Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 16

- Code generation (2)

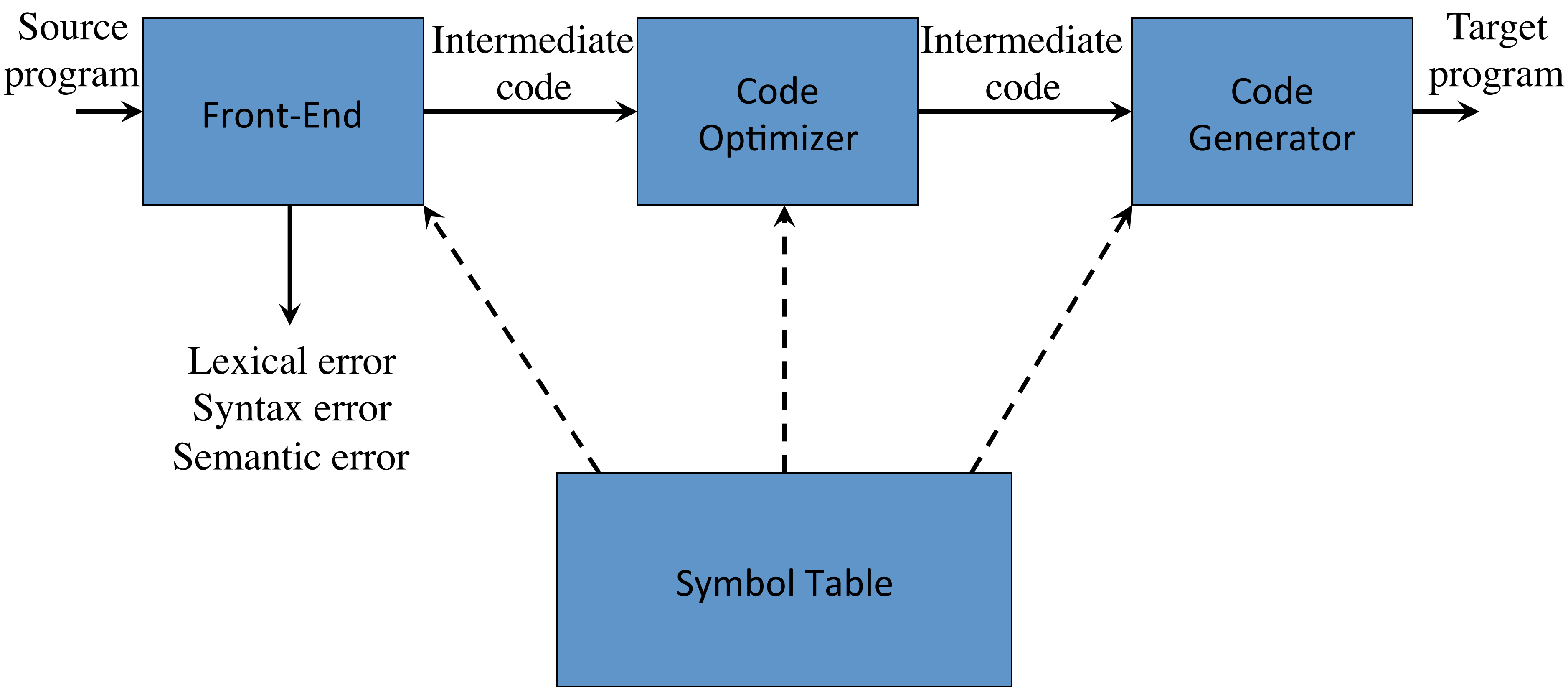
Recap (last lecture)

- Basics of Code Generation
- Code generation tasks:
  - Instruction selection
  - Register allocation and assigment
  - Instruction ordering
- Fixing Target Machine and Target Language
- Basic Blocks and Flow Graphs
- Local optimization: replacing basic blocks with equivalent ones

Summary

- Computing (local) “next use” and “live” info
- A Code Generator
  - Register allocation and assignment
    - Graph coloring
  - Instruction selection
    - Tree transducer
- An overview on Dataflow Analysis and some Global Optimization techniques

Position of a Code Generator in the Compiler Model



Transformations on Basic Blocks (recap)

- A *code-improving transformation* is a code optimization to improve speed or reduce code size
- *Global transformations* are performed across basic blocks
- *Local transformations* are only performed on single basic blocks
- We have seen several local optimization techniques:
  - Common subexpression elimination
  - Dead code elimination
  - Algebraic transformation, …
- To translate a simplified block we need additional info

(Local) Next-Use Information

- *Next-use information* is needed for dead-code elimination and register assignment
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement

  i: *x* := *y* op *z*

  – Add liveness/next-use info on *x*, *y*, and *z* to statement *I*
    - This info can be stored in the symbol table
  – Before going up to the previous statement (scan up):
    - Set *x* info to "not live" and "no next use"
    - Set *y* and *z* info to "live" and the next uses of *y* and *z* to *i*

Next-Use (Step 1)

i: **`b := b + 1`**

j: **`a := b + c`**

k: **`t := a + b`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true,
nextuse(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`t`**) = none]

Attach current live/next-use information
Because info is empty, assume variables are live
(Data flow analysis can provide accurate information)

Next-Use (Step 2)

i: **`b := b + 1`**

j: **`a := b + c`**

live(**`a`**) = true *nextuse*(**`a`**) = *k*
live(**`b`**) = true *nextuse*(**`b`**) = *k*
live(**`t`**) = false *nextuse*(**`t`**) = none

k: **`t := a + b`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true,
nextuse(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`t`**) = none]

Compute live/next-use information at *k*

Next-Use (Step 3)

i: **`b := b + 1`**

j: **`a := b + c`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`c`**) = true,
nextuse(**`a`**) = *k*, *nextuse*(**`b`**) = *k*, *nextuse*(**`c`**) = none]

k: **`t := a + b`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true,
nextuse(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`t`**) = none]

Attach current live/next-use information to *j*

Next-Use (Step 4)

live(**a**) = false	*nextuse*(**a**) = none
live(**b**) = true	*nextuse*(**b**) = *j*
live(**c**) = true	*nextuse*(**c**) = *j*
live(**t**) = false	*nextuse*(**t**) = none

i: `b := b + 1`

j: `a := b + c` [*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = false,
nextuse(**a**) = *k*, *nextuse*(**b**) = *k*, *nextuse*(**c**) = none]

k: `t := a + b` [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Compute live/next-use information *j*

Next-Use (Step 5)

i: **`b := b + 1`** [*live*(**`b`**) = true, *nextuse*(**`b`**) = *j*]

j: **`a := b + c`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`c`**) = false,
nextuse(**`a`**) = *k*, *nextuse*(**`b`**) = *k*, *nextuse*(**`c`**) = none]

k: **`t := a + b`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true,
nextuse(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`t`**) = none]

Attach current live/next-use information to *i*

A Simple Code Generator

- Algorithm for generating target code for a basic block (sequence of three-address statements) using next-use information
- Critical issue: how to use registers. Several competing uses:
  - To store operands of a target code operation
  - Registers make good temporaries
  - To hold (global) values computed in a block and used in another
  - To help runtime storage management (stack pointer, …)
- The algorithm will check if operands of three-address code are available in registers to avoid unnecessary stores and loads.

A Simple Code Generator (2)

- We assume that
  - A set of register can be used for values used within the block
  - The order of statements in the block is fixed
  - Each three-address operator corresponds to a single machine instruction
  - Machine instructions take operands in registers and leave the result in a register
- The algorithm makes use of *address* and *register descriptors*, and of function *getreg()* such that *getreg(x = y OP z)* returns the three registers to be used for x, y and z.

Register and Address Descriptors

- A *register descriptor RD* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.
- An *address descriptor AD* keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.
- Eg:

`LD R0,a` $RD(\mathbf{R0}) = \{\mathbf{a}\}$, $AD(\mathbf{a}) = AD(\mathbf{a}) \cup \{\mathbf{R0}\}$

`ST a,R0` $RD(\mathbf{R0}) = RD(\mathbf{R0}) \cup \{\mathbf{a}\}$, $AD(\mathbf{a}) = \{\mathbf{R0}\}$

The Code Generation Algorithm

For each statement $x := y$ op z

1. Use *getreg*(x := *y* OP *z*) to get registers *Rx*, *Ry* and *Rz*
2. If $Ry \notin AD(y)$ then emit **`LD`** *Ry*, *y'*
   where $y' \in AD(y)$, preferably a register
3. If $Rz \notin AD(z)$ then emit **`LD`** *Rz*, *z'*
   where $z' \in AD(z)$, preferably a register
4. Emit **`OP`** *Rx*, *Ry*, *Rz*
5. *Update the descriptors for the LD statements*
6. *RD(Rx) = {x}, AD(x) = {Rx}, remove Rx from other AD's*

The Code Generation Algorithm

For each copy statement *x :=y*

1. Use *getreg*(x := *y*) to get register *Ry* (= *Rx*)
2. If $Ry \notin AD(y)$ then emit **LD** *Ry*, *y'*
   where $y' \in AD(y)$, preferably a register
3. *Update the descriptors for operation LD*
4. $RD(Ry) = RD(Ry) \cup \{x\}$, $AD(x) = \{Ry\}$

At the end of the basic block

1. For each live variable x, if $x \notin AD(x)$
   emit **ST x,R**, where $R \in AD(x)$

Code Generation Example

Statements	*Code Generated*	*Register Descriptor*	*Address Descriptor*
		Registers empty	
`t := a - b`	**`LD R1,a`**	**`R0`** contains **`t`**	**`t`** in **`R0`**
	`LD R2,b`		
	`SUB R2,R1,R2`		
		`R0` contains **`t`** **`R1`** contains **`u`**	**`t`** in **`R0`** **`u`** in **`R1`**
`u := a - c`	**`LD R3,c`**		
	`SUB R1,R1,R3`		
`v := t + u`	**`ADD R3,R2,R1`**	**`R0`** contains **`v`** **`R1`** contains **`u`**	**`u`** in **`R1`** **`v`** in **`R0`**
`a := d`	**`LD R2,d`**		
`d := v + u`	**`ADD R1,R3,R1`**	**`R0`** contains **`d`**	**`d`** in **`R0`**
live(**`d`**)=true all other dead	**`ST d,R1`**		**`d`** in **`R0`** and memory

Example of code generation

R1	R2	R3	a	b	c	d	t	u	v
			a	b	c	d			

```
t = a - b
    LD R1, a
    LD R2, b
    SUB R2, R1, R2
```

R1	R2	R3	a	b	c	d	t	u	v
a	t		a, R1	b	c	d	R2		

```
u = a - c
    LD R3, c
    SUB R1, R1, R3
```

R1	R2	R3	a	b	c	d	t	u	v
u	t	c	a	b	c, R3	d	R2	R1	

```
v = t + u
    ADD R3, R2, R1
```

R1	R2	R3	a	b	c	d	t	u	v
u	t	v	a	b	c	d	R2	R1	R3

```
a = d
    LD R2, d
```

R1	R2	R3	a	b	c	d	t	u	v
u	a, d	v	R2	b	c	d, R2		R1	R3

```
d = v + u
    ADD R1, R3, R1
```

R1	R2	R3	a	b	c	d	t	u	v
d	a	v	R2	b	c	R1			R3

```
exit
    ST a, R2
    ST d, R1
```

R1	R2	R3	a	b	c	d	t	u	v
d	a	v	a, R2	b	c	d, R1			R3

The *getreg* algorithm

To compute *getreg*(x := *y* OP *z*)

1. If *y* is stored in a register *R*, return it as *Ry*
2. If *y* is not in a register, but exists *R* empty, return it as *Ry*
3. If y is not in a register and no register is empty, consider *R* and check any variable $v \in RD(R)$
   a. If *AD(v)* does not contain only *R*, OK.
   b. If *v* = *x* and *x* is not an operand in this instruction, OK.
   c. If *v* is not used later, then OK.
   d. Otherwise emit **`ST v,R`** this is a *spill*

   Choose *R* that minimizes the number of *spills* and return it as *Ry*
4. Same algorithm for determining *Rz*
5. For *Rx*, similar algorithm, but
   a. Any register containing *x* only is OK
   b. It is possible to return *Ry* for *Rx* if y is no more used and if *Ry* =*{y}*. Similarly for *Rz*.

To compute *getreg*(x := *y*)

1. Choose *Ry* as above
2. Choose *Rx* = *Ry*

Register Allocation and Assignment

- The code generation algorithm based on *getreg()* is not optimal
  - All live variables in registers are stored (flushed) at the end of a block: this could be not necessary
- *Global register allocation* assigns variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries
  - Keeping variables in registers in looping code can result in big savings

Allocating Registers in Loops: Usage Counts

- Suppose
  - not storing a variable *x* has a benefit of 2
  - accessing a variable in register instead of in memory has benefit 1
- Let
  - *use*(*x*, *B*) = number of uses of *x* in *B* before assignment
  - *live*(*x*, *B*) = *1* if *x* is assigned in *B* and live on exit from *B*
- Then the (approximate) *benefit* of allocating a register to a variable *x* within a loop *L* is

$$\sum_{B \in L} \left(use(x, B) + 2\, live(x, B) \right)$$

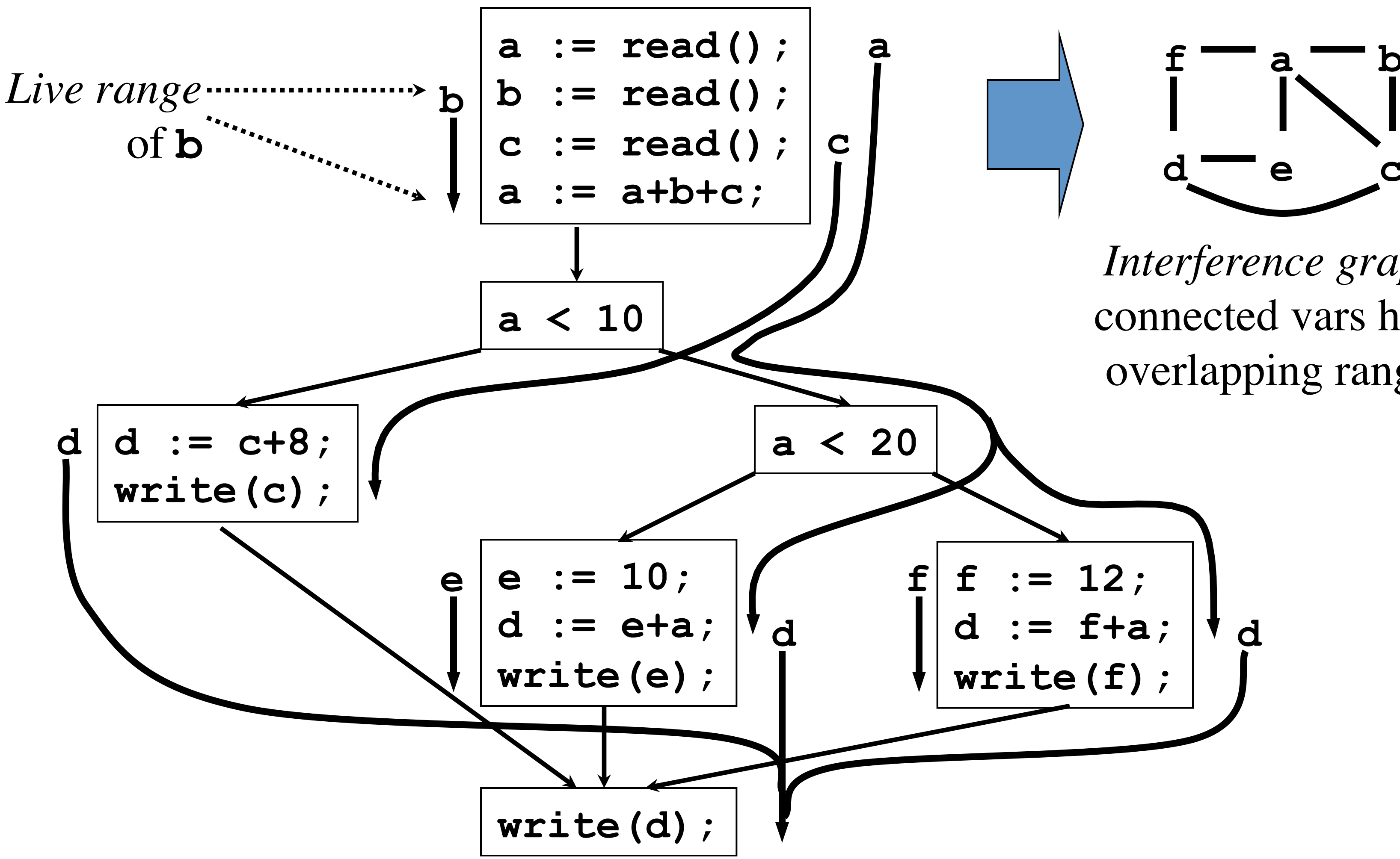
Global Register Allocation with Graph Coloring

- When a register is needed but all available registers are in use, the content of one of the used registers must be stored (*spilled*) to free a register
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a *conflict graph* (*interference graph*): two variables have an edge if one is live where the other is defined
- Find a *k*-coloring for the graph, with *k* the number of registers

Register Allocation with Graph Coloring: Example

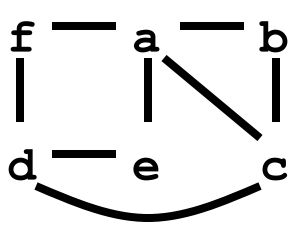
```
a := read();
b := read();
c := read();
a := a + b + c;
if (a < 10) {
    d := c + 8;
    write(c);
} else if (a < 20) {
    e := 10;
    d := e + a;
    write(e);
} else {
    f := 12;
    d := f + a;
    write(f);
}
write(d);
```

Register Allocation with Graph Coloring: Live Ranges



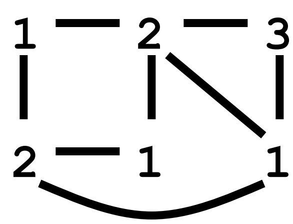
Interference graph: connected vars have overlapping ranges

Register Allocation with Graph Coloring: Solution



Interference graph

Solve



Three registers:

```
a = r2
b = r3
c = r1
d = r2
e = r1
f = r1
```

```
r2 := read();
r3 := read();
r1 := read();
r2 := r2 + r3 + r1;
if (r2 < 10) {
    r2 := r1 + 8;
    write(r1);
} else if (r2 < 20) {
    r1 := 10;
    r2 := r1 + r2;
    write(r1);
} else {
    r1 := 12;
    r2 := r1 + r2;
    write(r1);
}
write(r2);
```

Peephole Optimization

- Examines a short sequence of target instructions in a window (*peephole*) and replaces the instructions by a faster and/or shorter sequence when possible
- Applied to intermediate code or target code
- Typical optimizations:
  - Redundant instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms

Peephole Opt: Eliminating Redundant Loads and Stores

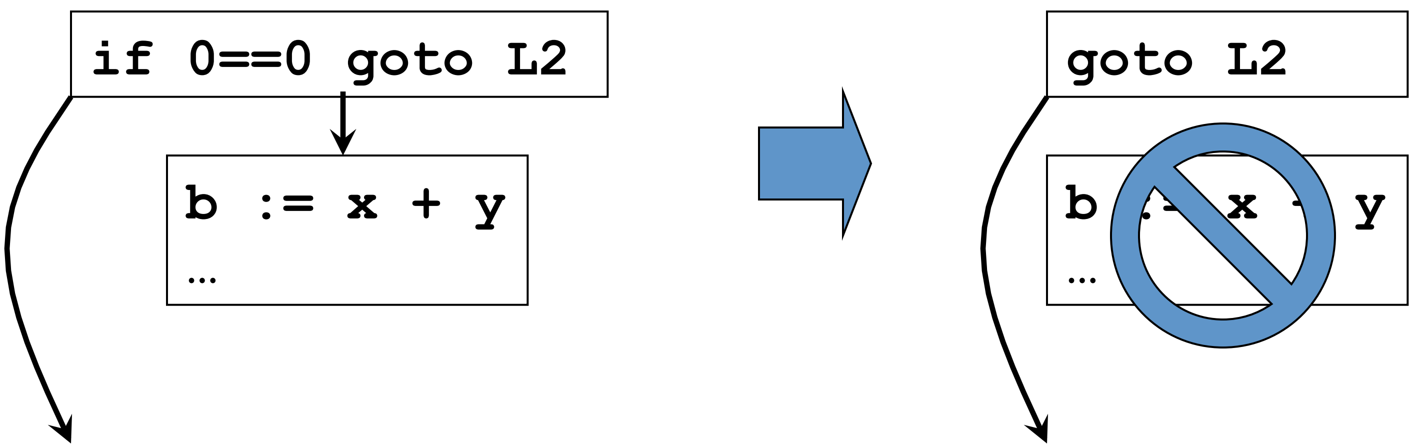
- Consider

```
MOV R0,a
MOV a,R0
```

- The second instruction can be deleted, but only if it is not labeled with a target label
  – Peephole represents sequence of instructions with at most one entry point
- The first instruction can also be deleted if *live*(**a**)=false

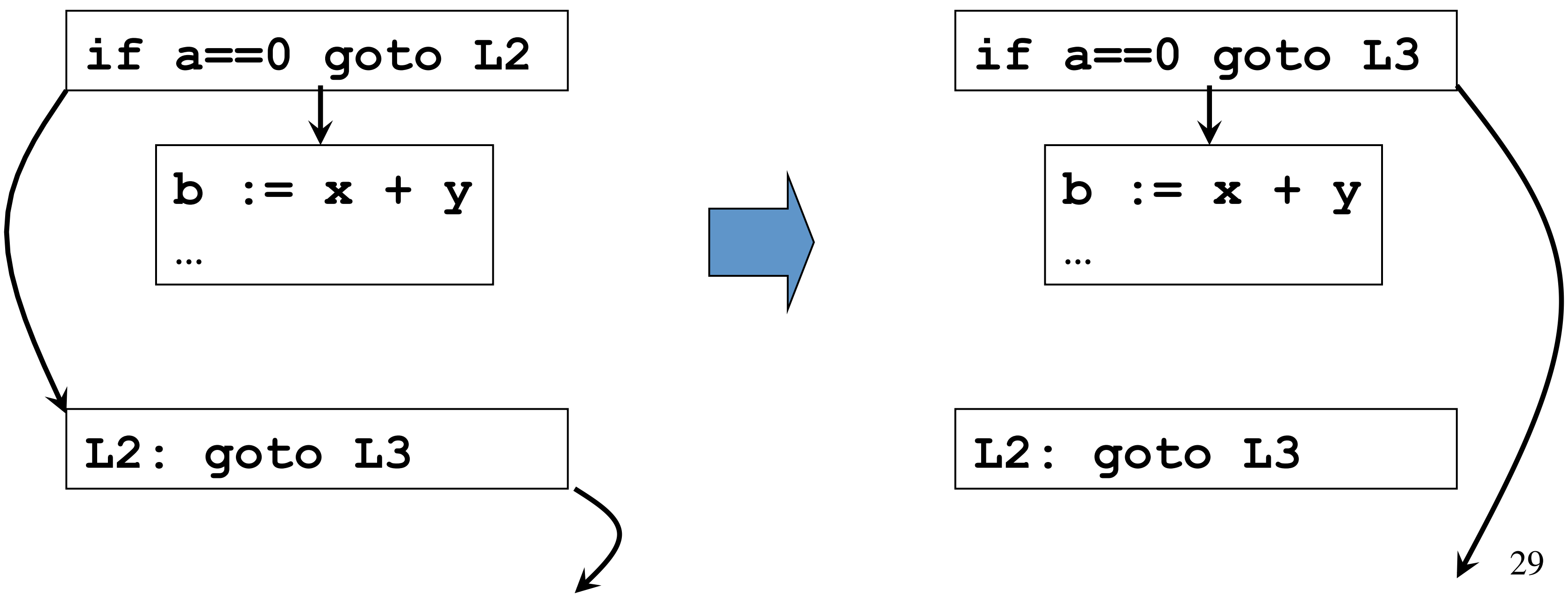
Peephole Optimization: Deleting Unreachable Code

- Unlabeled blocks can be removed



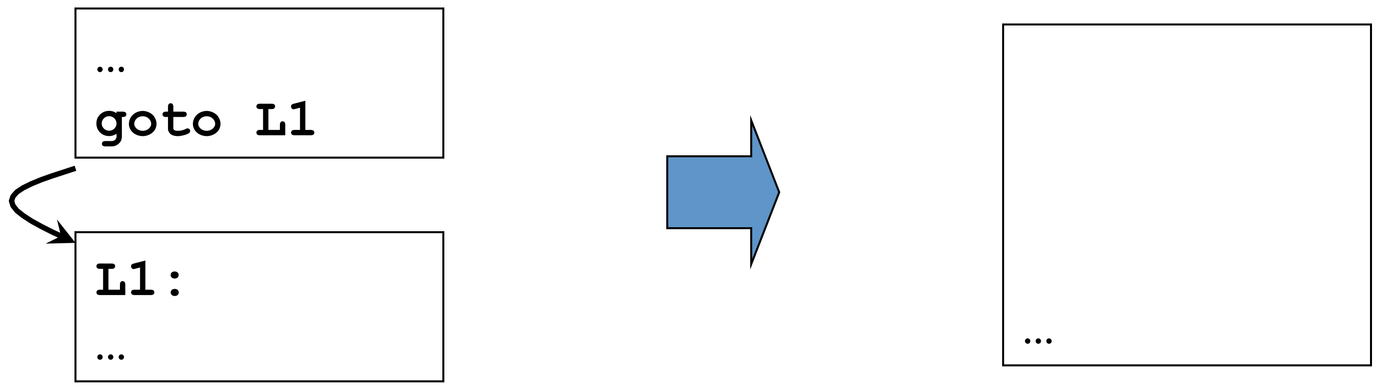
Peephole Optimization: Branch Chaining

- Shorten chain of branches by modifying target labels



Peephole Optimization: Other Flow-of-Control Optimizations

- Remove redundant jumps



Other Peephole Optimizations

- *Reduction in strength*: replace expensive arithmetic operations with cheaper ones

```
…
a := x ^ 2
b := y / 8
```

⇒

```
…
a := x * x
b := y >> 3
```

- Utilize machine idioms

```
…
a := a + 1
```

⇒

```
…
inc a
```

- Algebraic simplifications

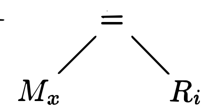
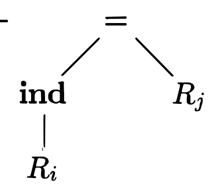
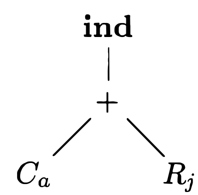
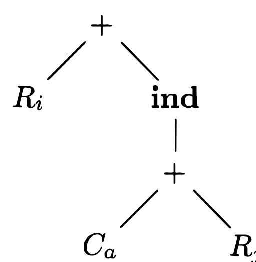
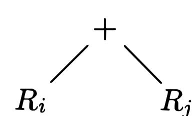
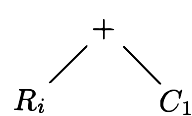
```
…
a := a + 0
b := b * 1
```

⇒

```
…
```

On Instruction Selection

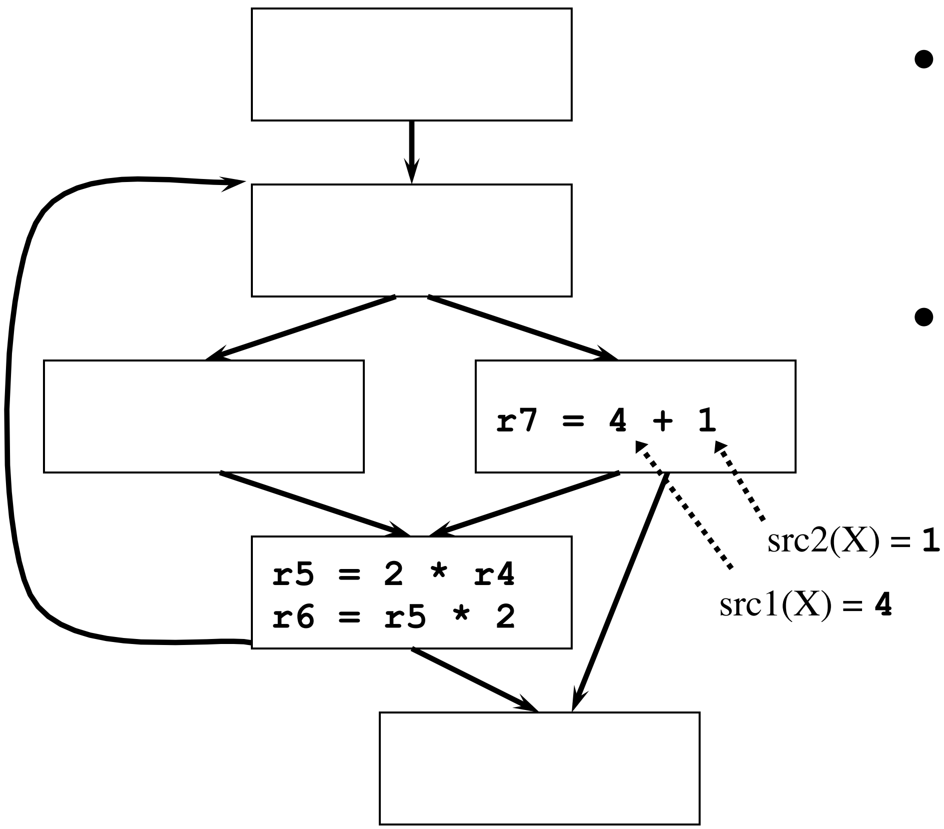
- Our simple algorithm uses a trivial Instruction Selection
- In practice it is a difficult problem, mainly for CISC machines with rich addressing mode
- Tree-rewriting rules can be used effectively for specifying the translation from IR to target code
- Tree-translation schemes can be handled with techniques similar to syntax-directed definitions: can be the basis of code generator generators
- Algorithms for pattern matching and general tree matching
- We can associate costs with the tree-rewriting rules and apply dynamic programming to obtain an optimal instruction selection

1)	$R_i \leftarrow C_a$	{ LD R*i*, #*a* }
2)	$R_i \leftarrow M_x$	{ LD R*i*, *x* }
3)	$M \leftarrow$ 	{ ST *x*, R*i* }
4)	$M \leftarrow$ 	{ ST *R*i*, R*j* }
5)	$R_i \leftarrow$ 	{ LD R*i*, *a*(R*j*) }
6)	$R_i \leftarrow$ 	{ ADD R*i*, R*i*, *a*(R*j*) }
7)	$R_i \leftarrow$ 	{ ADD R*i*, R*i*, R*j* }
8)	$R_i \leftarrow$ 	{ INC R*i* }

Classic Examples of Local and Global Code Optimizations

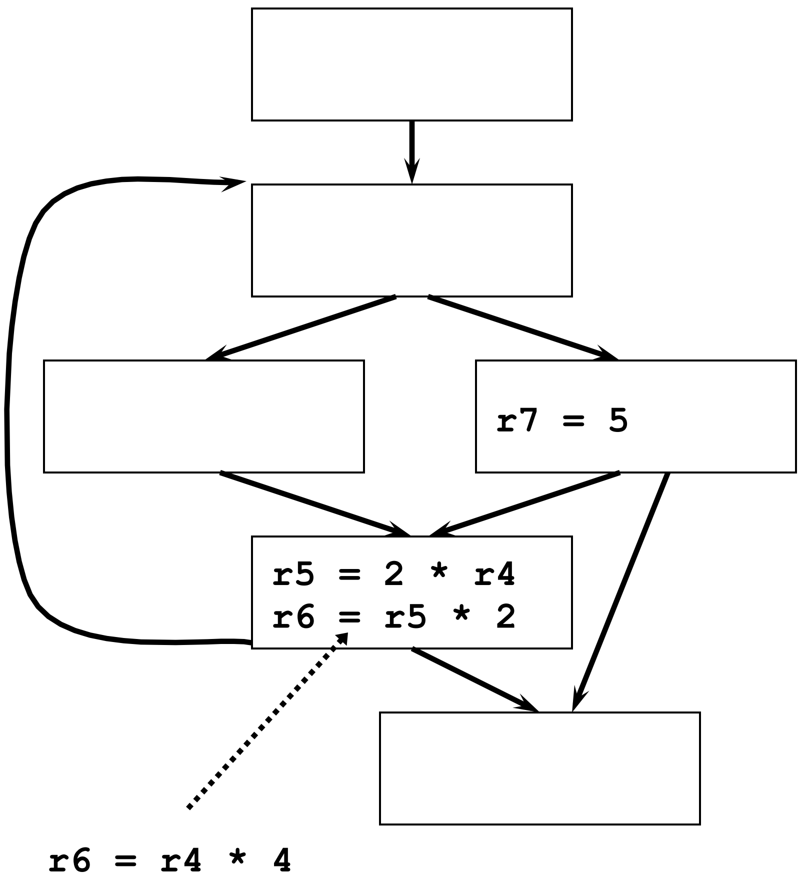
- Local
  - Constant folding
  - Constant combining
  - Strength reduction
  - Constant propagation
  - Common subexpression elimination
  - Backward copy propagation
- Global – based on data flow analysis
  - Dead code elimination
  - Constant propagation
  - Forward copy propagation
  - Common subexpression elimination
  - Code motion
  - Loop strength reduction
  - Induction variable elimination

Local: Constant Folding



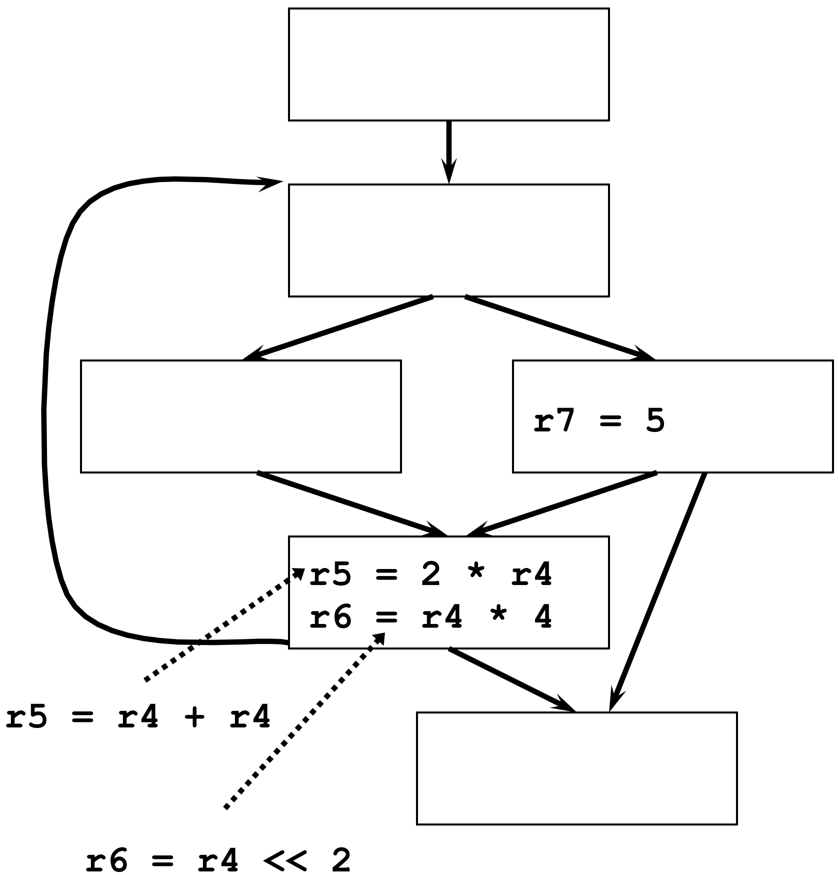
- Goal: eliminate unnecessary operations
- Rules:
  1. X is an arithmetic operation
  2. If src1(X) and src2(X) are constant, then change X by applying the operation

Local: Constant Combining



- Goal: eliminate unnecessary operations
  - First operation often becomes dead after constant combining
- Rules:
  1. Operations X and Y in same basic block
  2. X and Y have at least one literal src
  3. Y uses dest(X)
  4. None of the srcs of X have defs between X and Y (excluding Y)

Local: Strength Reduction



- Goal: replace expensive operations with cheaper ones
- Rules (common):
  1. X is an multiplication operation where src1(X) or src2(X) is a const 2^k integer literal
  2. Change X by using shift operation
  3. For k=1 can use add

Local: Constant Propagation

```
r1 = 5
r2 = _x
r3 = 7
r4 = r4 + r1    <····  r4 = r4 + 5
r1 = r1 + r2    <····  r1 = 5 + _x
r1 = r1 + 1     <····  r1 = 5 + _x + 1
r3 = 12
r8 = r1 - r2    <····  r8 = 5 + _x + 1 - _x
r9 = r3 + r5    <····  r9 = 12 + r5
r3 = r2 + 1     <····  r3 = _x + 1
r7 = r3 - r1    <····  r7 = _x + 1 - 5 - _x - 1
M[r7] = 0
```

- Goal: replace register uses with literals (constants) in a single basic block
- Rules:

1. Operation X is a move to register with src1(X) literal
2. Operation Y uses dest(X)
3. There is no def of dest(X) between X and Y (excluding defs at X and Y)
4. Replace dest(X) in Y with src1(X)

Local: Common Subexpression Elimination (CSE)

```
r1 = r2 + r3
r4 = r4 + 1
r1 = 6
r6 = r2 + r3
r2 = r1 - 1
r5 = r4 + 1
r7 = r2 + r3
r5 = r1 - 1   <···· r5 = r2
```

- Goal: eliminate re-computations of an expression
  - More efficient code
  - Resulting moves can get copy propagated (see later)
- Rules:
  1. Operations X and Y have the same opcode and Y follows X
  2. src(X) = src(Y) for all srcs
  3. For all srcs, no def of a src between X and Y (excluding Y)
  4. No def of dest(X) between X and Y (excluding X and Y)
  5. Replace Y with
     dest(Y) = dest(X)

Dataflow Analysis

- A data-flow analysis schema defines a value at each point in the program.
- Statements of the program have associated *transfer functions* that relate the value before the statement to the value after.
- Statements with more than one predecessor must have their value defined by combining the values at the predecessors, using a meet (or confluence) operator.
- Often *basic blocks* are annotated instead of individual statements.
- Useful for annotating the code with info needed for local or global optimization.

Dataflow analysis for Reaching Definitions and Live Variables

- ***Reaching Definitions***: Each statement is associated with the set of of definitions that are active.
- The transfer function for a block kills definitions of variables that are redefined in the block and adds definitions of variables that occur in the block.
- The confluence operator is union.
- ***Live Variables***: computes the variables that are *live* (will be used before redefinition) at each point.
- Similar to *reaching definitions*, but the transfer function runs backward. A variable is *live* at the beginning of a block if it is either used before definition in the block or is live at the end of the block and not redefined in the block.

Local: Backward Copy Propagation

```
r1 = r8 + r9
r2 = r9 + r1
r4 = r2
r6 = r2 + 1
r9 = r1
r7 = r6
r5 = r7 + 1
r4 = 0
r8 = r2 + r7
```

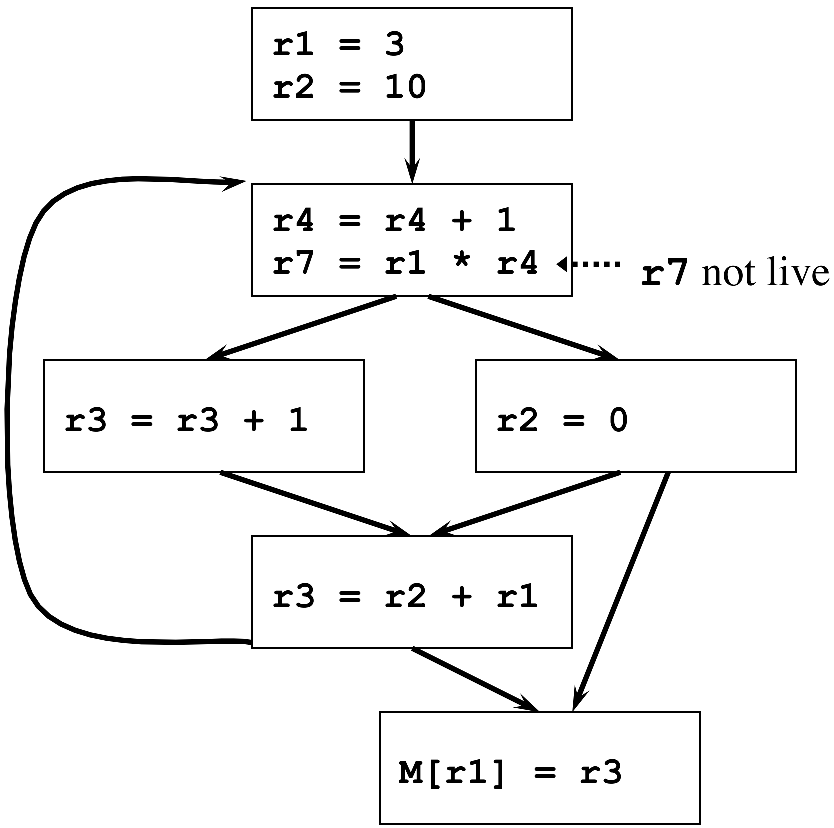
← `r7 = r2 + 1` (at `r6 = r2 + 1`)

← remove `r7 = r6`

r6 not live

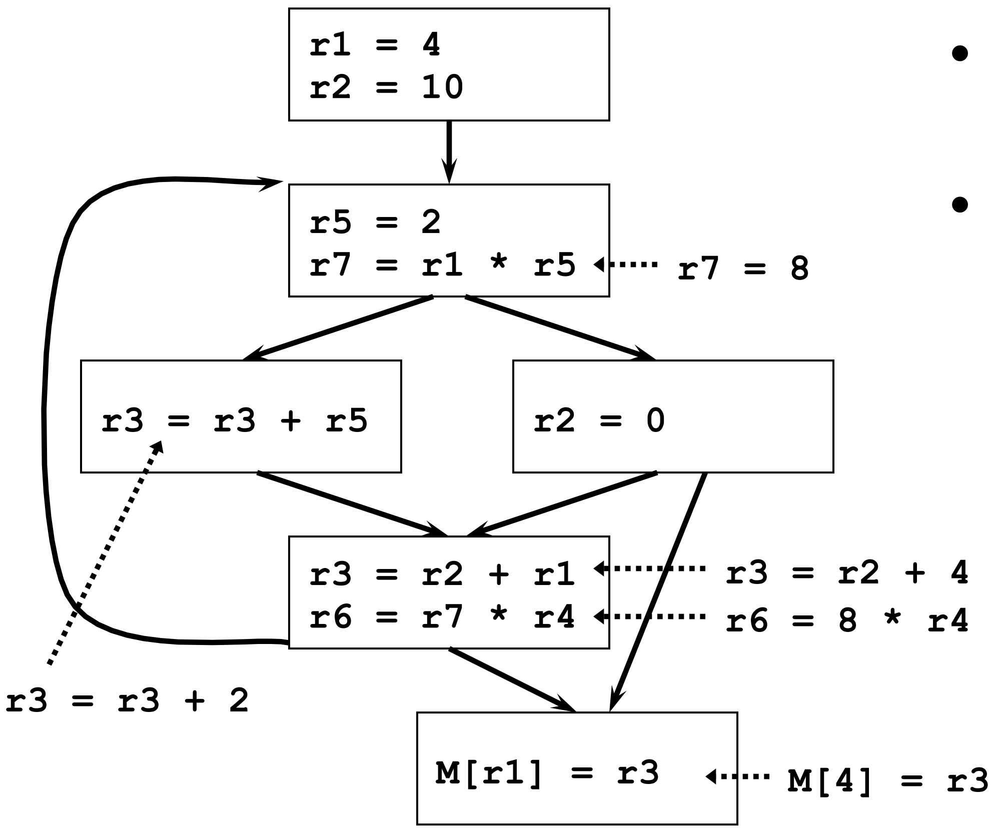
- Goal: propagate LHS of moves backward
  – Eliminates useless moves
- Rules (dataflow required)
  1. X and Y in same block
  2. Y is a move to register
  3. dest(X) is a register that is not live out of the block
  4. Y uses dest(X)
  5. dest(Y) not used or defined between X and Y (excluding X and Y)
  6. No uses of dest(X) after the first redef of dest(Y)
  7. Replace src(Y) on path from X to Y with dest(X) and remove Y

Global: Dead Code Elimination



- Goal: eliminate any operation who's result is never used
- Rules (dataflow required)
  1. X is an operation with dest(X) not live
  2. Delete X if removable (not a store or branch)
- Rules too simple!
  - Misses deletion of **r4**, even after deleting **r7**, since **r4** is live in loop

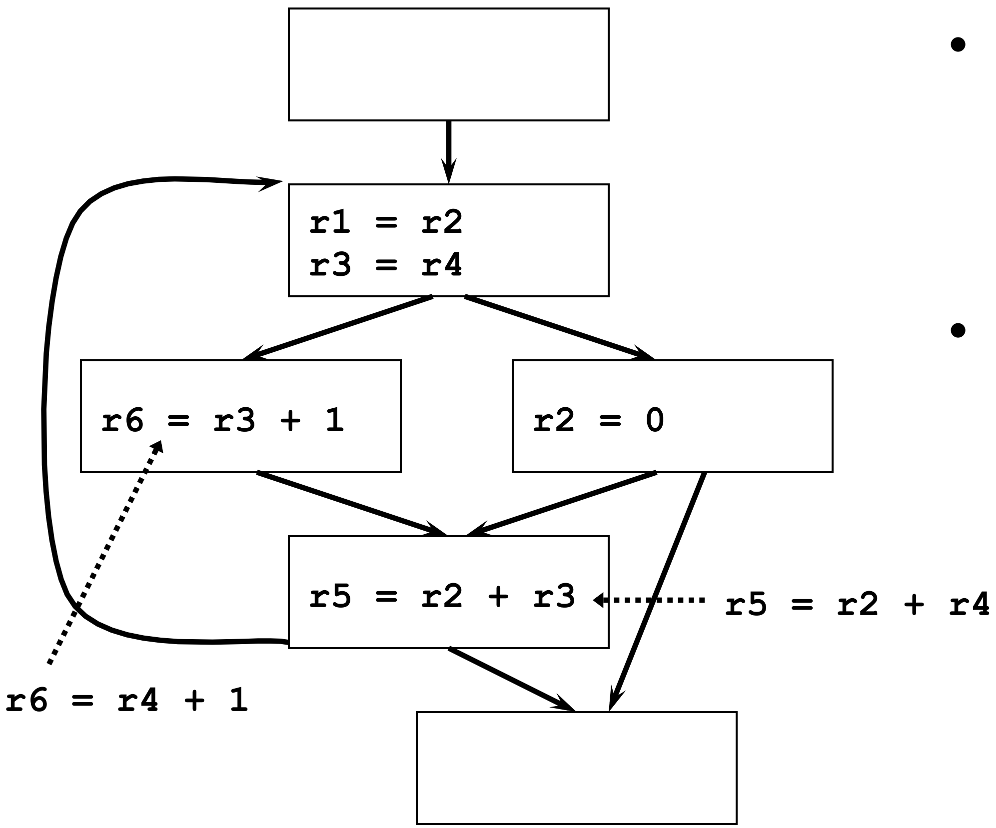
Global: Constant Propagation



- Goal: globally replace register uses with literals
- Rules (dataflow required)

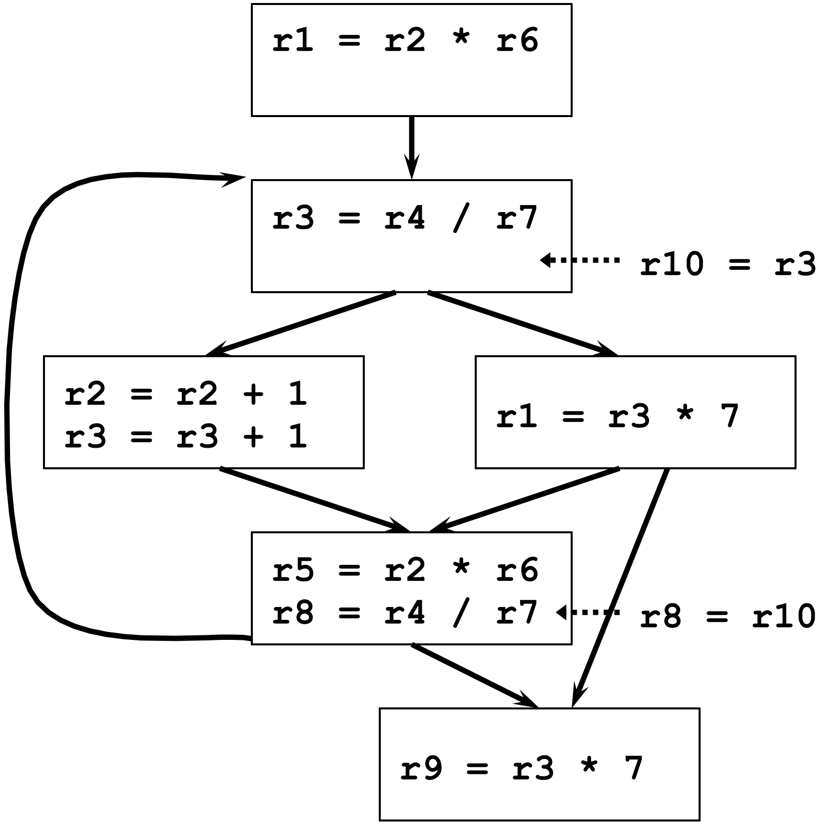
1. X is a load to a register with src1(X) literal
2. Y uses dest(X)
3. dest(X) has only one def at X for use-def (UD) chains to Y
4. Replace dest(X) in Y with src1(X)

Global: Forward Copy Propagation



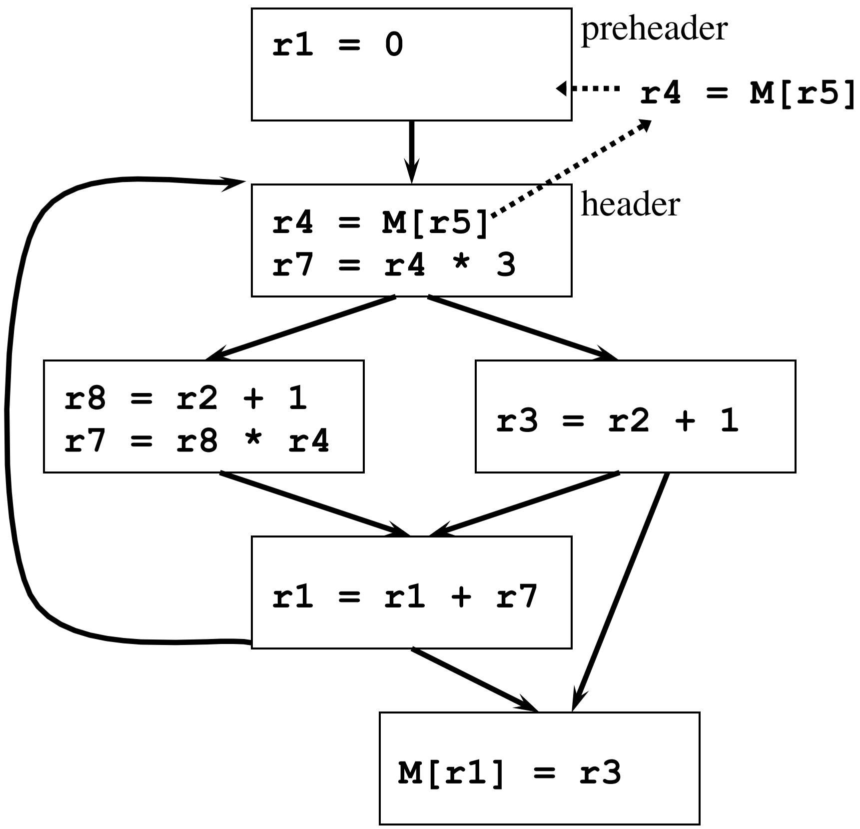
- Goal: globally propagate RHS of moves forward
  - Reduces dependence chain
  - May be possible to eliminate moves
- Rules (dataflow required)
  1. X is a move with src1(X) register
  2. Y uses dest(X)
  3. dest(X) has only one def at X for UD chains to Y
  4. src1(X) has no def on any path from X to Y
  5. Replace dest(X) in Y with src1(X)

Global: Common Subexpression Elimination (CSE)



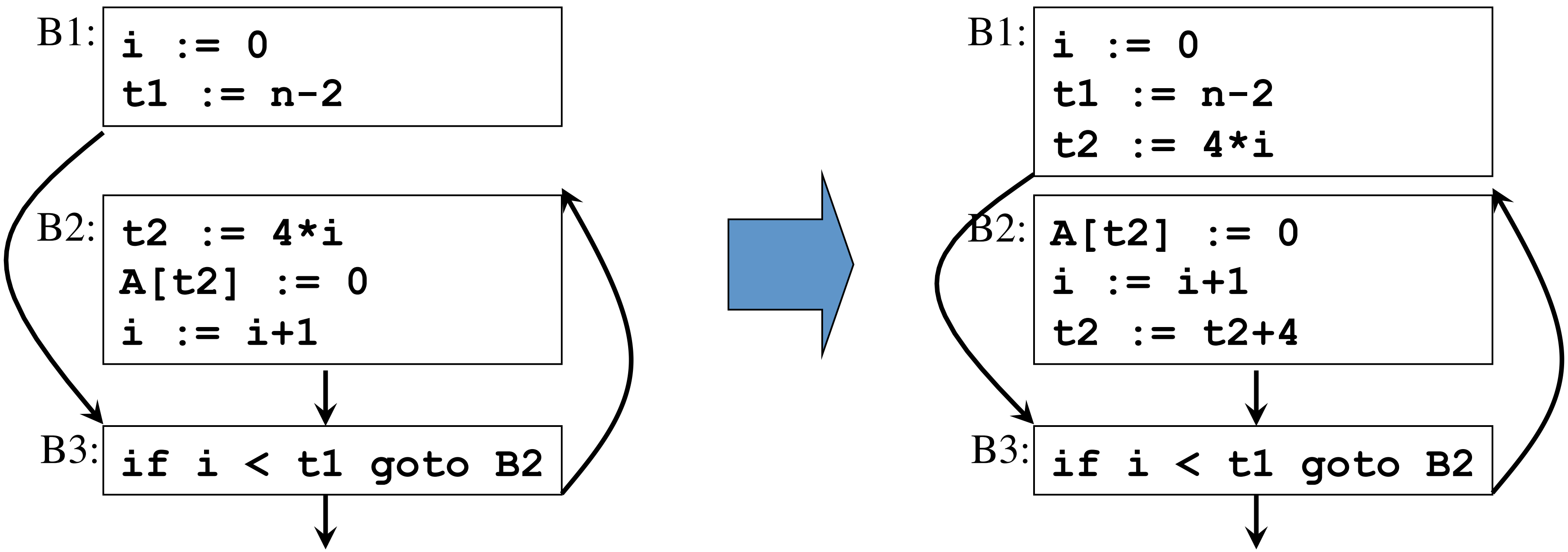
- Goal: eliminate recomputations of an expression
- Rules:
  1. X and Y have the same opcode and X dominates Y
  2. src(X) = src(Y) for all srcs
  3. For all srcs, no def of a src on any path between X and Y (excluding Y)
  4. Insert rx = dest(X) immediately after X for new register rx
  5. Replace Y with move dest(Y) = rx

Global: Code Motion



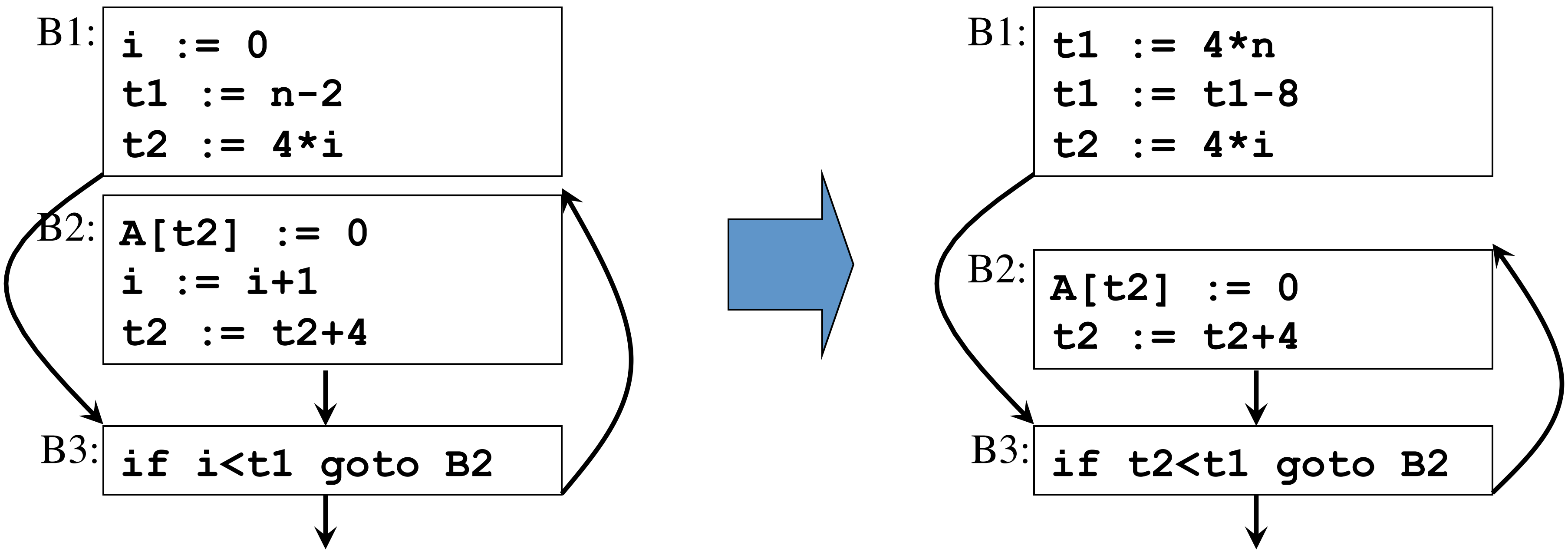
- Goal: move loop-invariant computations to preheader
- Rules:
  1. Operation X in block that dominates all exit blocks
  2. X is the only operation to modify dest(X) in loop body
  3. All srcs of X have no defs in any of the basic blocks in the loop body
  4. Move X to end of preheader
  5. Note 1: if one src of X is a memory load, need to check for stores in loop body
  6. Note 2: X must be movable and not cause exceptions

Global: Loop Strength Reduction



Replace expensive computations with *induction variables*

Global: Induction Variable Elimination



Replace induction variable in expressions with another

Generating Code for Stack Allocation of Activation Records

Three-address code	Target code	Comment
`t1 := a + b`	`100: ADD #16,SP`	Push frame
`param t1`	`108: MOV a,R0`	
`param c`	`116: ADD b,R0`	
`t2 := call foo,2`	`124: MOV R0,4(SP)`	Store a+b
…	`132: MOV c,8(SP)`	Store c
	`140: MOV #156,*SP`	Store return address
	`148: GOTO 500`	Jump to foo
`func foo`	`156: MOV 12(SP),R0`	Get return value
…	`164: SUB #16,SP`	Remove frame
`return t1`	`172: …`	
	`500: …`	
	`564: MOV R0,12(SP)`	Store return value
	`572: GOTO *SP`	Return to caller

Note: Language and machine dependent
Here we assume C-like implementation with SP and no FP