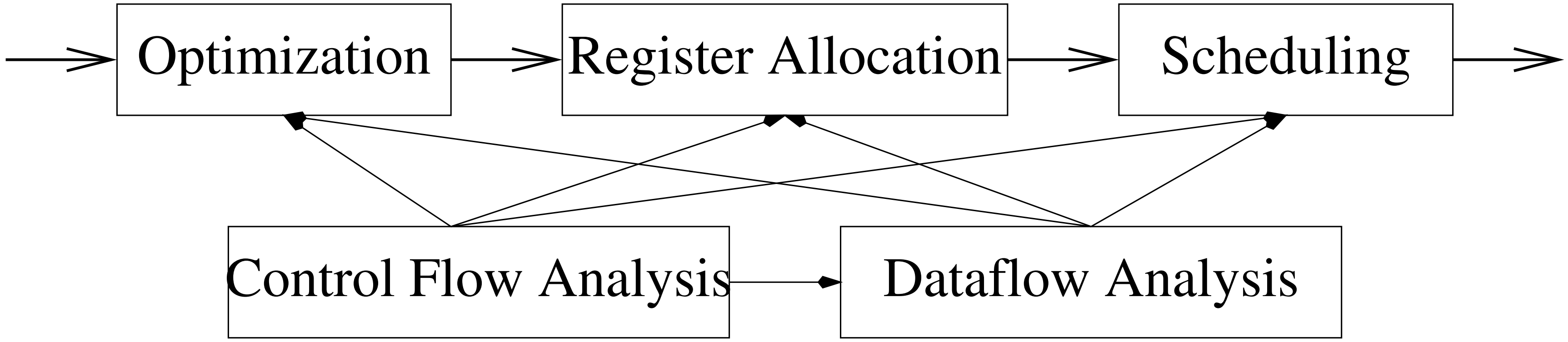
Analysis and Transformation

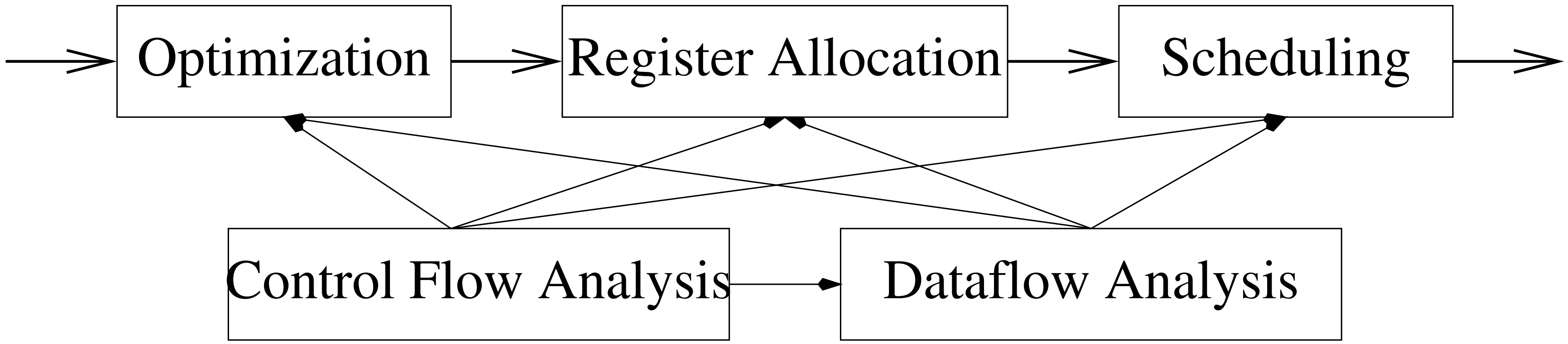
- Analysis:
  - Control Flow Analysis
  - Dataflow Analysis
- Transformation:
  - Register Allocation
  - Optimization
    - Copy propagation
    - Constant folding
    - Common subexpression elimination

Control Flow Analysis



- *Control Flow Analysis* determines the how instructions are *fetched* during execution.
  - *Control Flow Graph* - graph of instructions with directed edge $I_i \rightarrow I_j$ iff I_j can be executed immediately after I_i.
  - We use CFGs are a convenient program representation for analysis
  - We use CFGs to find loops where we concentrate optimization effort
  - Dominator information gives us guarantees about program control-flow that enables optimization.
- Control Flow Analysis precedes dataflow analysis.

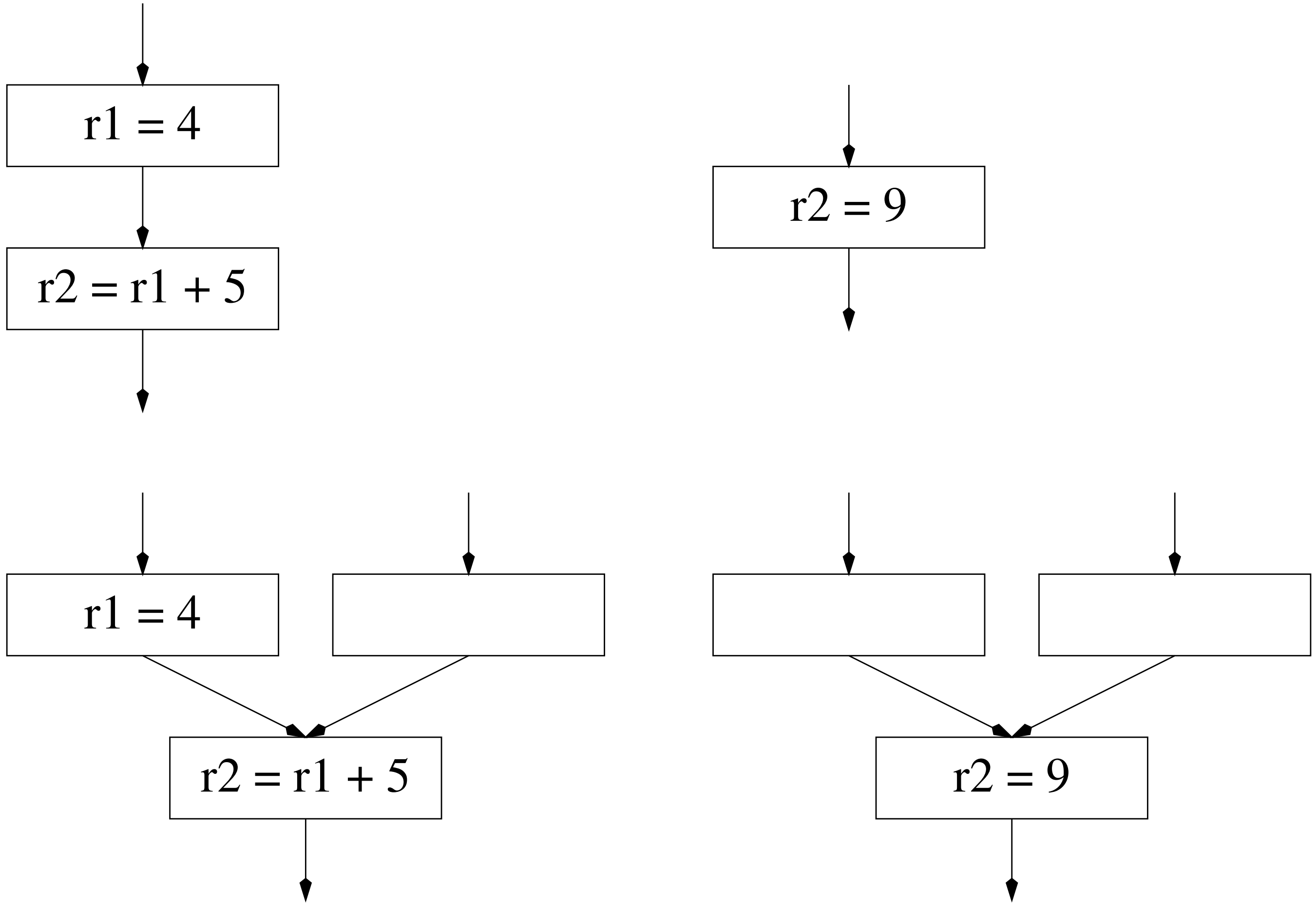
Control Flow Analysis



- *Dataflow analysis* determines how data flows among instructions.
- Questions answered by data flow analysis:
  - Is the contents of variable x at the program point immediately before the assignment y := x + 1 always 7? (allows constant folding)
  - Is y used after the instruction z := y * 2? (enables good register allocation)
  - Is the contents of variable y equal to the contents of variable z+1 each time around the loop? (allows hoisting out of a loop)
- Dataflow analysis precedes optimization, register allocation, and scheduling.

Dataflow Analysis Motivation

Constant Propagation and Dead Code Elimination:



Needs dominator, liveness, and reaching definition information.

Dataflow Analysis Motivation

Register Allocation:

- Infinite number of registers (virtual registers) must be mapped to a limited number of real registers.
- Pseudo-assembly must be examined by *live variable analysis* to determine which virtual registers contain values which may be used later.
- Virtual registers which are not simultaneously *live* may be mapped onto the same real register.

```
r2 = r1 + 1

r3 = M[r2]

r4 = r3 + 4

LOAD    r5 = M[r2 + r4]
```

Dataflow Analysis

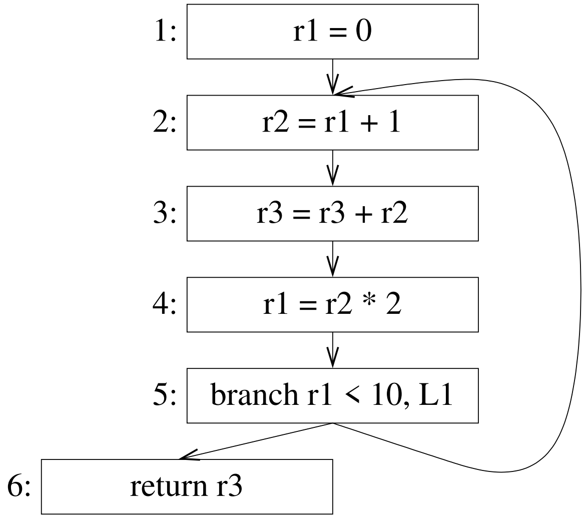
Three types we will cover:

- Live Variable
  - Live range for register allocation
  - Scheduling
  - Dead code elimination
- Reaching Definitions
  - Dead code elimination
  - Constant propagation
  - Constant folding
  - Copy propagation
- Available expressions
  - Common subexpression elimination

Constructing CFGs

```
  r1 = 0
LOOP:
  r2 = r1 + 1
  r3 = r3 + r2
  r1 = r2 * 2
  BRANCH r1 < 10, LOOP
```

```
RESULT:
  return r3
```



Definitions

Control Flow Definitions:

- CFG node has *out-edges* leading to *successor nodes*.
- CFG node has *in-edges* coming from *predecessor nodes*.
- For each CFG node n, $PRED[n]$ = set of all predecessors of n.
- For each CFG node n, $SUCC[n]$ = set of all successors of n.

Iterative Dataflow Analysis Framework

- These dataflow analyses are all very similar $\rightarrow$ define a framework.
- Specify:
  - Two *set definitions* - $A[n]$ and $B[n]$
  - A *transfer function* - $f\,(A, B, IN/OUT)$
  - A *confluence operator* - $\vee$.
  - A *direction* - FORWARD or REVERSE.
- For forward analyses:

$$IN[n] = \vee_{p \in PRED[n]} OUT[p]$$

$$OUT[n] = f\,(A, B, IN)$$

- For reverse analyses:

$$OUT[n] = \vee_{s \in SUCC[n]} IN[s]$$

$$IN[n] = f\,(A, B, OUT)$$

Iterative Dataflow Analysis Framework

- Iterative dataflow analysis equations are applied in an iterative fashion until IN and OUT sets do not change. (We reach a *fixed point*.)
- Typically done in (FORWARD or REVERSE) topological sort order of CFG for efficiency.
- IN and OUT sets often initialized to $\emptyset$; sometimes to maxset

```
For each node n {
  IN[n] = OUT[n] = {};
}
Repeat {
  For each node n in forward/reverse topological order {
    IN'[n] = IN[n];
    OUT'[n] = OUT[n];
    IN[n], OUT[n] = (Equations);
  }
} until IN'[n] = IN[n] and OUT'[n] = OUT[n] for all n.
```

Iterative Dataflow Analysis Framework

- Iterative dataflow analysis equations are applied in an iterative fashion until IN and OUT sets do not change.
- Question: how do we know the iterative data flow analysis will ever terminate?
  - It depends on the form of the equations. Inner loop of analysis:

    ```
    IN[n], OUT[n] = (Function of IN'[n], OUT'[n])
    ```

  - Conditions:
    * Equations are *monotonic* functions. They always add elements to the set or remove elements. More generally, they always increase IN[n] or decrease IN[n].
    * Sets have a finite maximum or minimum size.

Definitions

Liveness Definitions:

- A source (RHS) register t is a *use* of t.
- A destination (LHS) register t is a *definition* of t.

$r1 = r2 + r3$	r2, r3 are used; r1 is defined
compare $r1, r2$	r2, r3 are used; the condition code is defined
jmp $r3$	r3 is used
move $r3[r4] = 17$	r3,r4 are used; no registers are defined

- A register t is *live* on edge e if there exists a path from e to a use of t that does not go through a definition of t.
- Register t is *live-in* at CFG node n if t is live on any in-edge of n.
- Register t is *live-out* at CFG node n if t is live on any out-edge of n.

Live Variable Analysis

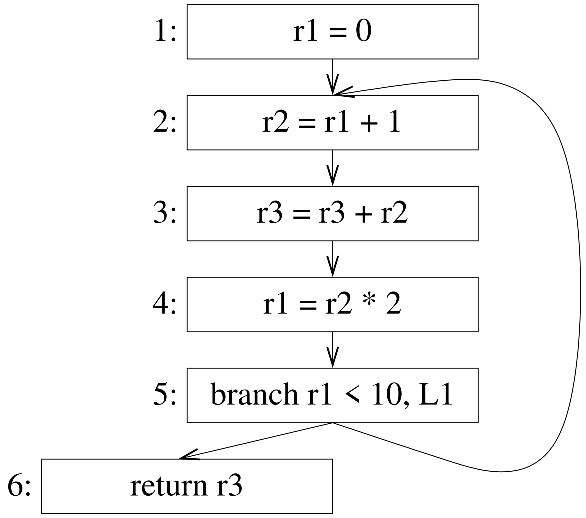
Live Variable Analysis Equation:

- Set definition ($A[n]$): $USE[n]$ - the set of registers that n uses.
- Set definition ($B[n]$): $DEF[n]$ - the set of registers that n defines.
- Transfer function ($f(A, B, OUT)$): $USE[n] \cup (OUT[n] - DEF[n])$
- Confluence operator ($\vee$): $\cup$
- Direction: REVERSE

$$OUT[n] = \cup_{s \in SUCC[n]} IN[s]$$

$$IN[n] = USE[n] \cup (OUT[n] - DEF[n])$$

Live Variable Analysis Example



Node	USE	DEF	OUT	IN	OUT	IN	OUT	IN
1								
2								
3								
4								
5								
6								

Live Variable Application 1: Register Allocation

Register Allocation:

1. Perform live variable analysis.
2. Build *interference graph*.
   - Variables interfere if they are live at the same time.
3. Color interference graph with real registers.
   - Choose non-interfering variables to place in the same concrete registers.

Interference Graph

- Node t corresponds to virtual register t.
- Edge $\langle t_i, t_j \rangle$ exists if registers t_i, t_j have overlapping live ranges.
- For some node n, if $DEF[n] = \{a\}$ and $OUT[n] = \{b_1, b_2, ...b_k\}$, then add interference edges: $\langle a, b_1 \rangle$, $\langle a, b_2 \rangle$, $\langle a, b_k \rangle$

Interference Graph For Example:

Node	DEF	OUT	IN
1	r1	r1,r3	r3
2	r2	r2,r3	r1,r3
3	r3	r2,r3	r2,r3
4	r1	r1,r3	r2,r3
5	-	r1, r3	r1,r3
6	-		r3

Virtual registers r1 and r2 may be mapped to same real registers.

Live Variable Application 2: Dead Code Elimination

- Given statement s with a definition and no side-effects:

  `r1 = r2 + r3, r1 = M[r2], or r1 = r2`

  If r1 is *not* live at the end of s, then the s is *dead*

- Dead statements can be deleted.

- Given statement s without a definition or side-effects:

  `r1 = call FUN_NAME, M[r1] = r2`

  Even if r1 is not live at the end of s, it is not dead.

Example:

```
r1 = r2 + 1
r2 = r2 + 2
r1 = r2 + 3
M[r1] = r2
```

Reaching Definition Analysis

Determines whether definition of register t directly affects use of t at some point in program.

Reaching Definition Definitions:

- Definition of d (of t) *reaches* statement u if a path of CFG edges exists from d to u that does not pass through a definition of t.
- Each definition of a variable will be associated with a *definition id* 1,2,3,4, etc.:

```
1: r1 = 5
2: r1 = 6
3,4:  r1,r2 = r2,r1
```

- In live variable analysis, we track variables
- In reaching definition analysis, we track definitions of variables

Reaching Definition Analysis

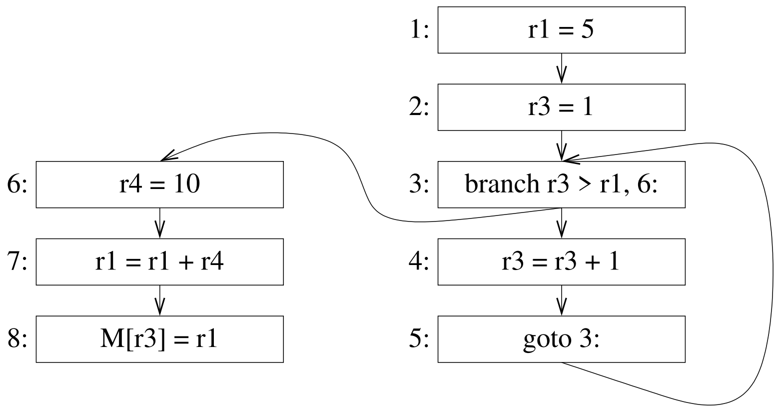
Reaching Definition Analysis Equation:

- Set definition ($A[n]$): $GEN[n]$ - the set of *definition id's* that n creates.
- Set definition ($B[n]$): $KILL[n]$ - the set of *definition id's* that n kills.
  - $defs(t)$ - set of all *definition id's* of register t.
- Transfer function ($f(A, B, IN)$): $GEN[n] \cup (IN[n] - KILL[n])$
- Confluence operator ($\vee$): $\cup$
- Direction: FORWARD

$$IN[n] = \cup_{p \in PRED[n]} OUT[p]$$

$$OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$$

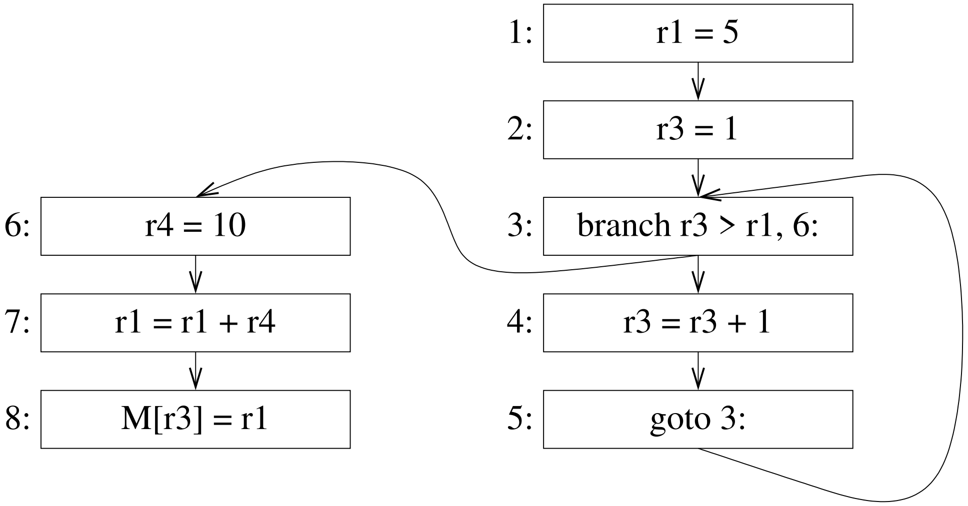
Reaching Definition Analysis Example



Node	GEN	$KILL$	IN	OUT	IN	OUT	IN	OUT
1								
2								
3								
4								
5								
6								
7								
8								

Reaching Definition Application 1: Constant Propagation

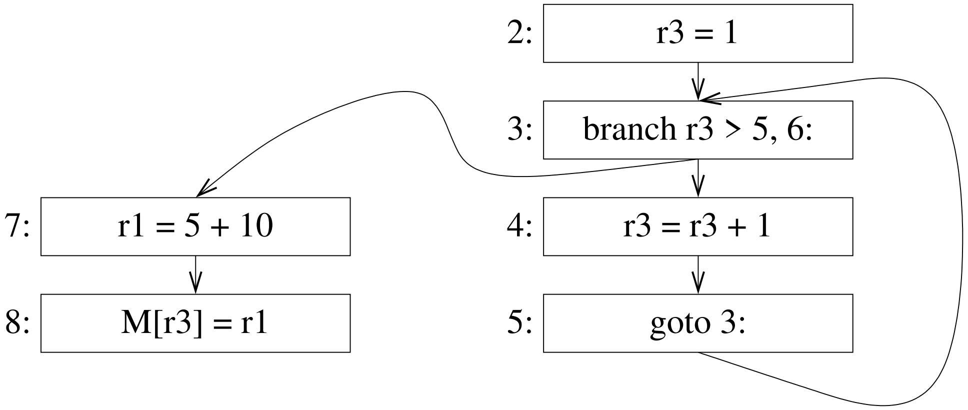
- Given Statement d: `a = c` where `a` is constant
- Given Statement u: `t = a op b`
- If statement d reach u and no other definition of `a` reaches u, then replace u by `t = c op b`.



Statements 1 and 6 are dead.

Constant Folding

- Given Statement d: `t = a op b`
- If `a` and `b` are constant, compute `c` as `a op b`, replace d by `t = c`



Common Subexpression Elimination

If `x op y` is computed multiple times, *common subexpression elimination* (CSE) attempts to eliminate some of the duplicate computations.

1: r1 = M[A]

2: r2 = r1 + 10

3: r3 = M[A]

4: r4 = r3 + 1

5: r5 = r4 + r2

Need to track expression propagation → available expression analysis

Definitions

Definitions:

- Expression `x op y` is *available* at CFG node n if, on every path from CFG entry node to n, `x op y` is computed at least once, and neither `x` nor `y` are defined since last occurrence of `x op y` on path.
- Can compute set of expressions available at each statement using system of dataflow equations.
- Statement `r1 = M[r2]`:
  - *generates* expression `M[r2]`.
  - *kills* all expressions containing `r1`.
- Statement `r1 = r2 + r3`:
  - *generates* expression `r2 + r3`.
  - *kills* all expressions containing `r1`.

Available Expression Analysis

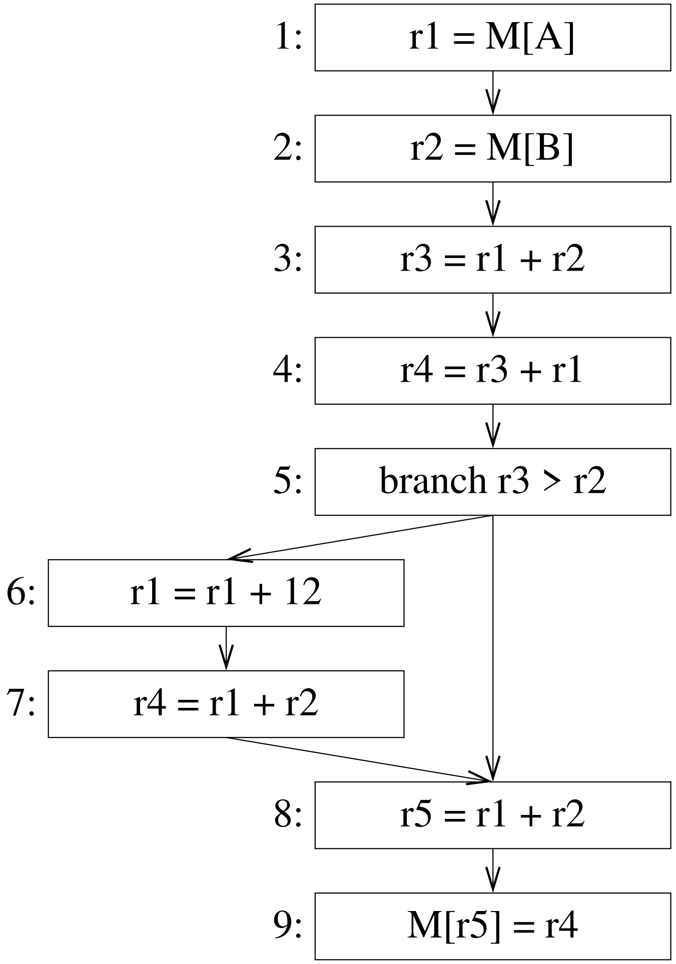
Available Expression Analysis:

- $exp(t)$ - set of all expressions containing t.
- Set definition ($A[n]$): $GEN[n]$ - the set of all expressions generated by n.
- Set definition ($B[n]$): $KILL[n]$ - the set of all expressions that n kills - $exp(n)$.
- Transfer function ($f(A, B, IN/OUT)$): $GEN[n] \cup (IN[n] - KILL[n])$
- Confluence operator ($\vee$): $\cap$
  - Use of $\cup$, required initialization of IN and OUT sets to $\emptyset$.
  - Use of $\cap$, requires initialization of IN and OUT sets to U (except for IN of entry node).
- Direction: FORWARD

$$IN[n] = \cap_{p \in PRED[n]} OUT[p]$$

$$OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$$

Available Expression Analysis Example

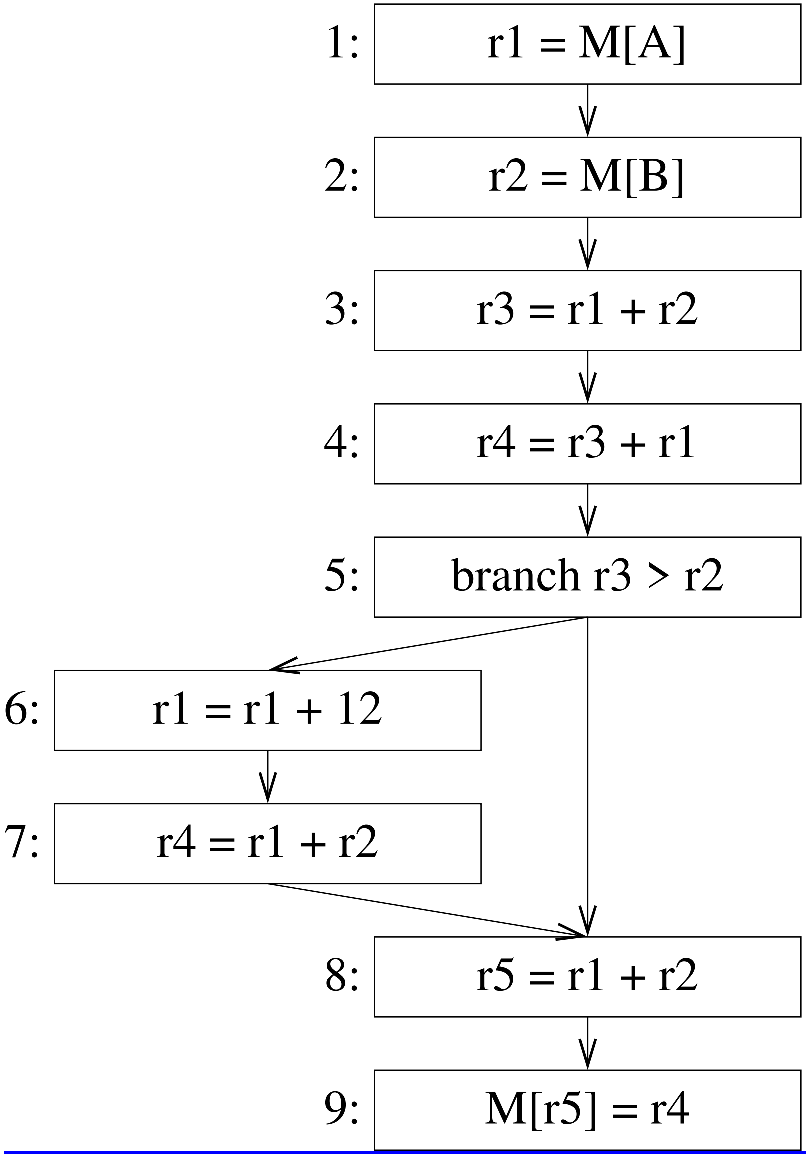


Common Subexpression Elimination

Given statement s: `t = x op y`:
If expression `x op y` is available at beginning of node s then:

1. starting from node s, traverse CFG edges backwards to find last occurrence of
   `x op y` on each path from entry node to s.
2. create new temporary `w`.
3. for each statement s': `v = x op y` found in (1), replace s' by:
   ```
   w = x op y
   v = w
   ```
4. replace statement s by: `t = w`

Common Subexpression Elimination - Example



`r1 + r2` **in node 8 is a common subexpression.**

The Phase Ordering Problem

- What order should we do optimizations in?
- Copy propagation, constant folding and constant propagation can create dead code
- Common sub-expression elimination can create opportunities for copy propagation.
- Inlining function bodies can create opportunities for all of the above.
- All of the above can make functions smaller and therefore more likely to be inlined ....