Compiler Optimization

Overview of Optimizations

- Compiler optimization is
  - to generate **better** code
  - not to generate **optimal** code
    - it is an NP-complete problem
- What is a better version?
  - Same result
  - Better one or more of the followings
    - Execution time
    - Memory usage
    - Energy/power consumption
    - Network messages
    - Other criteria

Types of Optimizations

- Compiler optimization is essentially transformation
  - Delete something
  - Add something
  - Move something
  - Modify something
- Transform code or data?
  - Data-related optimizations
  - Code-related optimizations

Data-Related Optimizations

Data-Related Optimizations

- Seeks to improve cache behavior by
  - changing the location and representation of data or code
  - exploiting knowledge of memory hierarchy layout (is inherently machine-dependent)
- Change code layout

```
f() {
 ... call h();
}
g() {
 ...
}
h() {
 ...
}
```

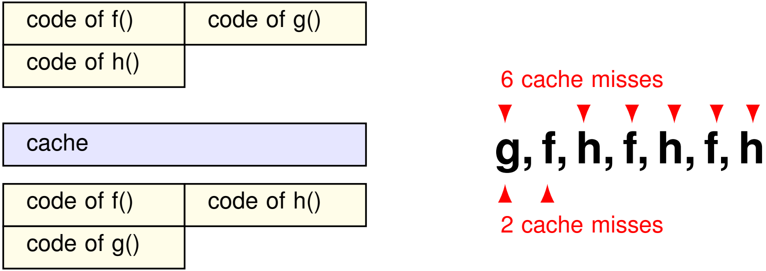
code of f()	code of g()
code of h()	

OR

code of f()	code of h()
code of g()	

Which Code Layout is Better?

- Assume
  - data cache has one N-word line
  - the size of each function is N/2-word long
  - access sequence is "**g, f, h, f, h, f, h**"



Data Layout Optimization

- Change the variable declaration order and/or the width

```
struct S {
    int x1;
    int x2[200];
    int x3;
} obj[100];
...
... obj[i].x1 + obj[i].x3
```

→

```
struct S {
    int x1;
    int x3;
    int x2[200];
} obj[100];
...
... obj[i].x1 + obj[i].x3
```

```
int flag1[200];
int flag2[200];
...
flag1[i] = 0 or 1;
...
flag2[i] = 0 or 1;
... flag1[i] + flag2[i]
```

→

```
char flag1[200]; // 8-bit, fast access
bit flag2[200]; // 1-bit, need bit operation
...
flag1[i] = 0 or 1;
...
flag2[i] = 0 or 1;
... flag1[i] + flag2[i]
```

Code-Related Optimizations

Code-Related Optimizations

- Modifying 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. **code motion**

  A=x*y; B=A+1; C=y; ≡ A=x*y; C=y; B=A+1;

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

```
while (p!=NULL)
   { ... p=p->next; }
≡
while (p!=NULL)
   { prefetch(p->next); ... p=p->next; }
```

Optimization Categories

- Optimize at what representation level?
  - Source code level
  - IR level
  - Binary code level
- Optimize for specific machine?
  - Machine independent — typically at IR or source level
  - Machine dependent — typically at machine code level
- Optimize across control flow?
  - Local optimization — scope within straight line code
  - Global optimization — scope across control structures
- Optimize across procedures?
  - Intra-procedural — scope within individual procedure
  - Inter-procedural — scope across different procedures (Analyze callee to optimize caller and vice versa)

Local Optimizations

Local Optimizations

- Optimizations where the scope includes no control flow
  - Limited in scope but can still do useful things

- **Strength Reduction**
  - The idea is to replace expensive operations (multiplication, division) by less expensive operations (add, sub, shift, mov)
  - Some are redundant and thus can be deleted
    e.g. x=x+0; y=y*1;
  - Some can be simplified
    e.g. x=x*0; y=y*8;
    can be replaced by x=0; y=y«3;
  - Is also machine-dependent since it uses knowledge about the underlying machine (e.g. multiplication is expensive)

More Local Optimizations

Constant folding

- Operations on constants can be computed at compile time
- In general, if **x= y op z** and y and z are constants then compute at compile time and replace
- Example:

```
#define LEN 100
x = 2 * LEN;
if (LEN < 0) print("error");
```

  Can be transformed to ...

```
x = 200;
if (false) print("error");
```

- Is machine-independent since it is beneficial regardless of machine

Global Optimizations and Control Flow Analysis

Global Optimizations and Control Flow

- Global optimization include more powerful optimizations
  - Can go across control structures
  - In effect, scope of optimization is the entire function (hence the name global)
  - E.g. Global Constant Propagation (GCP):
    - Replace variables with constants if value is known
      X = 7;
      ...
      Y = X + 3; // Can be replaced by Y = 10;
    - Needs knowledge of control flow
      (Whether evaluation at point A happens before point B)
- Global optimization requires control flow analysis
  - **Control flow analysis**: Compiler analysis that determines flow of control during execution of a function
  - Constructs a control flow graph that describes the flow

Basic Block

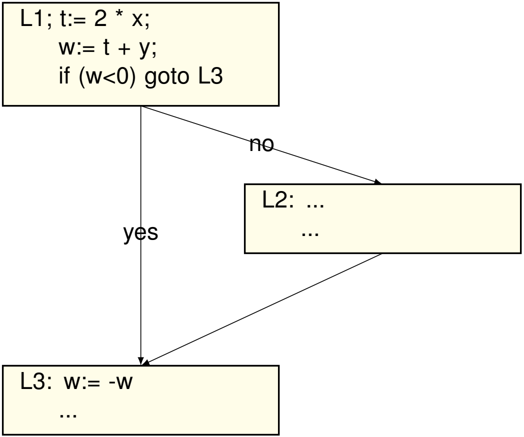
- A **basic block** is a maximal sequence of instructions that
  - Except the first instruction, there are no other labels;
  - Except the last instruction, there are no jumps;
- Therefore,
  - Can only jump into the beginning of a block
  - Can only jump out at the end of a block
  - All instructions of block execute or none at all
- Basic blocks are basic units of control flow which cannot be divided any further

Control Flow Graph

- A **control flow graph** is a directed graph in which
  - Nodes are basic blocks
  - Edges represent flow of execution
    - Control statements such as if-then-else, while-loop, for-loop introduce control flow edges
- CFG is widely used to represent a program
- CFG is widely used for program analysis, especially for global analysis/optimization

Example

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

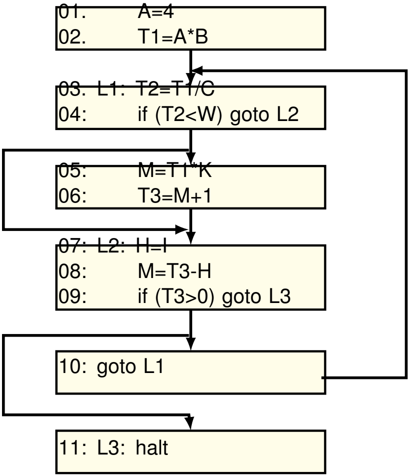


Construction of CFG

- Step 1: partition code into basic blocks
  - Identify **leader** instructions that are
    - the first instruction of a program, or
    - target instructions of jump instructions, or
    - instructions immediately following jump instructions
  - A basic block consists of a leader instruction and its subsequent instruction before the next leader
- Step 2: add an edge between basic blocks B1 and B2 if
  - there exist a jump from B1 to B2, or
  - B2 follows B1, and B1 does not end with unconditional jump
    - B1 ends with a conditional jump
    - B1 ends with a non-jump instruction (B2 is a target of a jump)

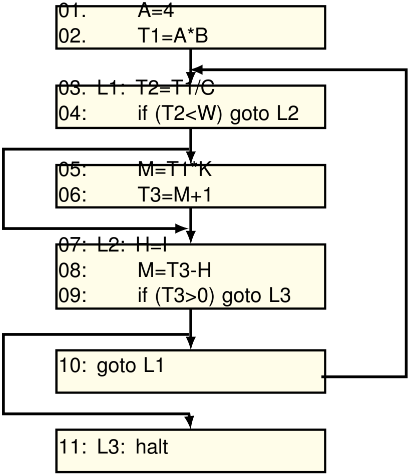
Example

```
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
```



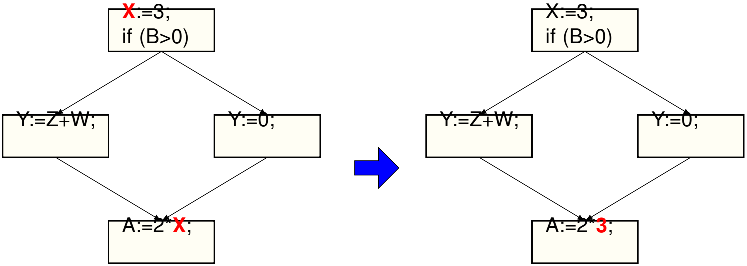
Example

```
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
```



Global Optimizations

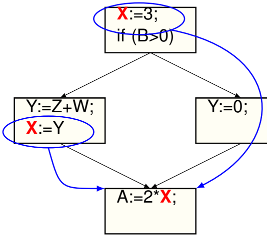
- Extend optimizations to flow of control, i.e. CFG



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

Correctness

- In particular, there are situations that prohibit this optimization



- To replace x by a constant C **correctly**, we must know
  - **Along all paths, the last assignment to X is "X:=C"**
  - **All paths** often include branches an even loops
    - Usually it is not trivial

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

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
  - Need to be **conservative** to ensure correctness
    - An optimization is enabled only when X is definitely true
    - If not sure if it is true or not, it is safe to say **don't know**
    - If you **don't know**, you don't do the optimization
    - May lose opt. opportunities but guarantees correctness

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
  - Need to be **conservative** to ensure correctness
    - An optimization is enabled only when X is definitely true
    - If not sure if it is true or not, it is safe to say **don't know**
    - If you **don't know**, you don't do the optimization
    - May lose opt. opportunities but guarantees correctness
- Property X often involves data flow of program
  - E.g. Global Constant Propagation (GCP):

    X = 7;
    ...
    Y = X + 3; // Replace by Y = 10, if X didn't change
    - Needs knowledge of data flow, as well as control flow (Whether data flow is interrupted between points A and B)

Global Optimizations and Data Flow Analysis

Dataflow Analysis Framework

- **Dataflow analysis**: Compiler analysis that determines what values get propagated from point A to point B
  - Requires CFG since values flow through control flow edges

Dataflow Analysis Framework

- **Dataflow analysis**: Compiler analysis that determines what values get propagated from point A to point B
  - Requires CFG since values flow through control flow edges
- **Dataflow analysis framework**: Framework for dataflow analysis that guarantees optimizations are **conservative**
  - Defined by: {**D**, **V**, $\wedge$, **F: V** $\rightarrow$ **V** }
  - **D**: Direction of propagation (forwards or backwards)
  - **V**: Set of values (depends on analyzed property)
    - Value for GCP: set of variables with constant values
  - $\wedge$: Meet operator (**V** $\wedge$ **V** $\rightarrow$ **V**)
    - Defines behavior when values meet at control flow merges
  - **F**: Transfer function **F: V** $\rightarrow$ **V**
    - Defines what happens to value within a basic block

Dataflow Analysis Framework

- **Dataflow analysis**: Compiler analysis that determines what values get propagated from point A to point B
  - Requires CFG since values flow through control flow edges
- **Dataflow analysis framework**: Framework for dataflow analysis that guarantees optimizations are **conservative**
  - Defined by: {**D**, **V**, $\wedge$, **F: V** $\rightarrow$ **V** }
  - **D**: Direction of propagation (forwards or backwards)
  - **V**: Set of values (depends on analyzed property)
    - Value for GCP: set of variables with constant values
  - $\wedge$: Meet operator (**V** $\wedge$ **V** $\rightarrow$ **V**)
    - Defines behavior when values meet at control flow merges
  - **F**: Transfer function **F: V** $\rightarrow$ **V**
    - Defines what happens to value within a basic block
- Goal: To assign **V** for every point in program $\rightarrow$ aids optimization

Global Constant Propagation (GCP)

- Rather than bore you with math, let's learn by example: **global constant propagation (GCP)**
- What is Global Constant Propagation?
  - At compile time, if the value of a variable is a constant, replace the variable with the constant
  - "Global" means we substitute across basic blocks and control flow
  - At compile time, we don't know which path is taken

Global Constant Propagation (GCP)

- Rather than bore you with math, let's learn by example: **global constant propagation (GCP)**
- What is Global Constant Propagation?
  - At compile time, if the value of a variable is a constant, replace the variable with the constant
  - "Global" means we substitute across basic blocks and control flow
  - At compile time, we don't know which path is taken
- Compiler can apply a dataflow analysis framework to the problem to ensure conservative optimization
  - What is **D**, **V**, $\wedge$, **F: V** $\rightarrow$ **V** in this context?

What is V?

- Definition: Set of values in property under analysis
- Property for GCP:
  - What are the variables with constant values?
  - And what are there values at the given point?

What is V?

- Definition: Set of values in property under analysis
- Property for GCP:
  - What are the variables with constant values?
  - And what are there values at the given point?
- A given variable can be in one of following states:

  x=1, x=2, ... // defined a constant
  x=* // not defined yet
  x=# // don't know (not provably constant)

What is V?

- Definition: Set of values in property under analysis
- Property for GCP:
  - What are the variables with constant values?
  - And what are there values at the given point?
- A given variable can be in one of following states:

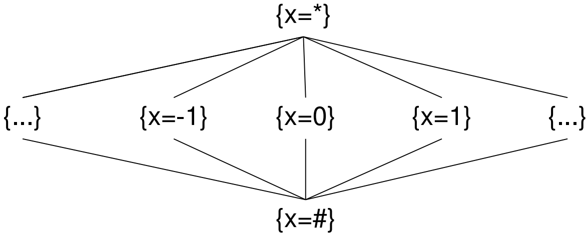
  x=1, x=2, ... // defined a constant
  x=* // not defined yet
  x=# // don't know (not provably constant)
- **V** for GCP: Set of values where each value is the set of variables and their respective states.
- Examples of values in **V**: {x=*, y=10, z=#}, {x=1, y=#, z=5}
- Goal for GCP is to assign a value to each point in program

What is $\wedge$?

- $\wedge$: Meet operator (**V** $\wedge$ **V** $\rightarrow$ **V**)
  - Defines behavior when values meet at control flow merges
  - Given
    - $\mathbf{V}_{in}\mathbf{(B)}$ — value at the entry of basic block **B**
    - $\mathbf{V}_{out}\mathbf{(B)}$ — value at the exit of basic block **B**
  - $\mathbf{V}_{in}\mathbf{(B)} = \wedge\ \mathbf{V}_{out}\mathbf{(P)}$ for each **P**, where **P** is a predecessor of **B**
- Example of $\wedge$ operator for GCP:
  {x=*, y=2, z=3} $\wedge$ {x=1, y=2, z=10} = {x=1, y=2, z=#}
- Relationship between values in **V** given by a meet operator is called a **Semilattice**

Semilattice

- Semilattice for GCP (when there is one variable):



- $\wedge$ operator is defined by the **Greatest Lower Bound (GLB)** between two values
  - {x=*} $\wedge$ {x=1} = {x=1}
  - {x=0} $\wedge$ {x=1} = {x=#}
  - Downward direction is always the conservative choice
  - In effect, GLB is the least **conservative** but **correct** choice
- Semilattice essentially defines what meet operator means

$\top$ and $\bot$ Values

- In a semilattice, there are two special values: $\top$ and $\bot$

$\top$ and $\perp$ Values

- In a semilattice, there are two special values: $\top$ and $\perp$
- $\top$: Called **Top Value**
  - Initial value when analysis begins
  - For GCP: {x=*, y=*, z=*}
  - Value is refined in the course of analysis

$\top$ and $\perp$ Values

- In a semilattice, there are two special values: $\top$ and $\perp$
- $\top$: Called **Top Value**
  - Initial value when analysis begins
  - For GCP: {x=*, y=*, z=*}
  - Value is refined in the course of analysis
- $\perp$: Called **Bottom Value**
  - Value which can be refined no further
  - For GCP: {x=#, y=#, z=#}
  - Meaning: none of the variables are provably constant
- Analysis iteratively refines values until they stabilize somewhere between $\top$ and $\perp$

What is F?

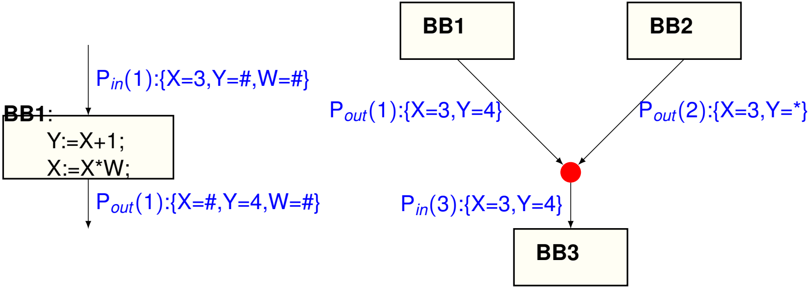
- **F**: Transfer function (**F: V** $\rightarrow$ **V**)
  - Defines what happens to value within a basic block
  - Given
    - $\mathbf{V}_{in}\mathbf{(B)}$ — value at the entry of basic block **B**
    - $\mathbf{V}_{out}\mathbf{(B)}$ — value at the exit of basic block **B**
  - $\mathbf{V}_{out}\mathbf{(B) = F(\ V}_{in}\mathbf{(B)\ )}$
- **F** for GCP:

  $\mathbf{V}_{out}\mathbf{(B) = (\ V}_{in}\mathbf{(B) - DEF}_{v}\mathbf{(B)\ )} \cup \mathbf{DEF}_{c}\mathbf{(B)}$

  where DEF_v(B) contains variable definitions in B
  DEF_c(B) contains constant definitions in B
- Easier to reason about if you treat each individual statement as a basic block

Propagation of Values for GCP

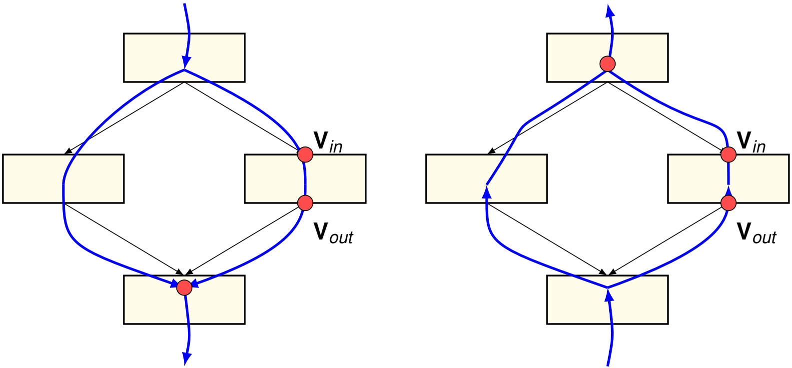
- There are two modes of propagation: **F** and $\wedge$



- Function **F**— propagates values through basic blocks
  - Variables in DEF_v are set to #
  - Variables in DEF_c are set to constant value
- $\wedge$ operator — propagates values through CFG edges
  - Merges values from multiple predecessor blocks

What is D?

- **D**: Direction of propagation (forwards or backwards)



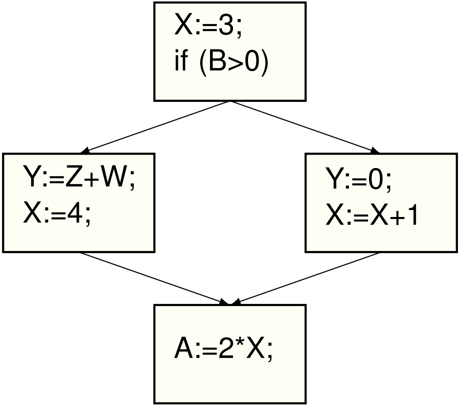
Forward Analsysis **Backward Analysis**

What is D?

- Values are propagated forward: **Forward Analysis**
- Values are propagated backward: **Backward Analysis**
- GCP is an example of a Forward Analysis
  - Starting from a constant definition, the 'constantness' of a variable propagates forward through CFG
- We will see an example of Backward Analysis soon

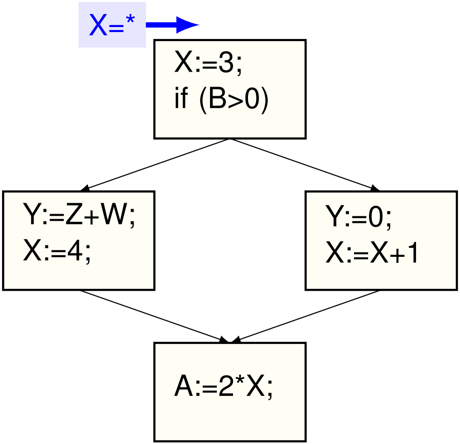
Example GCP without Loop

- In this example, constants can be propagated to **X+1**, **2*X**



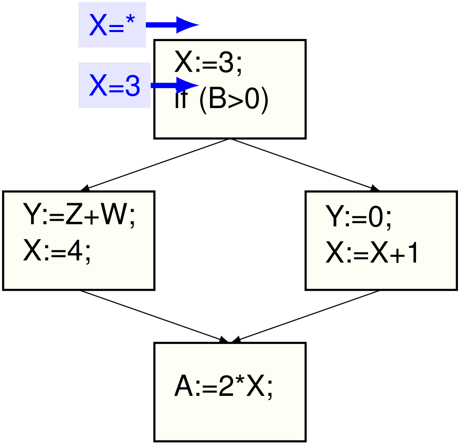
Example GCP without Loop

- In this example, constants can be propagated to **X+1**, **2*X**



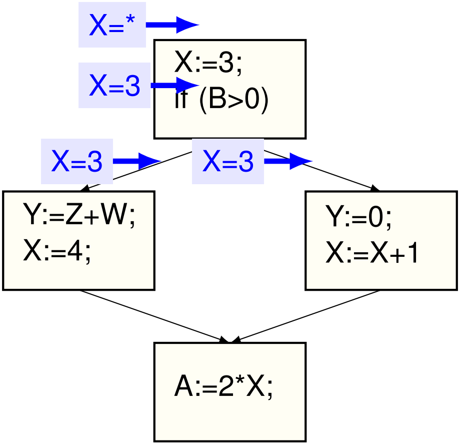
Example GCP without Loop

- In this example, constants can be propagated to **X+1**, **2*X**



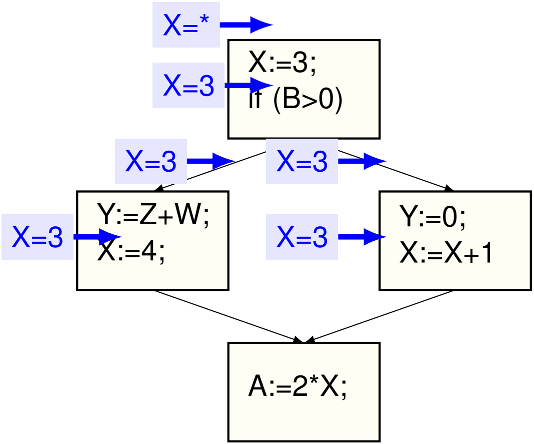
Example GCP without Loop

- In this example, constants can be propagated to **X+1**, **2*X**



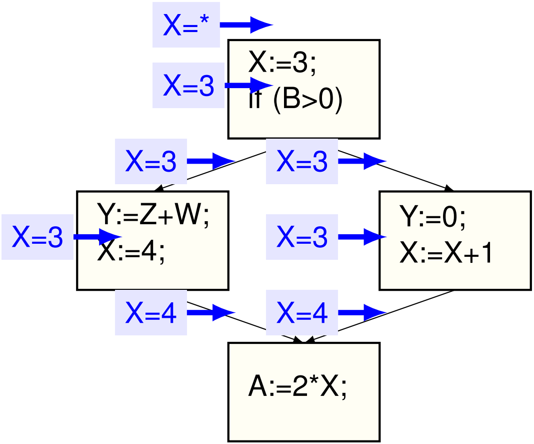
Example GCP without Loop

- In this example, constants can be propagated to **X+1**, **2*X**



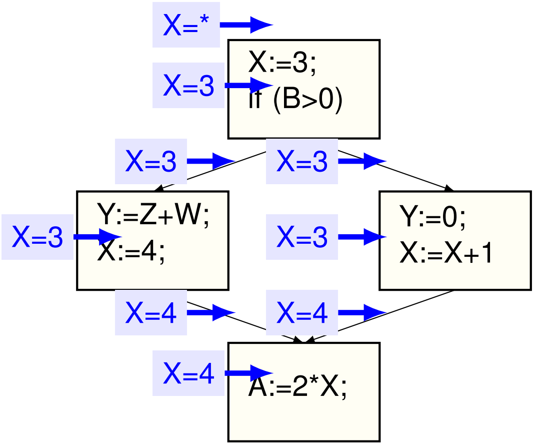
Example GCP without Loop

- In this example, constants can be propagated to **X+1**, **2*X**



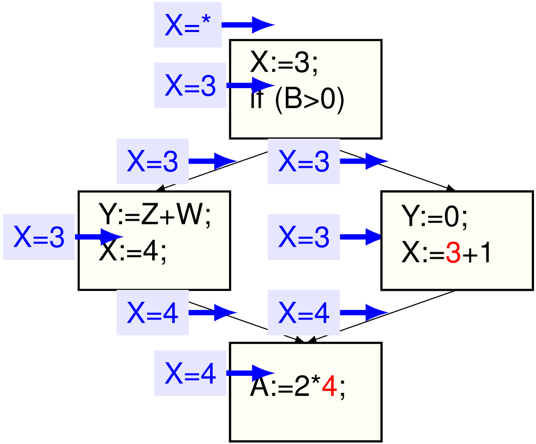
Example GCP without Loop

- In this example, constants can be propagated to **X+1**, **2*X**



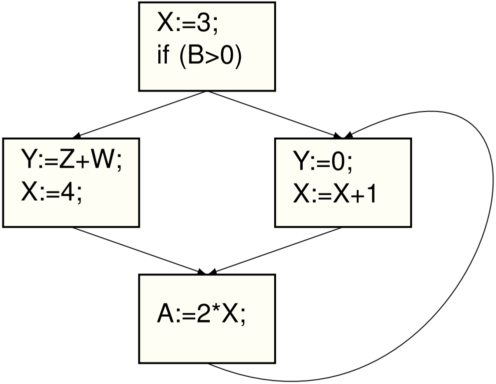
Example GCP without Loop

- In this example, constants can be propagated to **X+1**, **2*X**



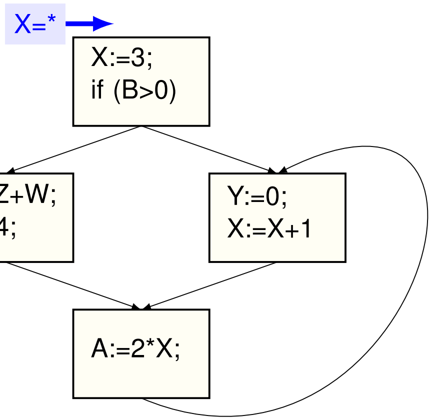
Example GCP with Loop

- In this example, loop prevents any constant propagation



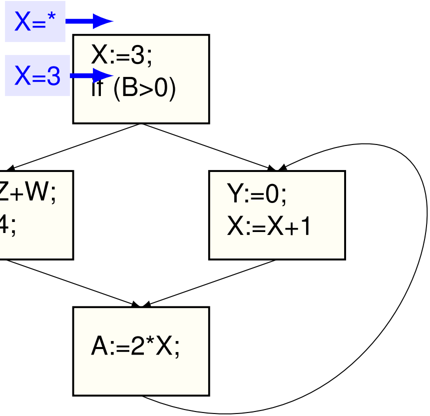
Example GCP with Loop

- In this example, loop prevents any constant propagation



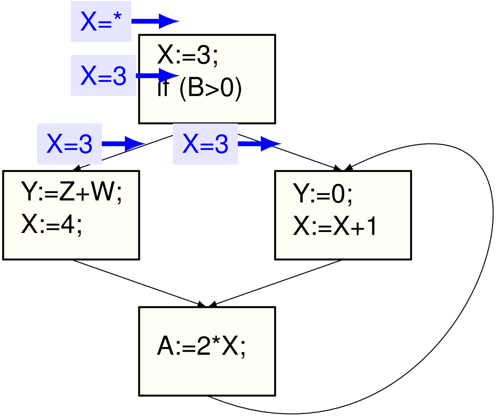
Example GCP with Loop

- In this example, loop prevents any constant propagation



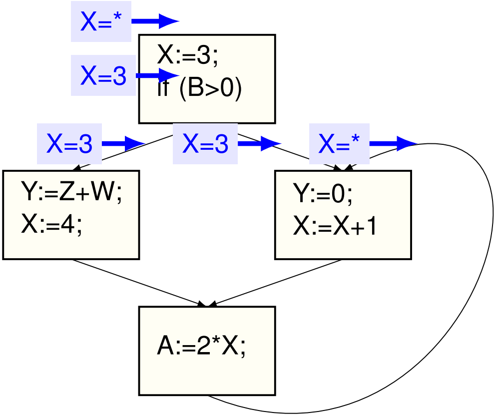
Example GCP with Loop

- In this example, loop prevents any constant propagation



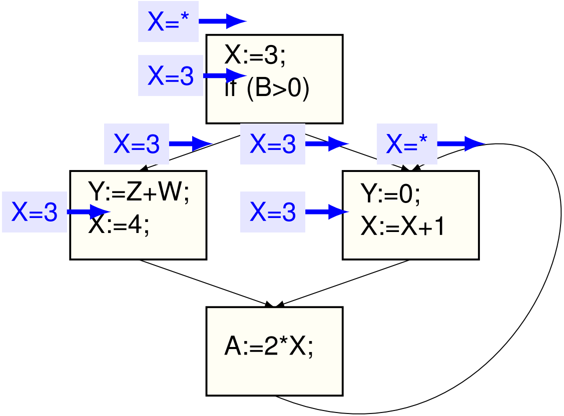
Example GCP with Loop

- In this example, loop prevents any constant propagation



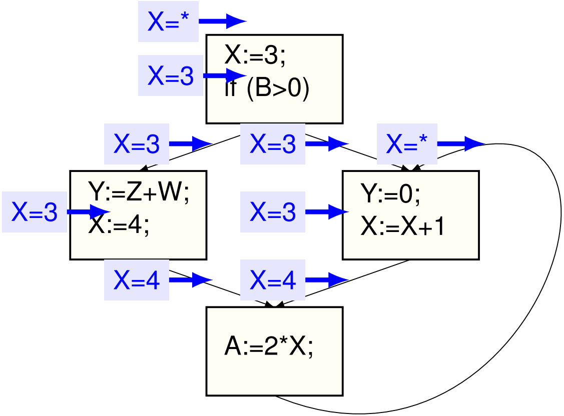
Example GCP with Loop

- In this example, loop prevents any constant propagation



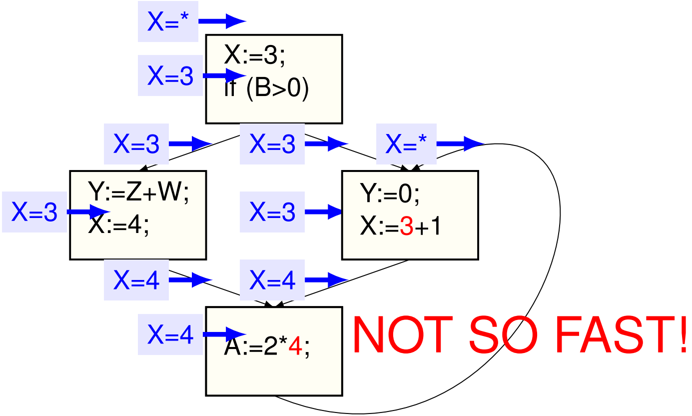
Example GCP with Loop

- In this example, loop prevents any constant propagation



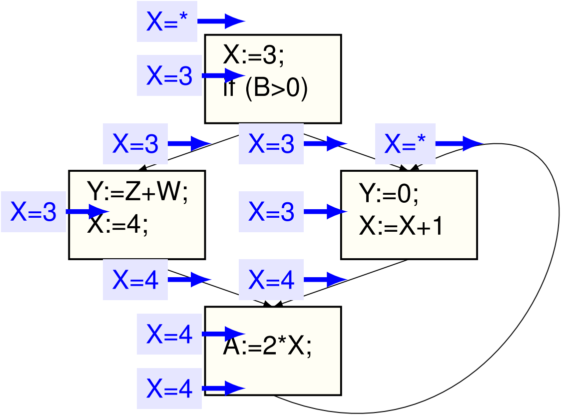
Example GCP with Loop

- In this example, loop prevents any constant propagation



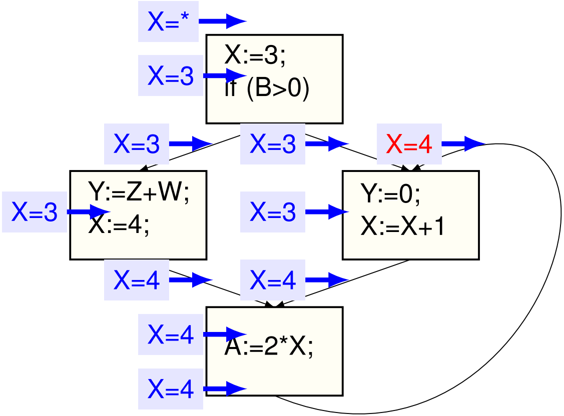
Example GCP with Loop

- In this example, loop prevents any constant propagation



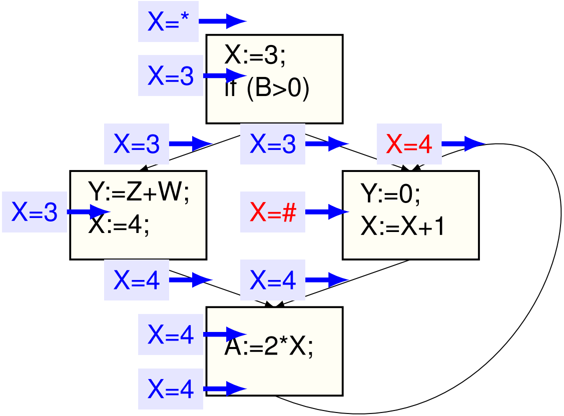
Example GCP with Loop

- In this example, loop prevents any constant propagation



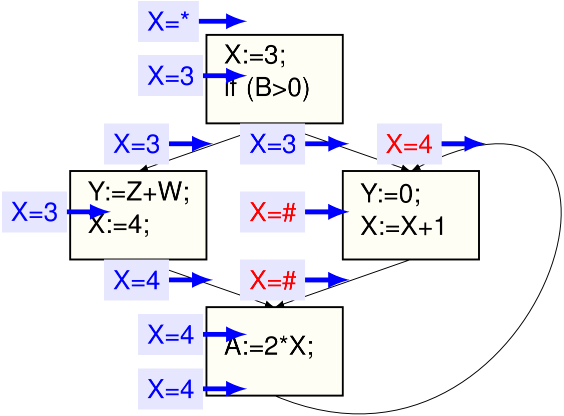
Example GCP with Loop

- In this example, loop prevents any constant propagation



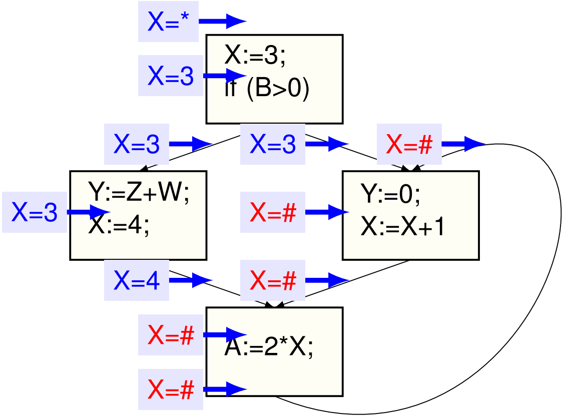
Example GCP with Loop

- In this example, loop prevents any constant propagation



Example GCP with Loop

- In this example, loop prevents any constant propagation



Forward Analysis Algorithm

- Pseudocode for Forward Analysis

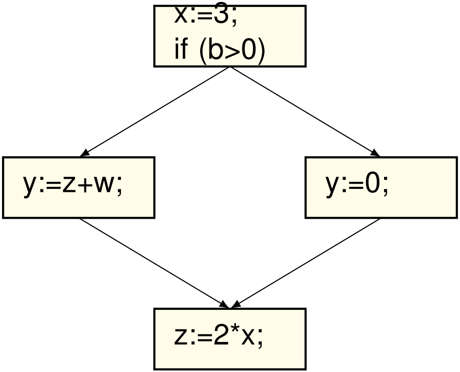
  for (each basic block B) $V_{out}(B) = \top$;
  while (changes to any $V_{out}(B)$ occur)
     for (each basic block B) {
        $V_{in}(B) = \wedge_{P\ is\ a\ predecessor\ of\ B}\ V_{out}(P)$
        $V_{out}(B) = F(V_{in}(B))$
     }
- $\wedge$ and **F** defined differently for each type of analysis
- Will forward analysis for GCP eventually stop?
  - If there are loops, we may go through the loop many times
  - Is there a possibility of an infinite loop?

Termination Problem

- Existence of $\perp$ value ensures termination
  - Values start from $\top$
  - Values can only go down in the semilattice
  - 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)

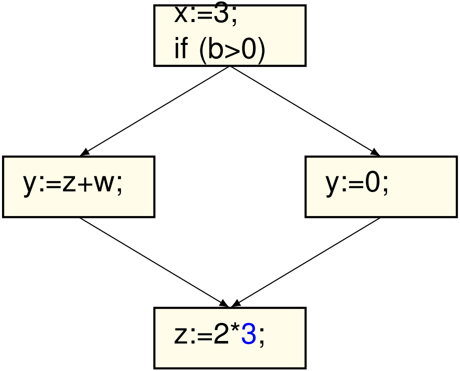
Another Analysis: Liveness Analysis

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



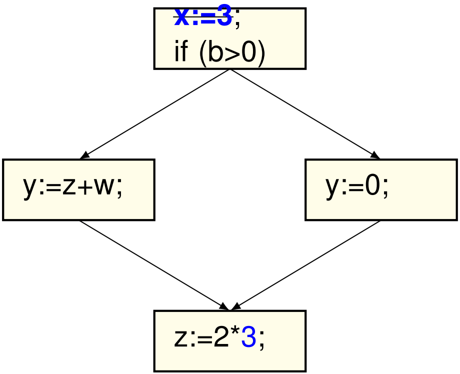
Another Analysis: Liveness Analysis

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



Another Analysis: Liveness Analysis

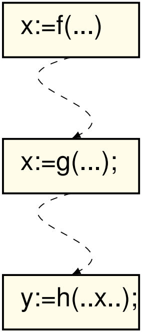
- Once constants have been globally propagated, we would like to eliminate the dead code



Live/Dead Statment

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

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



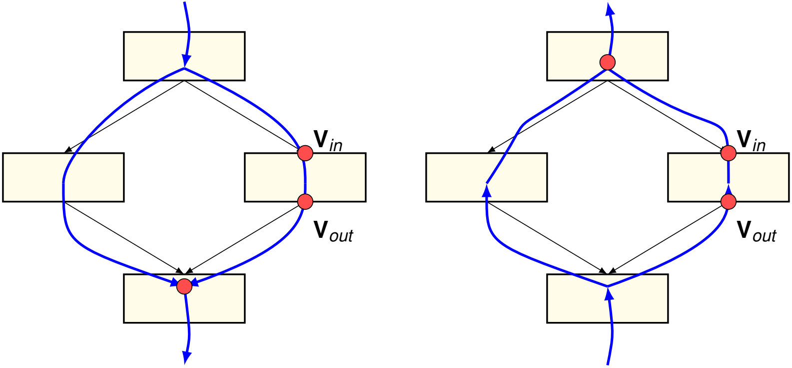
Liveness Analysis

- Global Liveness Analysis (GLA)
  - A variable X is live at statement S if
    - There exists a statement S2 after S that uses X
    - There is a path from S to S2
    - There is no intervening assignment to X between S and S2

Liveness Analysis

- Global Liveness Analysis (GLA)
  - A variable X is live at statement S if
    - There exists a statement S2 after S that uses X
    - There is a path from S to S2
    - There is no intervening assignment to X between S and S2
- Again a dataflow analysis framework can be applied to the problem
  - What is **D**, **V**, $\wedge$, **F: V** $\rightarrow$ **V** in this context?
- What is **D**?
  - Liveness Analysis is a **Backward Analysis**
    - Starting from a use, the 'liveness' of a variable propagates backward through CFG
  - Changes direction of $\wedge$ operator and transfer function

Forward and Backward Analysis Again



Forward Analsysis **Backward Analsysis**

What is V?

- Definition: Set of values in property under analysis
  - **V** for GLA: Each value is a set of live variables
  - Example values: {x, y, z}, {y}
- $\top$: initial value at the beginning
  - $\top$ for GLA = {}
  - Start with assumption that no variables are live
- $\perp$: the don't know value
  - $\perp$ for GLA = {all variables in function}
  - Meaning: none of the variables are provably dead

What is $\wedge$?

- $\wedge$: Meet operator ($\mathbf{V} \wedge \mathbf{V} \rightarrow \mathbf{V}$) for backward analysis
  - Defines behavior when values meet at control flow merges
  - Given
    - $\mathbf{V}_{in}$**(B)** — value at the entry of basic block **B**
    - $\mathbf{V}_{out}$**(B)** — value at the exit of basic block **B**
  - $\mathbf{V}_{out}\mathbf{(B)} = \wedge\ \mathbf{V}_{in}\mathbf{(S)}$ for each **S**, where **S** is successor of **B**
  - Note the reversal in direction! GLA is a backward analysis.
- $\wedge$ operator for GLA:
  - Meet operator is a simple union $\cup$
  - Example: $\{x, y\} \wedge \{y, z\} = \{x, y\} \cup \{y, z\} = \{x, y, z\}$
  - Union operation monotonically increases set, hence values form a semilattice from $\top$ to $\bot$

What is F?

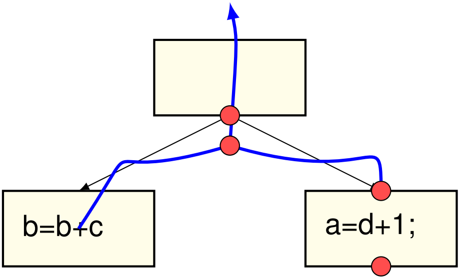
- **F**: Transfer function (**F: V** $\rightarrow$ **V**) for backward analysis
  - Defines what happens to value within a basic block
  - Given
    - $\mathbf{V}_{in}\mathbf{(B)}$ — value at the entry of basic block **B**
    - $\mathbf{V}_{out}\mathbf{(B)}$ — value at the exit of basic block **B**
  - $\mathbf{V}_{in}\mathbf{(B) = F(\ V}_{out}\mathbf{(B)\ )}$
  - Again note the reversal in direction!
- F for GLA:

  $\mathbf{V}_{in}\mathbf{(B) = (\ V}_{out}\mathbf{(B) - DEF(B)\ )} \cup \mathbf{USE(B)}$

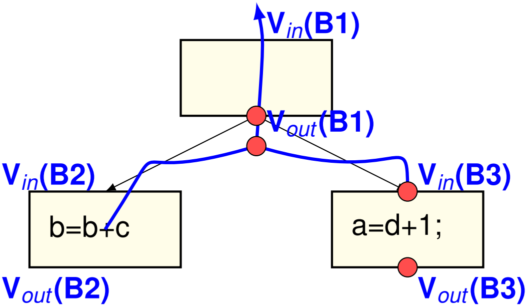
  where DEF(B) contains variable definitions in B

  USE(B) contains variable uses in B
- Easier to reason about if you treat each individual statement as a basic block

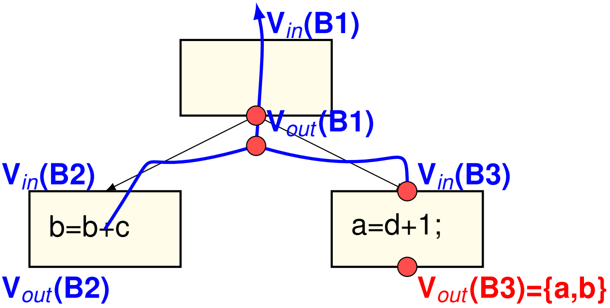
Liveness Example



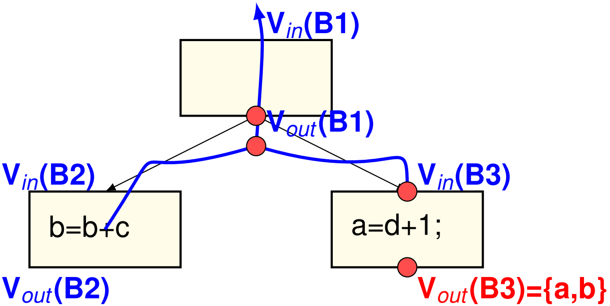
Liveness Example



Liveness Example

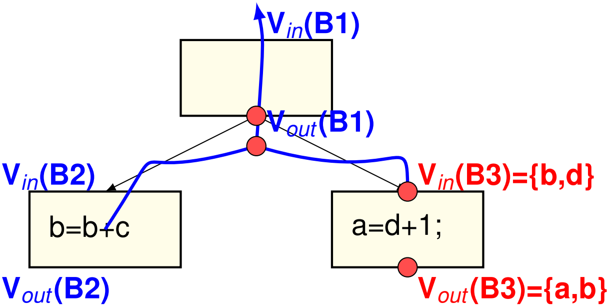


Liveness Example



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

Liveness Example

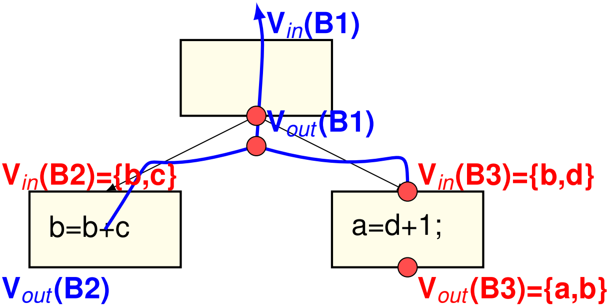


Two sets:

DEF={a}

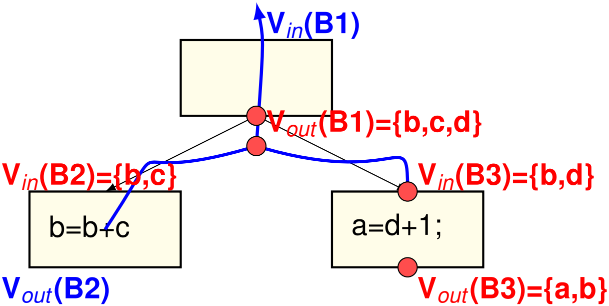
USE={d}

Liveness Example



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

Liveness Example



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

Backward Analysis Algorithm

- Pseudocode for Backward Analysis

  for (each basic block B) V_{in}(B) = $\top$;
  while (changes to any V_{in}(B) occur)
    for (each basic block B) {
      $V_{out}(B) = \wedge_{S\ is\ a\ successor\ of\ B} V_{in}(S)$
      $V_{in}(B) = F(V_{out}(B))$
    }

- Note the reversal in direction compared to forward analysis
- Will backward analysis for GLA eventually stop?
  - Again existence of $\perp$ value ensures termination
  - Value can change N times, where N is the number of variables used in function
  - The maximal number of steps is O(program_size * N)

Comparison of GCP and GLA

- **D**: Direction of propagation
  - GCP: Forward
  - GLA: Backward
- **V**: Set of values propagated
  - GCP: Whether each variable is constant, and if so the value
  - GLA: Set of live variables
- $\wedge$: Meet operator
  - GCP: Defined by semilattice (Top $\rightarrow$ Constant $\rightarrow$ Bottom)
  - GLA: Simply the set union operator
- **F**: Transfer function
  - GCP: Subtract variable definitions, add constant definitions
  - GLA: Add variable uses, subtract variable definitions

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

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

Register Allocation

What is Register Allocation?

- Process of assigning (a large number of) variables to (a small number of) CPU registers
- Registers are fast
  - access to memory: 100s of cycles
  - access to cache: a few to 10s of cycles
  - access to registers: 1 cycle
- But registers are limited in number
  - x86: 8 regs, MIPS: 32 regs, ARM: 32 regs ...
- Goals of register allocation:
  - Keep frequently accessed variables in registers
  - Keep variables in registers only as long as they are live

Local Register Allocation

- Allocate registers basic block by basic block
  - Makes allocation decisions on a per-block basis
  - Hence the prefix 'local'
  - Uses results of Global Liveness Analysis
- Requires only a single scan through each basic block
  - Keeps track of two tables:
    - Register table: which regs are currently allocated and where
    - Address table: location(s) where each variable is stored (locations can be: register, stack memory, global memory)
  - For every use of variable:
    - If variable is already in reg, no action
    - If not, allocate reg to variable from available regs
    - If no available regs, select reg for displacement

Local Register Allocation

- Which register should be displaced?
  - Register whose value is no longer live (given by GLA)
  - Register whose value has a copy in another location
  - These registers can be safely recycled
  - Otherwise the register needs to be spilled
- **Spill**: storing register back into own memory location
  - Generate store instruction to memory on assignment
  - Generate load instruction from memory on use
  - Own memory location can be in
    - Stack memory: local variables, temporary variables
    - Global memory: global variables
- At the end of basic block all live registes are spilled
  - Makes all registers available for next basic block allocation (Gives allocator clean slate for next basic block)
  - Can be source of inefficiency due to unnecessary spills $\rightarrow$ Addressed by Global Register Allocation

Global Register Allocation

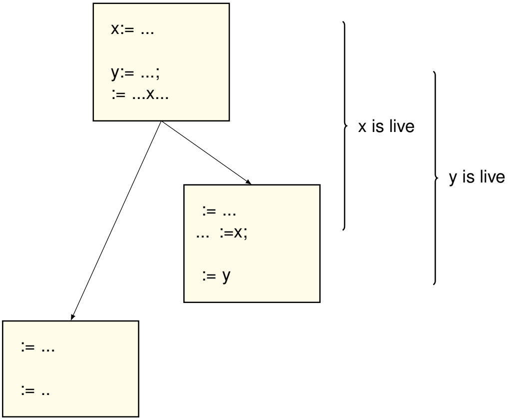
- Allocates registers across basic blocks
- Relies on Global Liveness Analysis just like local register allocation
- Three popular register allocation algorithms
  1. Graph coloring allocator
  2. Linear scan allocator
  3. ILP (Integer Linear Programming) allocator

Graph Coloring Allocator

- **Algorithm steps**:
  1. Identify live range interference using GLA
  2. Build register interference graph
  3. Attempt K-coloring of the graph
     - K is the number of available registers
  5. If none found, modify the program, rebuild graph until K-coloring can be obtained
     - Insert spill code to the program

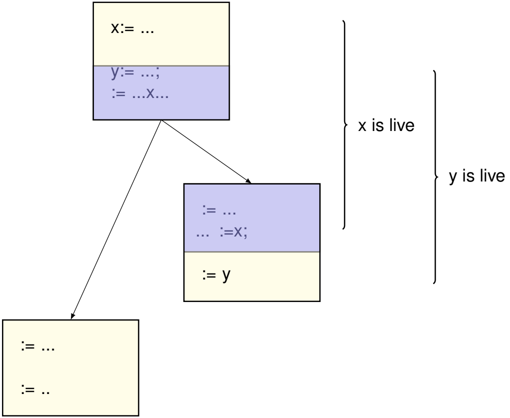
Live Range Interference

- **Live Range**: Set of program points where a variable is live
  - Two live ranges interfere if there is an overlap
  - Vars with interfering ranges cannot reside in same register



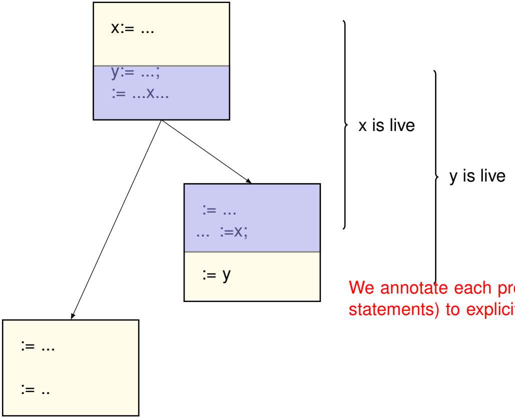
Live Range Interference

- **Live Range**: Set of program points where a variable is live
  - Two live ranges interfere if there is an overlap
  - Vars with interfering ranges cannot reside in same register



Live Range Interference

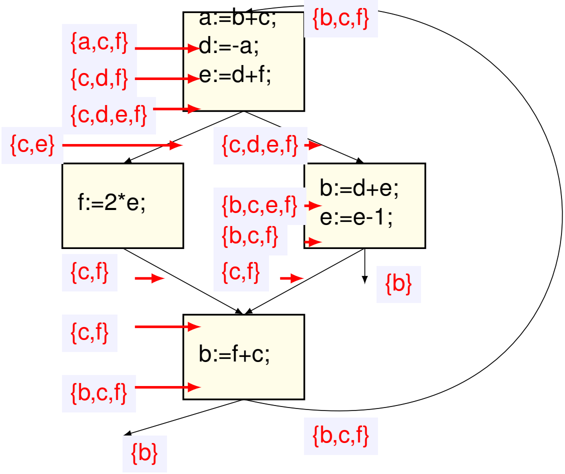
- **Live Range**: Set of program points where a variable is live
  - Two live ranges interfere if there is an overlap
  - Vars with interfering ranges cannot reside in same register



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

GLA and Live Range Interference

- Example of GLA and interfering live ranges

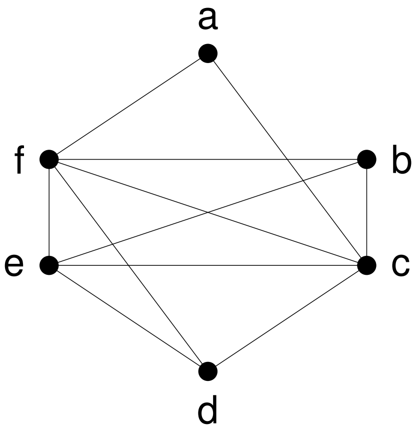


Register Interference Graph

- Construct **Register Interference Graph (RIG)** such that
  - Each node represents a variable
  - An edge between two nodes V_1 and V_2 represents an inteference in live ranges
- 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

- In the RIG for our example:
  - b,c cannot be in the same register
  - a,b,d can be in the same register

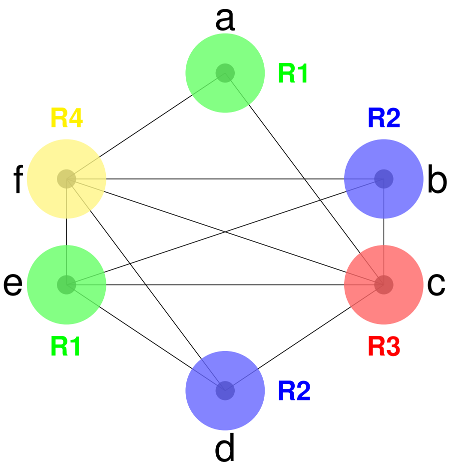


Allocating Registers using Graph Coloring

- Graph coloring is a theoretical problem where ...
  - 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
- Problem of register allocation in RIG maps to graph coloring problem
  - Instead of assigning k-colors, we need to assign k registers
  - 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 Result

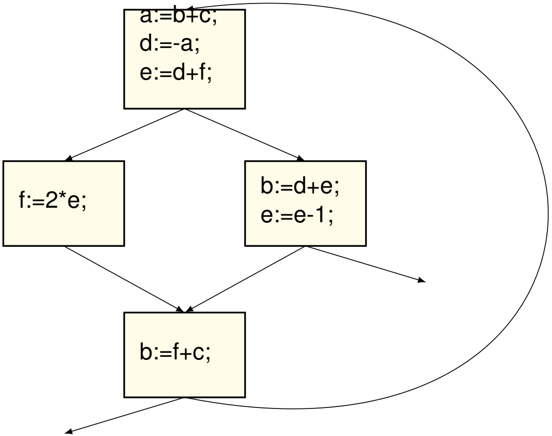
- This is an coloring of our example RIG using 4 colors
  - There is no solution with less than 4 colors



After Register Allocation

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

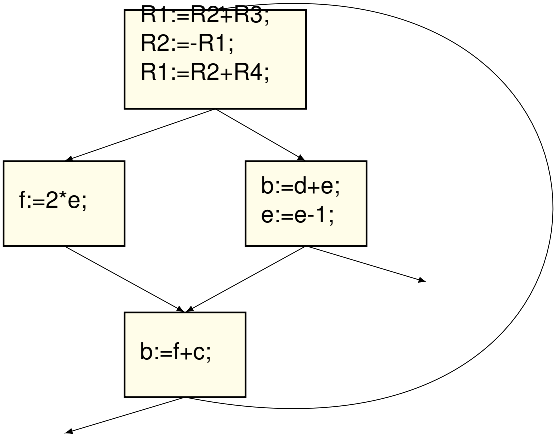
a–R1
b–R2
c–R3
d–R2
e–R1
f–R4



After Register Allocation

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

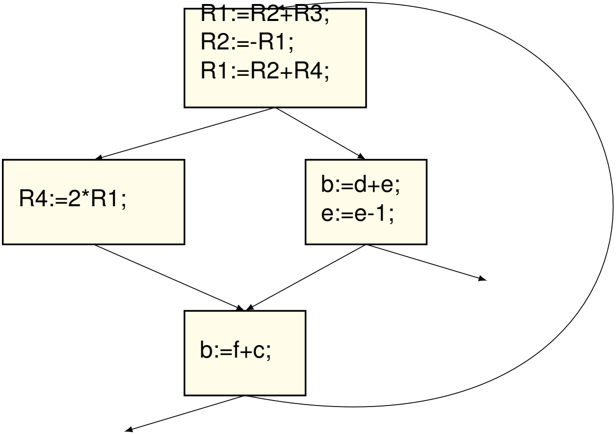
a–R1
b–R2
c–R3
d–R2
e–R1
f–R4



After Register Allocation

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

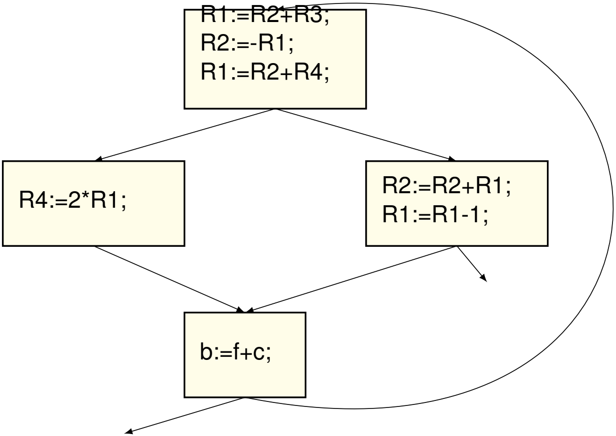
a–R1
b–R2
c–R3
d–R2
e–R1
f–R4



After Register Allocation

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

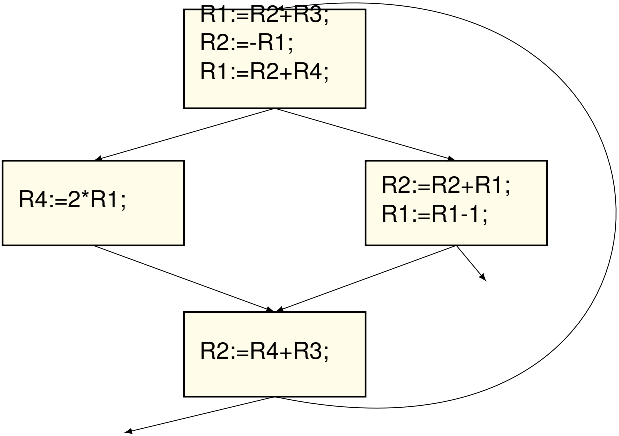
a–R1
b–R2
c–R3
d–R2
e–R1
f–R4



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

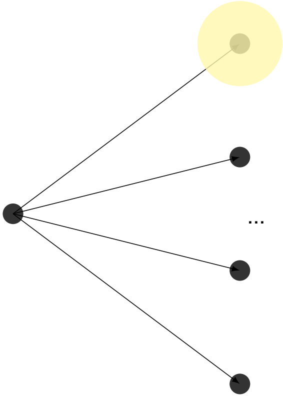


How is Graph Coloring Performed?

- For graph G and k>2, determining whether G is k-colorable is NP complete
  - Problem of k-register allocation is NP complete
  - In practice: use heuristical polynomial algorithm that gives close to optimal allocations most of the time
  - Chaitin's graph coloring is a popular heuristical algorithm
    - Most backends of GCC use Chaitin's algorithm by default
- What if k-register allocation does not exist?
  - Spill a register to memory to reduce RIG and try again

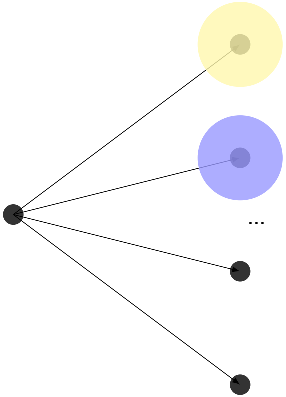
Chaitin's Graph Coloring

- **Observation**: for a **k**-coloring problem, a node with **k-1** neighbors can always be colored, no matter what



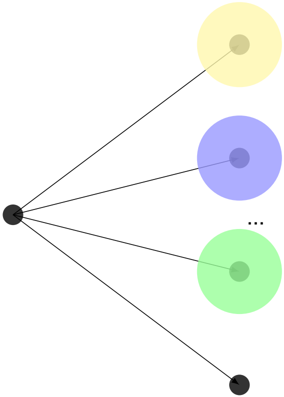
Chaitin's Graph Coloring

- **Observation**: for a **k**-coloring problem, a node with **k-1** neighbors can always be colored, no matter what



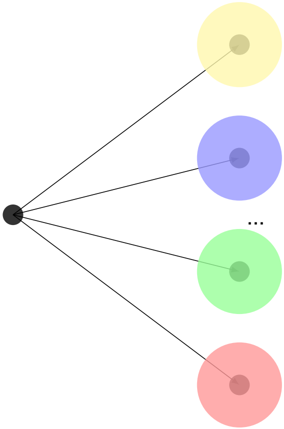
Chaitin's Graph Coloring

- **Observation**: for a **k**-coloring problem, a node with **k-1** neighbors can always be colored, no matter what



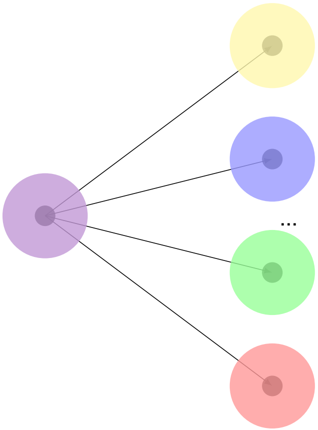
Chaitin's Graph Coloring

- **Observation**: for a **k**-coloring problem, a node with **k-1** neighbors can always be colored, no matter what



Chaitin's Graph Coloring

- **Observation**: for a **k**-coloring problem, a node with **k-1** neighbors can always be colored, no matter what

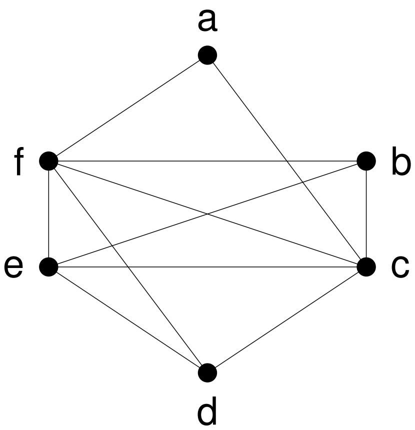


Chaitin's Graph Coloring

- **Corollary**: Given graph **G** for a **k**-coloring problem
  - Let **G'** be the graph after removing a node with fewer than **k** neighbors
  - If **G'** can be **k**-colored then **G** can be **k**-colored
- **Insight**: Solving for G' is easier than solving for G, so solve for G' instead of G
- **Algorithm**
  - Phase 1: Repeat until there are no nodes left
    - Pick a node V with fewer than k neighbors
    - Put V on a stack and remove it and its associated edges from the graph
  - Phase 2: Assign colors to nodes on the stack in LIFO order
    - Pick a color that is different from its neighbors
    - Such a color is guaranteed to exist due to corollary (Analogous to coloring G after adding removed node to G')

Chaitin's Graph Coloring Example

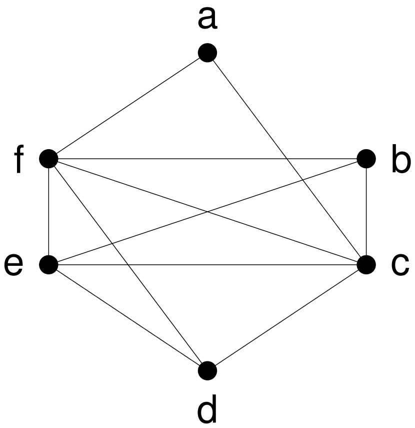
- Chaitin's algorithm applied to our example where k=4



Stack={}

Chaitin's Graph Coloring Example

- Chaitin's algorithm applied to our example where k=4

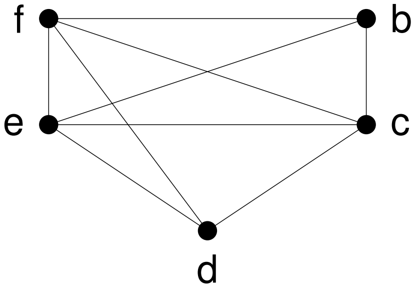


Stack={a}

Chaitin's Graph Coloring Example

- Chaitin's algorithm applied to our example where k=4

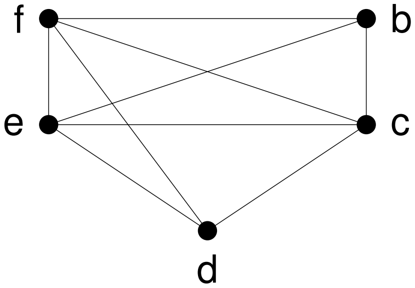
Stack={a}



Chaitin's Graph Coloring Example

- Chaitin's algorithm applied to our example where k=4

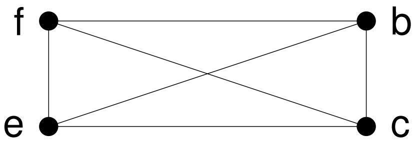
Stack={a,d}



Chaitin's Graph Coloring Example

- Chaitin's algorithm applied to our example where k=4

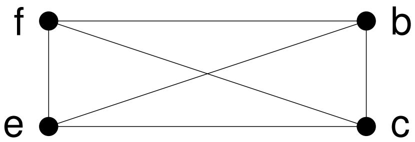
Stack={a,d}



Chaitin's Graph Coloring Example

- Chaitin's algorithm applied to our example where k=4

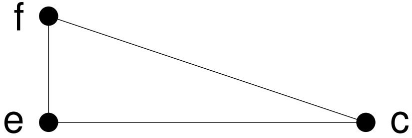
Stack={a,d,b}



Chaitin's Graph Coloring Example

- Chaitin's algorithm applied to our example where k=4

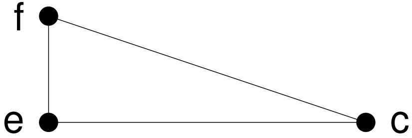
Stack={a,d,b}



Chaitin's Graph Coloring Example

- Chaitin's algorithm applied to our example where k=4

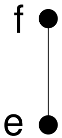
Stack={a,d,b,c}



Chaitin's Graph Coloring Example

- Chaitin's algorithm applied to our example where k=4

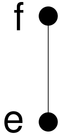
Stack={a,d,b,c}



Chaitin's Graph Coloring Example

- Chaitin's algorithm applied to our example where k=4

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



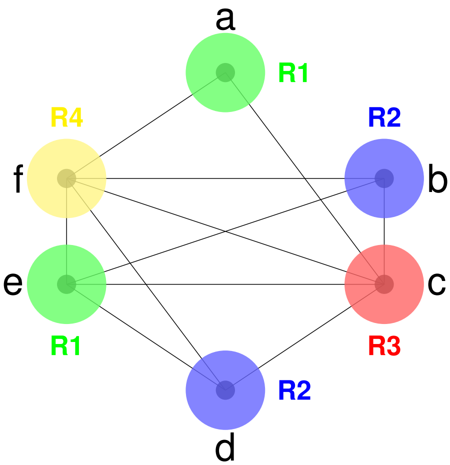
Chaitin's Graph Coloring Example

- Chaitin's algorithm applied to our example where k=4

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

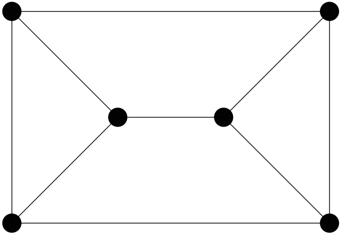
Coloring Result

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



Is Chaitin's Graph Coloring Optimal?

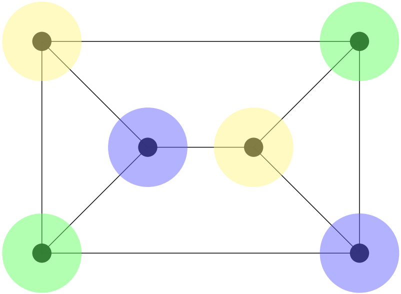
- According to Chaitin's algorithm:
  Every node has 3 outgoing edges, thus it is not 3-colorable



- However, it is 3-colorable as you can see above
- Chaitin's algorithm is not optimal

Is Chaitin's Graph Coloring Optimal?

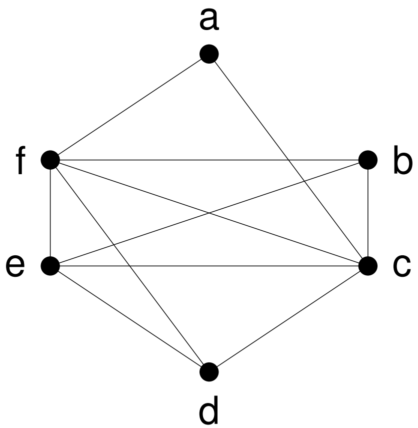
- According to Chaitin's algorithm:
  Every node has 3 outgoing edges, thus it is not 3-colorable



- However, it is 3-colorable as you can see above
- Chaitin's algorithm is not optimal

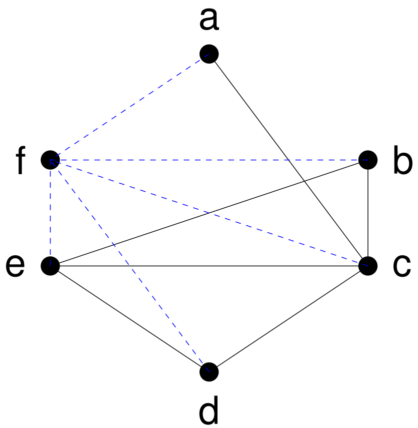
What if Coloring Fails?

- 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



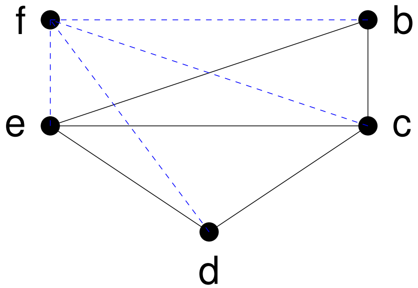
What if Coloring Fails?

- 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



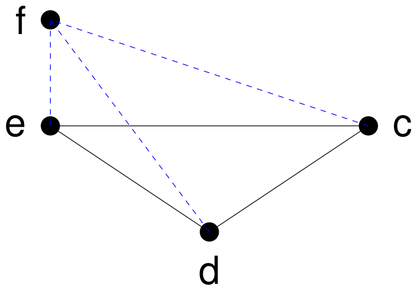
What if Coloring Fails?

- 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



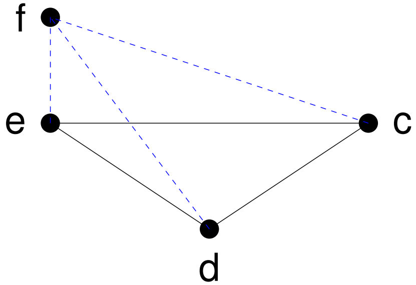
What if Coloring Fails?

- 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



What if Coloring 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

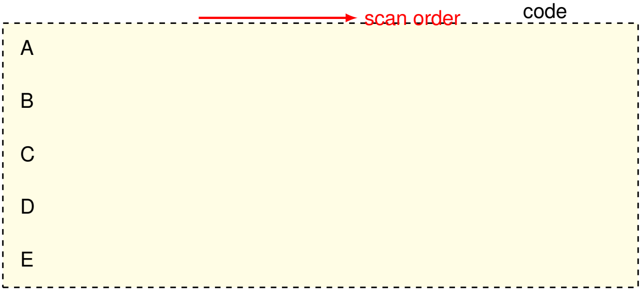


Linear Scan Register Allocation

- On-line compilers need to generate binary code quickly
  - Just-in-time compilation
  - Interactive environments e.g. IDE
- In these cases, it is beneficial to sacrifice code performance a bit for quicker compilation
  - A faster allocation algorithm
  - Not sacrificing too much in code quality
- Proposed in following publication:
  - Poletto, M., Sarkar, V., "Linear scan register allocation", in ACM Transactions on Programming Languages and Systems (TOPLAS), 1999

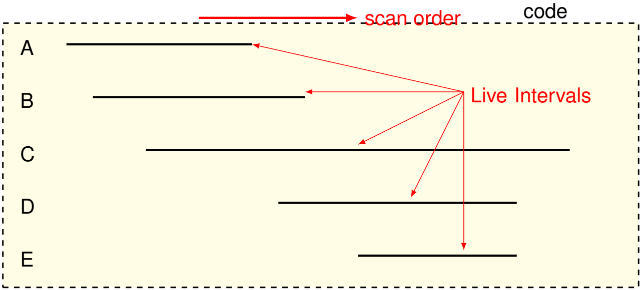
Linear Scan Register Allocation

- Layout the code in a certain linear order
- Do a single scan to allocate register for each **live interval**



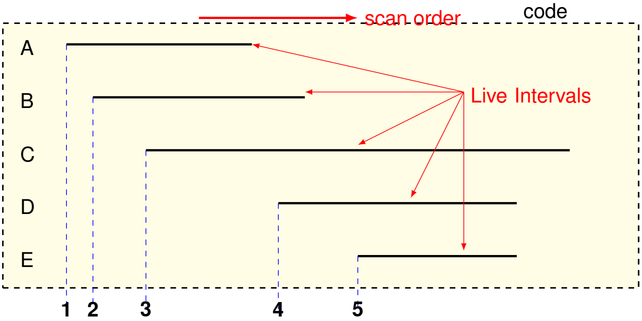
Linear Scan Register Allocation

- Layout the code in a certain linear order
- Do a single scan to allocate register for each **live interval**



Linear Scan Register Allocation

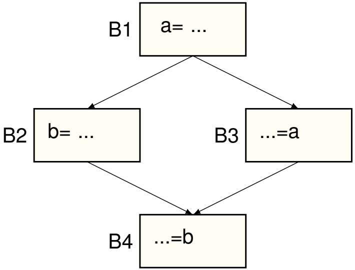
- Layout the code in a certain linear order
- Do a single scan to allocate register for each **live interval**



- Allocate greedily at each numbered point in program
  - **A** and **D** may be allocated to same register

Linear Scan and Live Intervals

- **Live Interval**: Smallest range of code containing live ranges
- Live range of a = {B1, B3}, b = {B2, B4}
- If code layout is "B1,B3,B2,B4", only 1 register is enough
  - Live interval of a = {B1, B3}, b = {B2, B4}
- If code layout is "B1,B2,B3,B4", then need 2 registers
  - Live interval of a = {B1, B2, B3}, b = {B2, B3, B4}



Linear Scan Algorithm

- Linear scan RA consists of four steps

  S1. Order all instructions in linear fashion
  - Order affects quality of allocation but not correctness

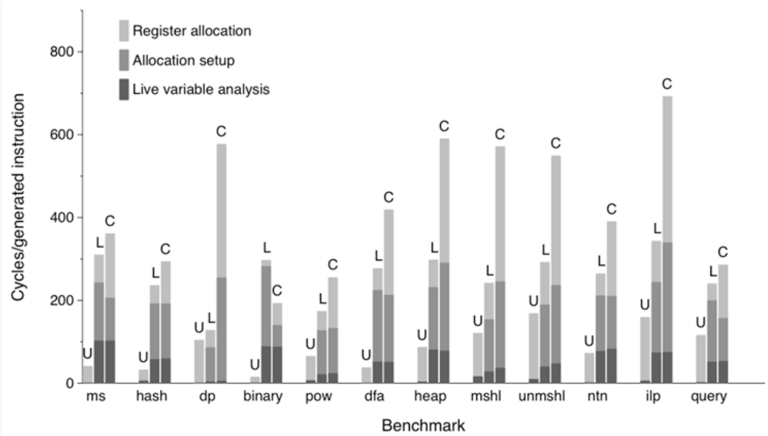
  S2. Calculate the set of live intervals
  - Each variable is given a live interval

  S3. Greedily allocate register to each interval in order
  - 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
  - CPU registers replace temporary or program variables
  - Spill code is generated

Register Allocation Time Comparison

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



ILP-based Register Allocation

- Uses linear programming to find the "optimal" register allocation
- Idea and steps:
  1. Convert RA problem to a ILP problem
  2. Solve ILP problem using widely known 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 $\longrightarrow$ does not scale well to large programs

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) ... = b + a ;

  ...

  - 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

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

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

Converting Register Allocation to ILP

S2: Constraints: clearly there are constraints for these variables

- $V^{Ri}_{var(location)}$ 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
  $V^{R1}_{b(10)} + V^{R1}_{a(10)} \leq 1$
- Any variable just needs to take one register
  $V^{R1}_{b(10)} + V^{R2}_{b(10)} + V^{R3}_{b(10)} + V^{R4}_{b(10)} = 1$
- and many more ...

S3: Define goal function

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

Conclusion

- Good Register Allocation is crucial to code quality
  - Accesses to memory are costly, even with caches
  - Even with only a handful of program variables, intermediate values introduce many more temporary variables adding to register pressure

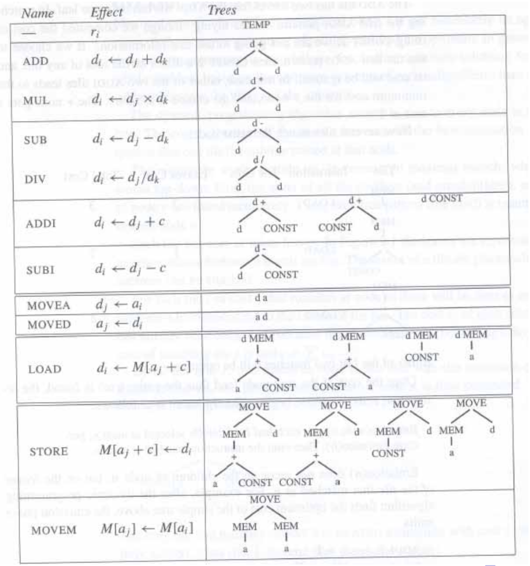
- Different algorithms make different trade-offs between allocation time (compiler performance) and code quality (application performance)

Instruction Selection

Instruction Selection

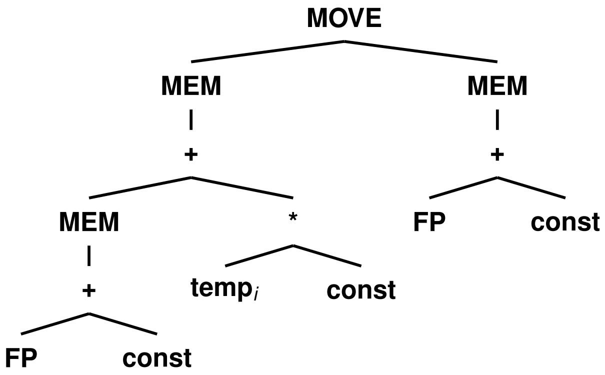
- Instruction selection is the task to select appropriate machine instructions to implement the operations in the intermediate representation (IR).
  - Very important for CISC machines, and machines with special purpose instructions (MMX)
  - X86, ARM, DSP, ...
- There are many semantically equivalent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

Some Instruction Patterns

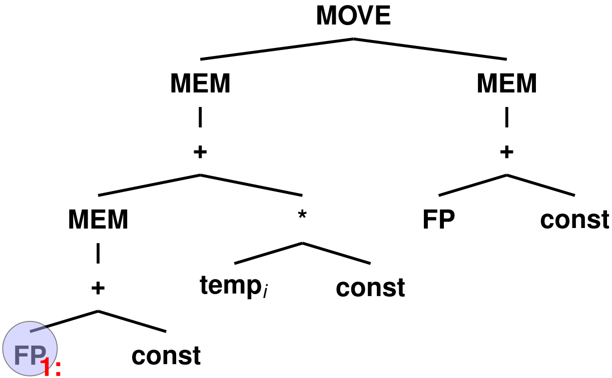


Name	Effect	Trees
—	r_i	TEMP
ADD	$d_i \leftarrow d_j + d_k$	d +(d, d)
MUL	$d_i \leftarrow d_j \times d_k$	d *(d, d)
SUB	$d_i \leftarrow d_j - d_k$	d -(d, d)
DIV	$d_i \leftarrow d_j / d_k$	d /(d, d)
ADDI	$d_i \leftarrow d_j + c$	d +(d, CONST); d +(CONST, d); d CONST
SUBI	$d_i \leftarrow d_j - c$	d -(d,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

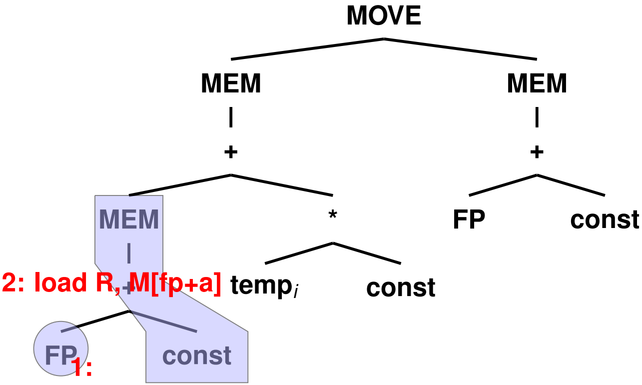
A Parse Tree to be Tiled



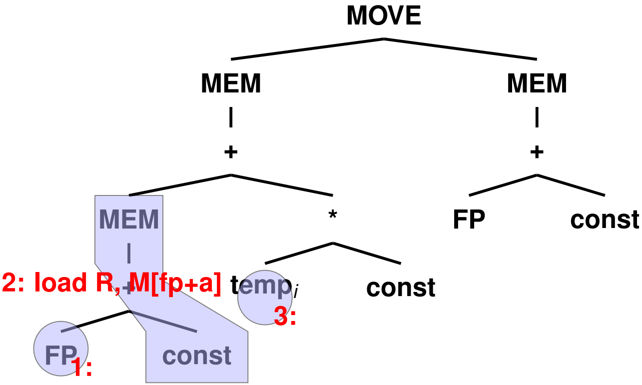
A Parse Tree to be Tiled



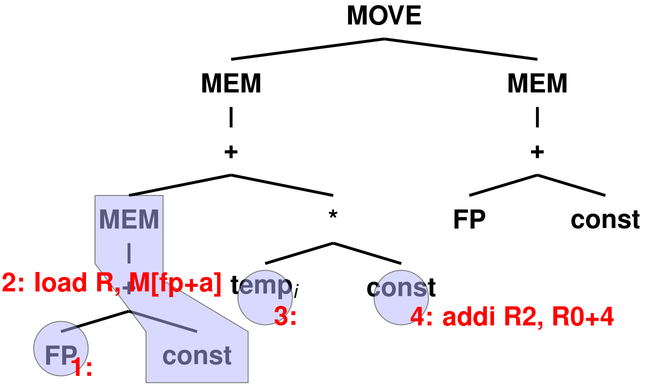
A Parse Tree to be Tiled



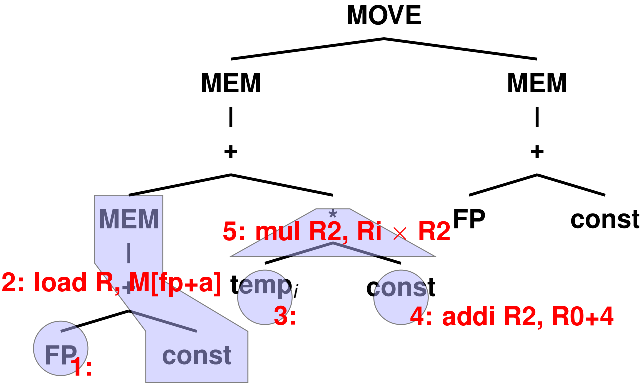
A Parse Tree to be Tiled



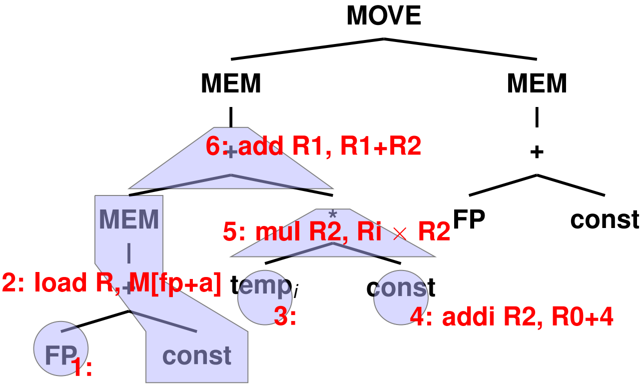
A Parse Tree to be Tiled



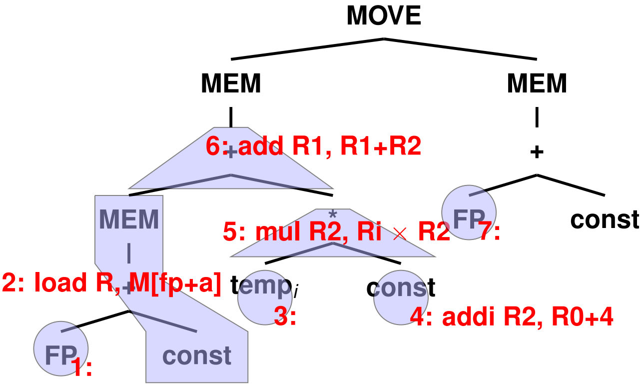
A Parse Tree to be Tiled



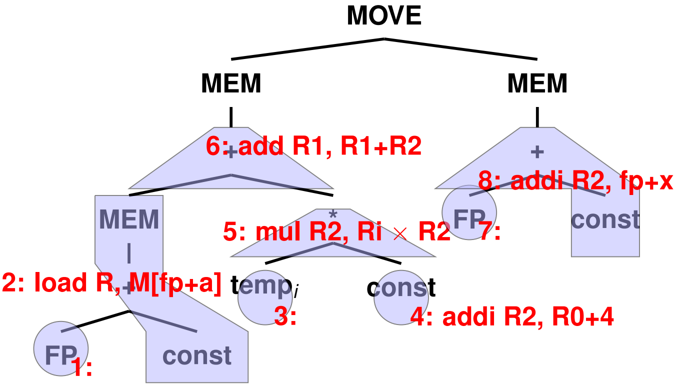
A Parse Tree to be Tiled



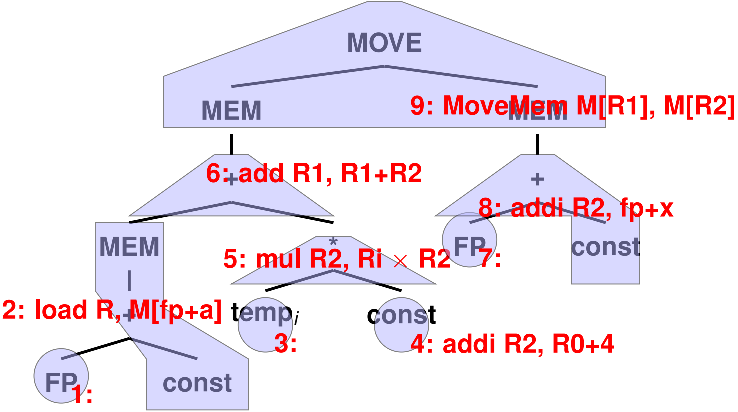
A Parse Tree to be Tiled



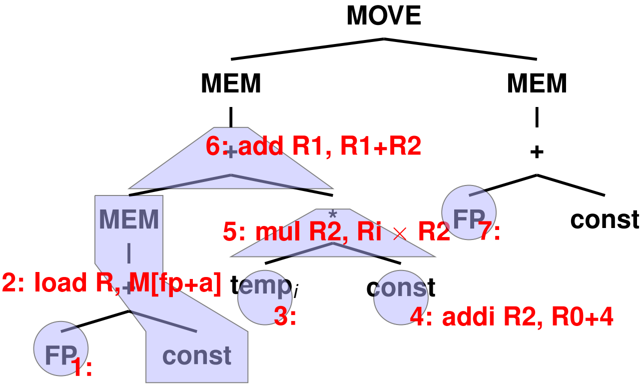
A Parse Tree to be Tiled



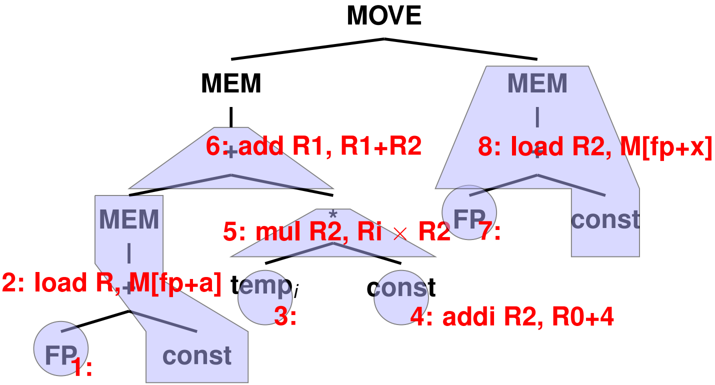
A Parse Tree to be Tiled



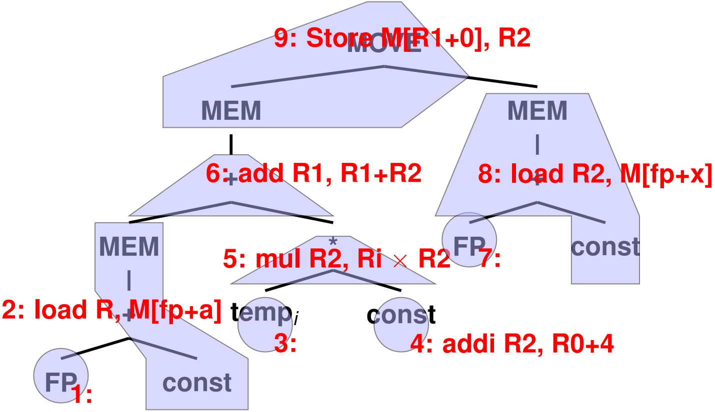
A Parse Tree to be Tiled



A Parse Tree to be Tiled



A Parse Tree to be Tiled



The END !