Functions:
Calling Conventions + Frames

Finishing Up

Then,

Dataflow Analysis

The power function

```
int pow(int b, int e)
//@requires e >= 0;
{
  if (e == 0)
    return 1;
  else
    return b * pow(b, e-1);
}
```

```
pow(b,e):
  if (e == 0) then done else recurse
done:
  ret 1
recurse:
  t0 <- e - 1
  t1 <- pow(b, t0)
  t2 <- b * t1
  ret t2
```

Initial Translation with def/use

program	def	use
pow(b, e) :	b, e	
if $(e == 0)$ then done else recurse		e
done :		
ret 1		
recurse :		
$t_0 \leftarrow e - 1$	t_0	e
$t_1 \leftarrow$ pow(b, t_0)	t_1	b, t_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1
ret t_2		t_2

Calculating liveness

program	def	use	live-in
$\mathsf{pow}(b, e)$:	b, e		
if $(e == 0)$ then done else recurse		e	
done :			
ret 1			
recurse :			
$t_0 \leftarrow e - 1$	t_0	e	
$t_1 \leftarrow \mathsf{pow}(b, t_0)$	t_1	b, t_0	
$t_2 \leftarrow b * t_1$	t_2	b, t_1	
ret t_2		t_2	t_2

Calculating liveness

program	def	use	live-in
pow(b, e) :	b, e		
if $(e == 0)$ then done else recurse		e	
done :			
ret 1			
recurse :			
$t_0 \leftarrow e - 1$	t_0	e	
$t_1 \leftarrow$ pow(b, t_0)	t_1	b, t_0	
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
ret t_2		t_2	t_2

Calculating liveness

program	def	use	live-in
pow(b, e) :	b, e		
if $(e == 0)$ then done else recurse		e	
done :			
ret 1			
recurse :			
$t_0 \leftarrow e - 1$	t_0	e	
$t_1 \leftarrow$ pow(b, t_0)	t_1	b, t_0	b, t_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
ret t_2		t_2	t_2

Calculating liveness

program	def	use	live-in
pow(b, e) :	b, e		
if $(e == 0)$ then done else recurse		e	
done :			
ret 1			
recurse :			
$t_0 \leftarrow e - 1$	t_0	e	b, e
$t_1 \leftarrow$ pow(b, t_0)	t_1	b, t_0	b, t_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
ret t_2		t_2	t_2

Calculating liveness

program	def	use	live-in
$\mathsf{pow}(b, e)$:	b, e		
if $(e == 0)$ then done else recurse		e	b, e
done :			
ret 1			
recurse :			b, e
$t_0 \leftarrow e - 1$	t_0	e	b, e
$t_1 \leftarrow \mathsf{pow}(b, t_0)$	t_1	b, t_0	b, t_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
ret t_2		t_2	t_2

Next: Arguments & retval explicit

program	def	use	live-in
pow(b, e) :	b, e		
if $(e == 0)$ then done else recurse		e	b, e
done :			
ret 1			
recurse :			b, e
$t_0 \leftarrow e - 1$	t_0	e	b, e
$t_1 \leftarrow$ pow(b, t_0)	t_1	b, t_0	b, t_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
ret t_2		t_2	t_2

Making argument's Explicit

program	def	use
pow :	arg_1, arg_2	
$b \leftarrow arg_1$	b	arg_1
$e \leftarrow arg_2$	e	arg_2
if $(e == 0)$ then done else recurse		
done :		
$res_0 \leftarrow 1$	res_0	
ret		res_0
recurse :		
$t_0 \leftarrow e - 1$	t_0	e
$arg_2 \leftarrow t_0$	arg_2	t_0
$arg_1 \leftarrow b$	arg_1	b
call pow	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8$	arg_1, arg_2
$t_1 \leftarrow res_0$	t_1	res_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1
$res_0 \leftarrow t_2$	res_0	t_2
ret		res_0

Missing a def

Where are callee save regs?

Liveness

program	def	use	live-in
pow :	arg_1, arg_2		
$b \leftarrow arg_1$	b	arg_1	
$e \leftarrow arg_2$	e	arg_2	
if $(e == 0)$ then done else recurse			
done :			
$res_0 \leftarrow 1$	res_0		
ret		res_0	
recurse :			
$t_0 \leftarrow e - 1$	t_0	e	
$arg_2 \leftarrow t_0$	arg_2	t_0	
$arg_1 \leftarrow b$	arg_1	b	
call pow	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8$	arg_1, arg_2	
$t_1 \leftarrow res_0$	t_1	res_0	
$t_2 \leftarrow b * t_1$	t_2	b, t_1	
$res_0 \leftarrow t_2$	res_0	t_2	
ret		res_0	res_0

Liveness

program	def	use	live-in
pow :	arg_1, arg_2		
$b \leftarrow arg_1$	b	arg_1	arg_1, arg_2
$e \leftarrow arg_2$	e	arg_2	b, arg_2
if $(e == 0)$ then done else recurse			b, e
done :			
$res_0 \leftarrow 1$	res_0		
ret		res_0	res_0
recurse :			b, e
$t_0 \leftarrow e - 1$	t_0	e	b, e
$arg_2 \leftarrow t_0$	arg_2	t_0	b, t_0
$arg_1 \leftarrow b$	arg_1	b	b, arg_2
call pow	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8$	arg_1, arg_2	b, arg_1, arg_2
$t_1 \leftarrow res_0$	t_1	res_0	b, res_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
$res_0 \leftarrow t_2$	res_0	t_2	t_2
ret		res_0	res_0

Liveness

program	def	use	live-in	Live-out
pow :	arg_1, arg_2			$arg_1 arg_2$
$b \leftarrow arg_1$	b	arg_1	arg_1, arg_2	b, arg_2
$e \leftarrow arg_2$	e	arg_2	b, arg_2	b, e
if $(e == 0)$ then done else recurse			b, e	b, e
done :				
$res_0 \leftarrow 1$	res_0			res_0
ret		res_0	res_0	
recurse :			b, e	b, e
$t_0 \leftarrow e - 1$	t_0	e	b, e	b, t_0
$arg_2 \leftarrow t_0$	arg_2	t_0	b, t_0	b, arg_2
$arg_1 \leftarrow b$	arg_1	b	b, arg_2	b, arg_1, arg_2
call pow	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8$	arg_1, arg_2	b, arg_1, arg_2	b, res_0
$t_1 \leftarrow res_0$	t_1	res_0	b, res_0	b, t_1
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1	t_2
$res_0 \leftarrow t_2$	res_0	t_2	t_2	res_0
ret		res_0	res_0	-

Calculating Interference Graph

program	def	use	Live-out
pow :	arg_1, arg_2		$arg_1 arg_2$
$b \leftarrow arg_1$	b	arg_1	b, arg_2
$e \leftarrow arg_2$	e	arg_2	b, e
if $(e == 0)$ then done else recurse			b, e
done :			
$res_0 \leftarrow 1$	res_0		res_0
ret		res_0	
recurse :			b, e
$t_0 \leftarrow e - 1$	t_0	e	b, t_0
$arg_2 \leftarrow t_0$	arg_2	t_0	b, arg_2
$arg_1 \leftarrow b$	arg_1	b	b, arg_1, arg_2
call pow	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8$	arg_1, arg_2	b, res_0
$t_1 \leftarrow res_0$	t_1	res_0	b, t_1
$t_2 \leftarrow b * t_1$	t_2	b, t_1	t_2
$res_0 \leftarrow t_2$	res_0	t_2	res_0
ret		res_0	-

$b — arg_2$

$e — b$

$t_0 — b$

$arg_1 — b$

temp	interfering with
b	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8, e, t_0, t_1$
e	b
t_0	b
t_1	b
t_2	

Where to put b?

program	def	use	live-in
pow :	arg_1, arg_2		
$b \leftarrow arg_1$	b	arg_1	arg_1, arg_2
$e \leftarrow arg_2$	e	arg_2	b, arg_2
if $(e == 0)$ then done else recurse			b, e
done :			
$res_0 \leftarrow 1$	res_0		
ret		res_0	res_0
recurse :			b, e
$t_0 \leftarrow e - 1$	t_0	e	b, e
$arg_2 \leftarrow t_0$	arg_2	t_0	b, t_0
$arg_1 \leftarrow b$	arg_1	b	b, arg_2
call pow	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8$	arg_1, arg_2	b, arg_1, arg_2
$t_1 \leftarrow res_0$	t_1	res_0	b, res_0
$t_2 \leftarrow b * t_1$	t_2	b, t_1	b, t_1
$res_0 \leftarrow t_2$	res_0	t_2	t_2
ret		res_0	res_0

temp	interfering with
b	$res_0, arg_1, arg_2, arg_3, arg_4, arg_5, arg_6, ler_7, ler_8, e, t_0, t_1$
e	b
t_0	b
t_1	b
t_2	

Where to put b?

program	live-in
pow :	arg_1, arg_2, lee_9
push lee_9	arg_1, arg_2, lee_9
$b \leftarrow arg_1$	arg_1, arg_2
$e \leftarrow arg_2$	b, arg_2
if $(e == 0)$ then done else recurse	b, e
done :	
$res_0 \leftarrow 1$	
goto exitpow	res_0
recurse :	b, e
$t_0 \leftarrow e - 1$	b, e
$arg_2 \leftarrow t_0$	b, t_0
$arg_1 \leftarrow b$	b, arg_2
call pow	b, arg_1, arg_2
$t_1 \leftarrow res_0$	b, res_0
$t_2 \leftarrow b * t_1$	b, t_1
$res_0 \leftarrow t_2$	t_2
goto exitpow	res_0
exitpow :	res_0
pop lee_9	res_0
ret	lee_9, res_0

- We added epilog
- save and restore lee_9
- Make all returns goto epilog

Post coloring

pow :
  push lee_9
  $lee_9 \leftarrow arg_1$
  $res_0 \leftarrow arg_2$
  if $(res_0 == 0)$ then done else recurse
done :
  $res_0 \leftarrow 1$
  goto exitpow
recurse :
  $res_0 \leftarrow res_0 - 1$
  $arg_2 \leftarrow res_0$
  $arg_1 \leftarrow lee_9$ (redundant)
  call pow
  $res_0 \leftarrow res_0$ (redundant)
  $res_0 \leftarrow lee_9 * res_0$
  $res_0 \leftarrow res_0$ (redundant)
  goto exitpow
exitpow :
  pop lee_9
  ret

Final

pow :
  push lee_9
  $lee_9 \leftarrow arg_1$
  $res_0 \leftarrow arg_2$
  if $(res_0 == 0)$ then done else recurse
done :
  $res_0 \leftarrow 1$
  goto exitpow
recurse :
  $res_0 \leftarrow res_0 - 1$
  $arg_2 \leftarrow res_0$
  $arg_1 \leftarrow lee_9$
  call pow
  $res_0 \leftarrow res_0$
  $res_0 \leftarrow lee_9 * res_0$
  $res_0 \leftarrow res_0$
  goto exitpow
exitpow :
  pop lee_9
  ret

```
pow:    pushq   %rbx
        movl    %edi, %ebx
        movl    %esi, %eax
        cmpl    $0, %eax
        jne L1
        movl    $1, %eax
        goto L2
L1:     subl    $1, %eax
        movl    %eax, %esi
        call    pow
        imull   %ebx, %eax
L2:     popq %rbx
        ret
```

Dataflow Analysis

15-411/15-611 Compiler Design

Seth Copen Goldstein

February 19, 2026

Today

- Full Employment Theorem
- Local Optimization: Value Numbering
- Dataflow Analysis
  - reaching definitions
  - liveness
  - available expressions
  - very busy expressions
- Framework

Optimizations

- Register Allocation
- Common subexpression elimination
- Constant Propagation
- Copy propagation
- Dead-code elimination
- Loop optimizations
  - Hoisting
  - Induction variable elimination
- And, many many more.

How many?

Infinitely Many!

- Lets assume there existed
  optc: A perfect optimizing compiler
- optc(P) $\Rightarrow$ the smallest possible P' which has equivalent behavior to P.
- Then, ?

Infinitely Many!

- Lets assume there existed
  optc: A perfect optimizing compiler
- optc(P) $\Rightarrow$ the smallest possible P' which has equivalent behavior to P.
- Then, if P never halts, it should produce:

```
L1: jmp L1
```

- So, instead we build "optimizing compilers" and there is always a job for a compiler writer to invent new optimizations!

Approach to Optimization

- Three parts to any optimization:
  - Determine if an optimization is legal
  - Determine if an optimization is profitable
  - Implement optimization
- Consider also compilation time

Scope of Optimization

- Local
  Within a basic block
- Global
  Within a function, across basic blocks
- Interprocedural
  The entire program (file), across functions and basic blocks.
- Whole program
  Across all the files

Value Numbering

- (Originally) A local optimization for:
  - common sub-expression elimination
  - constant folding
  - constant propagation
  - Copy propagation
  - dead-code elimination
  - (algebraic simplification)
- A long history
- An ancestor to SSA

Local Value Numbering

- Goal: recognize that $X \oplus Y$ is redundant and eliminate redundant operation
- Idea: Assign a value number, V(exp), to each expression such that:

  $V(exp) = V(X \oplus Y)$ iff

  $X \oplus Y$ has same value as exp (on all paths)

Beware:

$a \leftarrow x + y$

$b \leftarrow x + y$

$x \leftarrow 5$

$c \leftarrow x + y$

Value Numbering Example

```
void quicksort(int m, int n) {
   int i = m-1;
   int j = n;
   int v;
   int x;
   if n <= m then return;
   v = a[n];
   while (true) {
        i = i+1;
        while (a[i] < v) i = i+1;
        j = j-1;
        while (a[j] > v) j = j-1;
        if  i>=j break;
        x = a[i]; a[i] = a[j]; a[j] = x
   }
   x = a[i]; a[i] = a[n]; a[n] = x;
   quicksort(m,j);
   quicksort(i+1), n);
}
```

Example: CSE

```
void quicksort(int m, int n) {
   int i = m-1;
   int j = n;
   int v;
   int x;
   if n <= m then return;
   v = a[n];
   while (true) {
       i = i+1;
       while (a[i] < v) i = i+1;
       j = j-1;
       while (a[j] > v) j = j-1;
       if  i>=j break;
       x = a[i]; a[i] = a[j]; a[j] = x
   }
   x = a[i]; a[i] = a[n]; a[n] = x;
   quicksort(m,j);
   quicksort(i+1), n);
}
```

```
B1: i      = m - 1
    j      = n
    t1     = 4*n
    v      = a[t1]

B2: i      = i + 1
    t2     = 4 * i
    t3     = a[t2]
    cjump t3<v  B2, B3

B3: j      = j - 1
    t4     = 4 * j
    t5     = a[t4]
    cjump t5>v  B3, B4

B4: cjump i>=j  B6, B5
```

Example: CSE

```
void quicksort(int m, int n) {
   int i = m-1;
   int j = n;
   int v;
   int x;
   if n <= m then return;
   v = a[n];
   while (true) {
       i = i+1;
       while (a[i] < v) i = i+1;
       j = j-1;
       while (a[j] > v) j = j-1;
       if  i>=j break;
       x = a[i]; a[i] = a[j]; a[j] = x
   }
   x = a[i]; a[i] = a[n]; a[n] = x;
   quicksort(m,j);
   quicksort(i+1), n);
}
```

```
B5: t6      = 4*i
    x       = a[t6]
    t7      = 4 * i
    t8      = 4 * j
    t9      = a[t8]
    a[t7]   = t9
    t10     = 4*j
    a[t10] = x
    jump    B2

B6: t11     = 4*i
    x       = a[t11]
    t12     = 4 * i
    t13     = 4 * n
    t14     = a[t13]
    a[t12] = t14
    t15     = 4*n
    a[t15] = x
```

Example: CSE

```
B1:    i      = m - 1
       j      = n
       t1     = 4*n
       v      = a[t1]

B2:    i      = i + 1
       t2     = 4 * i
       t3     = a[t2]
       cjump  t3<v   B2, B3


B3:    j      = j - 1
       t4     = 4 * j
       t5     = a[t4]
       cjump  t5>v   B3, B4

B4:    cjump  i >= j B6, B5
```

```
B5:    t6     = 4*i
       x      = a[t6]
       t7     = 4 * i
       t8     = 4 * j
       t9     = a[t8]
       a[t7]  = t9
       t10    = 4*j
       a[t10] = x
       jump   B2

B6:    t11    = 4*i
       x      = a[t11]
       t12    = 4 * i
       t13    = 4 * n
       t14    = a[t13]
       a[t12] = t14
       t15    = 4*n
       a[t15] = x
```

Local CSE

```
B1:   i      = m - 1
      j      = n
      t1     = 4*n
      v      = a[t1]

B2:   i      = i + 1
      t2     = 4 * i
      t3     = a[t2]
      cjump  t3<v   B2, B3


B3:   j      = j - 1
      t4     = 4 * j
      t5     = a[t4]
      cjump  t5>v   B3, B4

B4:   cjump  i >= j B6, B5
```

```
B5:   t6     = 4*i
      x      = a[t6]
      t7     = 4 * i
      t8     = 4 * j
      t9     = a[t8]
      a[t7]  = t9
      t10    = 4*j
      a[t10] = x
      jump   B2

B6:   t11    = 4*i
      x      = a[t11]
      t12    = 4 * i
      t13    = 4 * n
      t14    = a[t13]
      a[t12] = t14
      t15    = 4*n
      a[t15] = x
```

Local CSE

```
B1:     i       = m - 1
        j       = n
        t1      = 4*n
        v       = a[t1]

B2:     i       = i + 1
        t2      = 4 * i
        t3      = a[t2]
        cjump   t3<v    B2, B3


B3:     j       = j - 1
        t4      = 4 * j
        t5      = a[t4]
        cjump   t5>v    B3, B4

B4:     cjump   i >= j B6, B5
```

```
B5:     t6      = 4*i
        x       = a[t6]
        t7      = 4 * i
        t8      = 4 * j
        t9      = a[t8]
        a[t6]   = t9
        t10     = 4*j
        a[t8]   = x
        jump    B2

B6:     t11     = 4*i
        x       = a[t11]
        t12     = 4 * i
        t13     = 4 * n
        t14     = a[t13]
        a[t11]  = t14
        t15     = 4*n
        a[t13]  = x
```

Local CSE

```
B1:     i       = m - 1
        j       = n
        t1      = 4*n
        v       = a[t1]

B2:     i       = i + 1
        t2      = 4 * i
        t3      = a[t2]
        cjump   t3<v    B2, B3


B3:     j       = j - 1
        t4      = 4 * j
        t5      = a[t4]
        cjump   t5>v    B3, B4

B4:     cjump   i >= j B6, B5
```

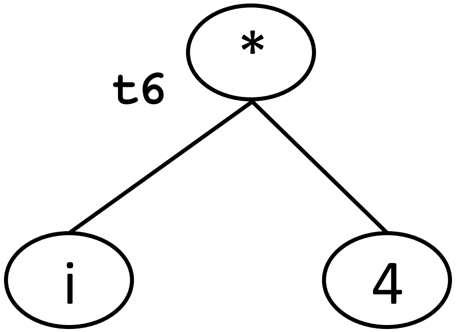
```
B5:     t6      = 4*i
        x       = a[t6]

        t8      = 4 * j
        t9      = a[t8]
        a[t6]   = t9

        a[t8]   = x
        jump    B2

B6:     t11     = 4*i
        x       = a[t11]

        t13     = 4 * n
        t14     = a[t13]
        a[t11] = t14

        a[t13] = x
```

How do we find this?

Build a DAG

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```

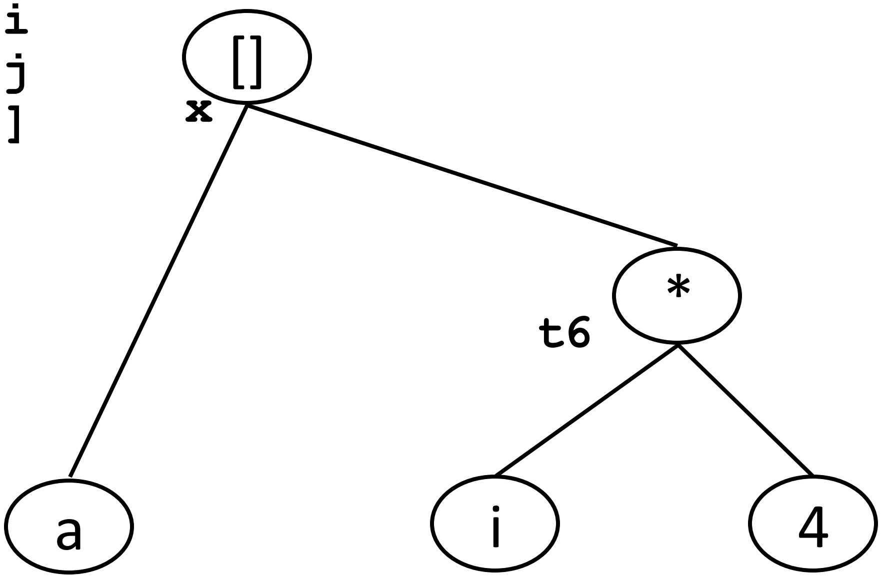
- For each var & constant not seen before create a leaf
- For each op, create a node and label with lvalue



Build a DAG

```
B5: t6      = 4*i
    x       = a[t6]
    t7      = 4 * i
    t8      = 4 * j
    t9      = a[t8]
    a[t7]   = t9
    t10     = 4*j
    a[t10] = x
    jump    B2
```

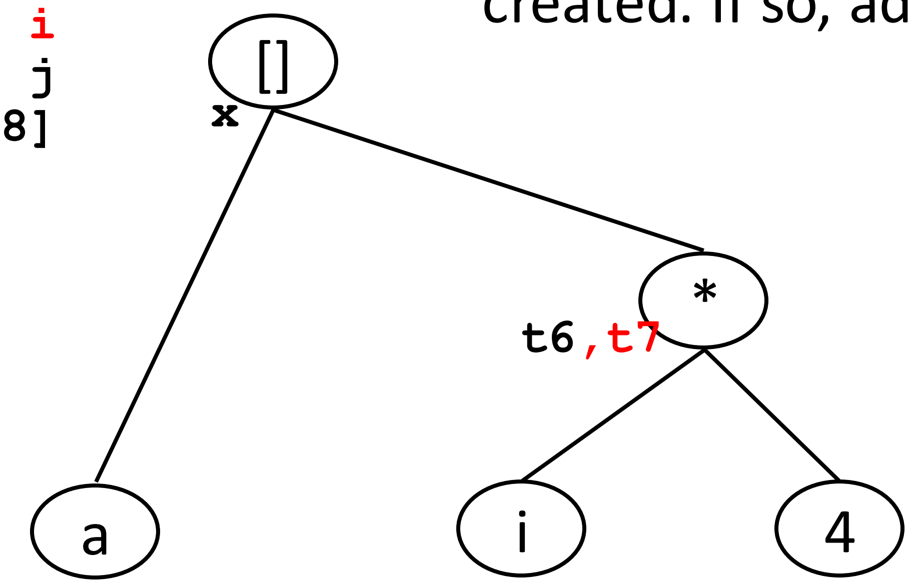
- For each var & constant not seen before create a leaf
- For each op, create a node and label with lvalue



Build a DAG

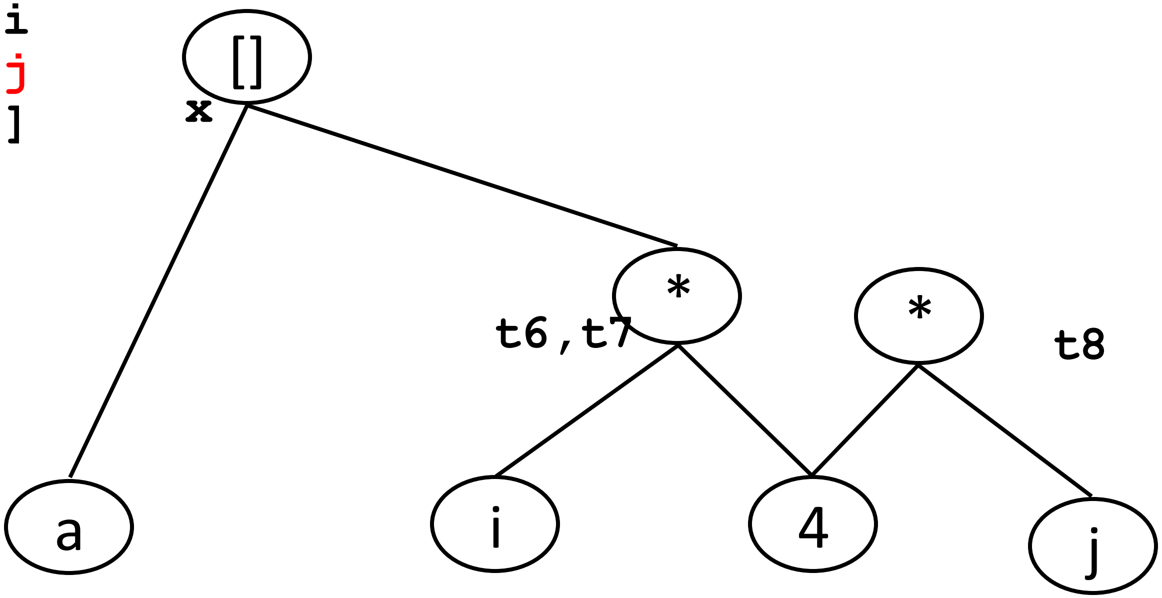
- If you have seen all rvalues before see if an interior node with same "op" and operands has already been created. If so, add a label.

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```



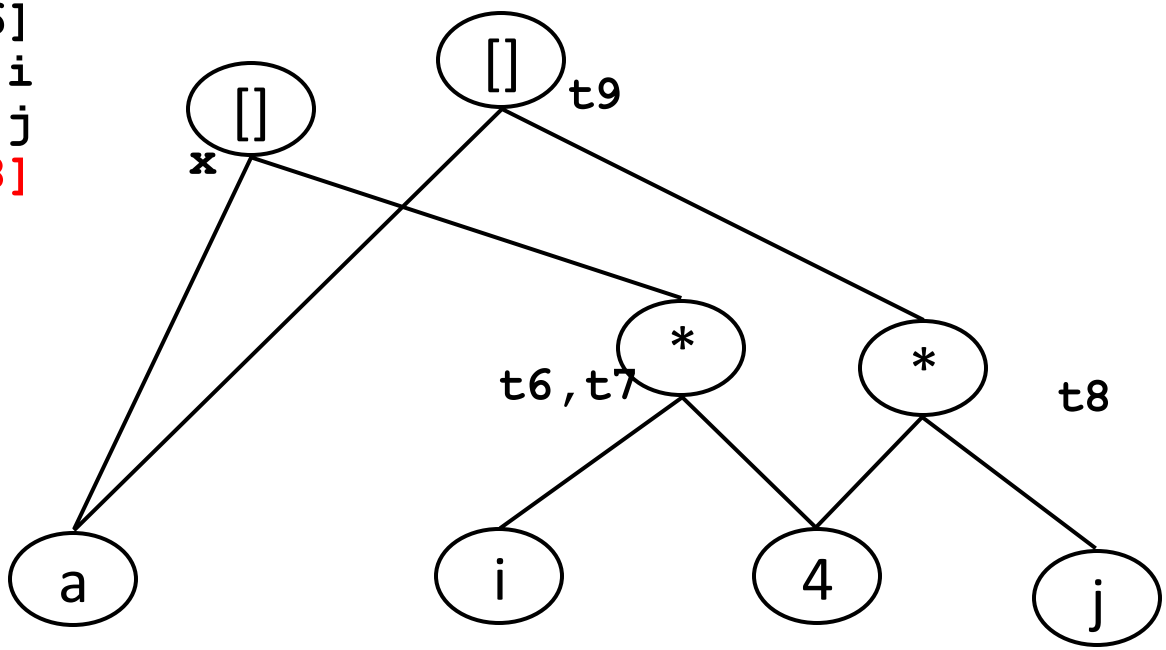
Build a DAG

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```



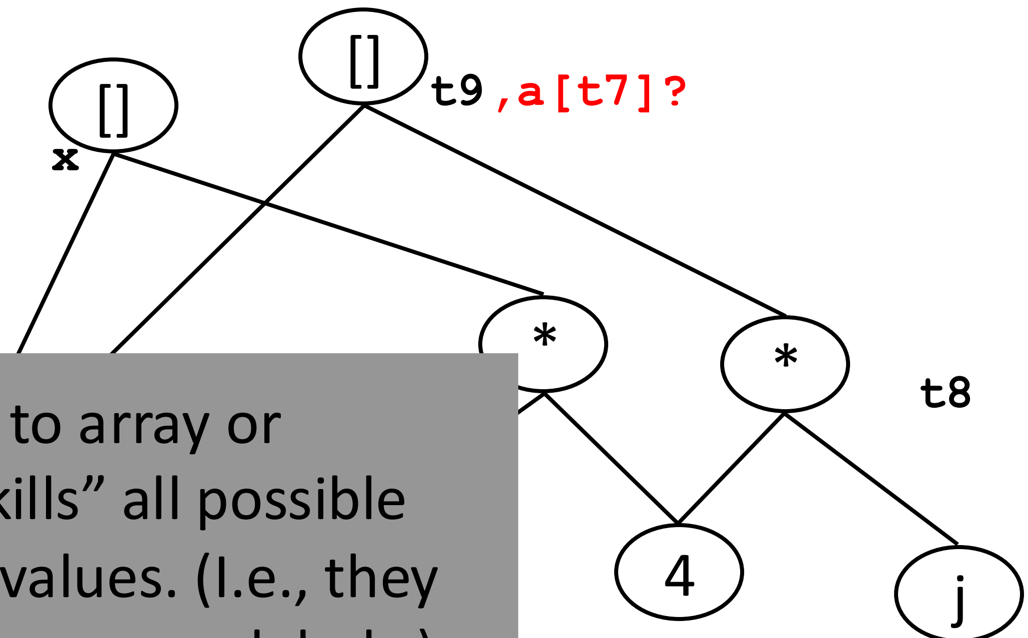
Build a DAG

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```



Build a DAG

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump
```



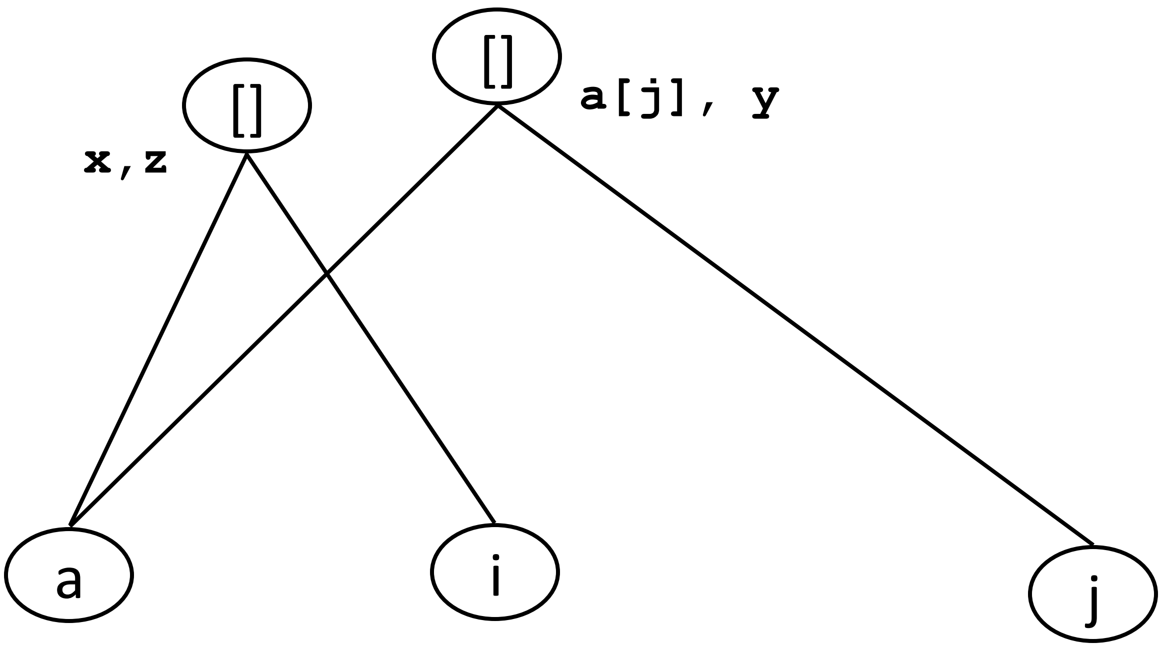
- Assigning to array or pointer “kills” all possible memory lvalues. (I.e., they can’t get any more labels.)

Memory References

```
x = a[i]
a[j] = y
z = a[i]
```

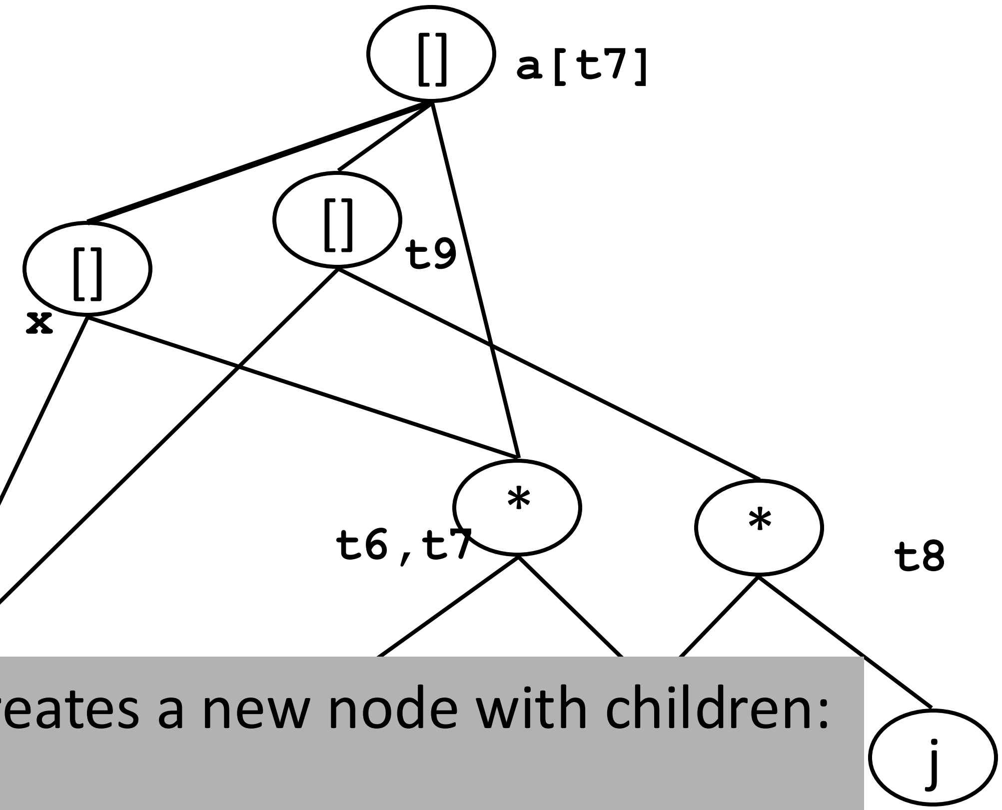
Becomes:

```
x = a[i]
z = x
a[j] = y
```



Build a DAG

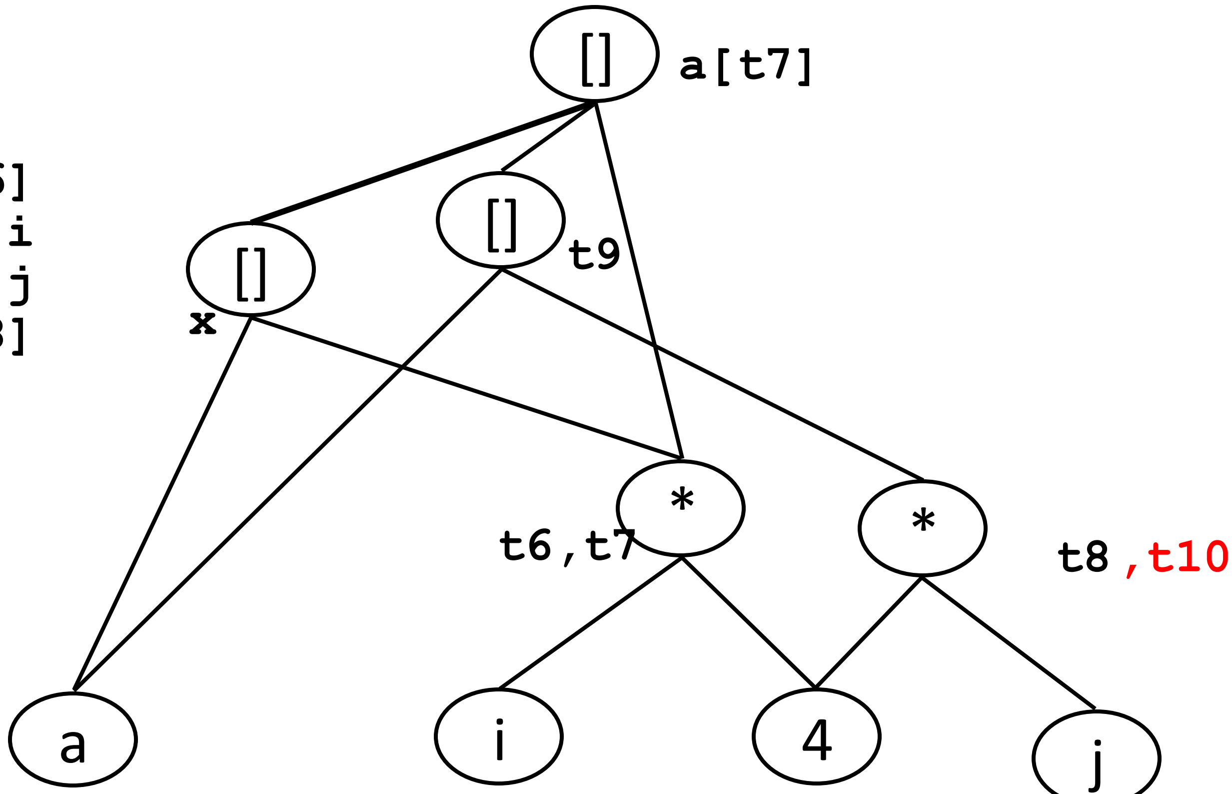
```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```



Assignment to an array creates a new node with children:
- index
- old value(s) of array
- value assigned

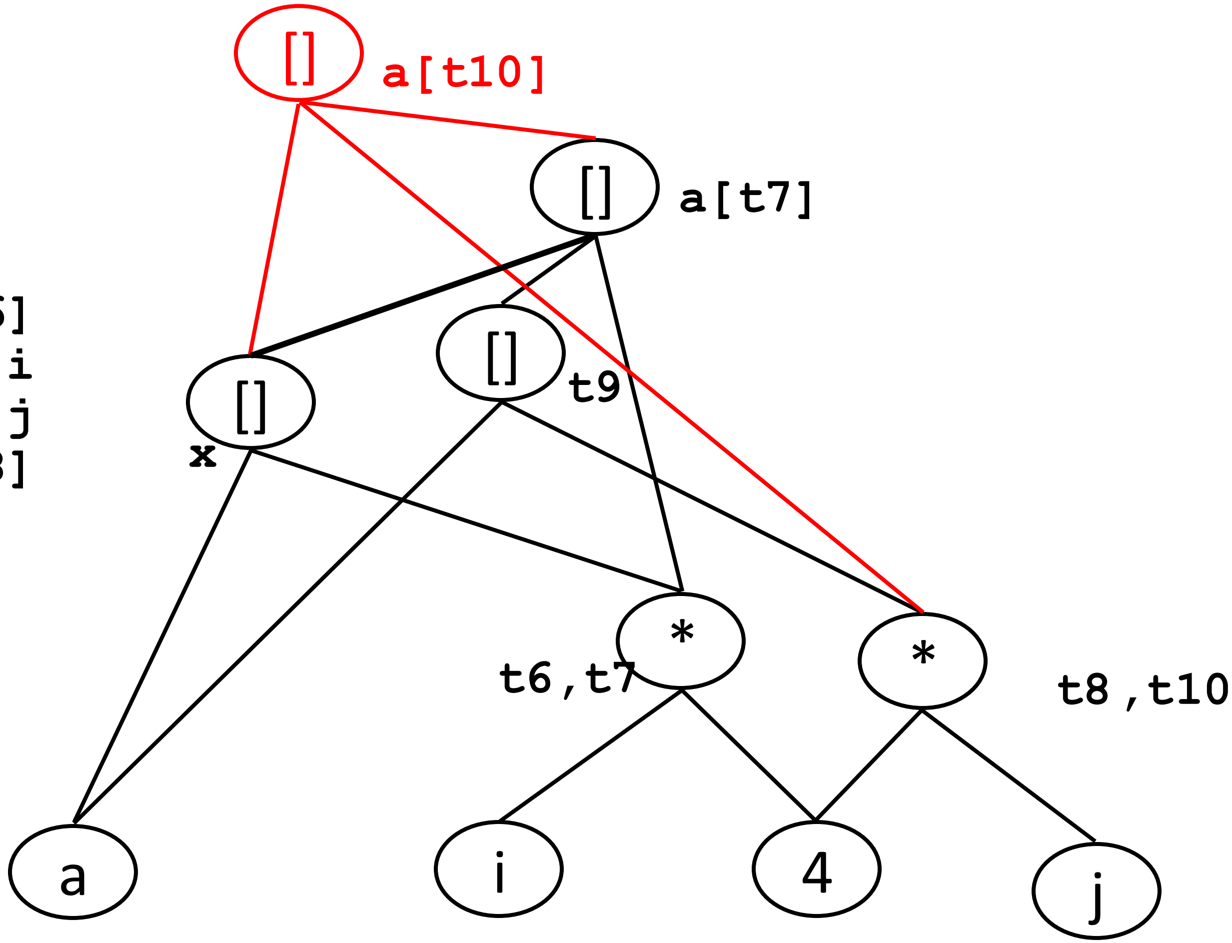
Build a DAG

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```



Build a DAG

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```



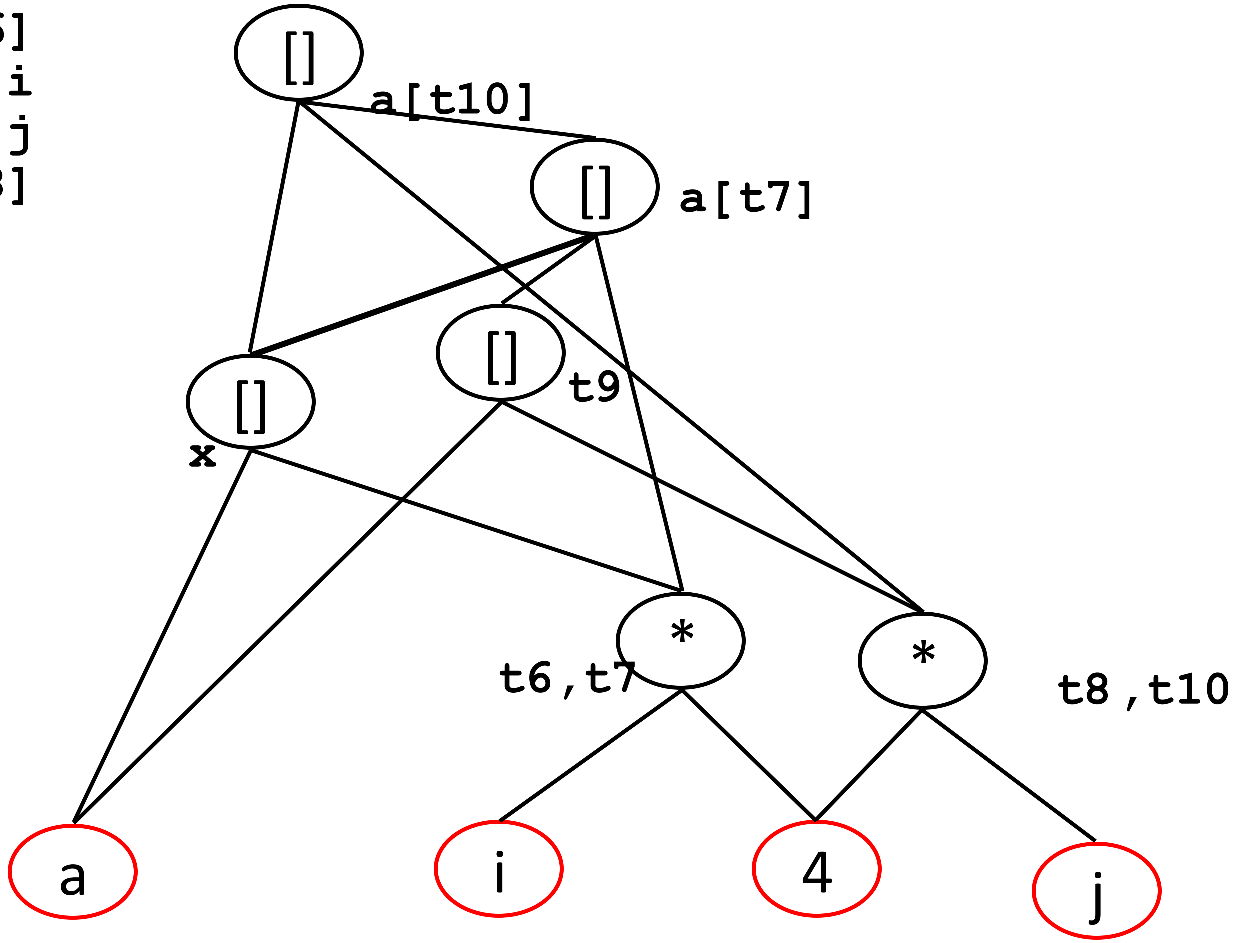
Using the DAG to recreate blocks

- Order of evaluation is any topological sort
- We pick a node. Assign it to ONE of the labels (hopefully one needed later in the program)
- If we end up with identifiers that are needed after this block, insert move statements.
- If a node has no identifiers, make up a new one.
- Caveats:
  - Procedure calls kill nodes
  - A[] = and *p = kill nodes

Recreating the Code

Original

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```

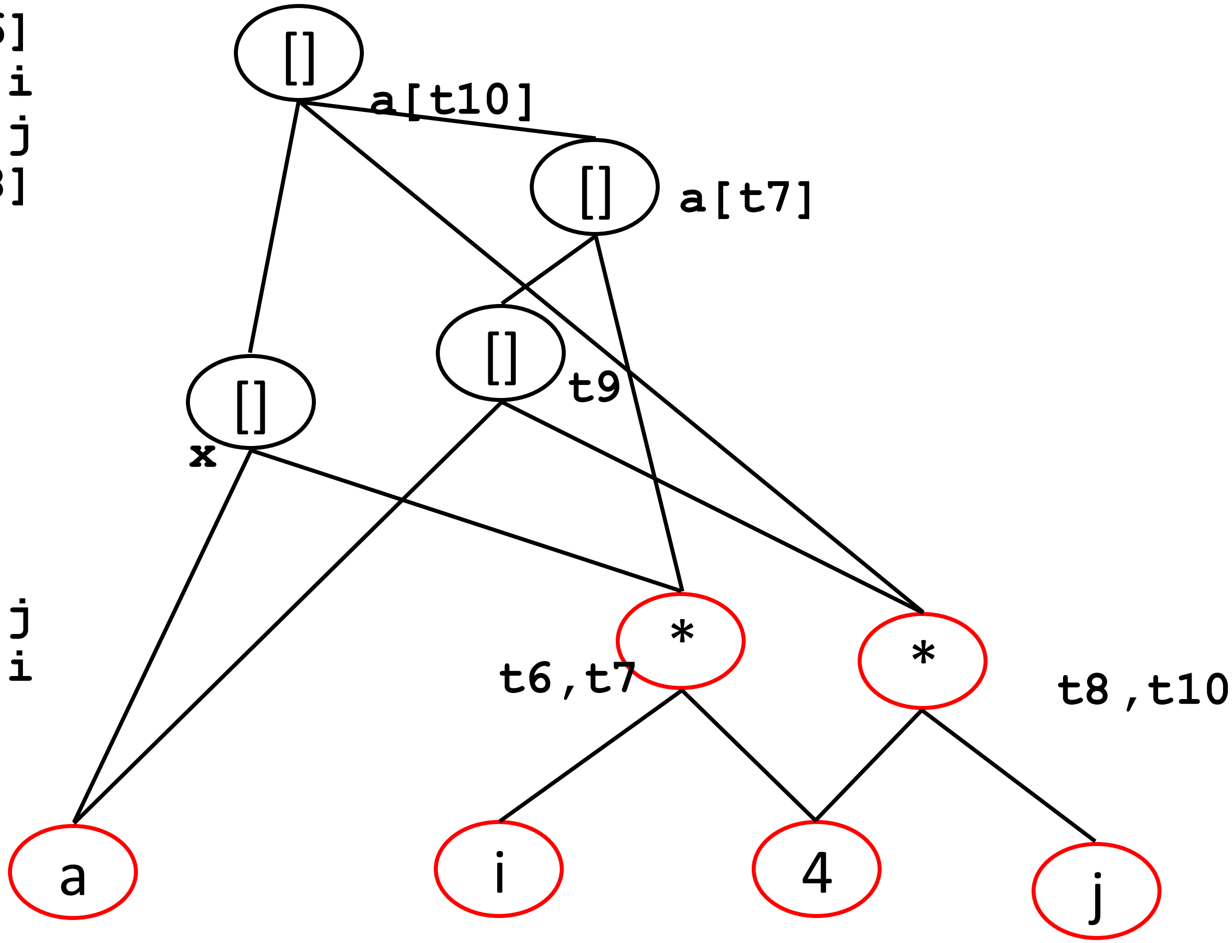


Recreating the Code

Original

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```

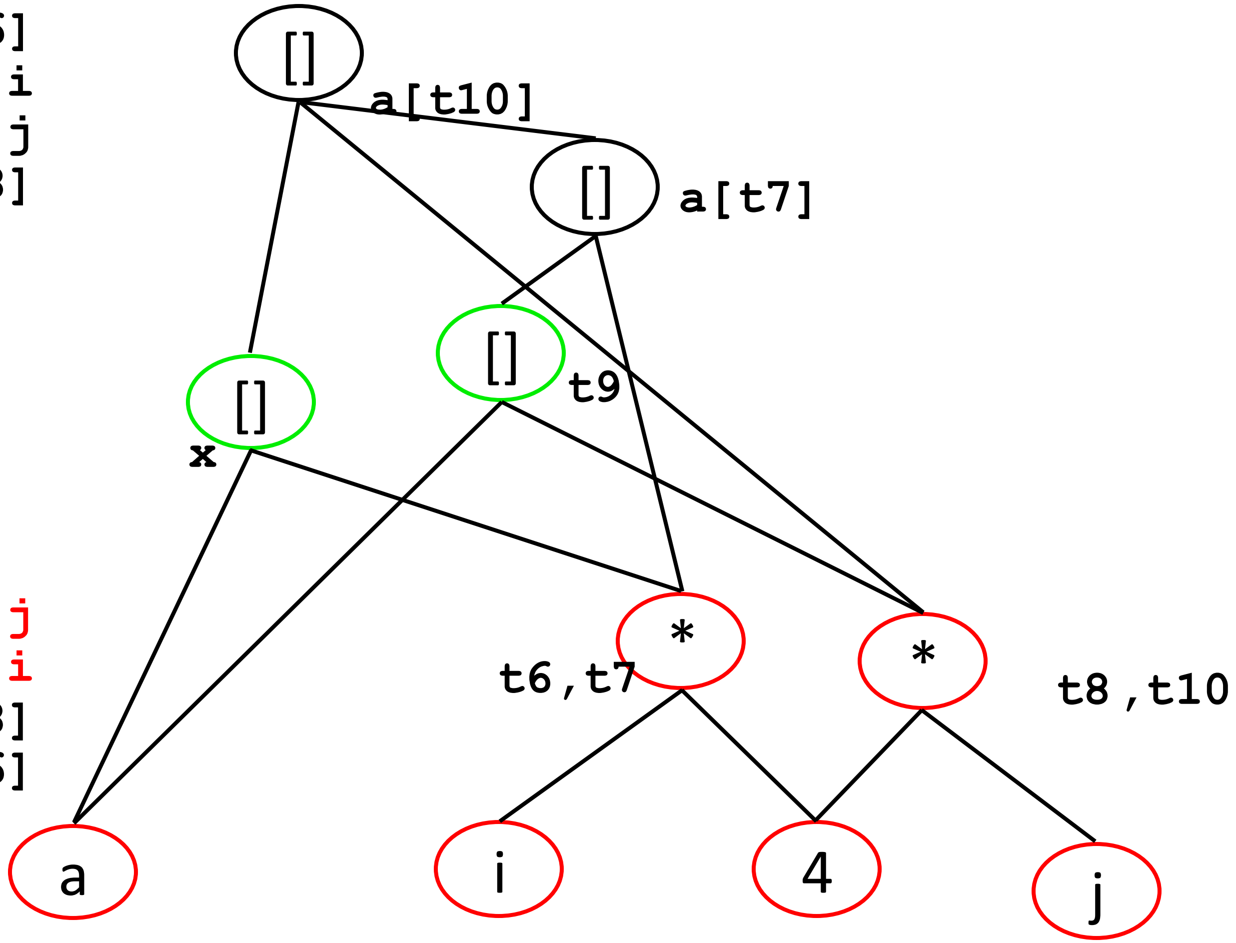
```
B5: t8     = 4 * j
    t6     = 4 * i
```



Recreating the Code

Original

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```

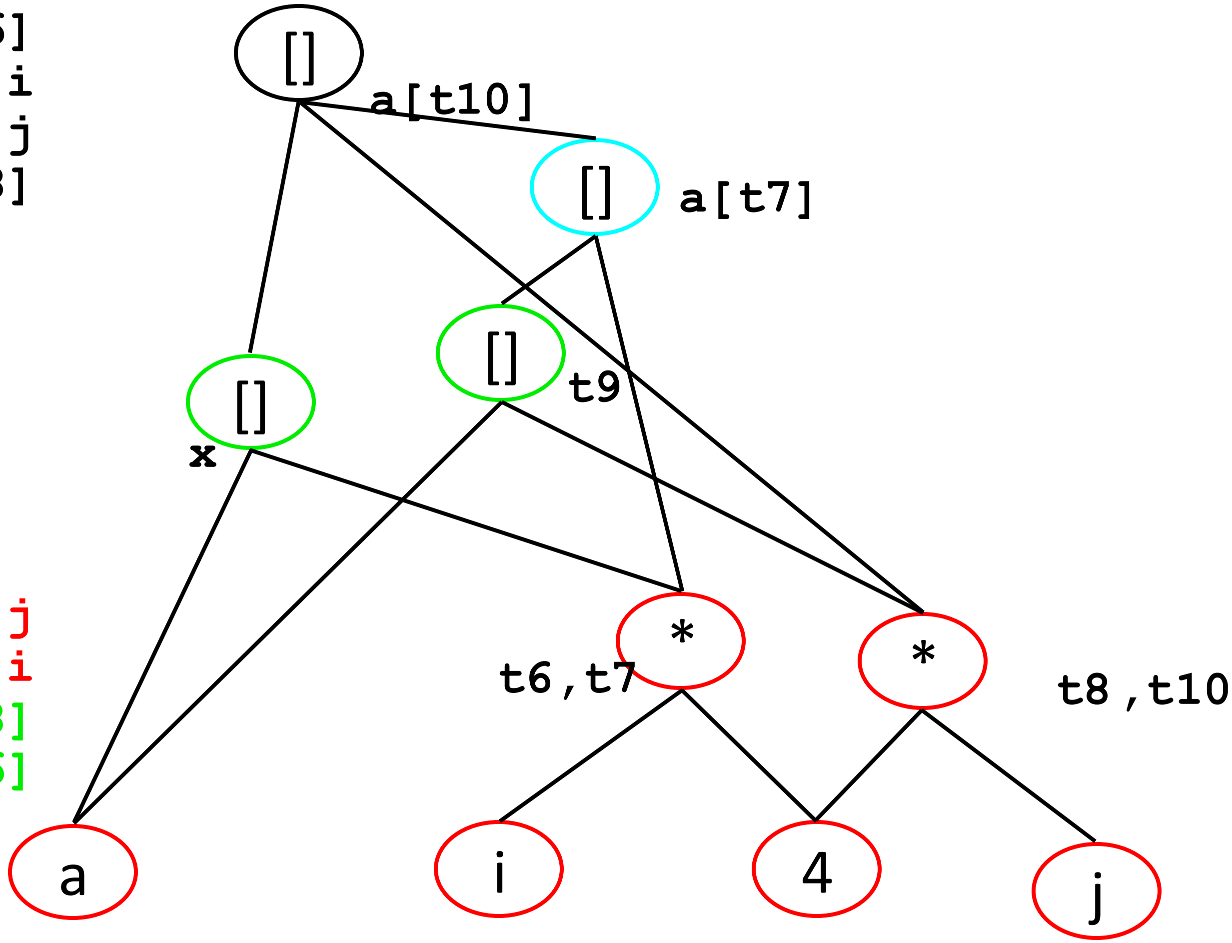



```
B5: t8     = 4 * j
    t6     = 4 * i
    t9     = a[t8]
    x      = a[t6]
```

Recreating the Code

Original

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```

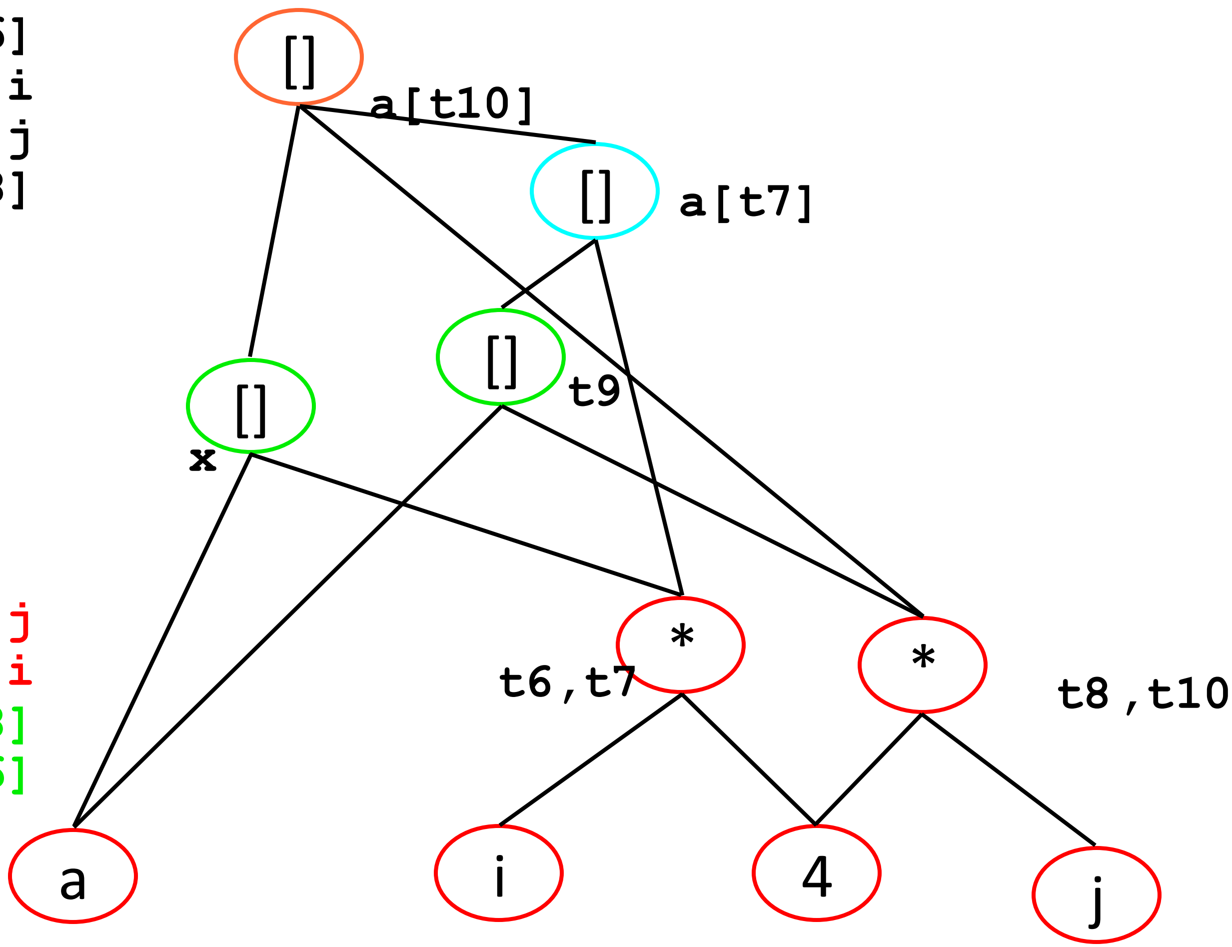



```
B5: t8     = 4 * j
    t6     = 4 * i
    t9     = a[t8]
    x      = a[t6]
    a[t6]  = t9
```

Recreating the Code

Original

```
B5: t6      = 4*i
    x       = a[t6]
    t7      = 4 * i
    t8      = 4 * j
    t9      = a[t8]
    a[t7]   = t9
    t10     = 4*j
    a[t10] = x
    jump    B2
```

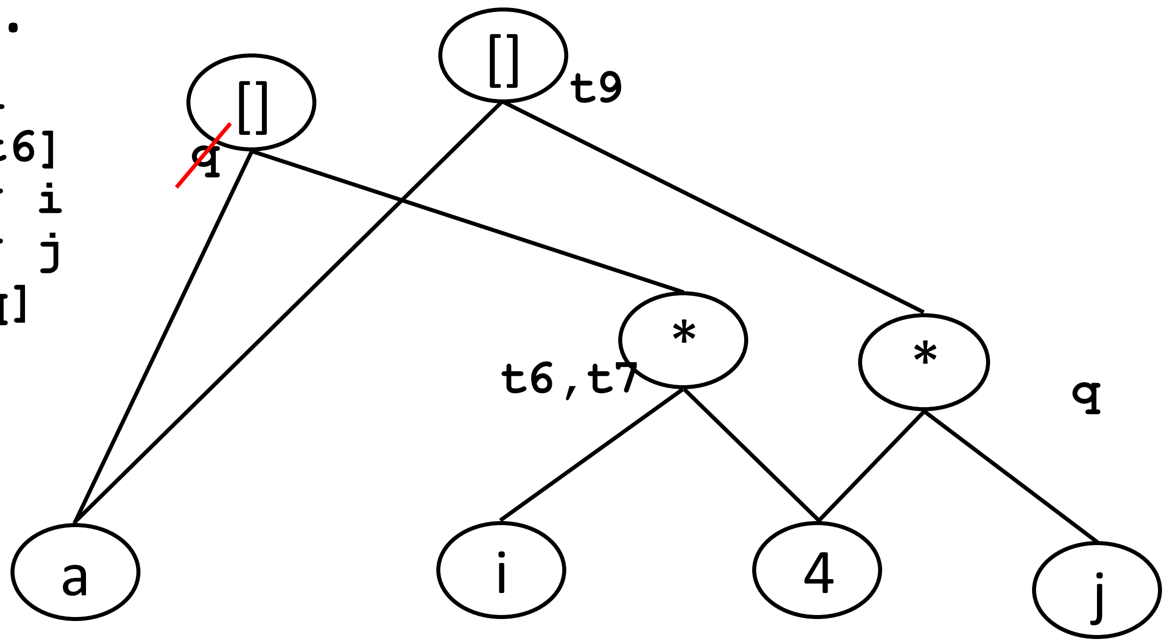



```
B5: t8      = 4 * j
    t6      = 4 * i
    t9      = a[t8]
    x       = a[t6]
    a[t6]   = t9
    a[t8]   = x
```

Other uses for DAGs

- Can determine those variables that *can* be live at end of a block.
- Can determine those variables that are live at start of block.

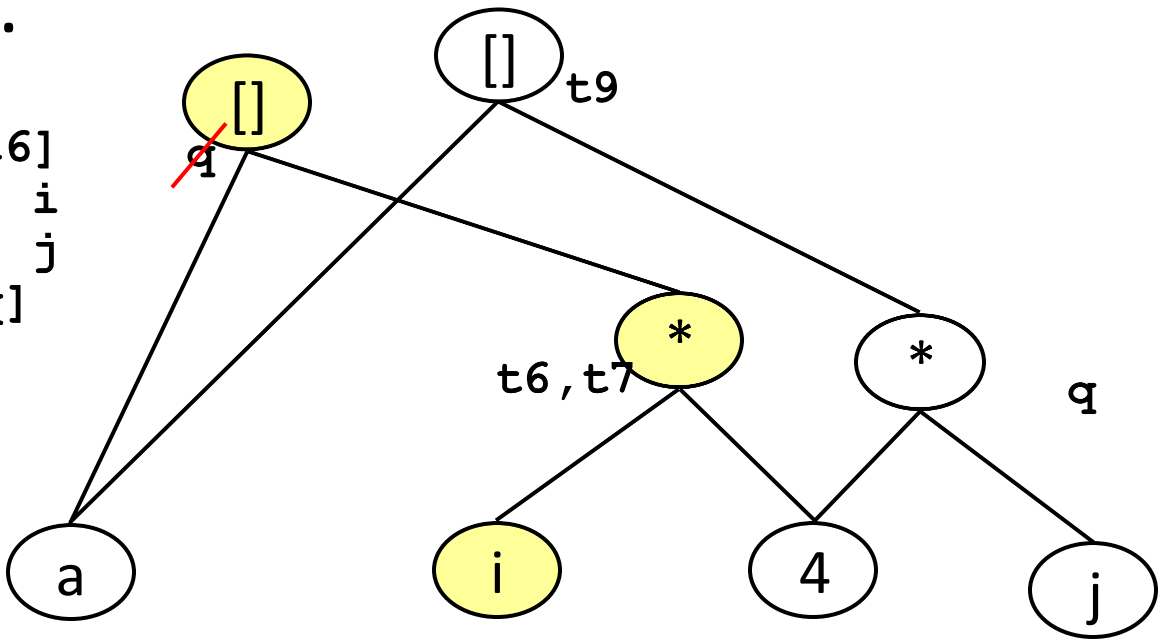
```
B8: t6     = 4*i
    q      = a[t6]
    t7     = 4 * i
    q      = 4 * j
    t9     = a[q]
    …
```



Dead code too?

- Can determine those variables that *can* be live at end of a block.
- Can determine those variables that are live at start of block.

```
B8: t6    = 4*i
    q     = a[t6]
    t7    = 4 * i
    q     = 4 * j
    t9    = a[q]
    …
```



Can we do better?

```
B1:     i       = m - 1                 B5:     t6      = 4 * i
        j       = n                             x       = a[t6]
        t1      = 4*n
        v       = a[t1]                         t8      = 4 * j
                                                t9      = a[t8]
B2:     i       = i + 1                         a[t6]   = t9
        t2      = 4 * i
        t3      = a[t2]                         a[t8]   = x
        cjump   t3<v    B2, B3                  jump    B2

                                        B6:     t11     = 4 * i
B3:     j       = j - 1                         x       = a[t11]
        t4      = 4 * j
        t5      = a[t4]                         t13     = 4 * n
        cjump   t5>v    B3, B4                  t14     = a[t13]
                                                a[t11]  = t14
B4:     cjump   i >= j B6, B5
                                                a[t13]  = x
```

Control Flow Graph

```
B1:     i       = m - 1
        j       = n
        t1      = 4*n
        v       = a[t1]
```

```
B2:     i       = i + 1
        t2      = 4 * i
        t3      = a[t2]
        cjump  t3<v   B2, B3
```

```
B3:     j       = j - 1
        t4      = 4 * j
        t5      = a[t4]
        cjump  t5>v   B3, B4
```

```
B4:     cjump  i >= j B6, B5
```

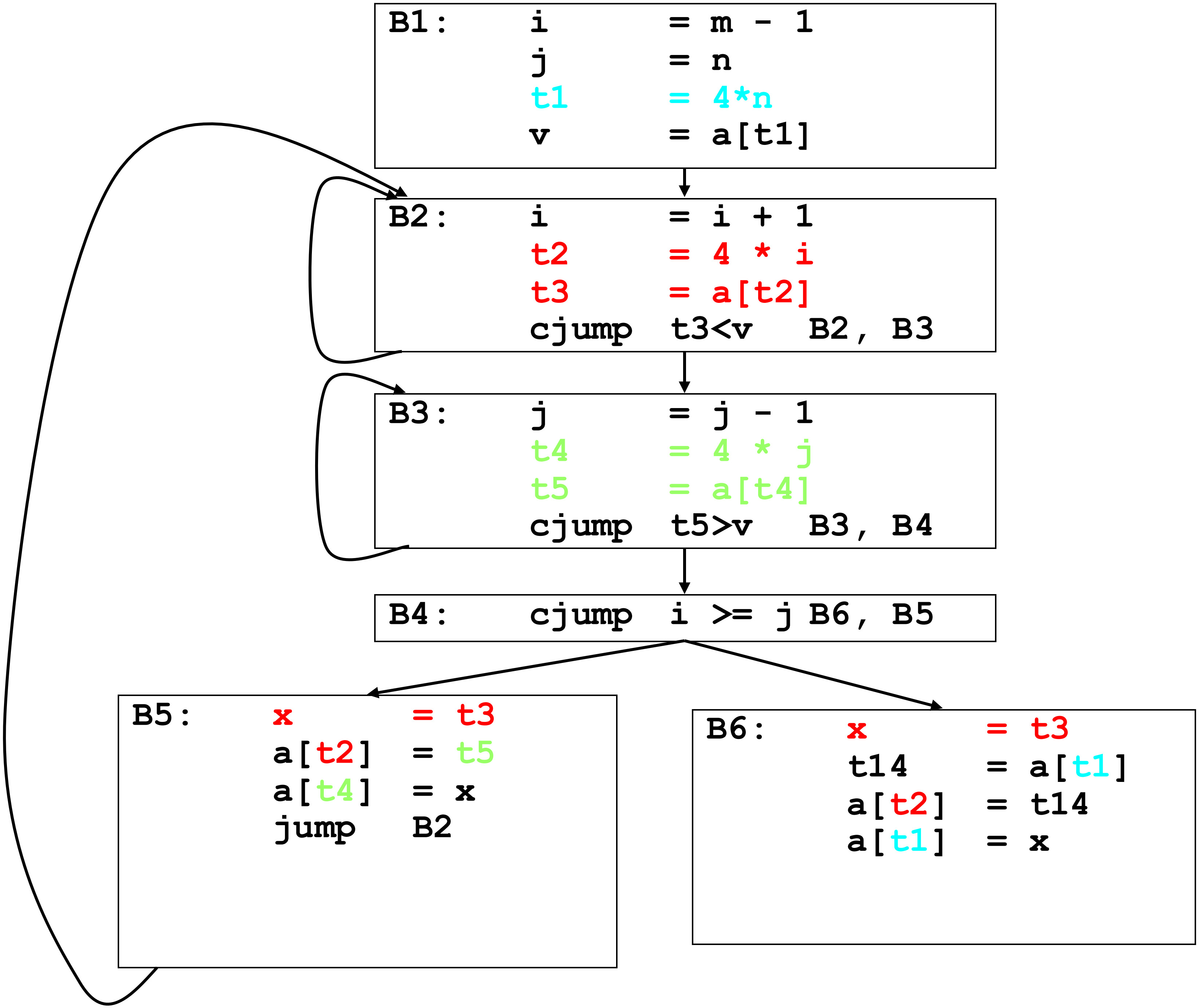
```
B5:     t6      = 4 * i
        x       = a[t6]
        t8      = 4 * j
        t9      = a[t8]
        a[t6]   = t9
        a[t8]   = x
        jump    B2
```

```
B6:     t11     = 4 * i
        x       = a[t11]
        t13     = 4 * n
        t14     = a[t13]
        a[t11]  = t14
        a[t13]  = x
```

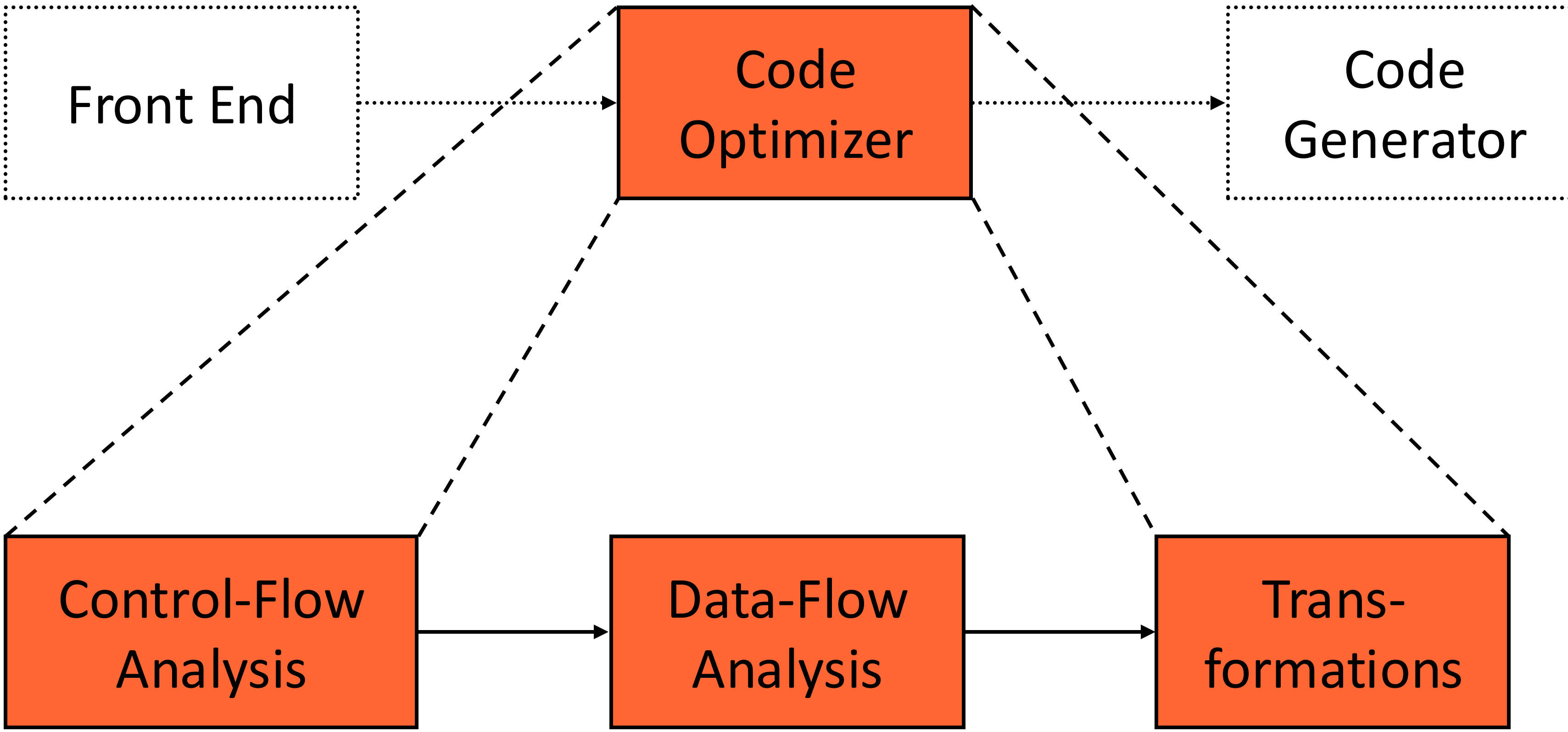
Control Flow Graph

```
B1:    i      = m - 1
       j      = n
       t1     = 4*n
       v      = a[t1]

B2:    i      = i + 1
       t2     = 4 * i
       t3     = a[t2]
       cjump  t3<v   B2, B3

B3:    j      = j - 1
       t4     = 4 * j
       t5     = a[t4]
       cjump  t5>v   B3, B4

B4:    cjump  i >= j B6, B5

B5:    t6     = 4 * i
       x      = a[t6]
       t8     = 4 * j
       t9     = a[t8]
       a[t6]  = t9
       a[t8]  = x
       jump   B2

B6:    t11    = 4 * i
       x      = a[t11]
       t13    = 4 * n
       t14    = a[t13]
       a[t11] = t14
       a[t13] = x
```

Key is data flow analysis


B1: i = m - 1
j = n
t1 = 4*n
v = a[t1]
B2: i = i + 1
t2 = 4 * i
t3 = a[t2]
cjump t3<v B2, B3
B3: j = j - 1
t4 = 4 * j
t5 = a[t4]
cjump t5>v B3, B4
B4: cjump i >= j B6, B5
B5: x = t3
a[t2] = t5
a[t4] = x
jump B2
B6: x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x

The Optimizer



Optimizations

- Register Allocation

  Liveness or reaching definitions
- Common subexpression elimination

  Available expressions and reaching expressions
- Constant Propagation
- Copy propagation
- Dead-code elimination
- Loop optimizations
  - Hoisting
  - Induction variable elimination

Dataflow Analysis

- Goal:
  - Answers: Is it legal to perform an optimization?
  - (Not answering: “it is beneficial?”)
- A framework for proving facts about program
- Reasons about lots of little facts
- Little or no interaction between facts
  - Works best on properties about how program computes
- Based on all paths through program
  - including infeasible paths

Today

- Dataflow Analysis
  - reaching definitions
  - liveness
  - available expressions
  - very busy expressions
- Framework

A sample program

```
int fib10(void) {
  int n = 10;
  int older = 0;
  int old = 1;
  int result = 0;
  int i;

  if (n <= 1) return n;
  for (i = 2; i<n; i++) {
      result = old + older;
      older = old;
      old = result;
  }
  return result;
}
```

```
1:    n <- 10
2:    older <- 0
3:    old <- 1
4:    result <- 0
5:    if n <= 1 goto 14
6:    i <- 2
7:    if i > n goto 13

8:    result <- old + older
9:    older <- old
10:   old <- result
11:   i <- i + 1
12:   JUMP 7
13:   return result
14:   return n
```

Simple Constant Propagation

- Can we do SCP?
- How do we recognize it?

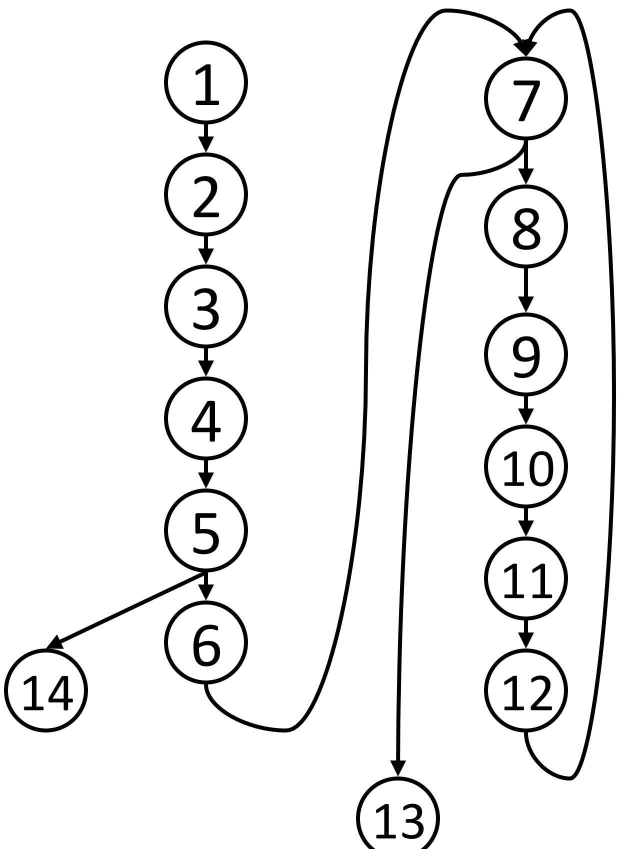
- What aren't we doing?

- Metanote:
  - keep opts simple!
  - Use combined power

```
1:     n <- 10
2:     older <- 0
3:     old <- 1
4:     result <- 0
5:     if n <= 1 goto 14
6:     i <- 2
7:     if i > n goto 13

8:     result <- old + older
9:     older <- old
10:    old <- result
11:    i <- i + 1
12:    JUMP 7
13:    return result
14:    return n
```

Reaching Definitions

A definition of variable v at program point d reaches program point u if there exists a path of control flow edges from d to u that does not contain a definition of v.




```
1:      n <- 10
2:      older <- 0
3:      old <- 1
4:      result <- 0
5:      if n <= 1 goto 14
6:      i <- 2
7:      if i > n goto 13
8:      result <- old + older
9:      older <- old
10:     old <- result
11:     i <- i + 1
12:     JUMP 7
13:     return result
14:     return n
```

Reaching Definitions (ex)

- 1 reaches 5, 7, and 14

14, Really?

Meta-notes:

- (almost) always conservative
- only know what we know
- Keep it simple:
  - What opt(s), if run before this would help
  - What about:

```
1: x <- 0
2: if (false) x<-1
3: … x …
```

  - Does 1 reach 3?
  - What opt changes this?

```
1:   n <- 10
2:   older <- 0
3:   old <- 1
4:   result <- 0
5:   if n <= 1 goto 14
6:   i <- 2
7:   if i > n goto 13
8:   result <- old + older
9:   older <- old
10:  old <- result
11:  i <- i + 1
12:  JUMP 7
13:  return result
14:  return n
```

Calculating Reaching Definitions

A definition of variable v at program point d reaches program point u if there exists a path of control flow edges from d to u that does not contain a definition of v.

- Build up RD stmt by stmt
- Stmt s, “d: v <- x op y”, generates d
- Stmt s, “d: v <- x op y”, kills all other defs(v)

Or,

- Gen[s] = { d }
- Kill[s] = defs(v) – { d }

Gen and kill for each stmt

Stmt	Gen	kill
`1: n <- 10`	1	
`2: older <- 0`	2	9
`3: old <- 1`	3	10
`4: result <- 0`	4	8
`5: if n <= 1 goto 14`		
`6: i <- 2`	6	11
`7: if i > n goto 13`		
`8: result <- old + older`	8	4
`9: older <- old`	9	2
`10: old <- result`	10	3
`11: i <- i + 1`	11	6
`12: JUMP 7`		
`13: return result`		
`14: return n`		

we determine the defs that reach a node by using:

- control flow information
- gen and kill info

Computing in[n] and out[n]

- In[n]: the set of defs that reach the beginning of node n
- Out[n]: the set of defs that reach the end of node n

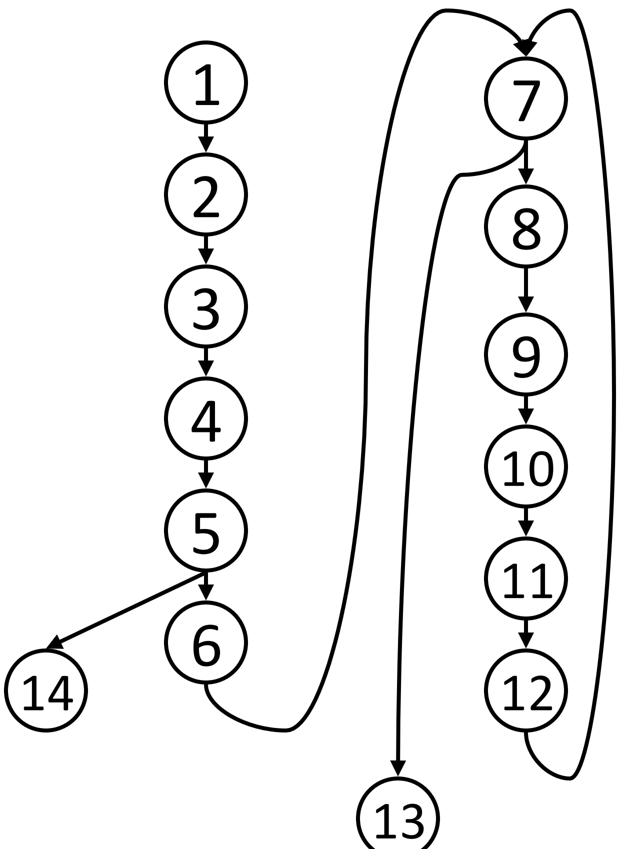
$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

- Initialize in[n]=out[n]={} for all n
- Solve iteratively

pred[n]?

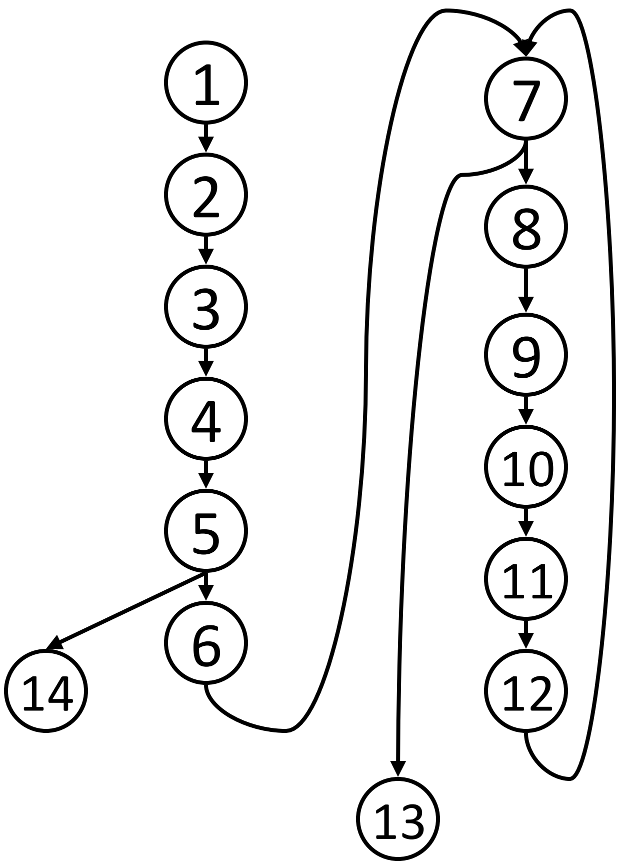
- Pred[n] are all nodes that can directly reach n in the control flow graph.
- E.g., pred[7] = { 6, 12 }




```
1:      n <- 10
2:      older <- 0
3:      old <- 1
4:      result <- 0
5:      if n <= 1 goto 14
6:      i <- 2
7:      if i > n goto 13
8:      result <- old + older
9:      older <- old
10:     old <- result
11:     i <- i + 1
12:     JUMP 7
13:     return result
14:     return n
```

What order to eval nodes?

- Does it matter?
- Lets do: 1,2,3,4,5,14,6,7,13,8,9,10,11,12




```
1:     n <- 10
2:     older <- 0
3:     old <- 1
4:     result <- 0
5:     if n <= 1 goto 14
6:     i <- 2
7:     if i > n goto 13
8:     result <- old + older
9:     older <- old
10:    old <- result
11:    i <- i + 1
12:    JUMP 7
13:    return result
14:    return n
```

Example:

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \qquad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9		
`3: old <- 1`	3	10		
`4: result <- 0`	4	8		
`5: if n <= 1 goto 14`				
`6: i <- 2`	6	11		
`7: if i > n goto 13`				
`8: result <- old + older`	8	4		
`9: older <- old`	9	2		
`10: old <- result`	10	3		
`11: i <- i + 1`	11	6		
`12: JUMP 7`				
`13: return result`				
`14: return n`				

Example:

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10		
`4: result <- 0`	4	8		
`5: if n <= 1 goto 14`				
`6: i <- 2`	6	11		
`7: if i > n goto 13`				
`8: result <- old + older`	8	4		
`9: older <- old`	9	2		
`10: old <- result`	10	3		
`11: i <- i + 1`	11	6		
`12: JUMP 7`				
`13: return result`				
`14: return n`				

Example:

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \cup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8		
`5: if n <= 1 goto 14`				
`6: i <- 2`	6	11		
`7: if i > n goto 13`				
`8: result <- old + older`	8	4		
`9: older <- old`	9	2		
`10: old <- result`	10	3		
`11: i <- i + 1`	11	6		
`12: JUMP 7`				
`13: return result`				
`14: return n`				

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \cup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`				
`6: i <- 2`	6	11		
`7: if i > n goto 13`				
`8: result <- old + older`	8	4		
`9: older <- old`	9	2		
`10: old <- result`	10	3		
`11: i <- i + 1`	11	6		
`12: JUMP 7`				
`13: return result`				
`14: return n`				

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \qquad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
1: n <- 10	1			1
2: older <- 0	2	9	1	1,2
3: old <- 1	3	10	1,2	1,2,3
4: result <- 0	4	8	1-3	1-4
5: if n <= 1 goto 14			1-4	1-4
6: i <- 2	6	11		
7: if i > n goto 13				
8: result <- old + older	8	4		
9: older <- old	9	2		
10: old <- result	10	3		
11: i <- i + 1	11	6		
12: JUMP 7				
13: return result				
14: return n			1-4	1-4

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \cup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`				
`8: result <- old + older`	8	4		
`9: older <- old`	9	2		
`10: old <- result`	10	3		
`11: i <- i + 1`	11	6		
`12: JUMP 7`				
`13: return result`				
`14: return n`			1-4	1-4

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6	1-4,6
`8: result <- old + older`	8	4		
`9: older <- old`	9	2		
`10: old <- result`	10	3		
`11: i <- i + 1`	11	6		
`12: JUMP 7`				
`13: return result`				
`14: return n`			1-4	1-4

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \cup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6	1-4,6
`8: result <- old + older`	8	4		
`9: older <- old`	9	2		
`10: old <- result`	10	3		
`11: i <- i + 1`	11	6		
`12: JUMP 7`				
`13: return result`			1-4,6	1-4,6
`14: return n`			1-4	1-4

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \qquad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
1: n <- 10	1			1
2: older <- 0	2	9	1	1,2
3: old <- 1	3	10	1,2	1,2,3
4: result <- 0	4	8	1-3	1-4
5: if n <= 1 goto 14			1-4	1-4
6: i <- 2	6	11	1-4	1-4,6
7: if i > n goto 13			1-4,6	1-4,6
8: result <- old + older	8	4	1-4,6	1-3,6,8
9: older <- old	9	2		
10: old <- result	10	3		
11: i <- i + 1	11	6		
12: JUMP 7				
13: return result			1-4,6	1-4,6
14: return n			1-4	1-4

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \qquad out[n] = gen[n] \cup (in[n] - kill[n])$$

	Gen	kill	in	out
1: n <- 10	1			1
2: older <- 0	2	9	1	1,2
3: old <- 1	3	10	1,2	1,2,3
4: result <- 0	4	8	1-3	1-4
5: if n <= 1 goto 14			1-4	1-4
6: i <- 2	6	11	1-4	1-4,6
7: if i > n goto 13			1-4,6	1-4,6
8: result <- old + older	8	4	1-4,6	1-3,6,8
9: older <- old	9	2	1-3,6,8	1,3,6,8,9
10: old <- result	10	3		
11: i <- i + 1	11	6		
12: JUMP 7				
13: return result			1-4,6	1-4,6
14: return n			1-4	1-4

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6	1-4,6
`8: result <- old + older`	8	4	1-4,6	1-3,6,8
`9: older <- old`	9	2	1-3,6,8	1,3,6,8,9
`10: old <- result`	10	3	1,3,6,8,9	1,6,8-10
`11: i <- i + 1`	11	6		
`12: JUMP 7`				
`13: return result`			1-4,6	1-4,6
`14: return n`			1-4	1-4

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
1: n <- 10	1			1
2: older <- 0	2	9	1	1,2
3: old <- 1	3	10	1,2	1,2,3
4: result <- 0	4	8	1-3	1-4
5: if n <= 1 goto 14			1-4	1-4
6: i <- 2	6	11	1-4	1-4,6
7: if i > n goto 13			1-4,6	1-4,6
8: result <- old + older	8	4	1-4,6	1-3,6,8
9: older <- old	9	2	1-3,6,8	1,3,6,8,9
10: old <- result	10	3	1,3,6,8,9	1,6,8-10
11: i <- i + 1	11	6	1,6,8-10	1,8-11
12: JUMP 7				
13: return result			1-4,6	1-4,6
14: return n			1-4	1-4

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
1: `n <- 10`	1			1
2: `older <- 0`	2	9	1	1,2
3: `old <- 1`	3	10	1,2	1,2,3
4: `result <- 0`	4	8	1-3	1-4
5: `if n <= 1 goto 14`			1-4	1-4
6: `i <- 2`	6	11	1-4	1-4,6
7: `if i > n goto 13`			1-4,6	1-4,6
8: `result <- old + older`	8	4	1-4,6	1-3,6,8
9: `older <- old`	9	2	1-3,6,8	1,3,6,8,9
10: `old <- result`	10	3	1,3,6,8,9	1,6,8-10
11: `i <- i + 1`	11	6	1,6,8-10	1,8-11
12: `JUMP 7`			1,8-11	1,8-11
13: `return result`			1-4,6	1-4,6
14: `return n`			1-4	1-4

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6	1-4,6
`8: result <- old + older`	8	4	1-4,6	1-3,6,8
`9: older <- old`	9	2	1-3,6,8	1,3,6,8,9
`10: old <- result`	10	3	1,3,6,8,9	1,6,8-10
`11: i <- i + 1`	11	6	1,6,8-10	1,8-11
`12: JUMP 7`			1,8-11	1,8-11
`13: return result`			1-4,6	1-4,6
`14: return n`			1-4	1-4

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \cup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6	1-4,6
`8: result <- old + older`	8	4	1-4,6	1-3,6,8
`9: older <- old`	9	2	1-3,6,8	1,3,6,8,9
`10: old <- result`	10	3	1,3,6,8,9	1,6,8-10
`11: i <- i + 1`	11	6	1,6,8-10	1,8-11
`12: JUMP 7`			1,8-11	1,8-11
`13: return result`			1-4,6	1-4,6
`14: return n`			1-4	1-4

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \qquad out[n] = gen[n] \cup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6,8-11	1-4,6,8-11
`8: result <- old + older`	8	4	1-4,6	1-3,6,8
`9: older <- old`	9	2	1-3,6,8	1,3,6,8,9
`10: old <- result`	10	3	1,3,6,8,9	1,6,8-10
`11: i <- i + 1`	11	6	1,6,8-10	1,8-11
`12: JUMP 7`			1,8-11	1,8-11
`13: return result`			1-4,6	1-4,6
`14: return n`			1-4	1-4

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \cup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6,8-11	1-4,6,8-11
`8: result <- old + older`	8	4	1-4,6	1-3,6,8
`9: older <- old`	9	2	1-3,6,8	1,3,6,8,9
`10: old <- result`	10	3	1,3,6,8,9	1,6,8-10
`11: i <- i + 1`	11	6	1,6,8-10	1,8-11
`12: JUMP 7`			1,8-11	1,8-11
`13: return result`			1-4,6,8-11	1-4,6,8-11
`14: return n`			1-4	1-4

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \qquad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6,8-11	1-4,6,8-11
`8: result <- old + older`	8	4	1-4,6,8-11	1-3,6,8-11
`9: older <- old`	9	2	1-3,6,8	1,3,6,8,9
`10: old <- result`	10	3	1,3,6,8,9	1,6,8-10
`11: i <- i + 1`	11	6	1,6,8-10	1,8-11
`12: JUMP 7`			1,8-11	1,8-11
`13: return result`			1-4,6,8-11	1-4,6,8-11
`14: return n`			1-4	1-4

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6,8-11	1-4,6,8-11
`8: result <- old + older`	8	4	1-4,6,8-11	1-3,6,8-11
`9: older <- old`	9	2	1-3,6,8-11	1,3,6,8-11
`10: old <- result`	10	3	1,3,6,8,9	1,6,8-10
`11: i <- i + 1`	11	6	1,6,8-10	1,8-11
`12: JUMP 7`			1,8-11	1,8-11
`13: return result`			1-4,6,8-11	1-4,6,8-11
`14: return n`			1-4	1-4

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \cup (in[n] - kill[n])$$

	Gen	kill	in	out
1: `n <- 10`	1			1
2: `older <- 0`	2	9	1	1,2
3: `old <- 1`	3	10	1,2	1,2,3
4: `result <- 0`	4	8	1-3	1-4
5: `if n <= 1 goto 14`			1-4	1-4
6: `i <- 2`	6	11	1-4	1-4,6
7: `if i > n goto 13`			1-4,6,8-11	1-4,6,8-11
8: `result <- old + older`	8	4	1-4,6,8-11	1-3,6,8-11
9: `older <- old`	9	2	1-3,6,8-11	1,3,6,8-11
10: `old <- result`	10	3	1,3,6,8-11	1,6,8-11
11: `i <- i + 1`	11	6	1,6,8-10	1,8-11
12: `JUMP 7`			1,8-11	1,8-11
13: `return result`			1-4,6,8-11	1-4,6,8-11
14: `return n`			1-4	1-4

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6,8-11	1-4,6,8-11
`8: result <- old + older`	8	4	1-4,6,8-11	1-3,6,8-11
`9: older <- old`	9	2	1-3,6,8-11	1,3,6,8-11
`10: old <- result`	10	3	1,3,6,8-11	1,6,8-11
`11: i <- i + 1`	11	6	1,6,8-11	1,8-11
`12: JUMP 7`			1,8-11	1,8-11
`13: return result`			1-4,6,8-11	1-4,6,8-11
`14: return n`			1-4	1-4

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \cup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6,8-11	1-4,6,8-11
`8: result <- old + older`	8	4	1-4,6,8-11	1-3,6,8-11
`9: older <- old`	9	2	1-3,6,8-11	1,3,6,8-11
`10: old <- result`	10	3	1,3,6,8-11	1,6,8-11
`11: i <- i + 1`	11	6	1,6,8-11	1,8-11
`12: JUMP 7`			1,8-11	1,8-11
`13: return result`			1-4,6,8-11	1-4,6,8-11
`14: return n`			1-4	1-4

An Improvement: Basic Blocks

- No need to compute this one stmt at a time
- For straight line code:
  - In[s1; s2] = in[s1]
  - Out[s1; s2] = out[s2]
- Combine the gen and kill sets into one per BB.

- Gen[BB]={2,3,4,5}
- Kill[BB]={1,8,11}

	Gen	kill
`1: i <- 1`	1	8,4
`2: j <- 2`	2	
`3: k <- 3 + i`	3	11
`4: i <- j`	4	1,8
`5: m <- i + k`	5	

General Procedure

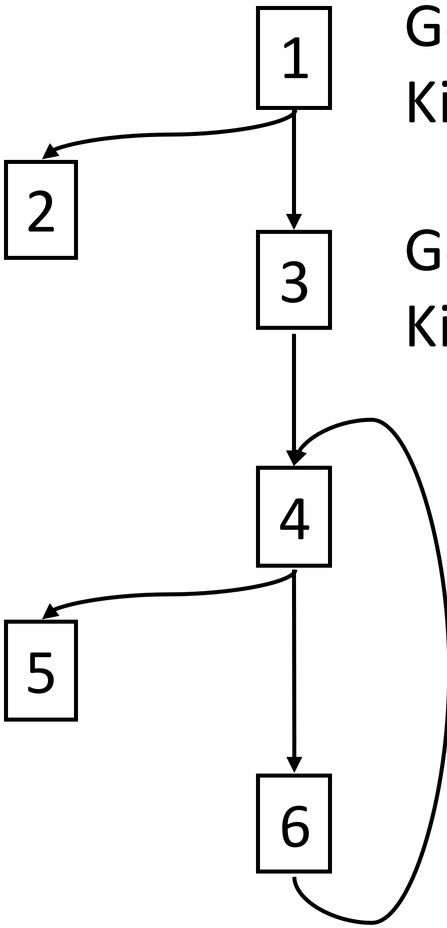
- Create Basic Blocks
- Create Control Flow Graph
- Summarize Basic Block into a single Gen set and a single Kill set
- Iterate over CFG until no changes
- create data flow facts for individual instructions as needed starting with in set of the basic block

BB sets

		Gen	kill		
	1: n <- 10	1			
	2: older <- 0	2	9		
1	3: old <- 1	3	10		
	4: result <- 0	4	8		
	5: if n <= 1 goto 14			1,2,3,4	8,9,10
3	6: i <- 2	6	11	6	11
4	7: if i > n goto 13				
	8: result <- old + older	8	4		
	9: older <- old	9	2		
6	10: old <- result	10	3		
	11: i <- i + 1	11	6		
	12: JUMP 7			8-11	2-4,6
5	13: return result				
2	14: return n				

BB sets

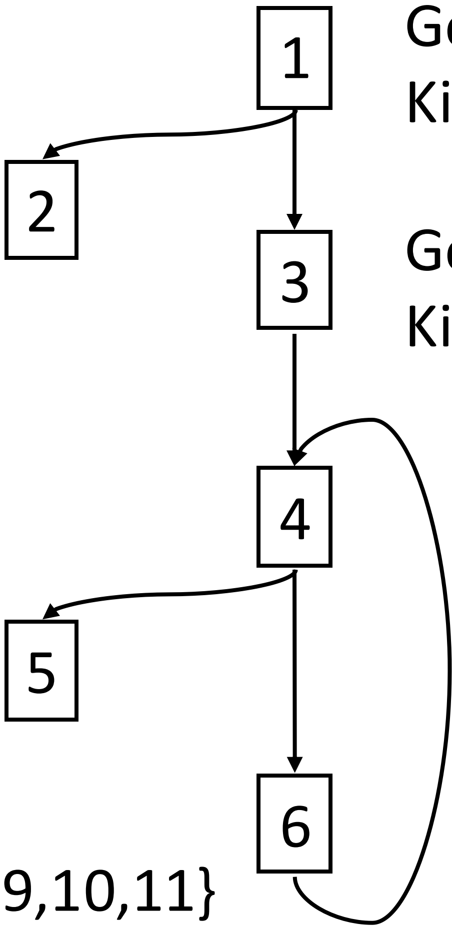
	In	out
Gen={1,2,3,4} Kill={8,9,10}		1,2,3,4



Gen={6}
Kill={11}

Gen={8,9,10,11}
Kill={2,3,4,6}

BB sets



Gen={1,2,3,4}
Kill={8,9,10}

Gen={6}
Kill={11}

Gen={8,9,10,11}
Kill={2,3,4,6}

In	out
	1,2,3,4
1,2,3,4	1,2,3,4,6
1-4,6,8-11	1-4,6,8-11
1-4,6,8-11	1,8-11

Forward Dataflow

- Reaching definitions is a forward dataflow problem:
  It propagates information from the predecessors of a node to the node
- Defined by:
  - Basic attributes: (gen and kill)
  - Transfer function: F_{bb} $\quad out[n] = gen[n] \cup (in[n] - kill[n])$
  - Meet operator: union $\quad in[n] = \bigcup_{p \in pred[n]} out[p]$
  - Set of values (a lattice, in this case powerset of program points)
  - Initial values for each node b
- Solve for fixed point solution

How to implement?

- Values?
- Gen?
- Kill?
- F_{bb}?
- Init?
- Order to visit nodes?
- When are we done?
  - In fact, do we know we terminate?

Implementing RD

- Values: bits in a bit vector
- Gen: 1 in each position generated, otherwise 0
- Kill: 0 in each position killed, otherwise 1
- F_{bb}: out[b] = gen[b] | (in[b] & kill[b])
- Init in[b]=out[b]=0

- When are we done?
- What order to visit nodes? Does it matter?

RD Worklist algorithm

Initialize: in[B] = out[b] = $\varnothing$

Initialize: in[entry] = $\varnothing$

Work queue, W = all Blocks in topological order

while (|W| != 0) {

    remove b from W

    old = out[b]

    in[b] = {over all pred(p) $\in$ b} $\cup$ out[p]

    out[b] = gen[b] $\cup$ (in[b] - kill[b])

    if (old != out[b]) W = W $\cup$ succ(b)

}

Storing Rd information

- Use-def chains: for each use of var x in s, a list of definitions of x that reach s

```
1:  n <- 10                  1
2:  older <- 0               1             1,2
3:  old <- 1                 1,2           1,2,3
4:  result <- 0              1-3           1-4
5:  if n <= 1 goto 14        1-4           1-4
6:  i <- 2                   1-4           1-4,6
7:  if i > n goto 13         1-4,6,8-11    1-4,6,8-11
8:  result <- old + older    1-4,6,8-11    1-3,6,8-11
9:  older <- old             1-3,6,8-11    1,3,6,8-11
10: old <- result            1,3,6,8-11    1,6,8-11
11: i <- i + 1               1,6,8-11      1,8-11
12: JUMP 7                   1,8-11        1,8-11
13: return result            1-4,6         1-4,6
14: return n                 1-4           1-4
```

Storing Rd information

- Use-def chains: for each use of var x in s, a list of definitions of x that reach s

`1: n <- 10`	1	
`2: older <- 0`	1	1,2
`3: old <- 1`	1,2	1,2,3
`4: result <- 0`	1-3	1-4
`5: if n <= 1 goto 14`	1-4	1-4
`6: i <- 2`	1-4	1-4,6
`7: if i > n goto 13`	1-4,6,8-11	1-4,6,8-11
`8: result <- old + older`	1-4,6,8-11	1-3,6,8-11
`9: older <- old`	1-3,6,8-11	1,3,6,8-11
`10: old <- result`	1,3,6,8-11	1,6,8-11
`11: i <- i + 1`	1,6,8-11	1,8-11
`12: JUMP 7`	1,8-11	1,8-11
`13: return result`	1-4,6	1-4,6
`14: return n`	1-4	1-4

Storing Rd information

- Use-def chains: for each use of var x in s, a list of definitions of x that reach s

```
1:  n <- 10                         1
2:  older <- 0                      1             1,2
3:  old <- 1                        1,2           1,2,3
4:  result <- 0                     1-3           1-4
5:  if n <= 1 goto 14               1-4           1-4
6:  i <- 2                          1-4           1-4,6
7:  if i > n goto 13                1-4,6,8-11    1-4,6,8-11
8:  result <- old + older           1-4,6,8-11    1-3,6,8-11
9:  older <- old                    1-3,6,8-11    1,3,6,8-11
10: old <- result                   1,3,6,8-11    1,6,8-11
11: i <- i + 1                      1,6,8-11      1,8-11
12: JUMP 7                          1,8-11        1,8-11
13: return result                   1-4,6         1-4,6
14: return n                        1-4           1-4
```

Storing Rd information

- Use-def chains: for each use of var x in s, a list of definitions of x that reach s

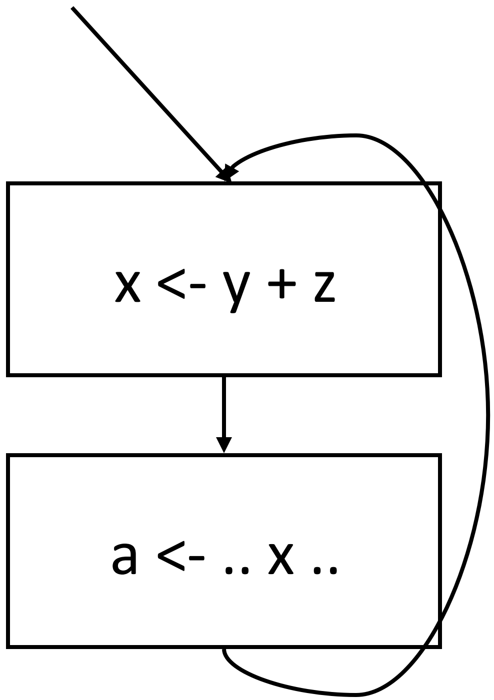
1: n <- 10	1	
2: older <- 0	1	1,2
3: old <- 1	1,2	1,2,3
4: result <- 0	1-3	1-4
5: if n <= 1 goto 14	1-4	1-4
6: i <- 2	1-4	1-4,6
7: if i > n goto 13	1-4,6,8-11	1-4,6,8-11
8: result <- old + older	1-4,6,8-11	1-3,6,8-11
9: older <- old	1-3,6,8-11	1,3,6,8-11
10: old <- result	1,3,6,8-11	1,6,8-11
11: i <- i + 1	1,6,8-11	1,8-11
12: JUMP 7	1,8-11	1,8-11
13: return result	1-4,6	1-4,6
14: return n	1-4	1-4

Constant Folding + DCE

~~1: n <- 10~~	~~1~~	
2: older <- 0	1	1,2
3: old <- 1	1,2	1,2,3
4: result <- 0	1-3	1-4
~~5: if 10 <= 1 goto 14~~	~~1-4~~	~~1-4~~
6: i <- 2	1-4	1-4,6
7: if i > 10 goto 13	1-4,6,8-11	1-4,6,8-11
8: result <- old + older	1-4,6,8-11	1-3,6,8-11
9: older <- old	1-3,6,8-11	1,3,6,8-11
10: old <- result	1,3,6,8-11	1,6,8-11
11: i <- i + 1	1,6,8-11	1,8-11
12: JUMP 7	1,8-11	1,8-11
13: return result	1-4,6	1-4,6
~~14: return 10~~	~~1-4~~	~~1-4~~

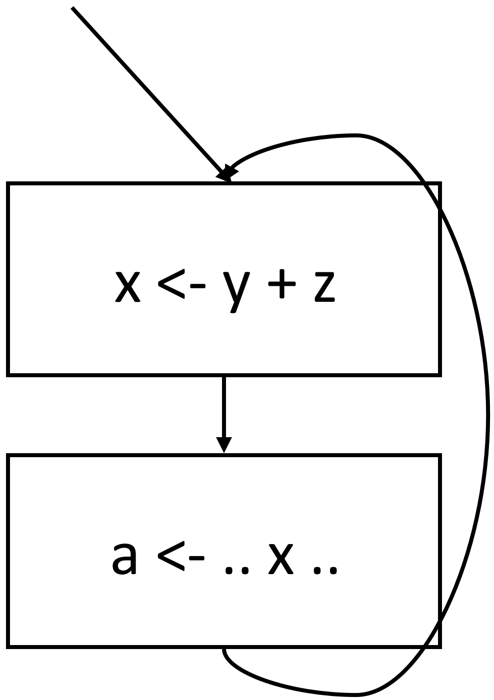
Loop Invariant Code Motion

- When can expression be moved out of a loop?



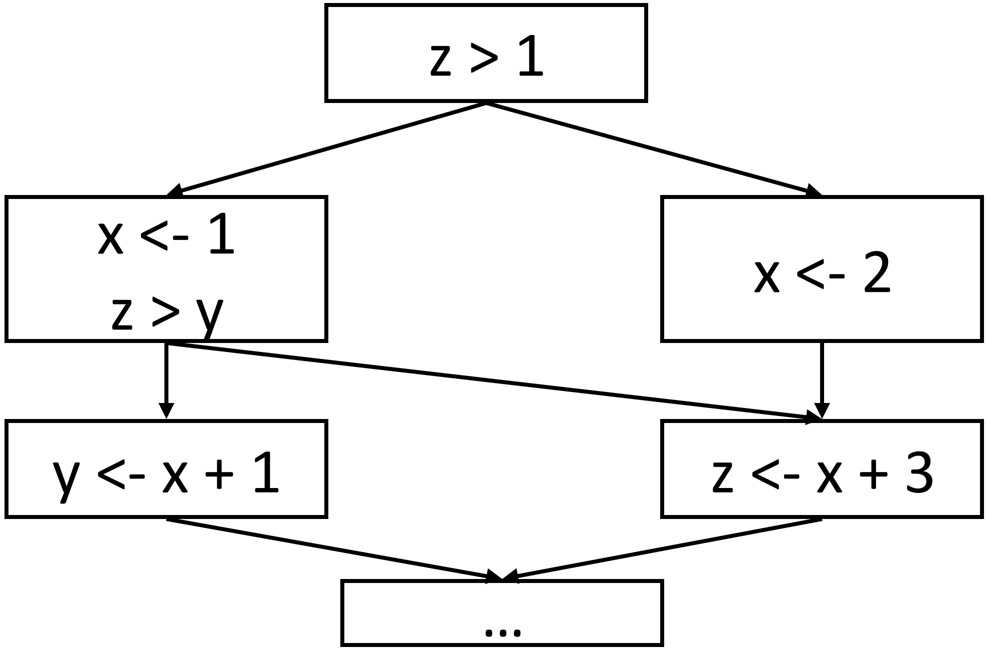
Loop Invariant Code Motion

- When can expression be moved out of a loop?
- When all reaching definitions of operands are outside of loop, expression is loop invariant
- Use ud-chains to detect



Def-use chains are valuable too

- Def-use chain: for each definition of var x, a list of all uses of that definition
- Computed from liveness analysis, a backward dataflow problem
- Def-use is NOT symmetric to use-def



Liveness (def-use chains)

- A variable x is live-out of a stmt s if x can be used along some path starting a s, otherwise x is dead.

Liveness as a dataflow problem

- This is a backwards analysis
  - A variable is live out if used by a successor
  - Gen: For a use: indicate it is live coming into s
  - Kill: Defining a variable v in s makes it dead before s (unless s uses v to define v)
  - Lattice is just live (top) and dead (bottom)
- Values are variables
- In[n] = variables live before n

$$= \left(out[n] - kill[n]\right) \cup gen[n]$$

- Out[n] = variables live after n

$$= \bigcup_{s \in succ(n)} In[s]$$

Dead Code Elimination

- Code is dead if it has no effect on the outcome of the program.
- When is code dead?

return

$x \leftarrow y \,\theta\, z$

Dead Code Elimination

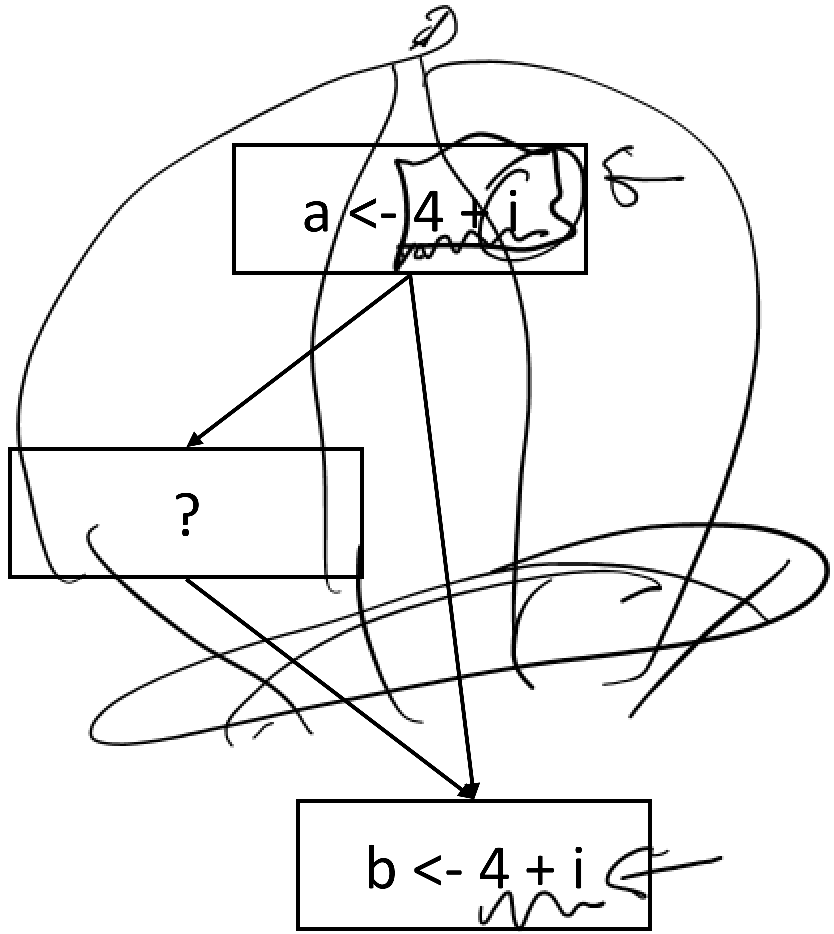
- Code is dead if it has no effect on the outcome of the program.
- When is code dead?
  - When the definition is dead, and
  - When the instruction has no side effects
- So:
  - run liveness
  - Construct def-use chains
  - Any instruction which has no users and has no side effects can be eliminated

So Far

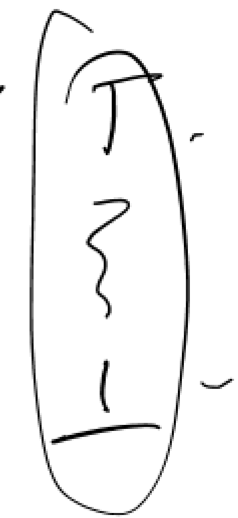
unified framework DF

	Union (may)	intersection (must)
Forward	Reaching defs	
Backward	Live variables	

When can we do CSE?



4+j



Available Expressions

- X+Y is “available” at statement S if
  - x+y is computed along every path from the start to S AND
  - neither x nor y is modified after the last evaluation of x+y

```
a <- b+c
b <- a-d
c <- b+c    not avai)
d <- a-d
```

Available Expressions

- X+Y is "available" at statement S if
  - x+y is computed along every path from the start to S AND
  - neither x nor y is modified after the last evaluation of x+y

a <- b+c

b <- a-d

c <- b+c ⟵ b+c Not available, since b redefined

d <- a-d ⟵ a-d is available

Computing Available Expressions

- Forward or backward?
- Values? exprs
- Lattice? $\top$ … $\bot$ ~~sets~~ powerset (all exprs)
- gen[b] = $\cancel{x \leftarrow}$ $\boxed{a+b}$
- kill[b] = kill all exprs that involve x
- out[b] = $gen[b] \cup (IN[b] - kill[b])$
- in[b] = $\bigcap_{p \in pred(b)} out[p]$
- initialization?

$out[b] = \{\}$

$In[m] = \bigcap_{p \in pred(m)} out[p]$

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- gen[b] = if b evals expr e and doesn't
  define variables used in e
- kill[b] = if b assigns to x,
  then all exprs using x are killed.
- out[b] = (in[b] – kill[b]) $\cup$ gen[b]
- in[b] = what to do at a join point?
- initialization?

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- gen[b] = if b evals expr e and doesn't define variables used in e
- kill[b] = if b assigns to x, exprs(x) are killed
- out[b] = (in[b] – kill[b]) $\cup$ gen[b]
- in[b] = An expr is avail only if avail on ALL edges, so: in[b] = $\cap$ over all p$\in$ pred(b), out[p]
- Initialization
  - All nodes, but entry are set to ALL avail
  - Entry is set to NONE avail

Constructing Gen & Kill

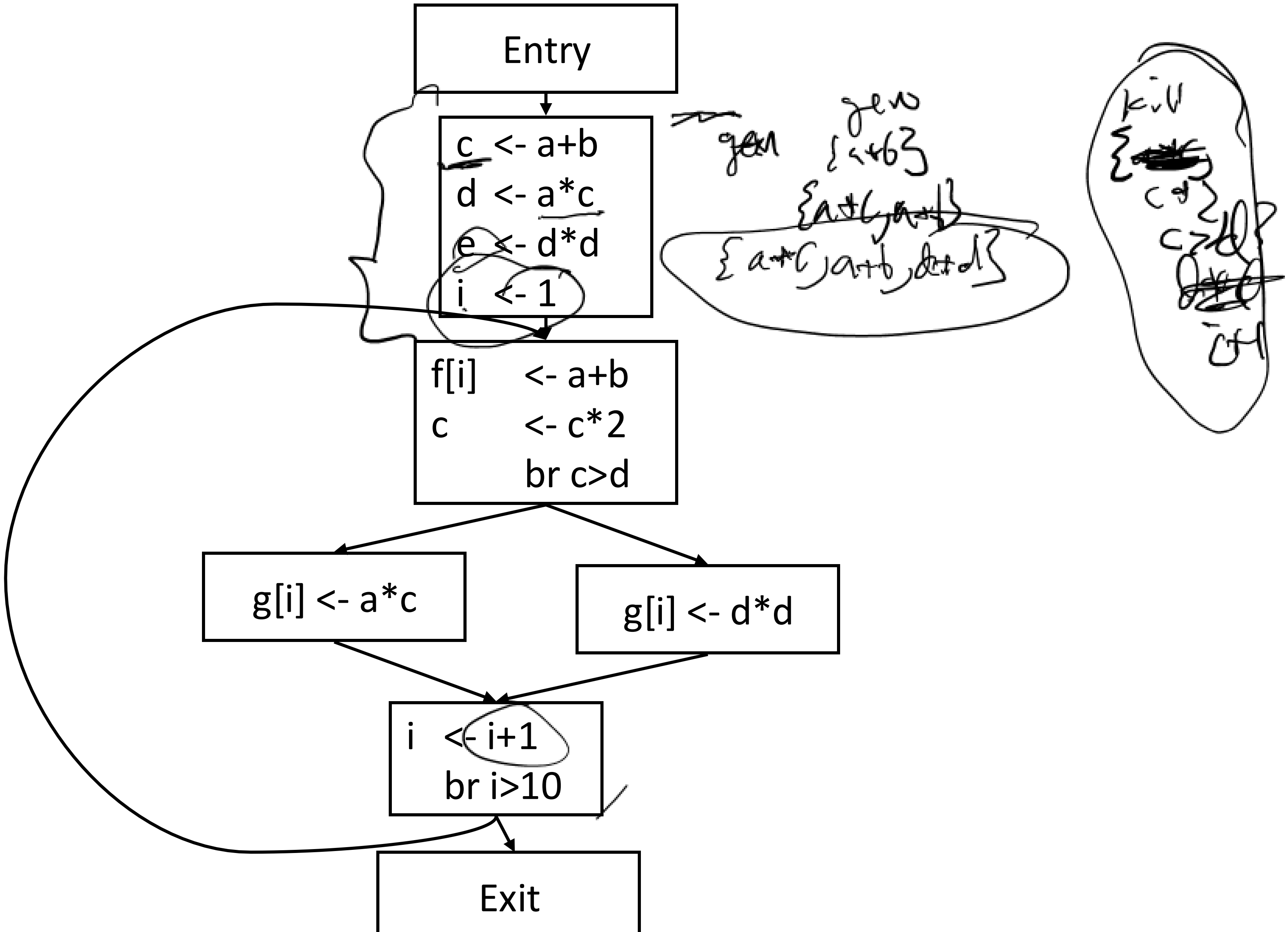
Stmt s	Gen	Kill
t <- x op y	{x op y}-kill[s]	{exprs containing t}
t <- M[a]	{M[a]}-kill[s]	{ " " " }
M[a] <- b	{}	{M[x] ∀x}
f(a, …)	{} ~~f(a, …)~~	{M[x] for all x}
t <- f(a,…)	{}	" & expr contnt

Constructing Gen & Kill

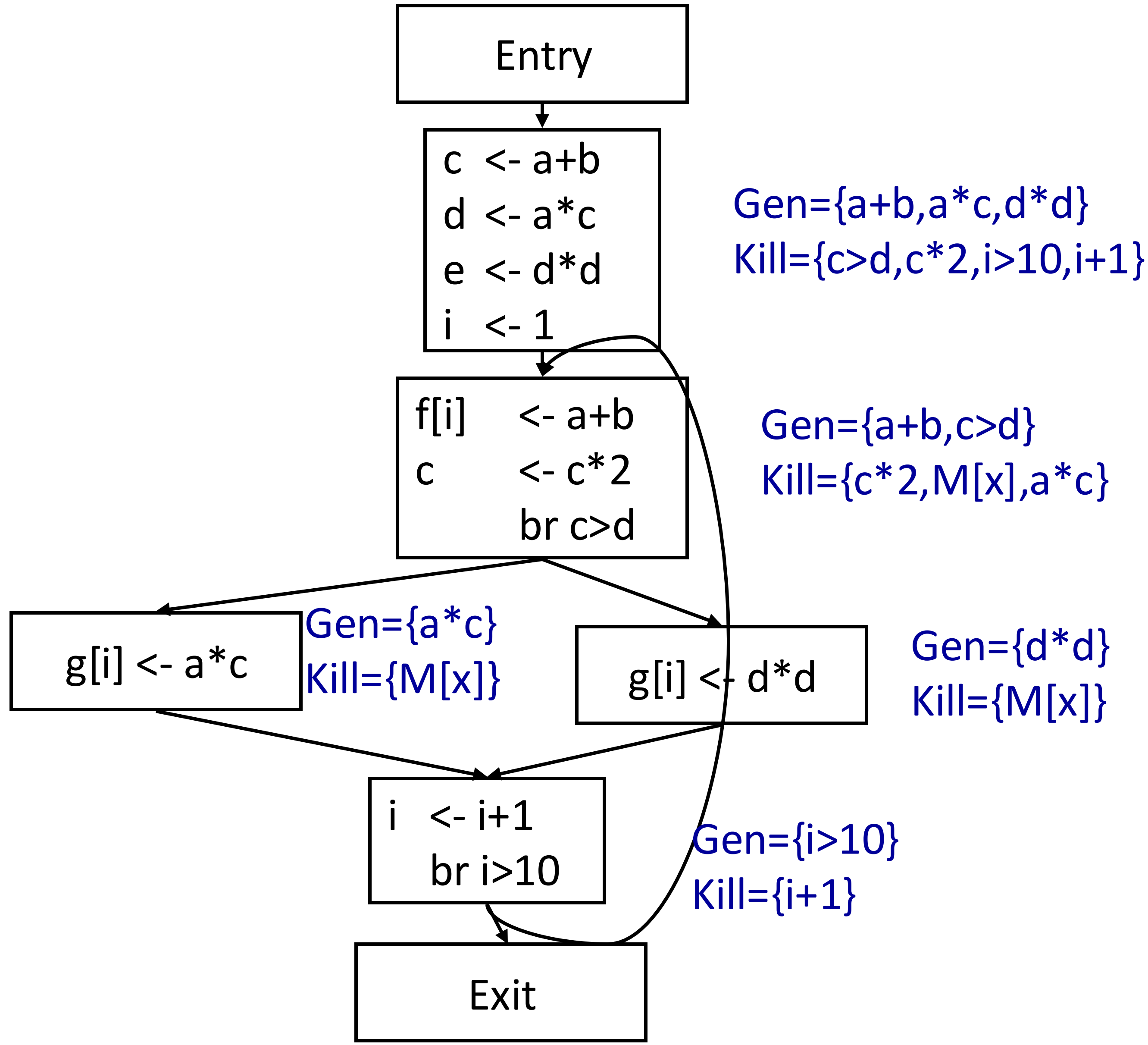
Stmt s	Gen	Kill
t <- x op y	{x op y}-kill[s]	{exprs containing t}
t <- M[a]	{M[a]}-kill[s]	{exprs containing t}
M[a] <- b	{}	{for all x, M[x]}
f(a, …)	{}	{for all x, M[x]}
t <- f(a,…)	{}	{exprs containing t for all x, M[x]}

Example

$$out[b] = (in[b] - kill[b]) \cup gen[b]$$
$$in[b] = \cap \text{ over } p \in pred(b),\ out[p]$$

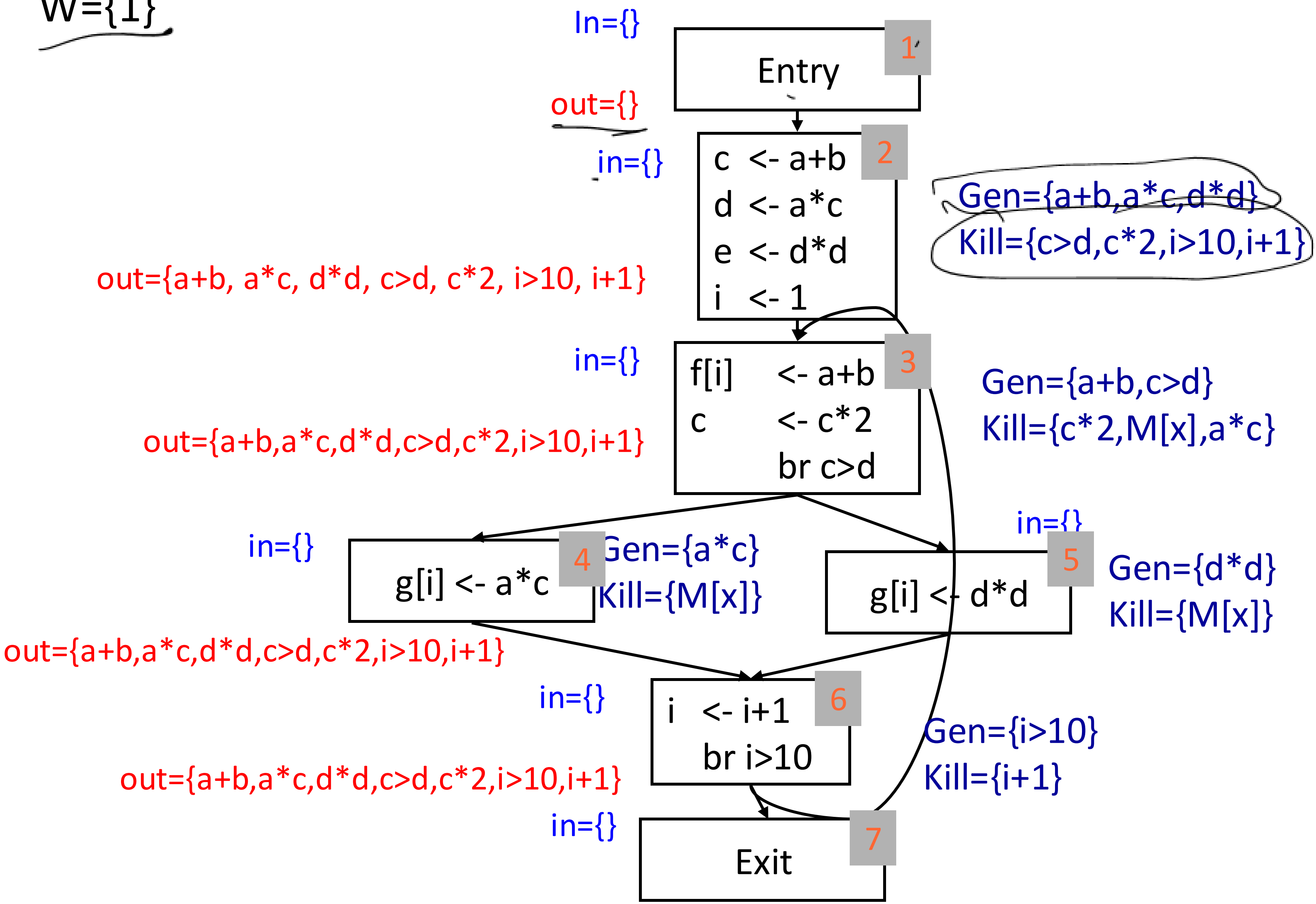


Example


Entry
c <- a+b
d <- a*c
e <- d*d
i <- 1
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
f[i] <- a+b
c <- c*2
br c>d
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
g[i] <- a*c
Gen={a*c}
Kill={M[x]}
g[i] <- d*d
Gen={d*d}
Kill={M[x]}
i <- i+1
br i>10
Gen={i>10}
Kill={i+1}
Exit

Example

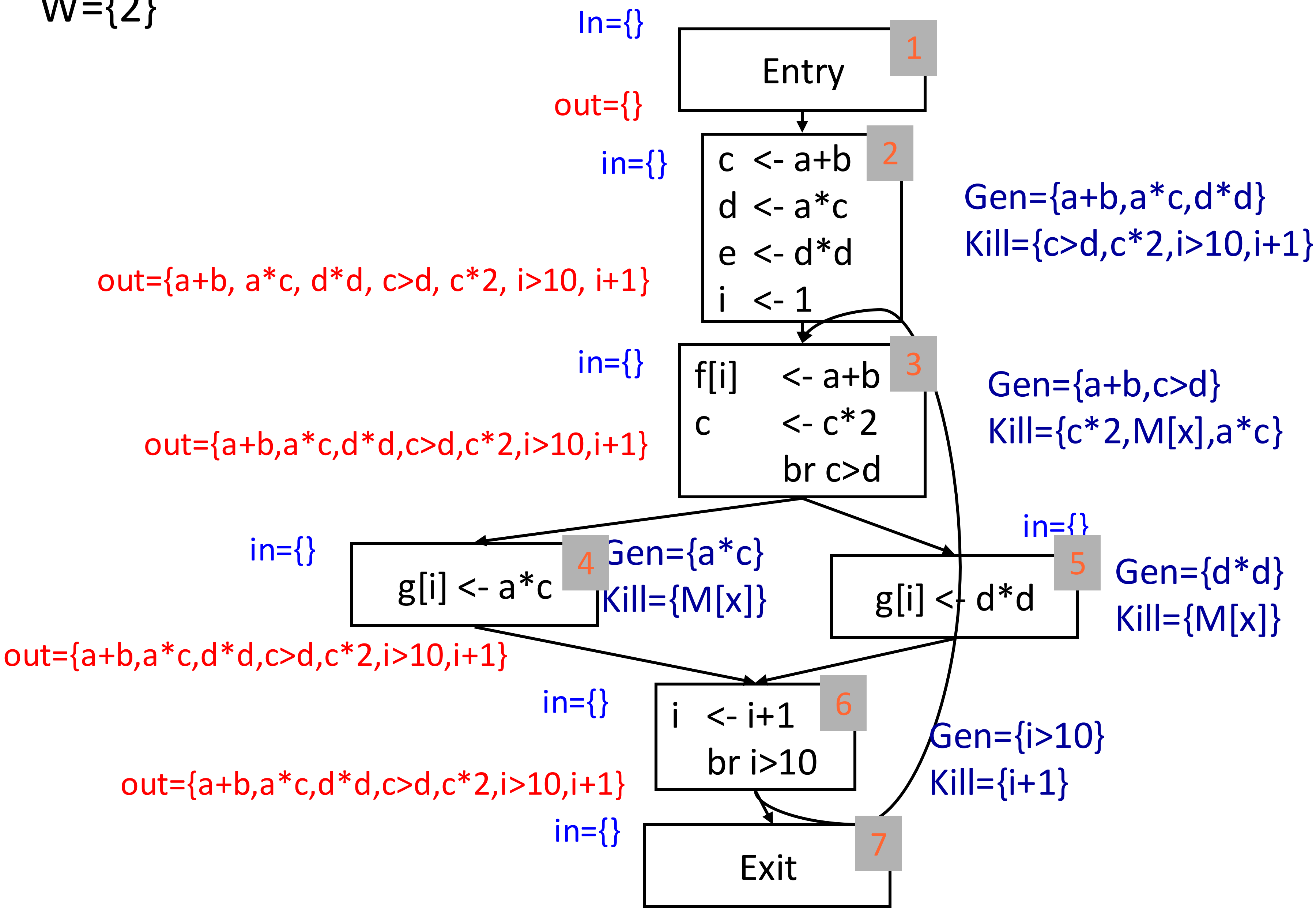
W={1}



In={}
Entry
1
out={}
in={}
c <- a+b
d <- a*c
e <- d*d
i <- 1
2
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d, c>d, c*2, i>10, i+1}
in={}
f[i] <- a+b
c <- c*2
br c>d
3
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
g[i] <- a*c
4
Gen={a*c}
Kill={M[x]}
in={}
g[i] <- d*d
5
Gen={d*d}
Kill={M[x]}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
i <- i+1
br i>10
6
Gen={i>10}
Kill={i+1}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
Exit
7

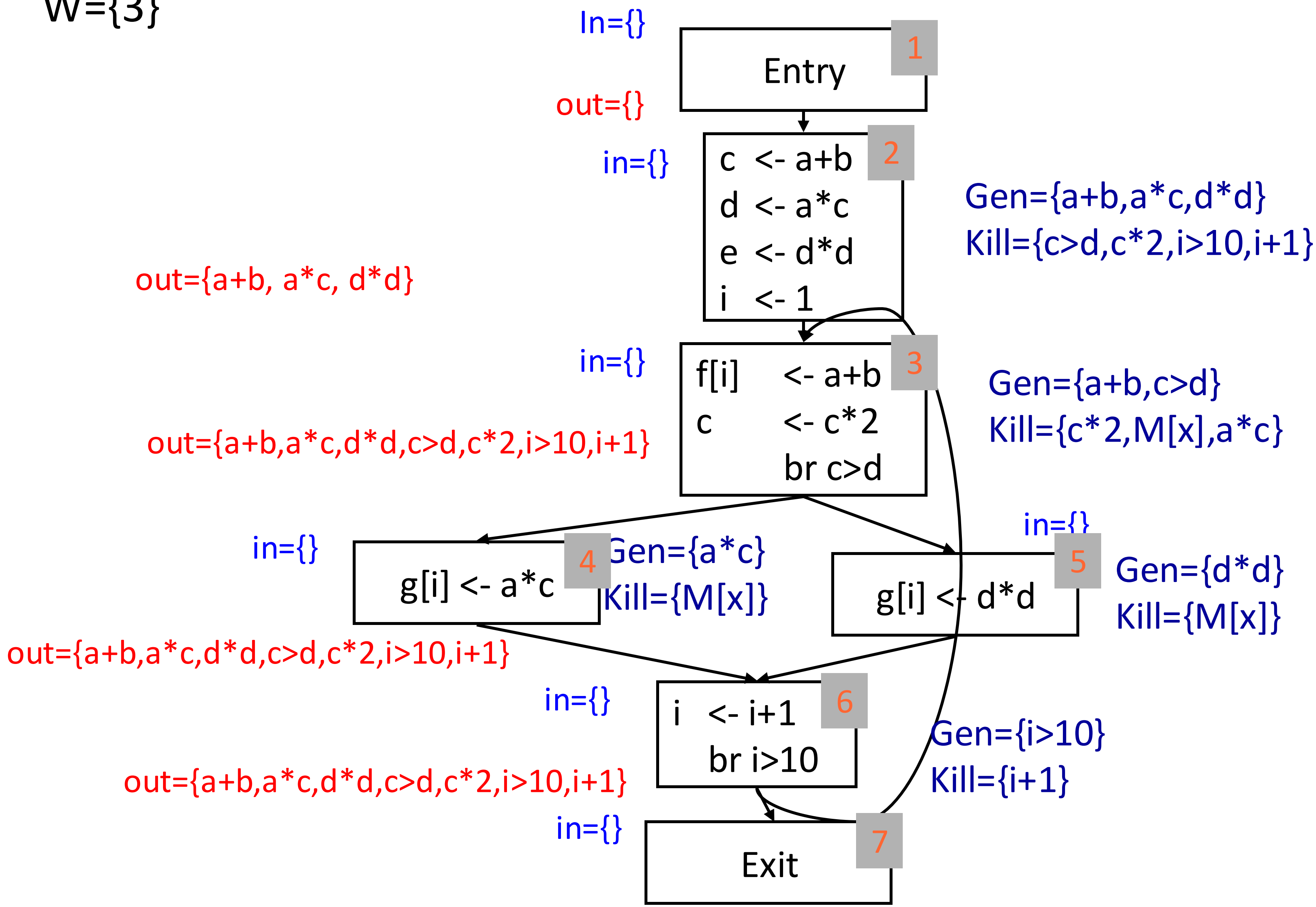
Example

W={2}


In={}
Entry
1
out={}
in={}
c <- a+b
d <- a*c
e <- d*d
i <- 1
2
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d, c>d, c*2, i>10, i+1}
in={}
f[i] <- a+b
c <- c*2
br c>d
3
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
g[i] <- a*c
4
Gen={a*c}
Kill={M[x]}
in={}
g[i] <- d*d
5
Gen={d*d}
Kill={M[x]}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
i <- i+1
br i>10
6
Gen={i>10}
Kill={i+1}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
Exit
7

Example

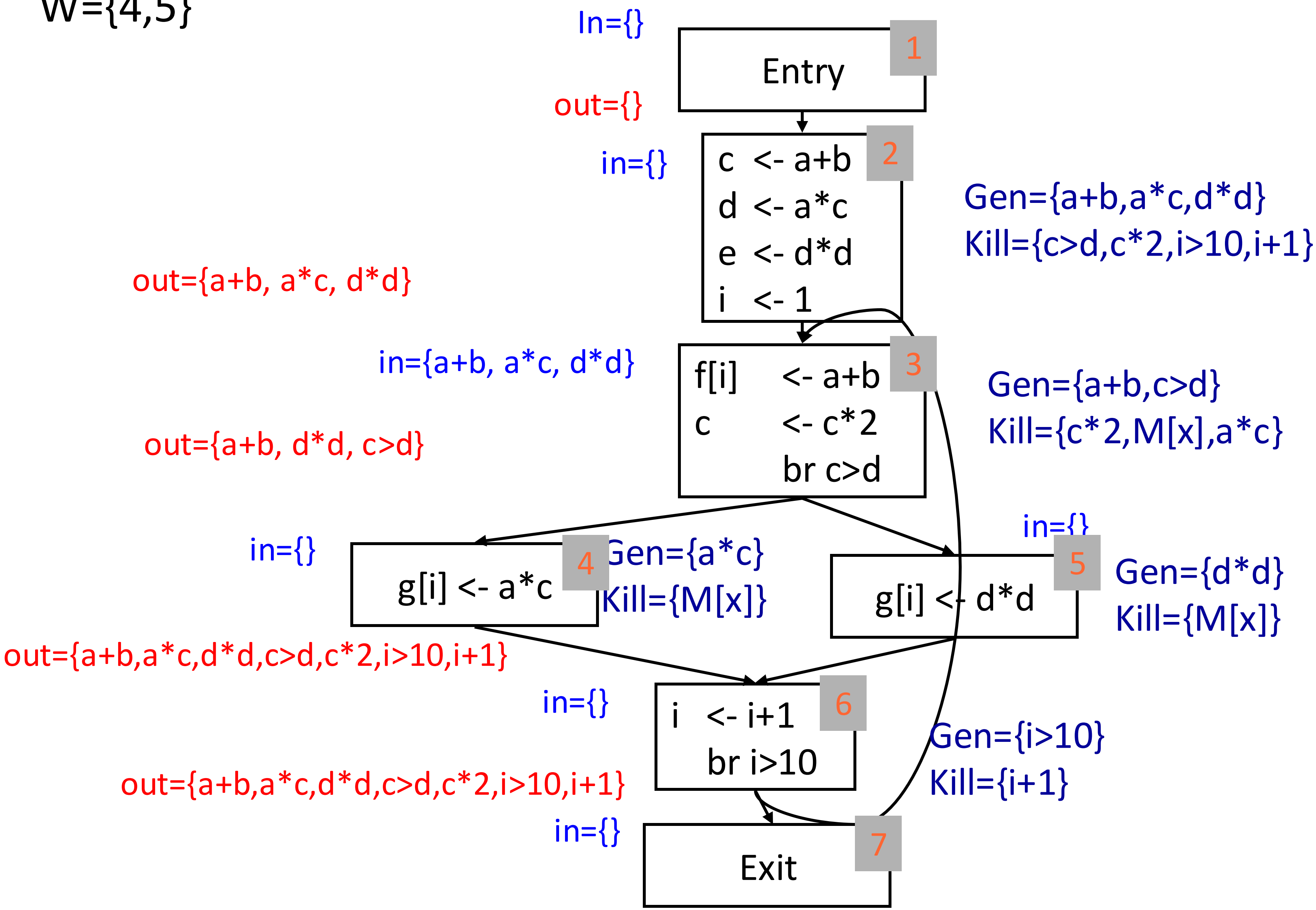
W={3}



In={}
Entry
1
out={}
in={}
c <- a+b
d <- a*c
e <- d*d
i <- 1
2
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d}
in={}
f[i] <- a+b
c <- c*2
br c>d
3
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
g[i] <- a*c
4
Gen={a*c}
Kill={M[x]}
in={}
g[i] <- d*d
5
Gen={d*d}
Kill={M[x]}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
i <- i+1
br i>10
6
Gen={i>10}
Kill={i+1}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
Exit
7

Example

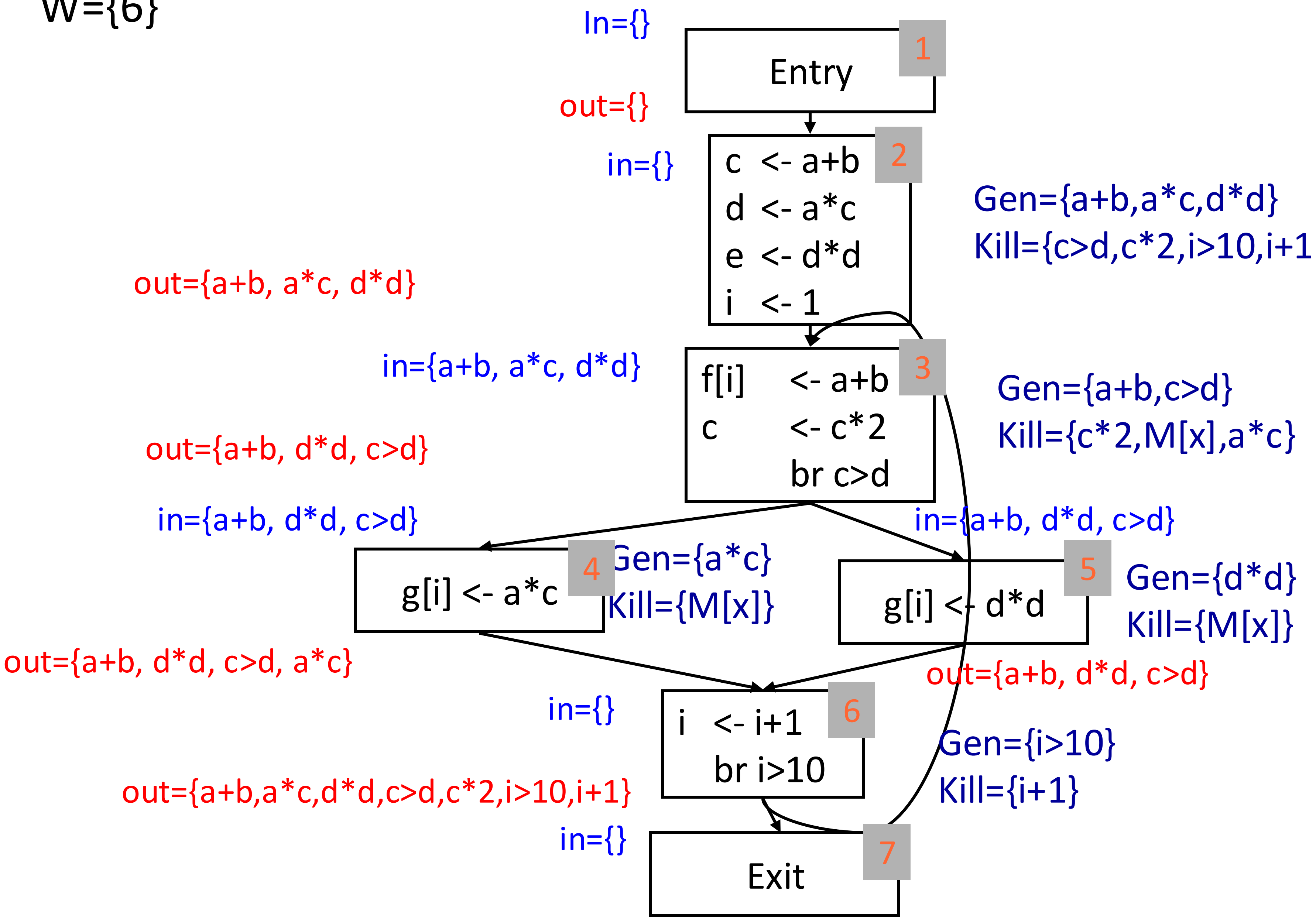
W={4,5}



In={}
1
Entry
out={}
in={}
2
c <- a+b
d <- a*c
e <- d*d
i <- 1
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d}
in={a+b, a*c, d*d}
3
f[i] <- a+b
c <- c*2
br c>d
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b, d*d, c>d}
in={}
4
g[i] <- a*c
Gen={a*c}
Kill={M[x]}
in={}
5
g[i] <- d*d
Gen={d*d}
Kill={M[x]}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
6
i <- i+1
br i>10
Gen={i>10}
Kill={i+1}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
7
Exit

Example

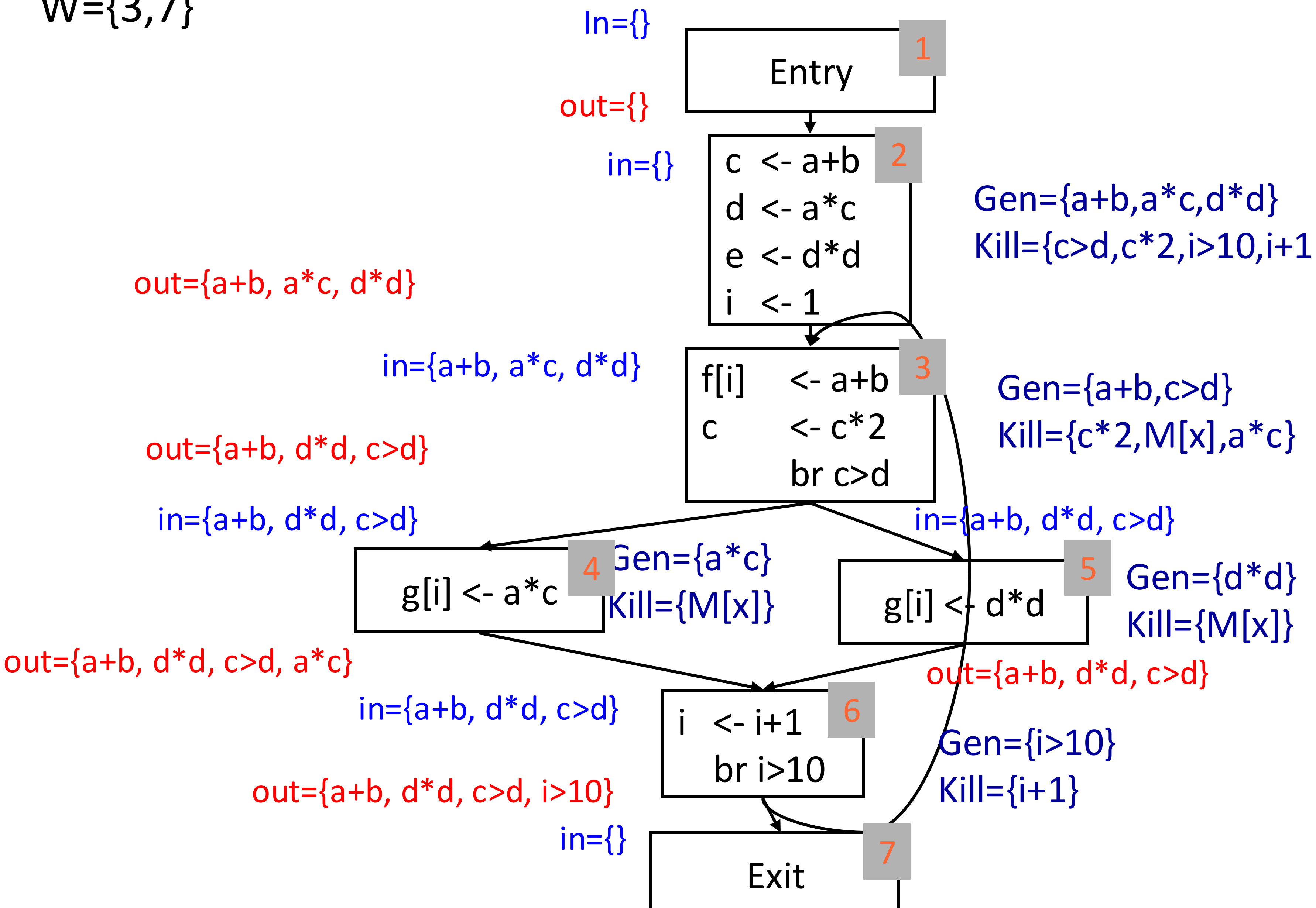
W={6}



In={}
Entry
1
out={}
in={}
c <- a+b
d <- a*c
e <- d*d
i <- 1
2
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d}
in={a+b, a*c, d*d}
f[i] <- a+b
c <- c*2
br c>d
3
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b, d*d, c>d}
in={a+b, d*d, c>d}
in={a+b, d*d, c>d}
g[i] <- a*c
4
Gen={a*c}
Kill={M[x]}
g[i] <- d*d
5
Gen={d*d}
Kill={M[x]}
out={a+b, d*d, c>d, a*c}
out={a+b, d*d, c>d}
in={}
i <- i+1
br i>10
6
Gen={i>10}
Kill={i+1}
out={a+b,a*c,d*d,c>d,c*2,i>10,i+1}
in={}
Exit
7

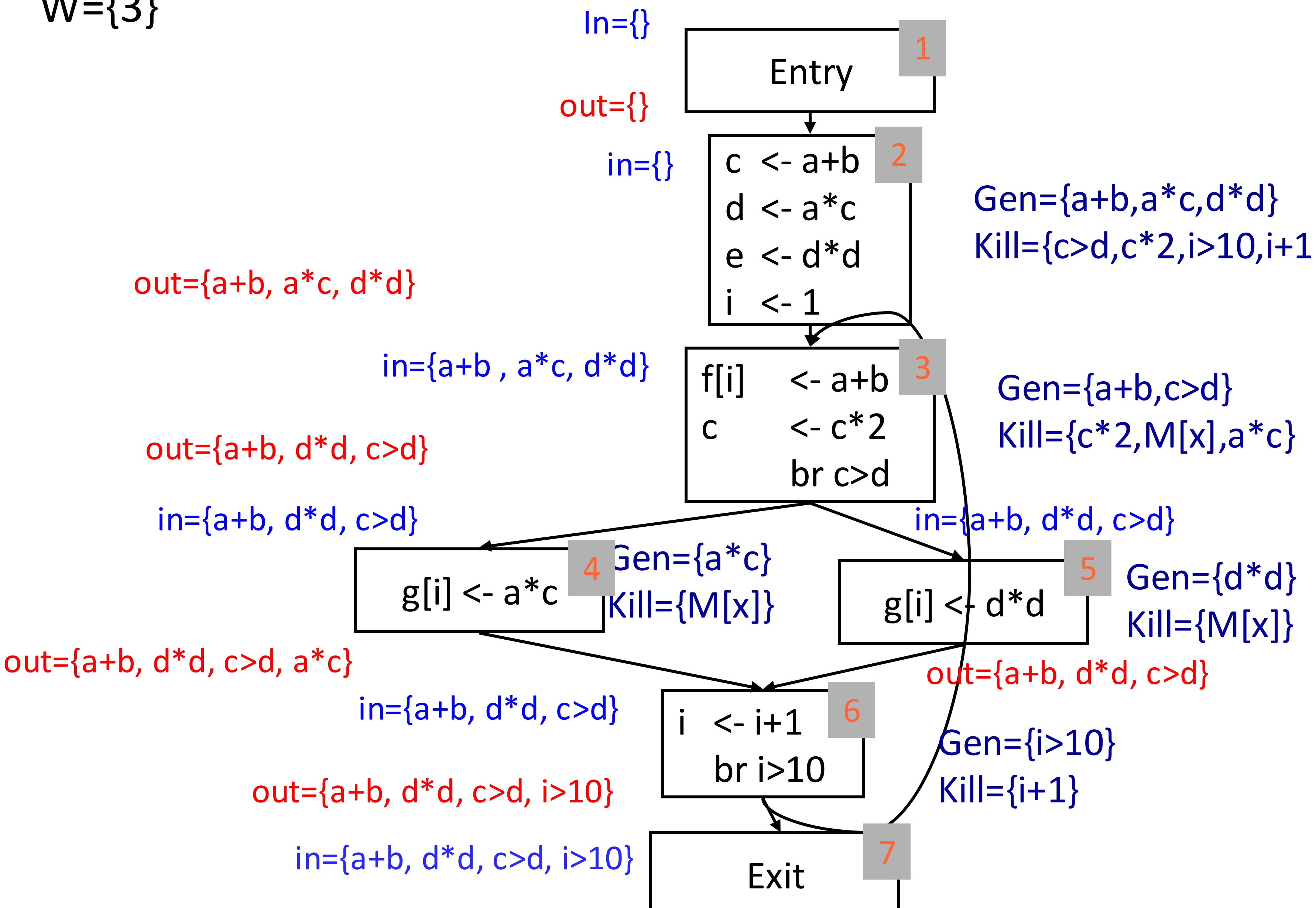
Example

W={3,7}

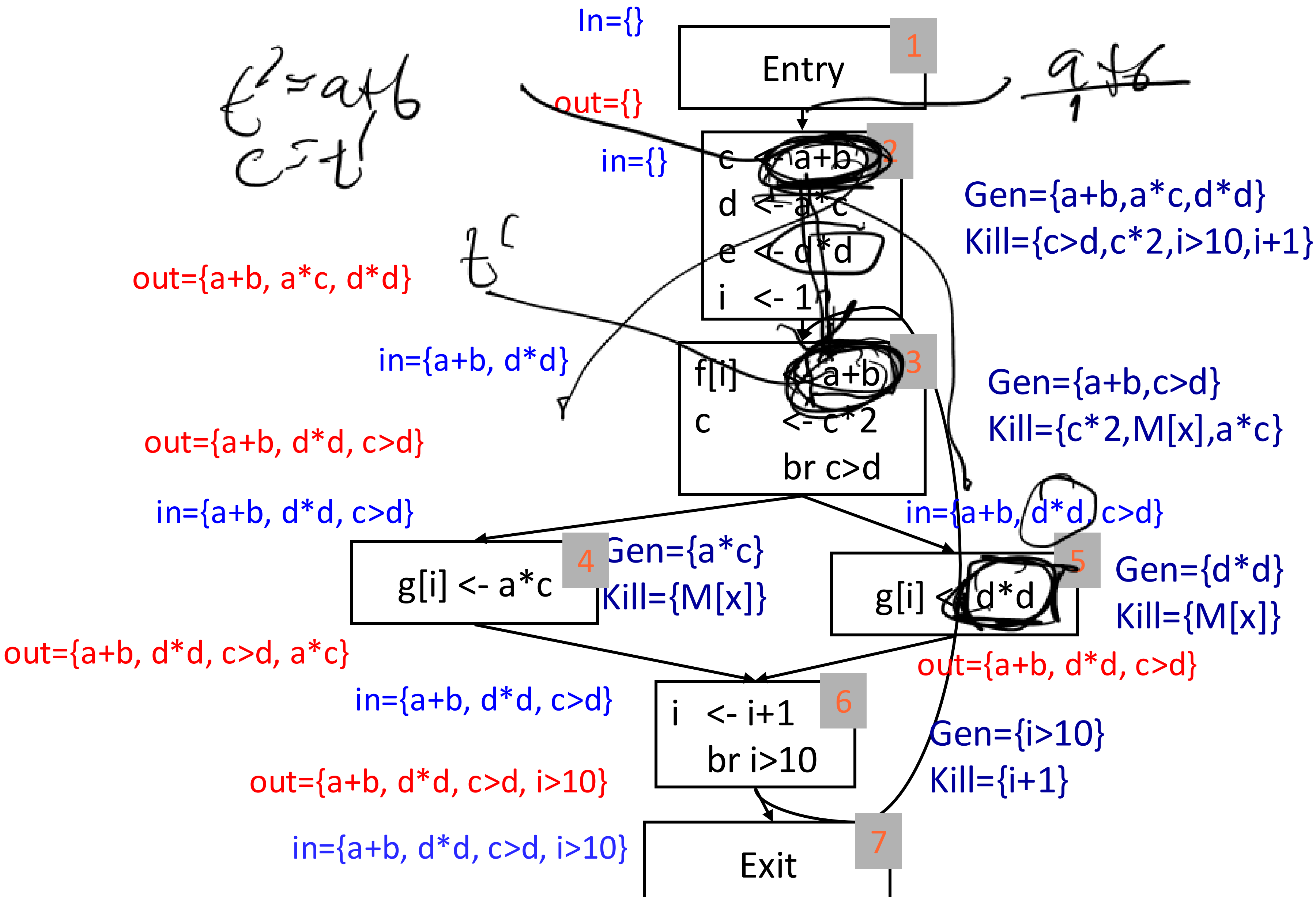

In={}
Entry
1
out={}
in={}
c <- a+b
d <- a*c
e <- d*d
i <- 1
2
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d}
in={a+b, a*c, d*d}
f[i] <- a+b
c <- c*2
br c>d
3
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b, d*d, c>d}
in={a+b, d*d, c>d}
g[i] <- a*c
4
Gen={a*c}
Kill={M[x]}
in={a+b, d*d, c>d}
g[i] <- d*d
5
Gen={d*d}
Kill={M[x]}
out={a+b, d*d, c>d, a*c}
out={a+b, d*d, c>d}
in={a+b, d*d, c>d}
i <- i+1
br i>10
6
Gen={i>10}
Kill={i+1}
out={a+b, d*d, c>d, i>10}
in={}
Exit
7

Example

W={3}


In={}
Entry
1
out={}
in={}
c <- a+b
d <- a*c
e <- d*d
i <- 1
2
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d}
in={a+b , a*c, d*d}
f[i] <- a+b
c <- c*2
br c>d
3
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b, d*d, c>d}
in={a+b, d*d, c>d}
in={a+b, d*d, c>d}
g[i] <- a*c
4
Gen={a*c}
Kill={M[x]}
g[i] <- d*d
5
Gen={d*d}
Kill={M[x]}
out={a+b, d*d, c>d, a*c}
out={a+b, d*d, c>d}
in={a+b, d*d, c>d}
i <- i+1
br i>10
6
Gen={i>10}
Kill={i+1}
out={a+b, d*d, c>d, i>10}
in={a+b, d*d, c>d, i>10}
Exit
7

Example


In={}
Entry
1
out={}
in={}
c <- a+b
d <- a*c
e <- d*d
i <- 1
2
Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}
out={a+b, a*c, d*d}
in={a+b, d*d}
f[i] <- a+b
c <- c*2
br c>d
3
Gen={a+b,c>d}
Kill={c*2,M[x],a*c}
out={a+b, d*d, c>d}
in={a+b, d*d, c>d}
g[i] <- a*c
4
Gen={a*c}
Kill={M[x]}
in={a+b, d*d, c>d}
g[i] <- d*d
5
Gen={d*d}
Kill={M[x]}
out={a+b, d*d, c>d, a*c}
out={a+b, d*d, c>d}
in={a+b, d*d, c>d}
i <- i+1
br i>10
6
Gen={i>10}
Kill={i+1}
out={a+b, d*d, c>d, i>10}
in={a+b, d*d, c>d, i>10}
Exit
7
t' = a+b
c = t'
t'
~~a+b~~

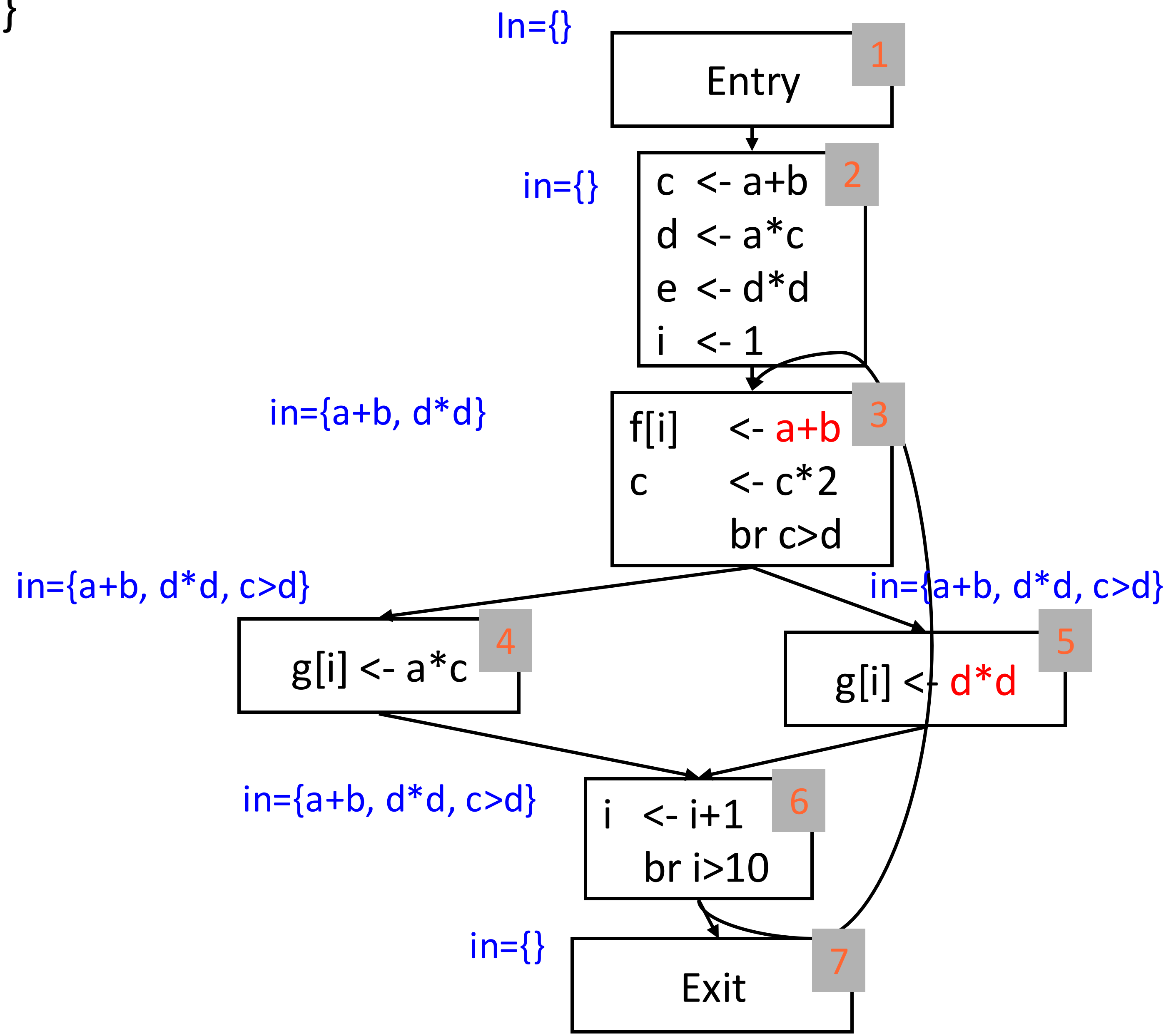
CSE

- Calculate Available expressions
- For every stmt in program

```
If expression, x op y, is available {
    Compute reaching expressions for "x op y"
        at this stmt
    foreach stmt in RE of the form t <- x op y
            rewrite at:   t' <- x op y
                          t <- t'
    }
    replace "x op y" in stmt with t'
}
```

Find x op y available

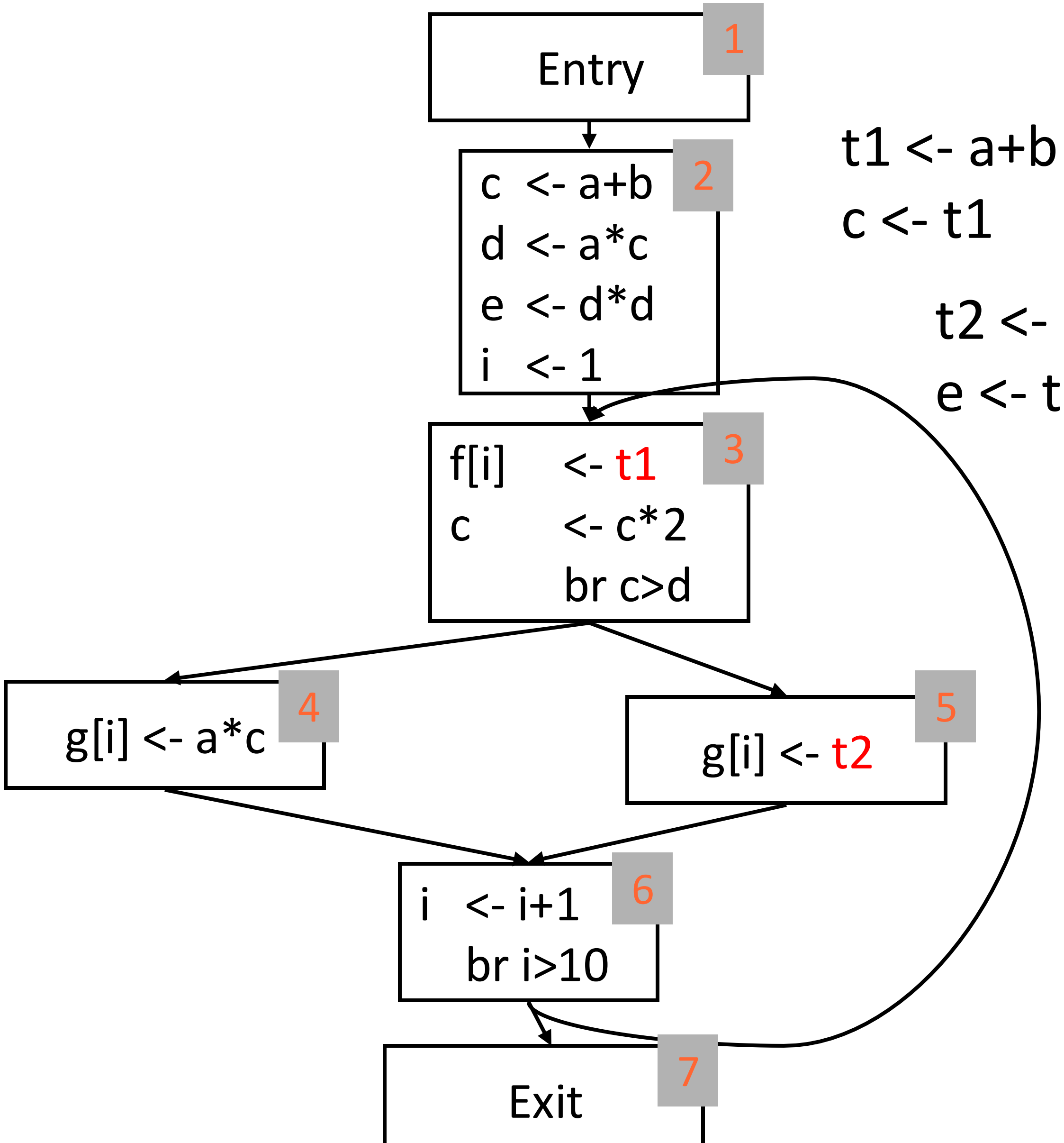
W={3}



Calculating Reaching Expressions

- Could be dataflow problem, but not needed enough, so …
- To find RE for “x op y” at stmt S
  - traverse cfg backward from S until
    - reach t <- x + y (& put into RE)
    - reach definition of x or y

Example



t1 <- a+b
c <- t1

t2 <- d*d
e <- t2

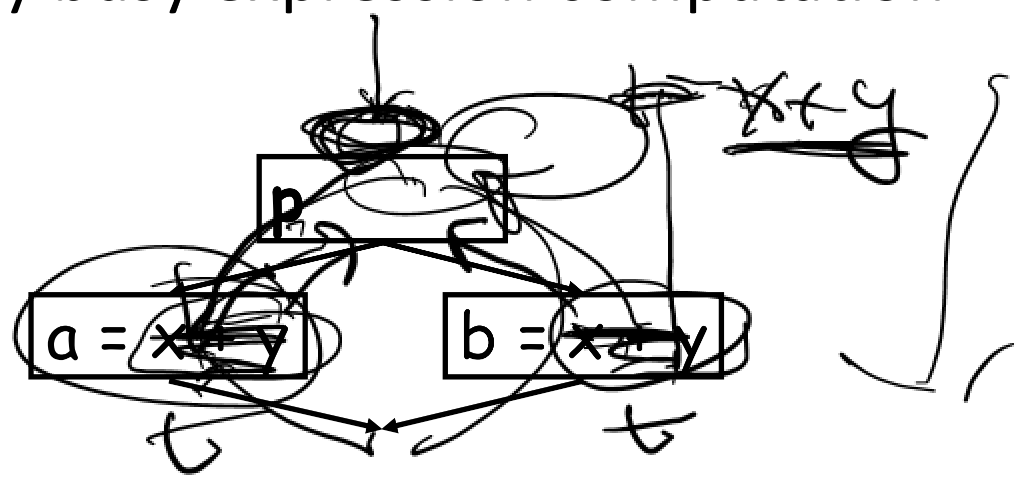
Dataflow Summary

	Union (may)	intersection (must)
Forward	Reaching defs	Available exprs
Backward	Live variables	very busy exprs

Later in course we look at bidirectional dataflow

Very Busy Expressions

- A Backward, Must data flow analysis
- An expression e is *very busy at point p* if On every path from p, e is evaluated before the value of e is changed
- Optimization
  - Can hoist very busy expression computation



Forward Must Data Flow Algorithm

Out(s) = Gen(s) for all statements s

W = {all statements}

Repeat

    Take s from W

    In(s) = $\bigcap_{s' \in \textbf{pred(s)}}$ Out(s')

    Temp = Gen(s) $\cup$ (In(s) – Kill(s))

    If (temp != Out(s)) {

        Out(s) = temp

        W = W $\cup$ succ(s)

    }

Until W = $\varnothing$

Forward May Data Flow Algorithm

Out(s) = Gen(s) for all statements s

W = {all statements}

Repeat

  Take s from W

  In(s) = $\cup_{s' \in pred(s)}$ Out(s')

  Temp = Gen(s) $\cup$ (In(s) – Kill(s))

  If (temp != Out(s)) {

    Out(s) = temp

    W = W $\cup$ succ(s)

  }

Until W = $\varnothing$

Backward May Data Flow Algorithm

In(s) = Gen(s) for all statements s

W = {all statements} (worklist)

Repeat

  Take s from W

  Out(s) = $\bigcup_{s' \in succ(s)}$ In(s')

  Temp = Gen(s) $\cup$ (Out(s) – Kill(s))

  If (temp != In(s)) {

    In(s) = temp

    W = W $\cup$ pred(s)

  }

Until W = $\varnothing$

Backward Must Data Flow Algorithm

In(s) = Gen(s) for all statements s

W = {all statements} (worklist)

Repeat

    Take s from W

    $\text{Out}(s) = \bigcap_{s' \in \text{succ}(s)} \text{In}(s')$

    $\text{Temp} = \text{Gen}(s) \cup (\text{Out}(s) - \text{Kill}(s))$

    If (temp != In(s)) {

        In(s) = temp

        $W = W \cup \text{pred}(s)$

    }

Until $W = \varnothing$