Tour of common optimizations

Simple example

```
foo(z) {
    x := 3 + 6;
    y := x - 5
    return z * y
}
```

Simple example

```
foo(z) {
    x := 3 + 6;          x:=9;
    y := x - 5;
    return z * y
}
```

Applying Constant Folding

Simple example

```
foo(z) {
   x := 9;
   y := x – 5;
   return z * y
}
```

```
y:= 9 - 5;
```

By Constant Propagation

Simple example

```
foo(z) {

   x := 9;

   y := 9 - 5;      y:=4;
   
   return z * y

}
```

Applying Constant Folding

Simple example

```
foo(z) {
   x := 9;
   y := 4;
   return z * y;        return z*4;
}
```

By Constant Propagation

Simple example

```
foo(z) {
    x := 9;
    y := 4;
    return z*4;
}
```

`return z << 2;` By Strenght Reduction

Simple example

```
foo(z) {
    x := 9;
    y := 4;
    return z << 2;
}
```

By Dead Assignment Elimination

Simple example

```
foo(z) {

   x := 3 + 6;

   y := x - 5

   return z * y

}
```

```
foo(z) {

   return z << 2;

}
```

Constant Folding
Constant Propagation
Strenght Reduction
Dead Assignment Elimination

Peephole optimizations

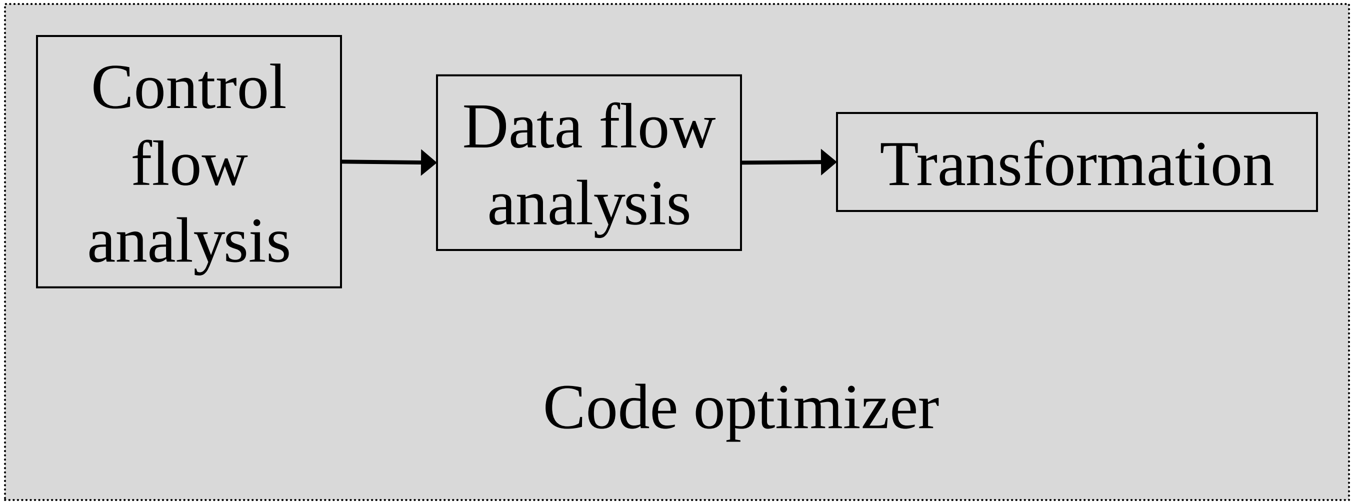
- Concerns with machine-independent code optimization
  - 90-10 rule: execution spends 90% time in 10% of the code.
  - It is moderately easy to achieve 90% optimization. The rest 10% is very difficult.
  - Identification of the 10% of the code is not possible for a compiler – it is the job of a profiler.
- In general, loops are the hot-spots

Introduction

- Criterion of code optimization
  - Must preserve the semantic equivalence of the programs
  - The algorithms should not be modified
  - Transformation, on average should speed up the execution of the program
  - Transformations should be simple enough to have a good effect

Introduction

- Organization of an optimizing compiler



Themes behind Optimization

- Avoid redundancy: something already computed need not be computed again
- Smaller code: less work for CPU, cache, and memory!
- Less jumps: jumps interfere with code pre-fetch
- Code locality: codes executed close together in time is generated close together in memory – increase locality of reference
- Extract more information about code: More info – better code generation

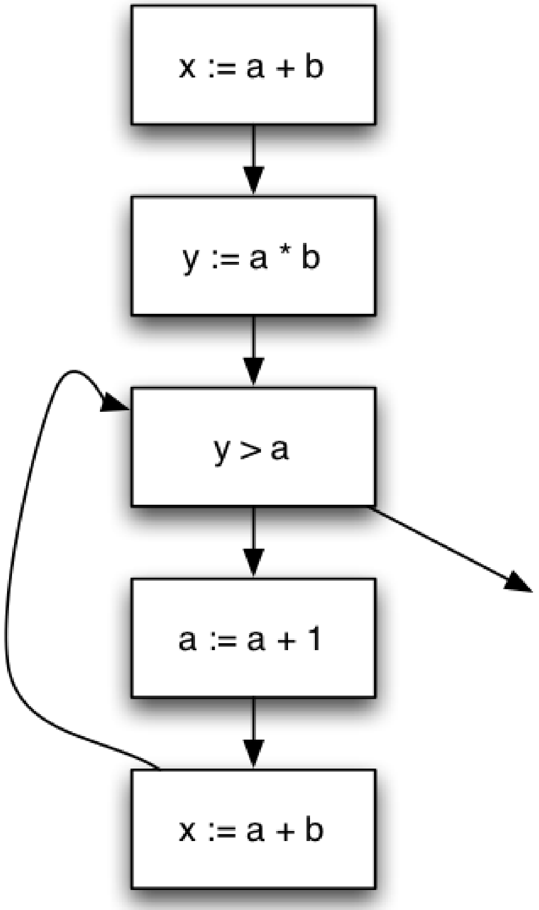
Control-Flow Graph (CFG)

- A directed graph where
  - Each node represents a statement
  - Edges represent control flow

- Three adress code: Statements may be
  - Assignments x = y op z or x = op z
  - Copy statements x = y
  - Branches goto L or if relop y goto L
  - etc

Control-flow Graph Example

```
x := a + b;
y := a * b
while (y > a){
   a := a +1;
   x := a + b
}
```



Variations on CFGs

- Usually don't include declarations (e.g. int x;).
- May want a unique entry and exit point.
- May group statements into *basic blocks.*
  - A *basic block* is a sequence of instructions with no branches into or out of the block.

Basic Blocks

A basic block is a sequence of *consecutive* intermediate language statements in which flow of control can only *enter at the beginning* and *leave at the end*.

Only the last statement of a basic block can be a branch statement and only the first statement of a basic block can be a target of a branch.

In some frameworks, procedure calls may occur within a basic block.

Basic Block Partitioning Algorithm

1. Identify leader statements (i.e. the first statements of basic blocks) by using the following rules:

(i) The *first statement* in the program is a leader

(ii) Any statement that is the *target of a branch* statement is a leader (for most intermediate languages these are statements with an associated label)

(iii) Any statement that *immediately follows a branch or return statement* is a leader

Example: Finding Leaders

The following code computes the inner product of two vectors.

```
begin
   prod := 0;
   i := 1;
   do begin
         prod := prod + a[i] * b[i];
         i = i+ 1;
   end
   while i <= 20
end
```

Source code

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)   t7 := i + 1
(11)   i := t7
(12)   if i <= 20 goto (3)
```

Three-address code

Example: Finding Leaders

The following code computes the inner product of two vectors.

```
begin
   prod := 0;
   i := 1;
   do begin
         prod := prod + a[i] * b[i];
         i = i+ 1;
   end
   while i <= 20
end
```

Source code

Rule (i)

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
(13)  …
```

Three-address code

Example: Finding Leaders

The following code computes the inner product of two vectors.

Rule (i)

Rule (ii)

```
begin
   prod := 0;
   i := 1;
   do begin
         prod := prod + a[i] * b[i];
         i = i+ 1;
   end
   while i <= 20
end
```

Source code

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)   t7 := i + 1
(11)   i := t7
(12)   if i <= 20 goto (3)
(13)   …
```

Three-address code

Example: Finding Leaders

The following code computes the inner product of two vectors.

```
begin
   prod := 0;
   i := 1;
   do begin
        prod := prod + a[i] * b[i];
        i = i+ 1;
   end
   while i <= 20
end
```

Source code

Rule (i)	(1)	prod := 0
	(2)	i := 1
Rule (ii)	(3)	t1 := 4 * i
	(4)	t2 := a[t1]
	(5)	t3 := 4 * i
	(6)	t4 := b[t3]
	(7)	t5 := t2 * t4
	(8)	t6 := prod + t5
	(9)	prod := t6
	(10)	t7 := i + 1
	(11)	i := t7
	(12)	if i <= 20 goto (3)
Rule (iii)	(13)	…

Three-address code

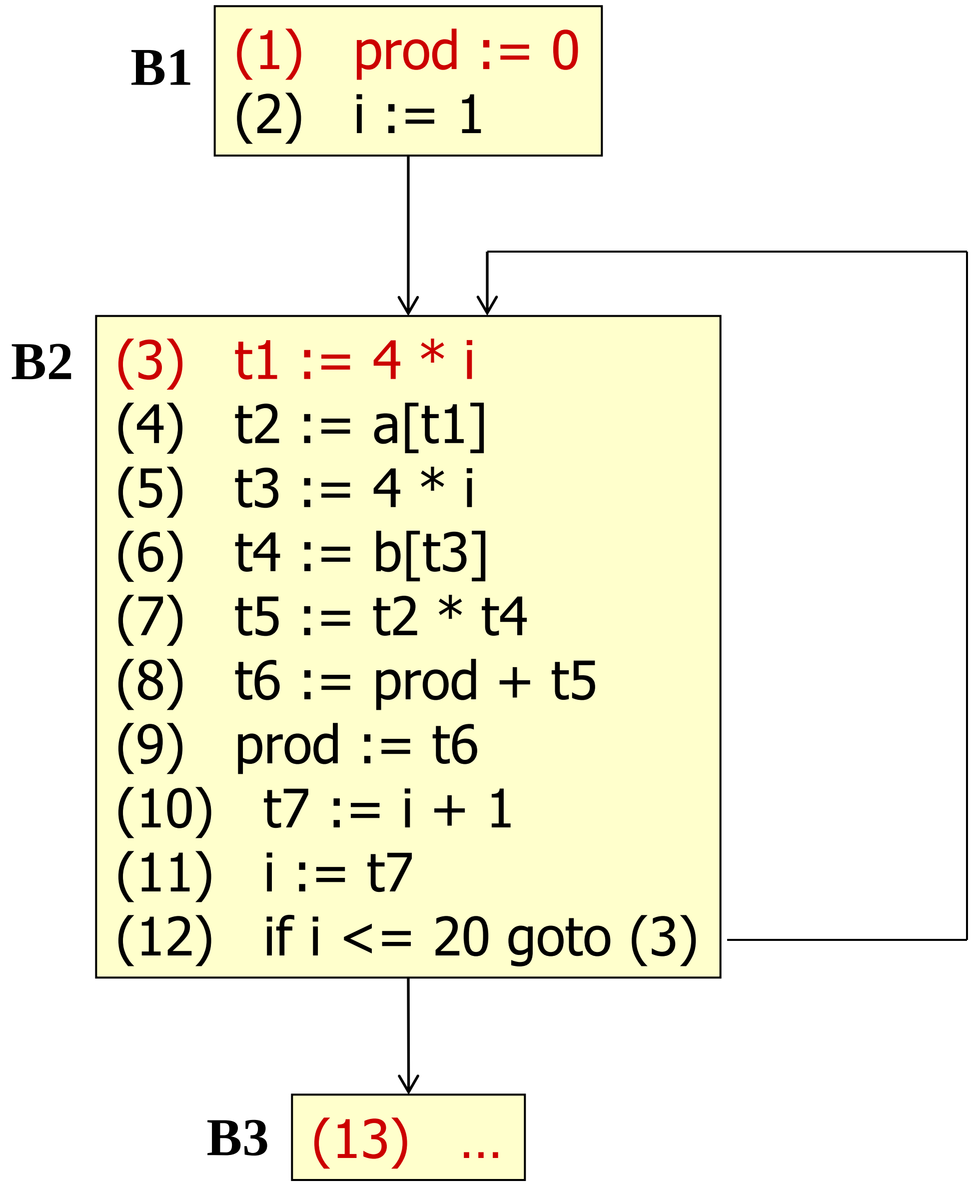
Forming the Basic Blocks

Now that we know the leaders, how do we form the basic blocks associated with each leader?

2. The basic block corresponding to a leader consists of the leader, plus all statements up to *but not including* the next leader or up to the end of the program.

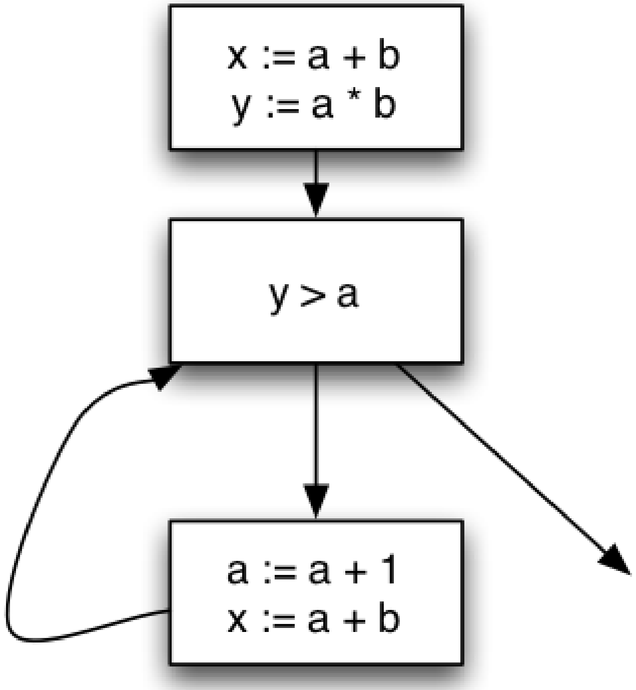
Example: Forming the Basic Blocks

Control flow diagram



Control-Flow Graph with Basic Blocks

```
x := a + b;
y := a * b
while (y > a){
   a := a +1;
   x := a + b
}
```



Can lead to more efficient implementations

Classical Optimization

- Types of classical optimizations
  - *Operation level*: one operation in isolation
  - *Local*: optimize pairs of operations in same basic block (with or without dataflow analysis)
  - *Global*: optimize pairs of operations spanning multiple basic blocks and must use dataflow analysis in this case, e.g. *reaching definitions, UD/DU chains,* or *SSA forms*
  - *Loop*: optimize loop body and nested loops

Redundancy elimination

- Redundancy elimination = determining that two computations are equivalent and eliminating one.
- There are several types of redundancy elimination:
  - Common subexpression elimination
    - Identifies expressions that have operands with the same name
  - Constant Folding and Constant/Copy propagation
    - Identifies variables that have constant/copy values and uses the constants/copies in place of the variables.

Compile-Time Evaluation

- **Constant folding**: Evaluation of an expression with constant operands to replace the expression with single value
- Example:

```
area := (22.0/7.0) * r ** 2
                ↓
area := 3.14286 * r ** 2
```

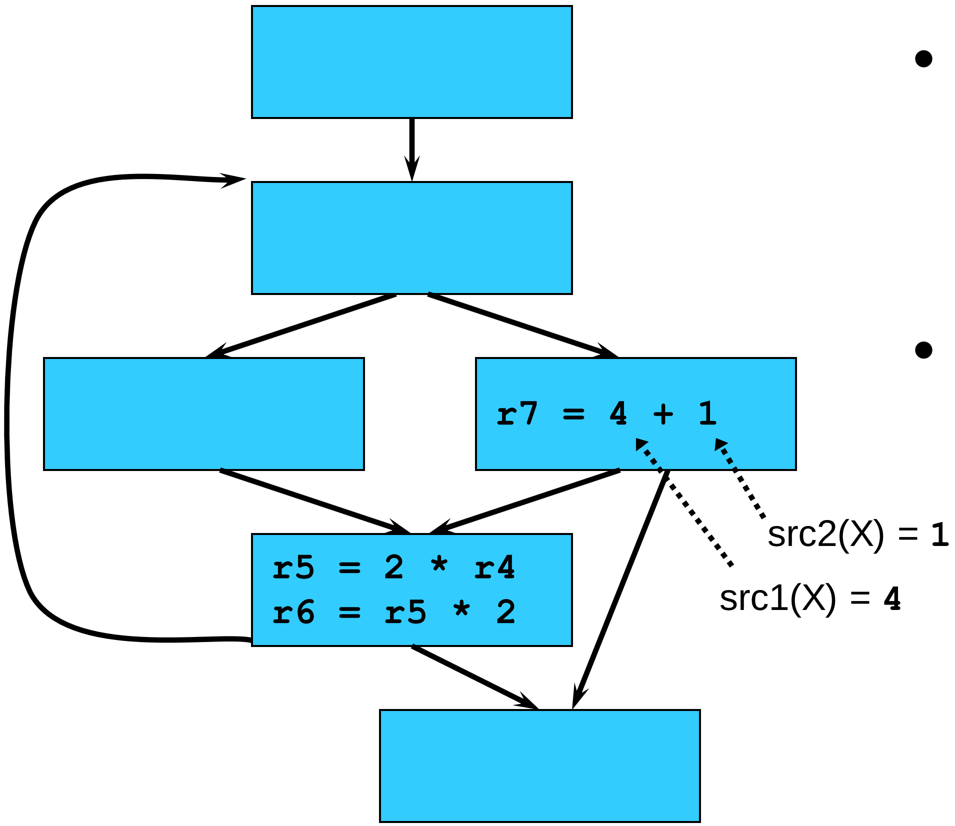
Compile-Time Evaluation

- **Constant Folding**: Replace a variable with constant which has been assigned to it earlier.
- Example:

```
pi := 3.14286
area = pi * r ** 2
```

→ `area = 3.14286 * r ** 2`

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

Constant Propagation

- What does it mean?
  - Given an assignment x = c, where c is a constant, replace later uses of x with uses of c, provided there are no intervening assignments to x.
    - Similar to copy propagation
    - Extra feature: It can analyze constant-value conditionals to determine whether a branch should be executed or not.
- When is it performed?
  - Early in the optimization process.
- What is the result?
  - Smaller code
  - Fewer registers

Common Sub-expression Evaluation

- Identify common sub-expression present in different expression, compute once, and use the result in all the places.
  – The *definition* of the variables involved should not change

Example:

```
a := b * c                 temp := b * c
…                          a := temp
…            ⟶            …
x := b * c + 5             x := temp + 5
```

Common Subexpression Elimination

- Local common subexpression elimination
  - Performed within basic blocks
  - Algorithm sketch:
    - Traverse BB from top to bottom
    - Maintain table of expressions evaluated so far
      - if any operand of the expression is redefined, remove it from the table
    - Modify applicable instructions as you go
      - generate temporary variable, store the expression in it and use the variable next time the expression is encountered.

```
x = a + b
...
y = a + b
```

```
t = a + b
x = t
...
y = t
```

Common Subexpression Elimination

Example

r2:= r1 * 5		r4:= r1*5
r2:= r2 + r3	may be transformed in	r2:= r4 + r3
r3:= r1 * 5		r3:= r4

Common Subexpression Elimination

```
c = a + b
d = m * n
e = b + d
f = a + b
g = - b
h = b + a
a = j + a
k = m * n
j = b + d
a = - b
if m * n go to L
```

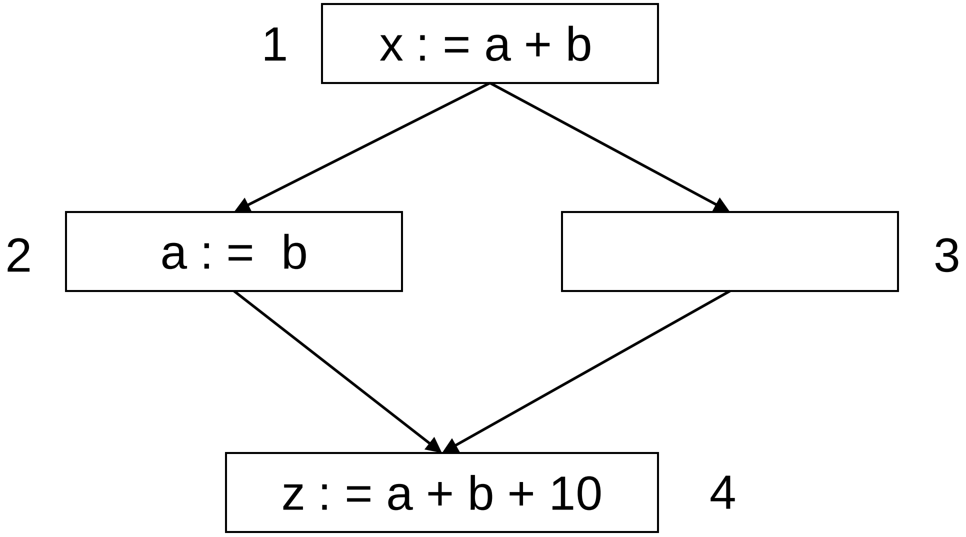
```
t1 = a + b
c = t1
t2 = m * n
d = t2
t3 = b + d
e = t3
f = t1
g = -b
h = t1 /* commutative */
a = j + a
k = t2
j = t3
a = -b
if t2 go to L
```

the table contains quintuples:
(pos, opd1, opr, opd2, tmp)

Common Subexpression Elimination

- Global common subexpression elimination
  - Performed on flow graph
  - Requires available expression information
    - In addition to finding what expressions are available at the endpoints of basic blocks, we need to know where each of those expressions was most recently evaluated (which block and which position within that block).

Common Sub-expression Evaluation



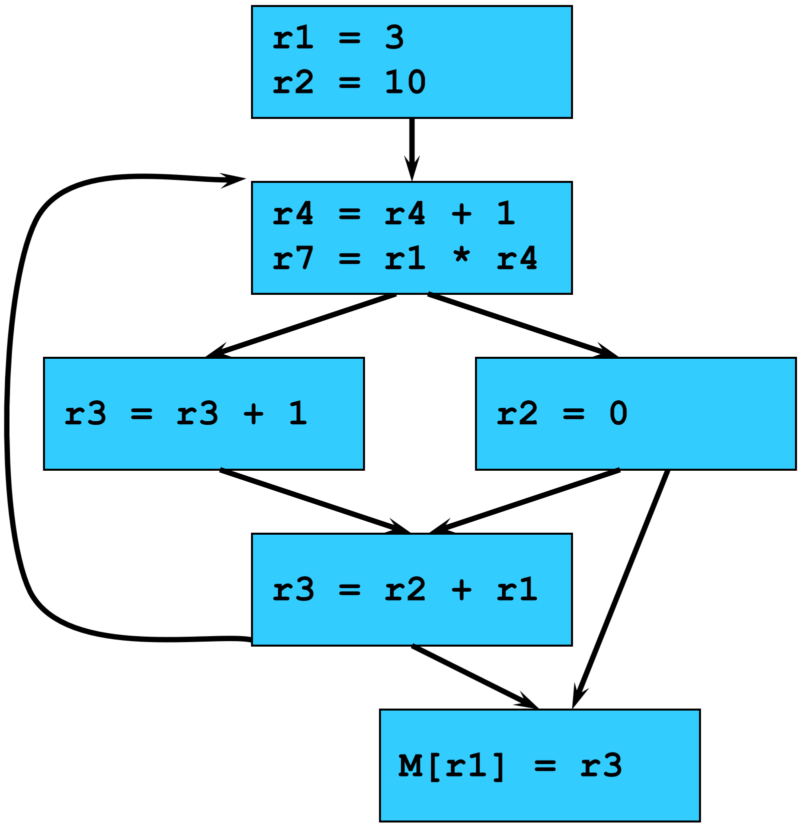
"a + b" is not a common sub-expression in 1 and 4

None of the variable involved should be modified in any path

Dead Code Elimination

- Dead Code are portion of the program which will not be executed in any path of the program.
  - Can be removed
- Examples:
  - No control flows into a basic block
  - A variable is dead at a point -> its value is not used anywhere in the program
  - An assignment is dead -> assignment assigns a value to a dead variable

Dead Code Elimination



- Goal: eliminate any operation who's result is never used
- Rules (dataflow required)
  1. X is an operation with no use in DU chain, i.e. dest(X) is not live
  2. Delete X if removable (not a mem store or branch)
- Rules too simple!
  - Misses deletion of **`r4`**, even after deleting **`r7`**, since **`r4`** is live in loop
  - Better is to trace UD chains backwards from "critical" operations

Dead Code Elimination

- Examples:

```
DEBUG:=0
if (DEBUG) print
```

Can be eliminated

Copy Propagation

- What does it mean?
  - Given an assignment x = y, replace later uses of x with uses of y, provided there are no intervening assignments to x or y.
- When is it performed?
  - At any level, but usually early in the optimization process.
- What is the result?
  - Smaller code

Copy Propagation

- `f := g` are called copy statements or copies
- Use of `g` for `f`, whenever possible after copy statement

Example:

x[i] = a;	x[i] = a;
sum = x[i] + a;	sum = a + a;

- May not appear to be code improvement, but opens up scope for other optimizations.

Local Copy Propagation

- <u>Local</u> copy propagation
  - Performed within basic blocks
  - Algorithm sketch:
    - traverse BB from top to bottom
    - maintain table of copies encountered so far
    - modify applicable instructions as you go

Copy Propagation

r2:= r1		r2:= r1		
r3:= r1 + r2	gets	r3:= r1+r1	gets	r3:= r1+r1
r2:= 5		r2:= 5		r2:= 5

By Copy Propagation

By Dead Assignment Elimination

Loop Optimization

- Decrease the number if instruction in the inner loop
- Even if we increase no of instructions in the outer loop
- Techniques:
  - Code motion
  - Induction variable elimination
  - Strength reduction

Optimization themes

- Don’t compute if you don’t have to
  - unused assignment elimination
- Compute at compile-time if possible
  - constant folding, loop unrolling, inlining
- Compute it as few times as possible
  - CSE, PRE, PDE, loop invariant code motion
- Compute it as cheaply as possible
  - strength reduction
- Enable other optimizations
  - constant and copy prop, pointer analysis
- Compute it with as little code space as possible
  - unreachable code elimination