP

- There are situations that prohibit GCP:



- To replace X by a constant C **correctly**, we must know
  - **Along all paths**, the last assignment to X is “X = C”

- Paths may go through loops and/or branches
  - When two paths **meet**, need to make a **conservative** choice

Global Optimizations need to be Conservative

- Many compiler optimizations depend on knowing some property X at a particular point in program execution
  - Need to prove at that point property X holds along all paths

- To ensure correctness, optimization must be **conservative**
  - An optimization is enabled only when X is definitely true
  - If not sure, be conservative and say **don't know**
  - **Don't know** usually disables the optimization

Dataflow Analysis Framework

- **Dataflow analysis**: discovering properties about values
  - ... at certain points in the CFG to enable optimizations
  - E.g. discovering a value is constant at a statement
  - Done by observing the flow of data through the CFG

- **Dataflow analysis framework**:
  - A framework for describing different dataflow analyses
  - Can be defined using the following 4 components:

    $\{$ **D**, **V**, $\wedge$: (V, V) $\rightarrow$ V, **F**: V $\rightarrow$ V $\}$

    - **D**: direction of dataflow (forward or backward)
    - **V**: domain of values denoting property
    - $\wedge$: **meet operator** that merges values when paths meet
    - **F**: **flow propagation function** that propagates values through a basic block

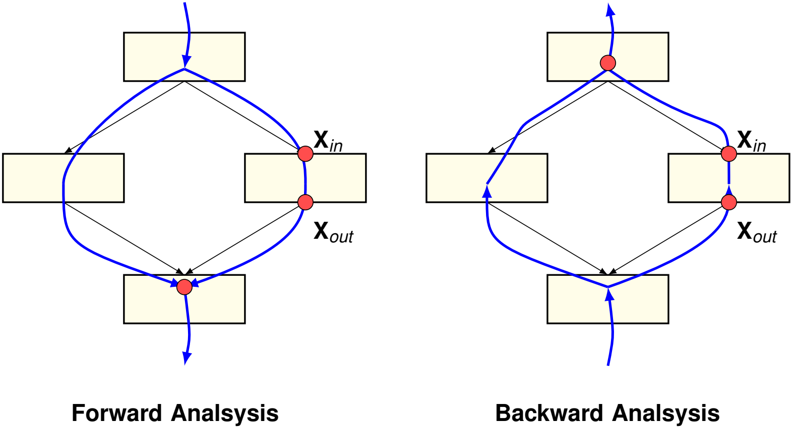
Global Constant Propagation (GCP)

- Let's use **GCP** to study dataflow analysis framework

- We will define each component one by one for GCP
  - **D**: direction of dataflow for constant property
  - **V**: domain of values denoting constant property
  - $\wedge$: **meet operator** that merges values when paths meet
  - **F**: **flow propagation function** for GCP

Direction D for GCP

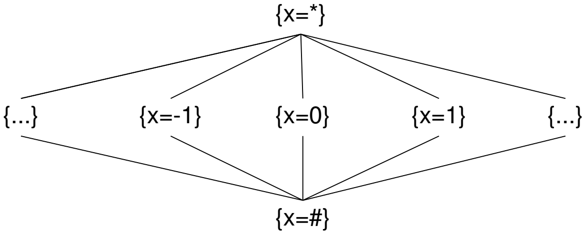
- Is GCP a forward or backward analysis?



- Forward, since "constantness" of a variable flows forward to subsequent instructions starting from assignment

V and meet operator ∧ for GCP

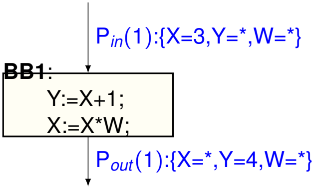
- Given an integer variable x, domain V is the set:
  - ..., {x=-1}, {x=0}, {x=1}, ... /* a constant */
  - {x=*} /* not a constant */
  - {x=#} /* x is not assigned on any path*/
- Meet operator ∧ is given by this **semi-lattice**:
  - a ∧ b = least upper bound in the below graph

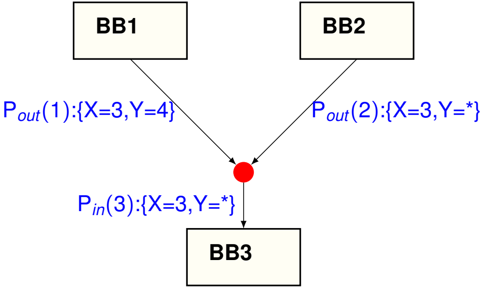


  - {x=0} ∧ {x=1} = {x=*}: Different values on each path
  - {x=#} ∧ {x=1} = {x=1}: Constant definition on one path

Dataflow Equations for GCP

- There are two types of flow functions





- ➢ Flow transfer function F: V $\rightarrow$ V
  - Computes data flow within basic blocks
  - Remove those that become variables, add new constants
- ➢ Meet operator $\wedge$: (V, V) $\rightarrow$ V
  - Computes data flow across basic blocks
  - Merge values from two paths using the previous semi-lattice

Flow Transfer Function F for GCP

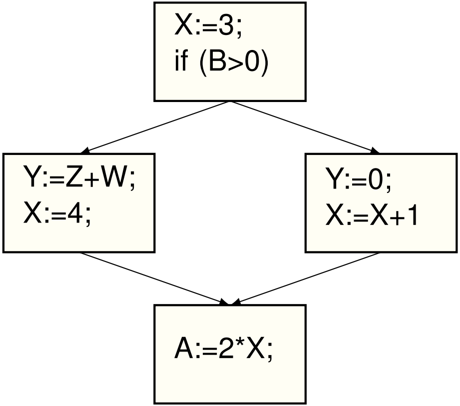
- **X(i)**: dataflow property X of basic block **i**
  - $\mathbf{X_{in}(i)}$: at the entry of basic block **i**
  - $\mathbf{X_{out}(i)}$: at the exit of basic block **i**

- F for Global constant propagation (GCP)

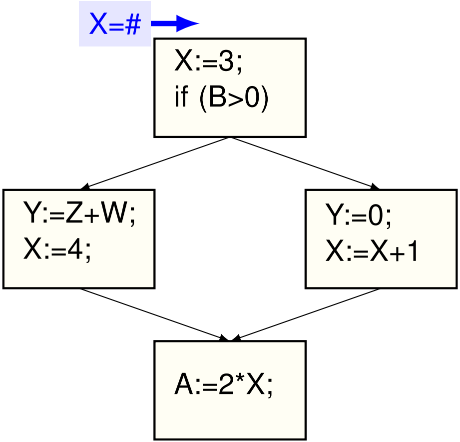
$$\mathbf{GCP_{out}(i) = (\ GCP_{in}(i) - DEF_{v}(i)\ ) \cup DEF_{c}(i)}$$

where DEF_v(i) contains variable definitions in basic block i
DEF_c(i) contains constant definitions in basic block i

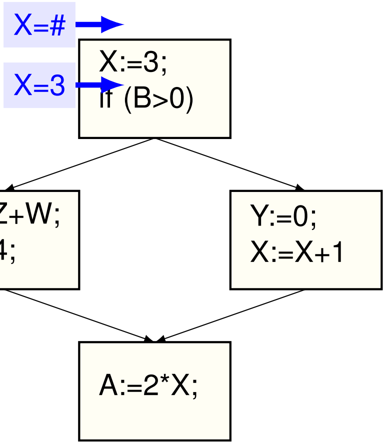
GCP Propagation without loops



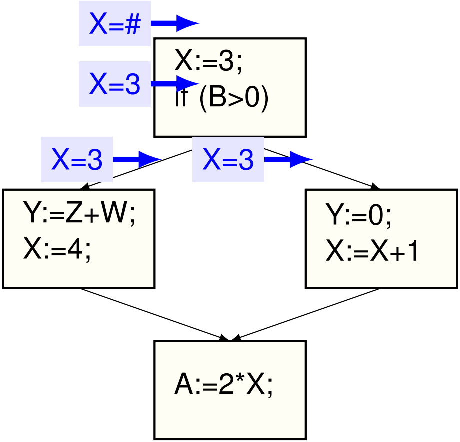
GCP Propagation without loops



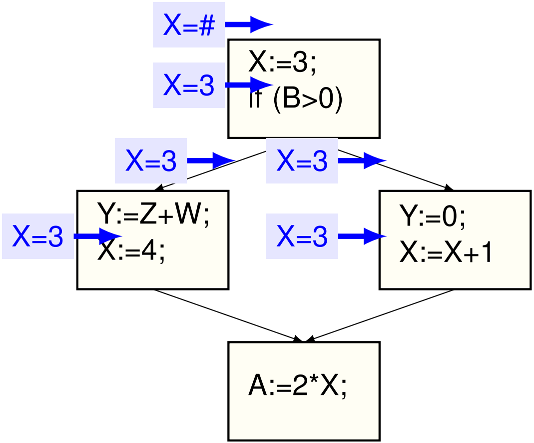
GCP Propagation without loops



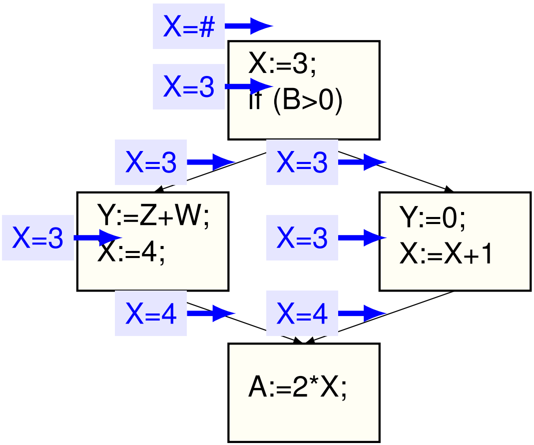
GCP Propagation without loops



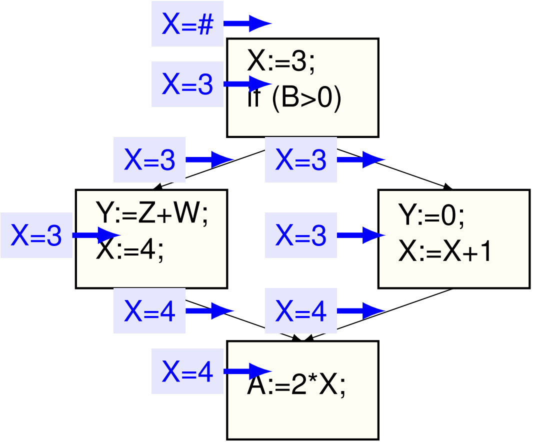
GCP Propagation without loops



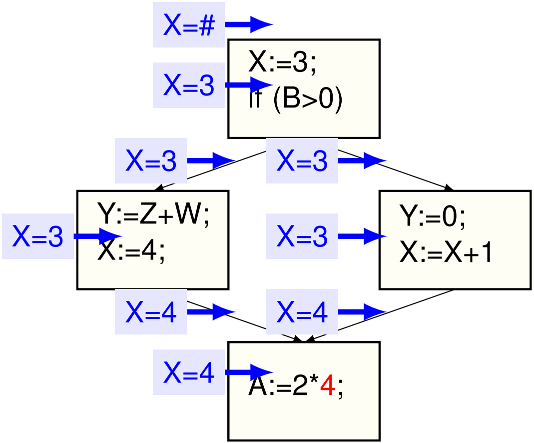
GCP Propagation without loops



GCP Propagation without loops

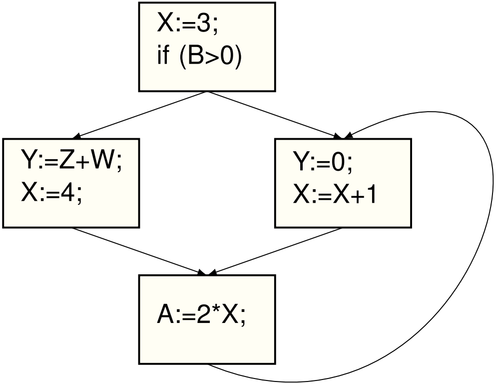


GCP Propagation without loops



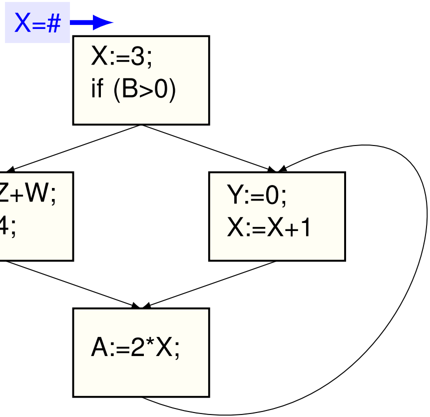
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



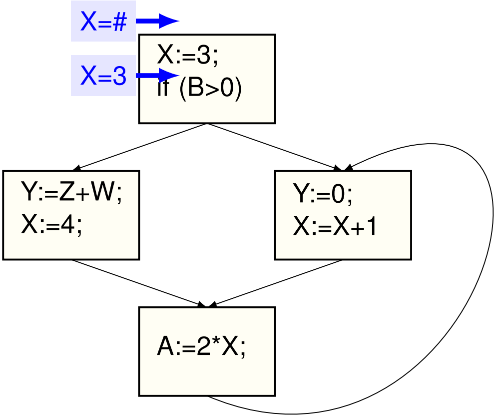
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



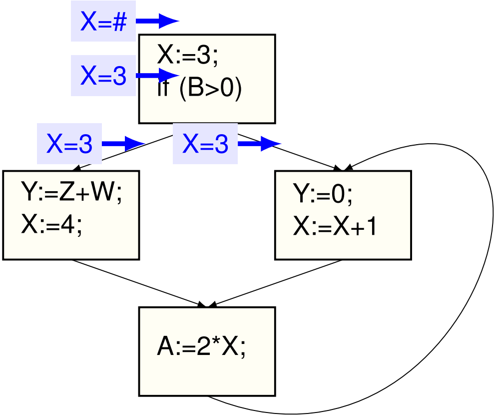
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



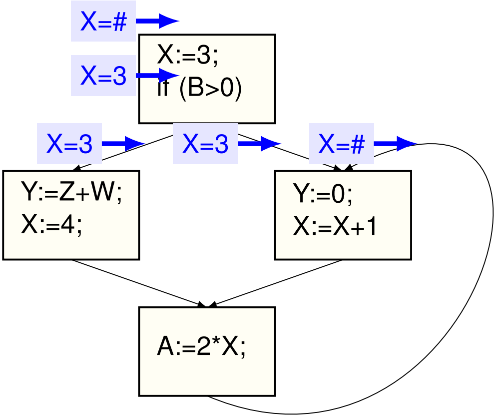
GCP Propagation with loops

- Iterate until there are no changes to values
  - ➢ This is called the **maximum fixed point** solution



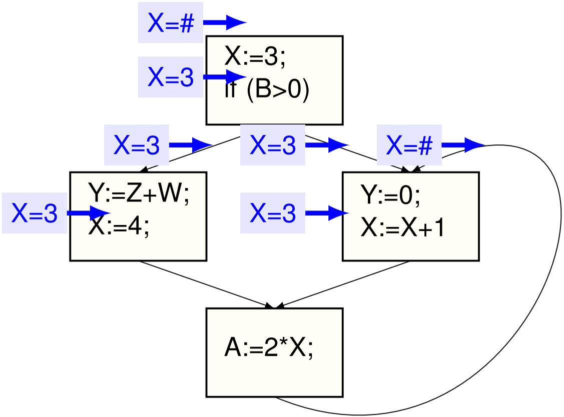
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



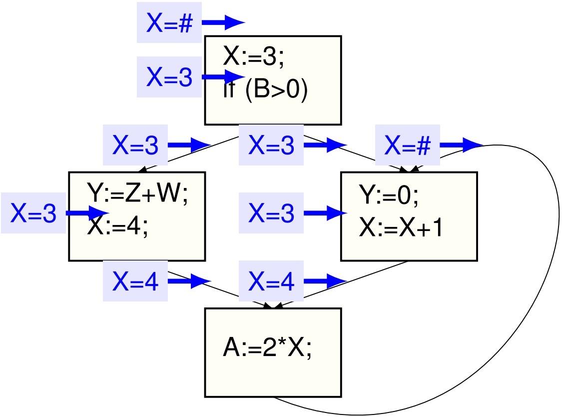
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



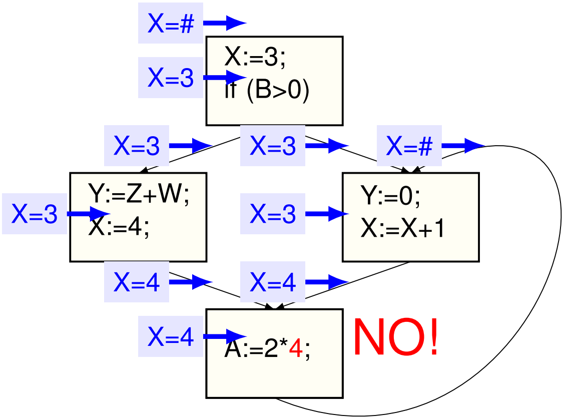
GCP Propagation with loops

- Iterate until there are no changes to values
  - ➢ This is called the **maximum fixed point** solution



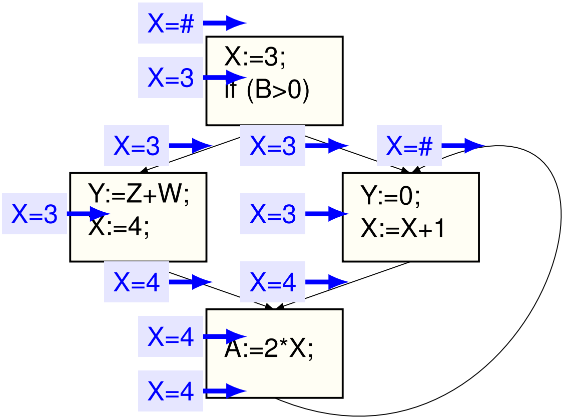
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



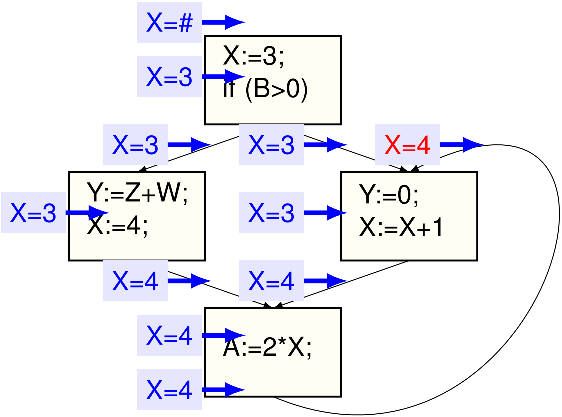
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



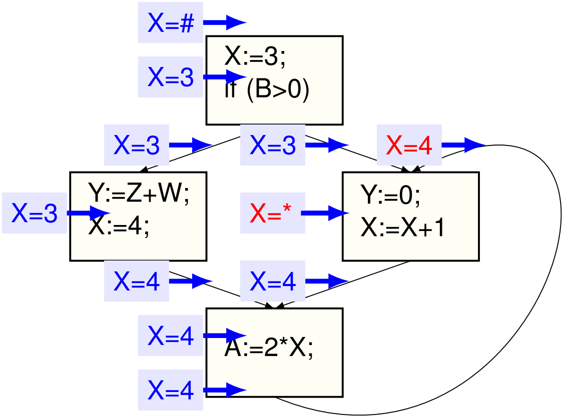
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



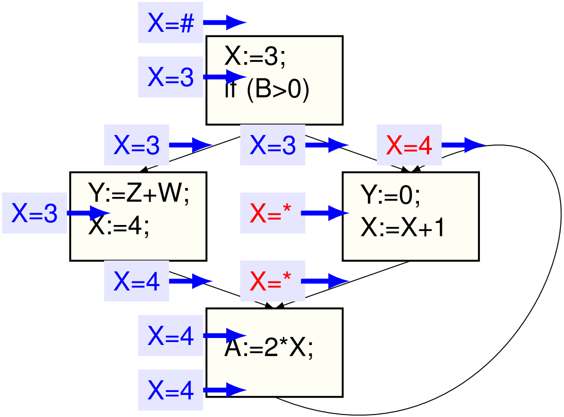
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



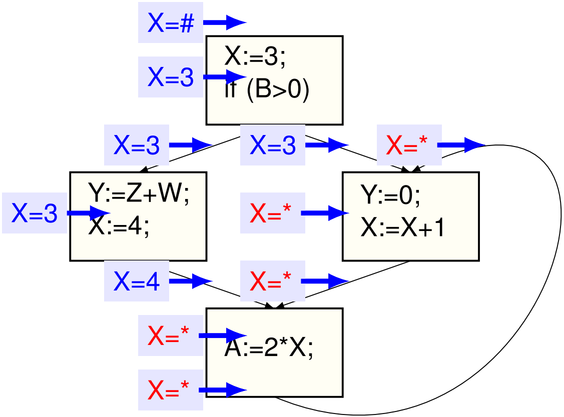
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



Analysis Algorithm for GCP

- GCP Algorithm
  (1). Set {x=#} at all the points in the procedure
  (2). Propagate the dataflow property along the control flow
  (3). Repeat step (2) until there are no changes

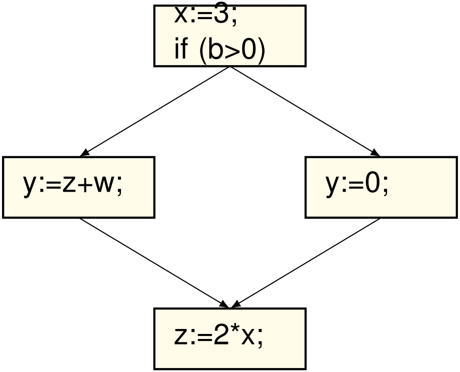
- Will GCP eventually stop?
  - If there are loops, we may propagate the loop many times
  - Is there a possibility to run into an endless loop?

Termination Problem

- **Least upper bound** ensures the termination
  - Values starts from #
  - Values can only increase in the hierarchy
  - Any values can change at most twice in our example
    ... from # to C, and from C to *
- The maximal number of steps is O(program_size)

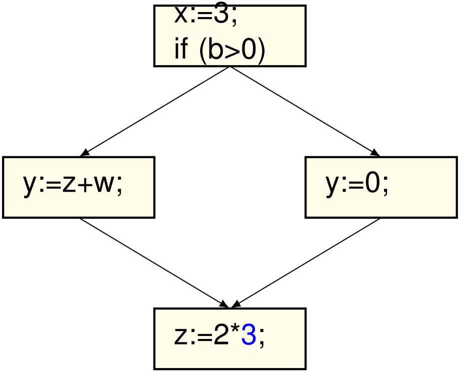
Another Analysis: Liveness Analysis

- Once constants have been globally propagated, we would like to eliminate the dead code



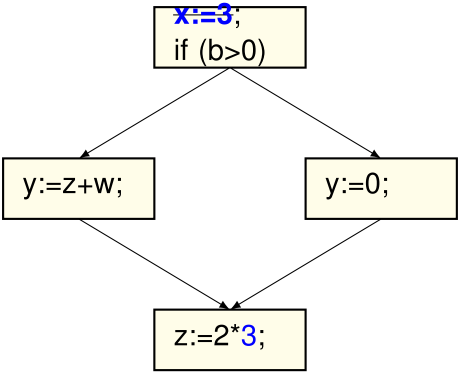
Another Analysis: Liveness Analysis

- Once constants have been globally propagated, we would like to eliminate the dead code



Another Analysis: Liveness Analysis

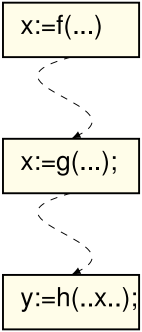
- Once constants have been globally propagated, we would like to eliminate the dead code



Live/Dead Statment

- A **dead statement** calculates a value that is not used later, or output to file
- Otherwise, it is a **live statement**

In the example,
the 1st statement is dead,
the 2nd statement is live



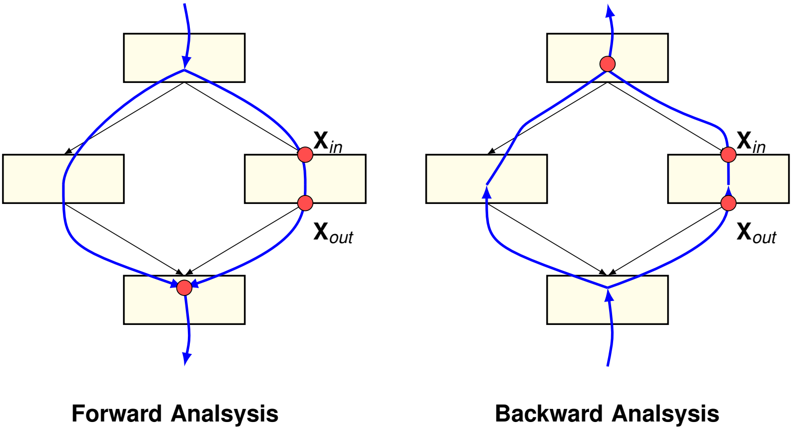
Liveness Analysis

- We can form liveness analysis as a dataflow analysis
  - Propagate the information along control flow
    - “x is dead”, “x is live”, “y is dead”, “y is live”
  - Liveness is simpler than constant propagation
    - It is a boolean property
  - Liveness is different from constant propagation
    - Liveness analysis is a union problem
      ☞ x is alive if x is alive along one path
    - Constant propagation is an intersection problem
      ☞ x is NOT a constant if x is NOT a constant along one path
      ...

Forward and Backward Analysis

- The most significant difference is
  - Liveness analysis wants to know if it is used some time later
    - Use information after this statement to decide
    - **Backward analysis**
  - Constant analysis wants to know if it is constant for all possible executions at this point
    - Use information before this statement to decide
    - **Forward analysis**

Graphic View of Forward and Backward Analysis

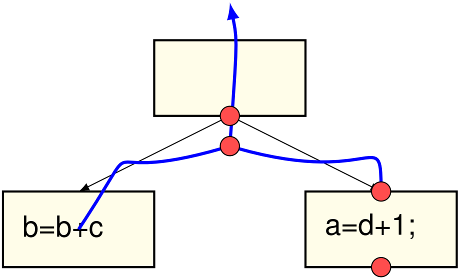


Global Liveness Analysis

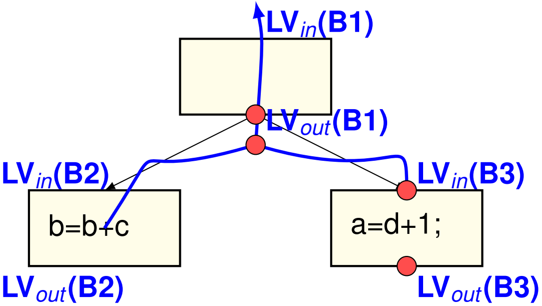
- Global liveness analysis
  - A variable x is live at statement s if
    - There exists a statement ss after s that use x
    - There is a path from s to ss
    - That path has no intervening assignment to x

- A backward dataflow analysis
  {**L**, ($\top$, $\bot$), ($\sqcap$, $\sqcup$), **f: L** $\rightarrow$ **L** }
  - Lattice, top and bottowm items
  - Operators
  - Dataflow functions

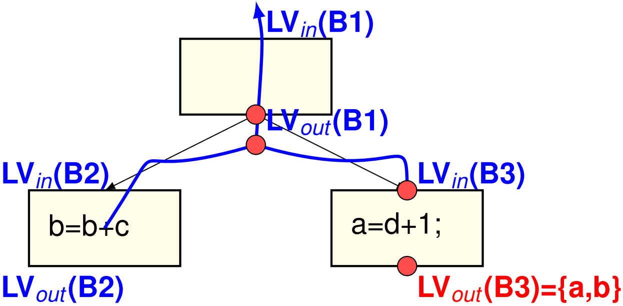
Liveness Example



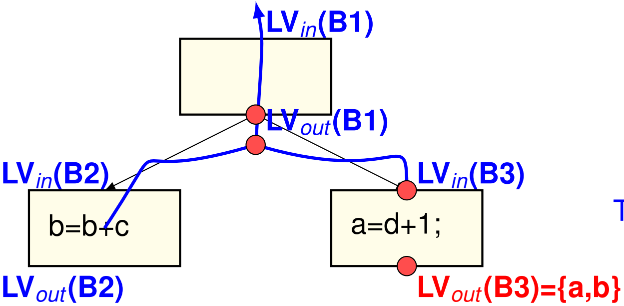
Liveness Example



Liveness Example

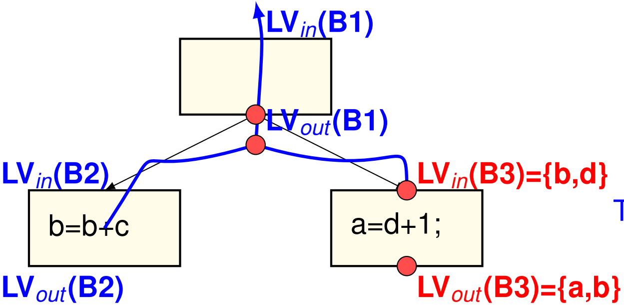


Liveness Example



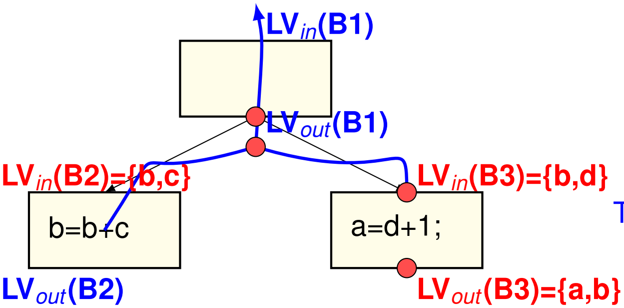
Two sets:
DEF={a}
USE={d}

Liveness Example



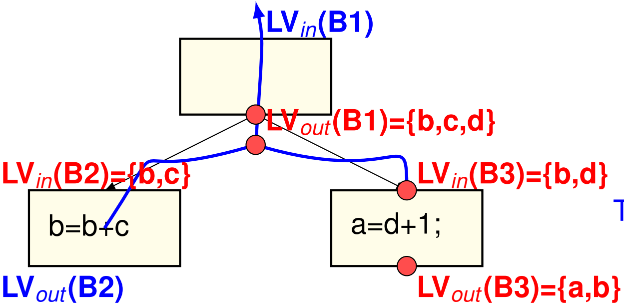
Two sets:
DEF={a}
USE={d}

Liveness Example



Two sets:
DEF={a}
USE={d}

Liveness Example



Two sets:
DEF={a}
USE={d}

Dataflow Equations for Liveness Analysis

- **X(i)** — dataflow property of basic block **i**
  - $\mathbf{X_{in}(i)}$ — at the entry of basic block **i**
  - $\mathbf{X_{out}(i)}$ — at the exit of basic block **i**
- Flow equations
  - flow transfer function
    $\mathbf{LV_{in}(i) = (\ LV_{out}(i) - DEF(i)\ ) \cup USE(i)}$
  - flow propagation function
    $\mathbf{LV_{out}(i) = \cup\ LV_{in}(k)}$ where **k** is successor of **i**

Comparison with Dataflow Equations for GCP

- **X(i)** — dataflow property of basic block **i**
  - $\mathbf{X}_{in}\mathbf{(i)}$ — at the entry of basic block **i**
  - $\mathbf{X}_{out}\mathbf{(i)}$ — at the exit of basic block **i**
- Global constant propagation (GCP)
  - flow transfer function
    $\mathbf{GCP}_{out}\mathbf{(i) = (\ GCP}_{in}\mathbf{(i)\ -\ DEF}_{v}\mathbf{(i)\ )} \cup \mathbf{DEF}_{c}\mathbf{(i)}$

    where DEF_v(i) contains variable definitions in basic block i
    DEF_c(i) contains constant definitions in basic block i
  - flow propagation function
    $\mathbf{GCP}_{in}\mathbf{(i) =} \cap\ \mathbf{GCP}_{out}\mathbf{(k)}$ where **k** is predecessor of **i**

Application of Liveness Analysis

- Global dead code elimination is based on global liveness analysis (GLA)
  - Dead code detection
    - A statement x:= ... is dead code if x is dead after this statement
    - Dead statement can be deleted from the program (Didn't consider side-effect)

- Global register allocation is also based on GLA
  - Live variables should be placed in registers
  - Registers holding dead variables can be reused

Summary of Dataflow Analysis

- There are many other global dataflow analysis
- Classification
  - Forward/Backward analysis
  - Union analysis — some property is true if it is true along at least one path Intersection analysis — some property is true if it is true along all paths
- Very useful in
  - compiler optimization
  - software engineering
  - debugging

Register Allocation

Why Register Allocation?

- Register allocation is a very important optimization
  - Register are important hardware resources
  - RISC computer: all ALU instructions use registers only
  - Fast but a fixed small number of registers
    - from memory: one instruction and 20-100+ cycles
    - from cache: one instruction and 1 cycle
    - from register: same instruction cycle
- Goal of register allocation
  - Improve performance by keeping right values in registers
  - Compute in registers and keep it as long as it will be used

Local Register Allocation

- Allocate registers only to values within basic block
  - Must put values back into memory at the end of each basic block
  - It is helpful to distinguish
    - Short lived temporary variables: only used in basic block
    - Long lived temporary variables: used later in following basic blocks
    - Normal variables
- Backward scan
  - Keep variables in registers if they are used later

Global Register Allocation

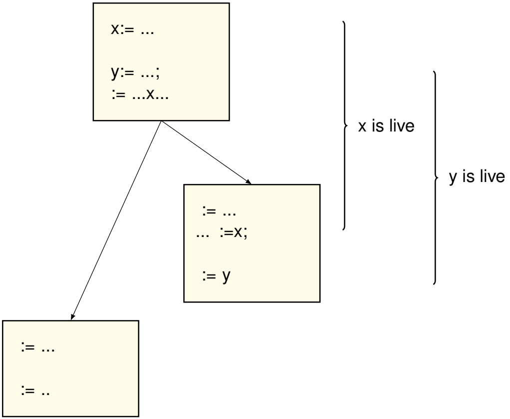
- Allocate registers across basic blocks
- Need liveness information
  - Global liveness analysis
- How to analyze?
  1. Graph coloring
  2. Linear scan
  3. ILP

Allocate Registers using Graph Coloring

- **Algorithm steps**:
  1. Global liveness analysis and identify interference
  2. Build register interference graph
  3. Coalesce nodes of graph
  4. Attempt N-coloring of the graph
     - N is the number of available registers
  5. If none found, modify the program, rebuild graph until N-coloring can be obtained
     - Insert spill code to the program

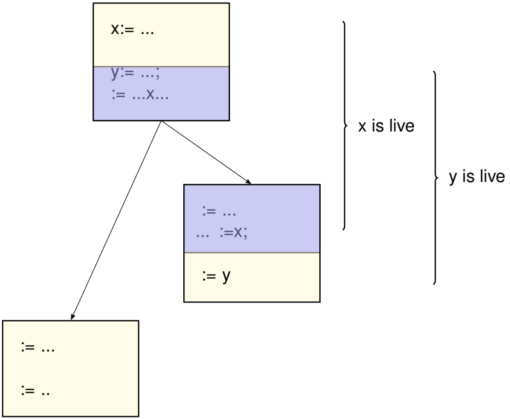
Global Liveness and Interference Analysis

- A variable is alive till its last usage
- Two variables interfere when their values cannot reside in the same register



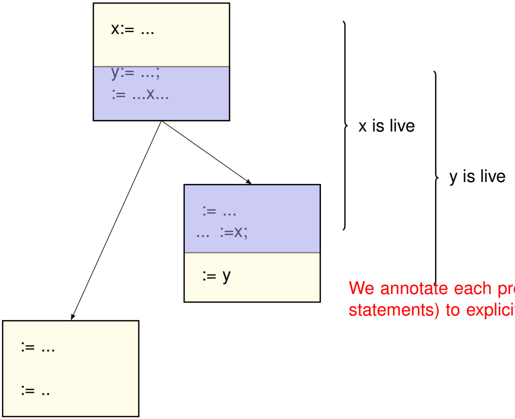
Global Liveness and Interference Analysis

- A variable is alive till its last usage
- Two variables interfere when their values cannot reside in the same register



Global Liveness and Interference Analysis

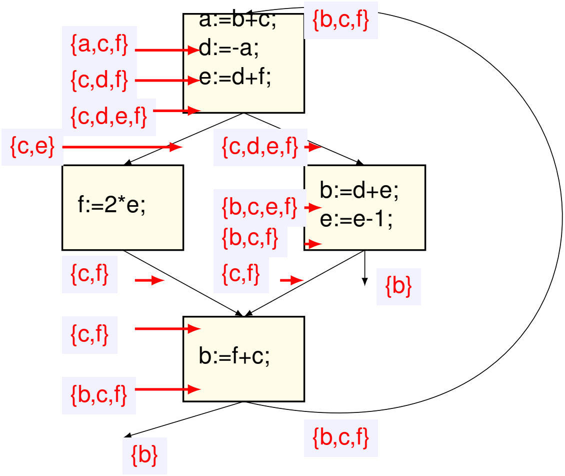
- A variable is alive till its last usage
- Two variables interfere when their values cannot reside in the same register



We annotate each program point (between two statements) to explicitly show the interference

Point of Definition Interference

- At each definition point P, compute live variables and determine the interference

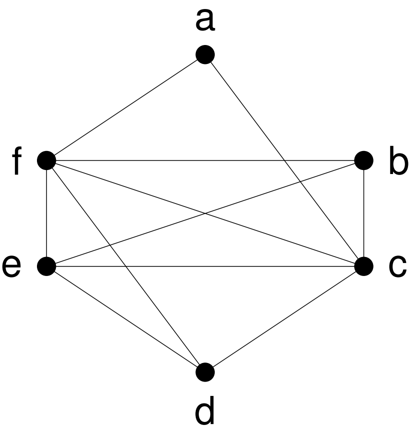


Register Interference Graph

- We can construct **Register Interference Graph (RIG)** such that
  - Each node represents a variable
  - An edge between two nodes V_1 and V_2 if they live simultaneously at some program point
- Based on RIG,
  - Two variables can be allocated in the same register if there is no edge between them
  - Otherwise, they cannot be allocated in the same register

RIG Example

- For our example,
  - b,c cannot be in the same register
  - a,d,d can be in the same register

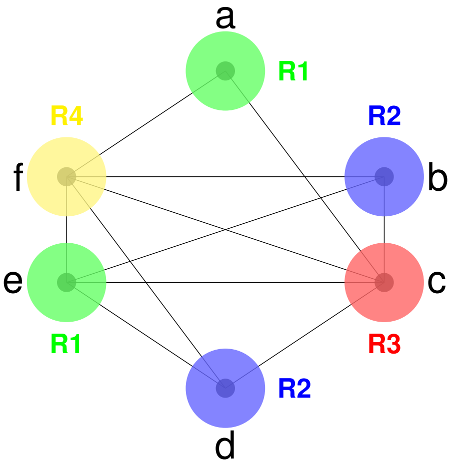


Allocating Registers using Graph Coloring

- Graph coloring is a basic register allocation scheme
  - A coloring of a graph is an assignment of colors to nodes such that nodes connected by an edge have different colors
  - A graph is k-colorable if it has a coloring with k colors
- In the algorithm, colors = registers
  - We need to assign k registers to graph nodes
  - K is the number of available machine registers
  - If the graph is k-colorable, we have a register assignment that uses no more than k registers

Coloring Results

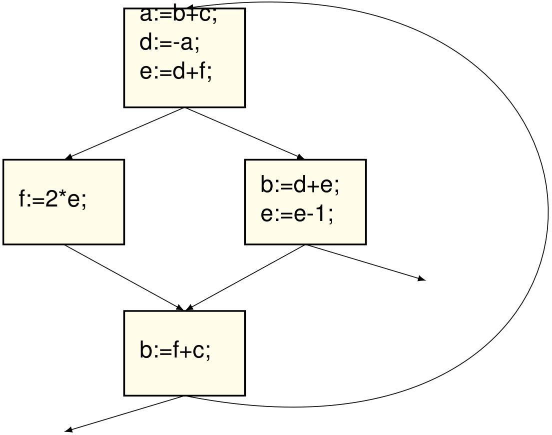
- For our example,
  - There are 4 colors in the coloring result
  - There is no solution with less than 4 colors



After Register Allocation

- Using the coloring result, map it back to the code

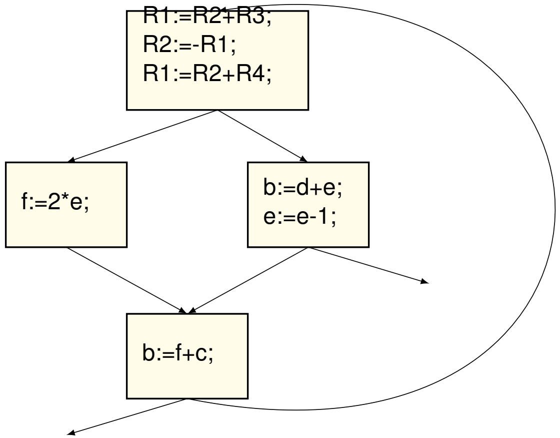
a–R1
b–R2
c–R3
d–R2
e–R1
f–R4



After Register Allocation

- Using the coloring result, map it back to the code

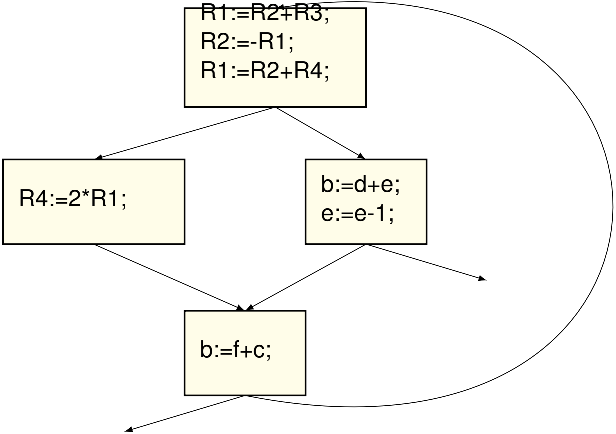
a–R1
b–R2
c–R3
d–R2
e–R1
f–R4



After Register Allocation

- Using the coloring result, map it back to the code

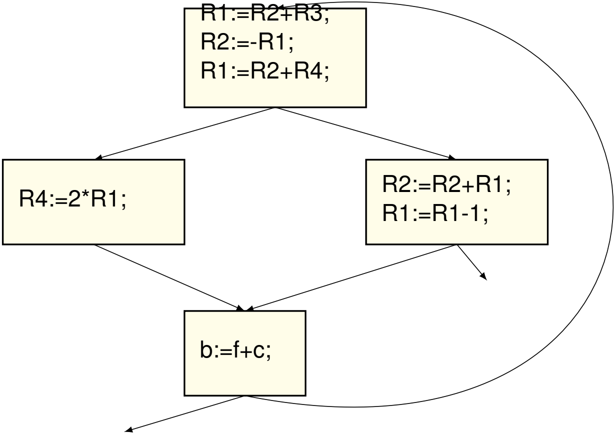
a–R1
b–R2
c–R3
d–R2
e–R1
f–R4



After Register Allocation

- Using the coloring result, map it back to the code

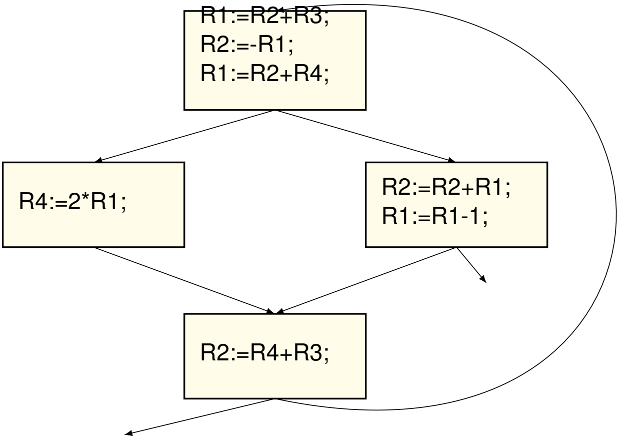
a–R1
b–R2
c–R3
d–R2
e–R1
f–R4



After Register Allocation

- Using the coloring result, map it back to the code

a–R1
b–R2
c–R3
d–R2
e–R1
f–R4

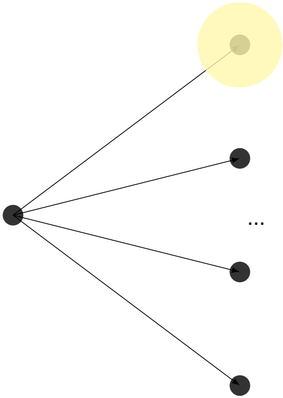


How to Perform Graph Coloring

- Now, the problem is how to compute a coloring from an interference graph
  - For graph G and N>2, determining whether G is N colorable is NP complete
  - A coloring might not exist for a given number or registers
- In practice
  - For the first problem, use heuristics
  - For the second problem, generate spill code

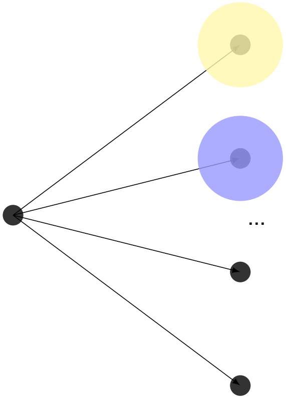
Graph Coloring Heuristics

- **Observation**: given a node with **k-1** neighbors, if there are **k** colors, then no matter how the neighbors were colored, there might be a unused color for this node



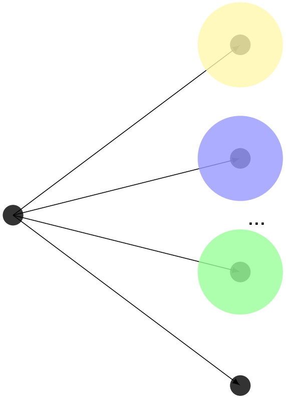
Graph Coloring Heuristics

- **Observation**: given a node with **k-1** neighbors, if there are **k** colors, then no matter how the neighbors were colored, there might be a unused color for this node



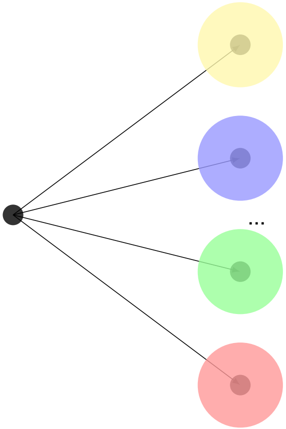
Graph Coloring Heuristics

- **Observation**: given a node with **k-1** neighbors, if there are **k** colors, then no matter how the neighbors were colored, there might be a unused color for this node



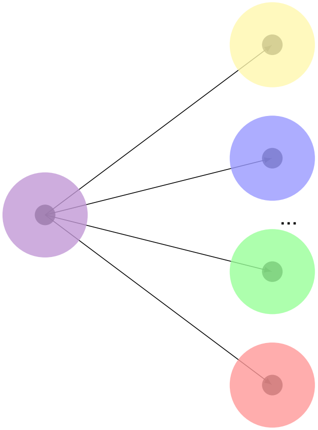
Graph Coloring Heuristics

- **Observation**: given a node with **k-1** neighbors, if there are **k** colors, then no matter how the neighbors were colored, there might be a unused color for this node



Graph Coloring Heuristics

- **Observation**: given a node with **k-1** neighbors, if there are **k** colors, then no matter how the neighbors were colored, there might be a unused color for this node



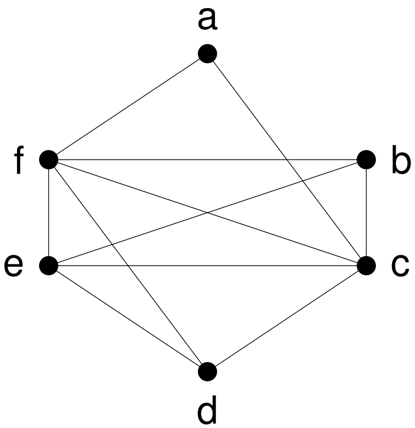
Heuristic Algorithm

To determine if a graph can be colored with **k** colors,

- repeat until there is only one node left
  - Pick a node t with fewer than k neighbors
  - Put t on a stack and remove it and its associated edges from the graph
- Starting assigning colors to nodes on the stack
  - Starting from the last nodes added to the stack
  - Pick a color that is different from its neighbors

Example

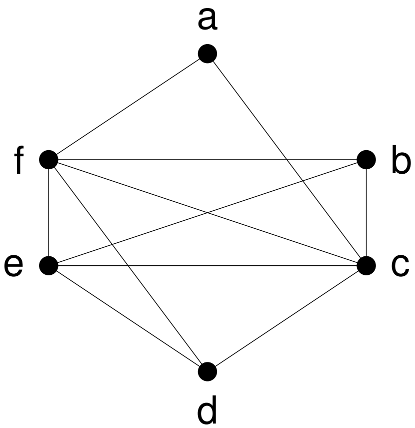
- To test if K=4 works in our example



Stack={}

Example

- To test if K=4 works in our example

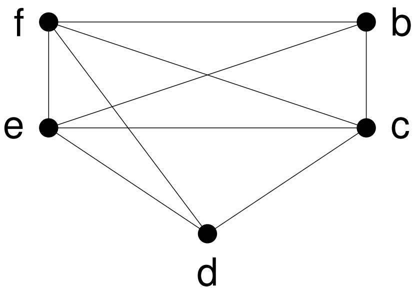


Stack={a}

Example

- To test if K=4 works in our example

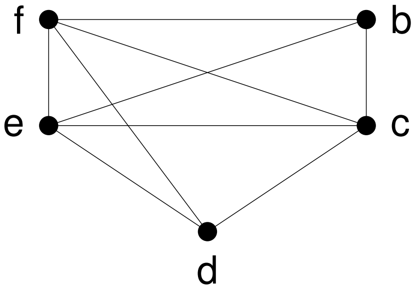
Stack={a}



Example

- To test if K=4 works in our example

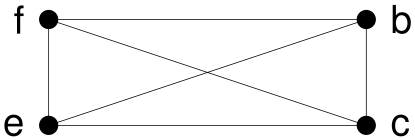
Stack={a,d}



Example

- To test if K=4 works in our example

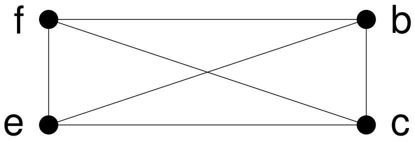
Stack={a,d}



Example

- To test if K=4 works in our example

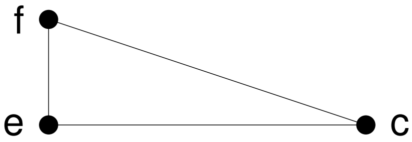
Stack={a,d,b}



Example

- To test if K=4 works in our example

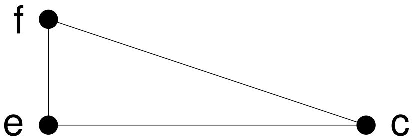
Stack={a,d,b}



Example

- To test if K=4 works in our example

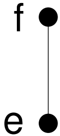
Stack={a,d,b,c}



Example

- To test if K=4 works in our example

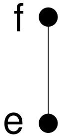
Stack={a,d,b,c}



Example

- To test if K=4 works in our example

Stack={a,d,b,c,e}



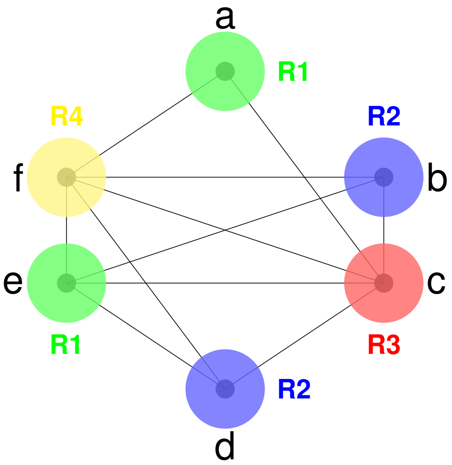
Example

- To test if K=4 works in our example

Stack={a,d,b,c,e}

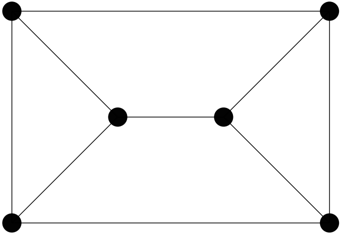
Coloring Results

- Starting assigning colors to **f,e,b,c,d,a**



What if the Heuristic Fails?

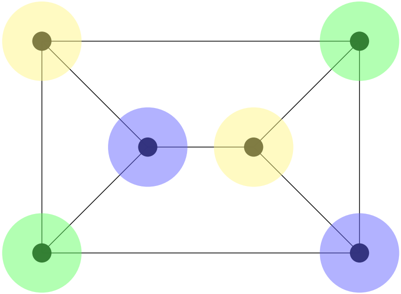
- Given the example, is it 3-colorable?
  - Every node has 3 outgoing edges, thus it is not 3-colorable



- However, it is 3-colorable ...

What if the Heuristic Fails?

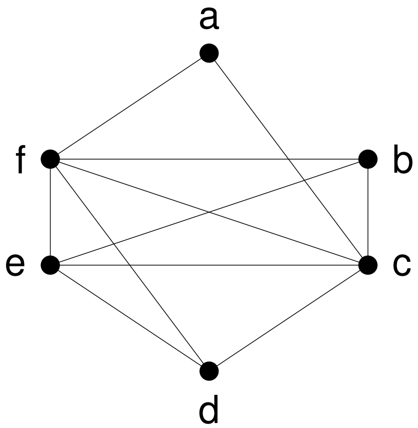
- Given the example, is it 3-colorable?
  - Every node has 3 outgoing edges, thus it is not 3-colorable



- However, it is 3-colorable ...

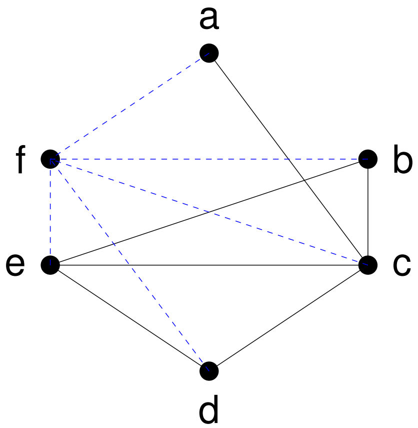
When the Heuristic Fails ...

- Spill the variable to memory
  - a spilled variable temporarily **lives** in memory
  - e.g. to color the previous graph using 3 colors
    - spill "f" into memory



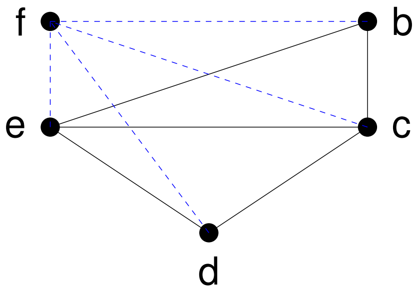
When the Heuristic Fails ...

- Spill the variable to memory
  - a spilled variable temporarily **lives** in memory
  - e.g. to color the previous graph using 3 colors
    - spill “f” into memory



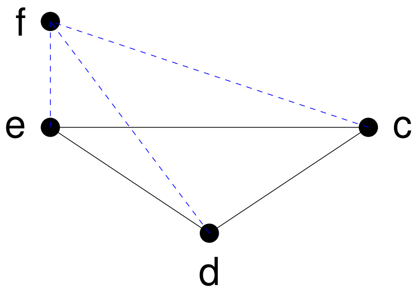
When the Heuristic Fails ...

- Spill the variable to memory
  - a spilled variable temporarily **lives** in memory
  - e.g. to color the previous graph using 3 colors
    - spill “f” into memory



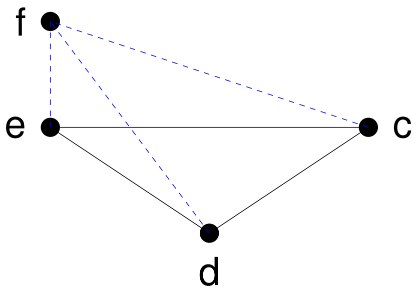
When the Heuristic Fails ...

- Spill the variable to memory
  - a spilled variable temporarily **lives** in memory
  - e.g. to color the previous graph using 3 colors
    - spill “f” into memory



When the Heuristic Fails ...

- Spill the variable to memory
  - a spilled variable temporarily **lives** in memory
  - e.g. to color the previous graph using 3 colors
    - spill “f” into memory

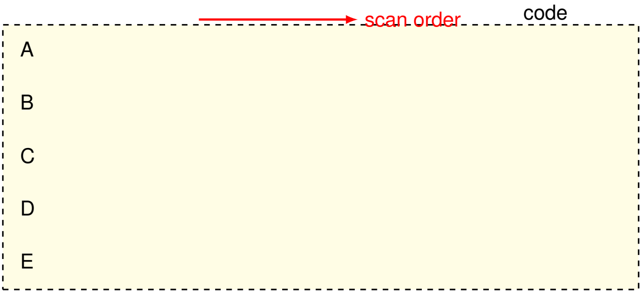


Linear Scan Register Allocation

- On-line compilers need to generate binary code quickly
  - Just-in-time compilation
  - Interactive environments e.g. IDE
  - Dynamic code generation in language extensions
- In these cases, it is beneficial to sacrifice code performance a bit for quicker compilation
  - A faster allocation algorithm
  - Not sacrificing too much

Linear Scan Register Allocation

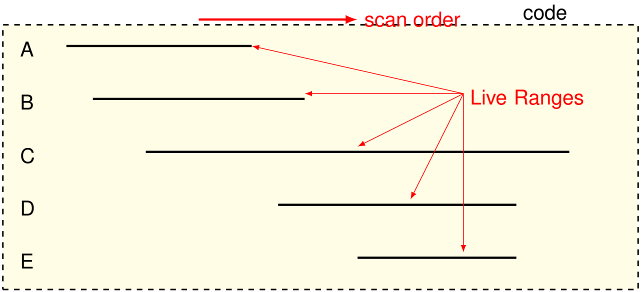
- Live range — the linear code range that a variable is alive



- Allocate at each numbered place
  - **A** and **D** may be allocated to the same register as at location “4”, **A** is dead

Linear Scan Register Allocation

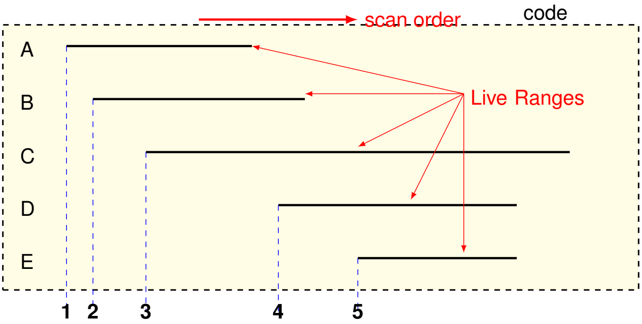
- Live range — the linear code range that a variable is alive



- Allocate at each numbered place
  - **A** and **D** may be allocated to the same register as at location “4”, **A** is dead

Linear Scan Register Allocation

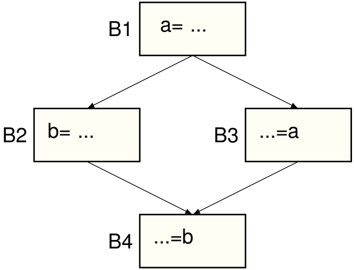
- Live range — the linear code range that a variable is alive



- Allocate at each numbered place
  - **A** and **D** may be allocated to the same register as at location “4”, **A** is dead

Linear Scan and Live Ranges

- Live range of a = {B1, B3}
- Live range of b = {B2, B4}
  - No interference between a and b, such that only one register is enough
- **However**, if code layout is “B1,B2,B3,B4”, then we need 2 registers



Linear Scan Algorithm

- Linear scan RA consists of four steps

S1. Order the instructions in linear fashion
  - Many have proposed heuristics for finding the best linear order

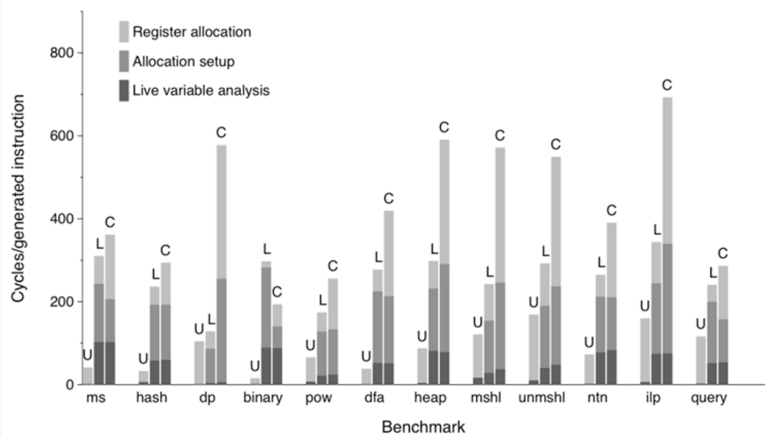
S2. Calculate the set of live intervals
  - Each temporary is given a live interval

S3. Allocate a register to each interval
  - If a register is available then allocation is possible
  - If a register is not available then an already allocated register is chosen (register spill occurs)

S4. Rewrite the code according to the allocation
  - Actual registers replace temporary or virtual registers
  - Spill code is generated

Implementing Type Checking on AST

- **U**sage Counts, **L**inear Scan, and Graph **C**oloring shown
- Linear Scan allocation is always faster than Graph Coloring



ILP-based Register Allocation

- Another types of register allocation that targets at
  - finding “optimal” register allocation result
- Idea and steps:
  1. Convert RA problem to a ILP problem
  2. Solve ILP problem using widely used ILP solvers
  3. Map the ILP solution back to register assignment
- Major problem that restricts its wide adoption
  - ILP problem is NP-hard
  - Solving ILP problem is slow

What is Integer Linear Programming (ILP)?

- Integer Linear Programming (ILP)

  Variables:

  a, b

  Constraints:

  $0 \leq a \leq 10$

  $0 \leq b \leq 29$

  $a + b \leq 36$

  Goal function

  minimize $f(a,b) = 3a + 4b$

- It is trivial if a and b can take real values
- It is NP hard if a and b can only take integer values

How to Convert Register Allocation to ILP?

- An example

  ...
  (10) a = b + ... ;
  ...

  - Want to know to which register b should be allocated i.e.
    load Rx, addr(b)

- Let us form an ILP problem
  - assume there are four free registers R1, R2, R3, R4

S1: Define variables

$X^{Ri}_{var(location)}$ — we allocate **var** at **location** to **Ri**

$X^{R1}_{b(10)}, X^{R2}_{b(10)}, X^{R3}_{b(10)}, X^{R4}_{b(10)}$

Converting Register Allocation to ILP

S2: Constraints: clearly there are constraints for these variables

- $X^{Ri}_{var(location)}$ only takes value 0 or 1
  0 — not allocate to that register at the place
  1 — is allocated to that register at the place
- Any register can hold only one variable at any place
  $X^{R1}_{b(10)} + X^{R1}_{a(10)} \leq 1$
- Any variable just needs to take one register
  $X^{R1}_{b(10)} + X^{R2}_{b(10)} + X^{R3}_{b(10)} + X^{R4}_{b(10)} = 1$
- and many more ...

S3: Define goal function

- to minimize memory operations
  $f_{cost} = (\sum X^{Ri}_{v(mem.p)})$ * $LOAD_{cost}$ + ... (store cost) ...

Conclusion

- Register Allocation is a “must have” optimization in most compilers
  - Because intermediate code uses too many temporaries
  - Because it makes a big difference in performance
- Different algorithms have been developed to satisfy different needs

Instruction Selection

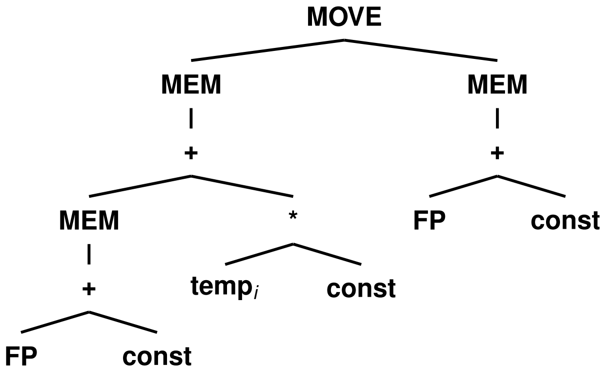
Instruction Selection

- Instruction selection is the task to select appropriate machine instructions to implement the operations in the intermediate representation (IR).
  - ➢ Very important for CISC machines, and machines with special purpose instructions (MMX)
  - ☞ X86, ARM, DSP, ...
- There are many semantically equivalent instruction sequences
  - ➢ How to find the “minimal cost” sequence?

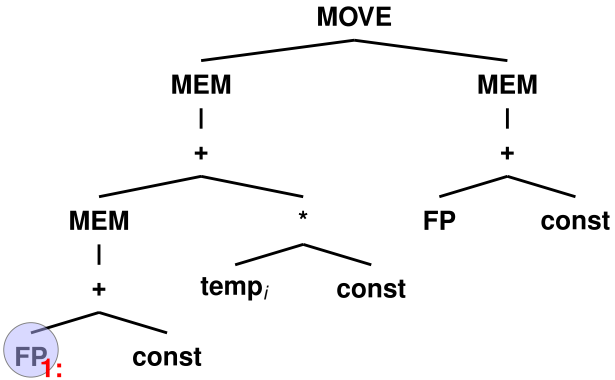
Some Instruction Patterns

Name	Effect	Trees
—	r_i	TEMP
ADD	$d_i \leftarrow d_j + d_k$	d + ; d d
MUL	$d_i \leftarrow d_j \times d_k$	d * ; d d
SUB	$d_i \leftarrow d_j - d_k$	d - ; d d
DIV	$d_i \leftarrow d_j / d_k$	d / ; d d
ADDI	$d_i \leftarrow d_j + c$	d + ; d CONST — d + ; CONST d — d CONST
SUBI	$d_i \leftarrow d_j - c$	d - ; d CONST
MOVEA	$d_j \leftarrow a_i$	d a
MOVED	$a_j \leftarrow d_i$	a d
LOAD	$d_i \leftarrow M[a_j + c]$	d MEM ; + ; a CONST — d MEM ; + ; CONST a — d MEM ; CONST — d MEM ; a
STORE	$M[a_j + c] \leftarrow d_i$	MOVE ; MEM d ; + ; a CONST — MOVE ; MEM d ; + ; CONST a — MOVE ; MEM d ; CONST — MOVE ; MEM d ; a
MOVEM	$M[a_j] \leftarrow M[a_i]$	MOVE ; MEM MEM ; a a

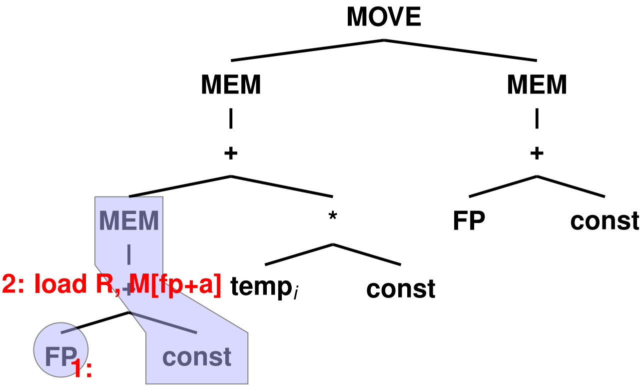
A Parse Tree to be Tiled



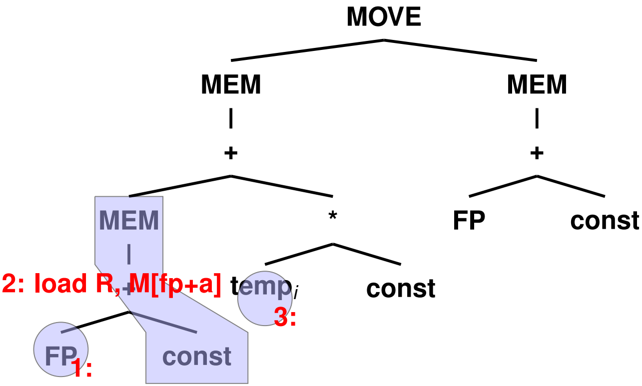
A Parse Tree to be Tiled



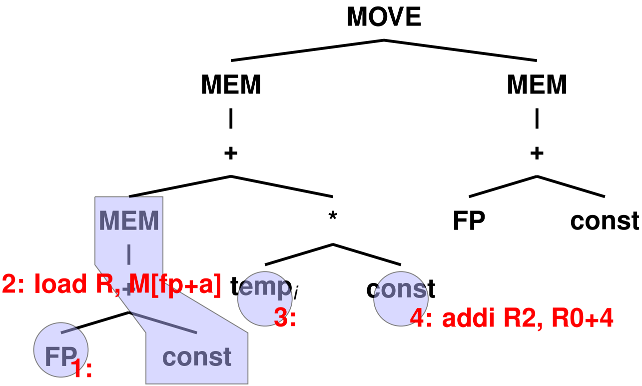
A Parse Tree to be Tiled



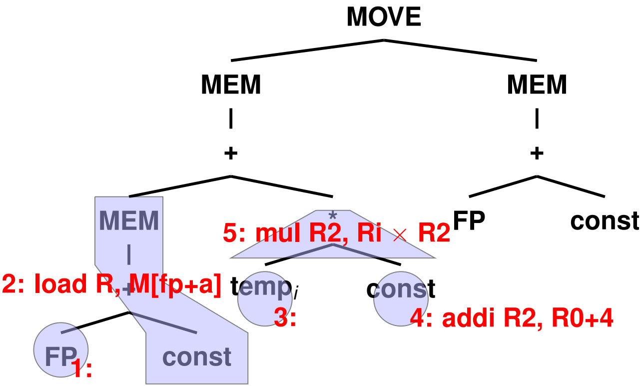
A Parse Tree to be Tiled



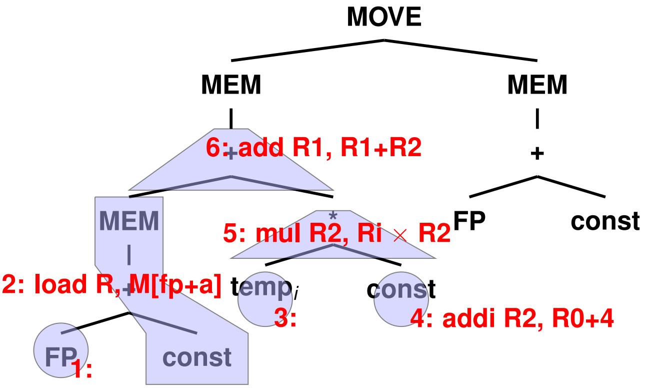
A Parse Tree to be Tiled



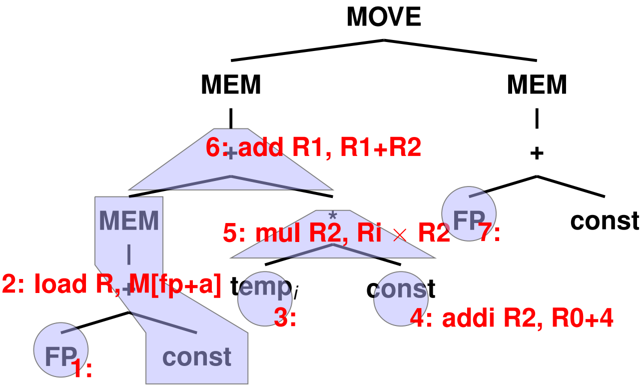
A Parse Tree to be Tiled



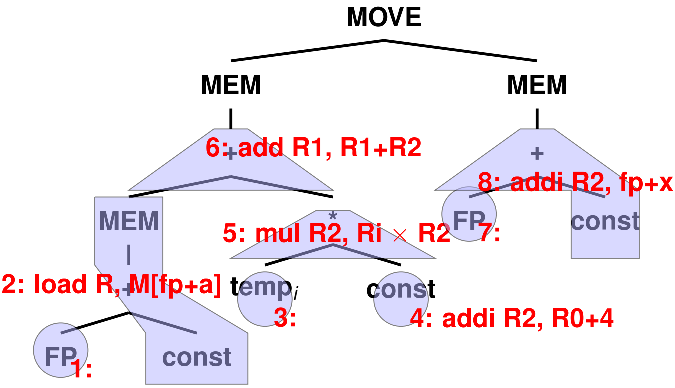
A Parse Tree to be Tiled



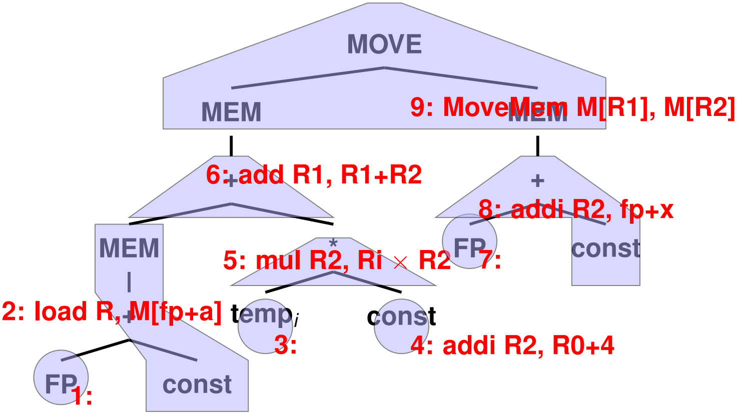
A Parse Tree to be Tiled



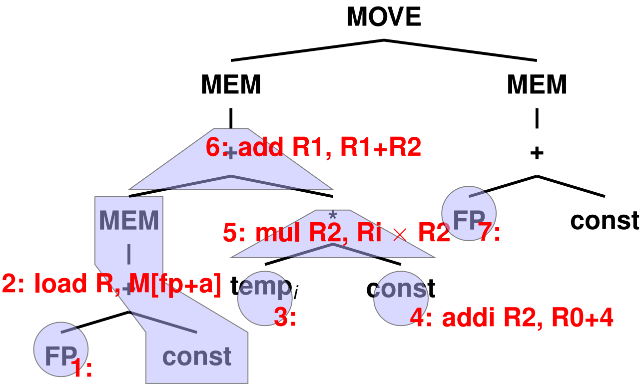
A Parse Tree to be Tiled



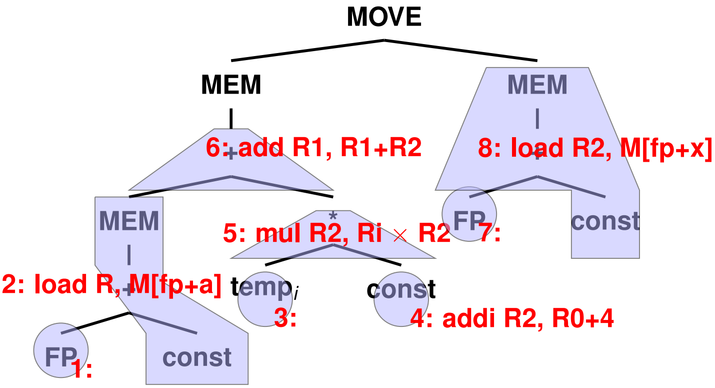
A Parse Tree to be Tiled



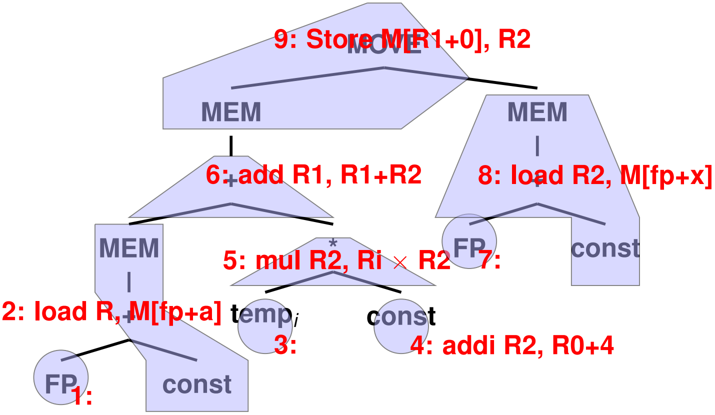
A Parse Tree to be Tiled



A Parse Tree to be Tiled



A Parse Tree to be Tiled



The END !