Using Program Analysis for Optimization

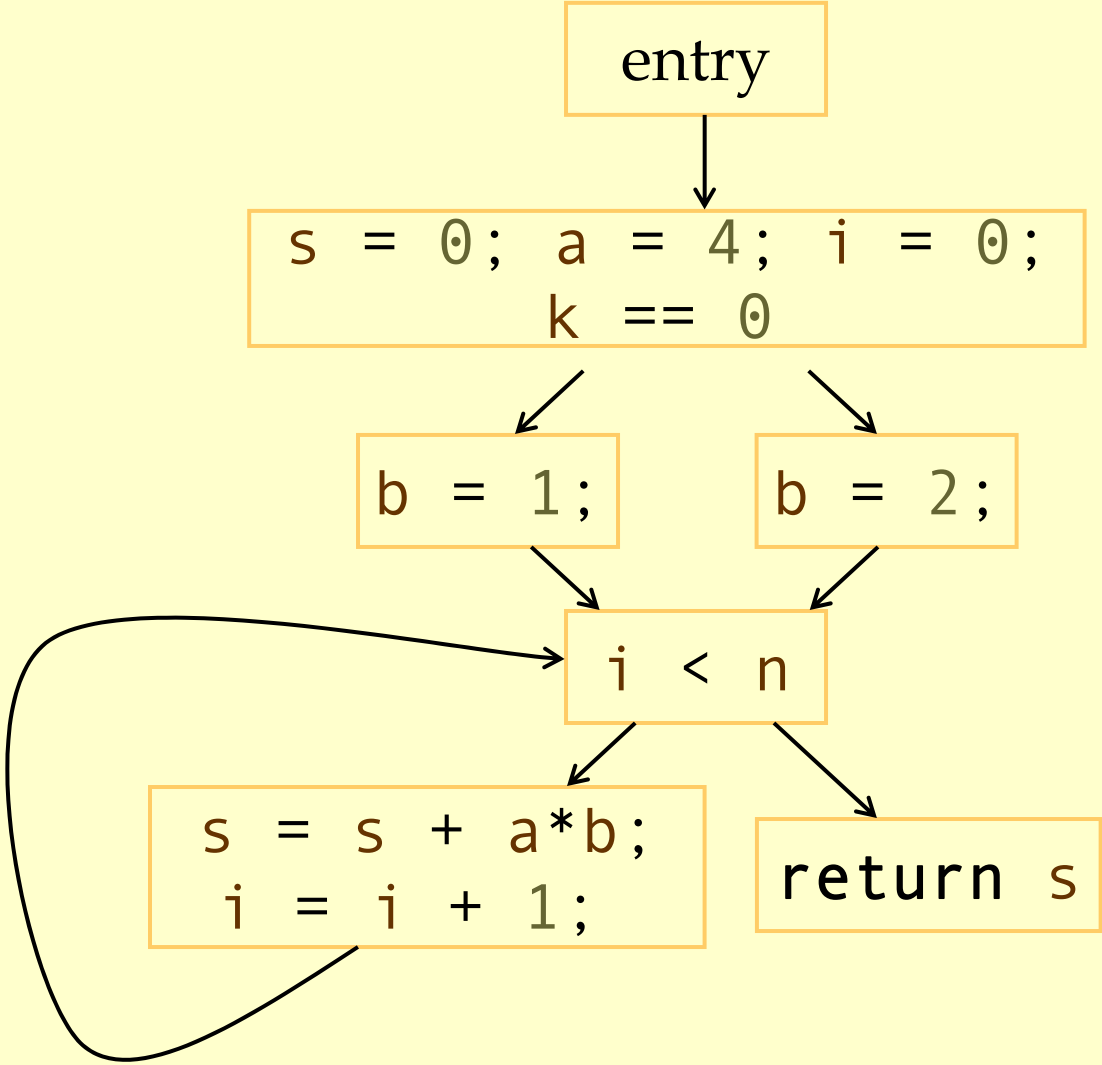
This lecture is primarily based on Konstantinos Sagonas set of slides
(**Advanced Compiler Techniques**, (2AD518)
at Uppsala University, January-February 2004).
Used with kind permission.

Analysis and Optimizations

- Program Analysis
  - Discover properties of a program.
- Optimizations
  - Use analysis results to transform the program.
  - Goal: improve some aspect of the program
    - number of executed instructions, number of cycles
    - cache hit rate
    - memory space (code or data)
    - power consumption
  - Has to be safe: Keep the semantics of the program.

Control Flow Graph

```
int add(n, k) {
  s = 0; a = 4; i = 0;
  if (k == 0)
      b = 1;
  else
      b = 2;
  while (i < n) {
      s = s + a*b;
      i = i + 1;
  }
  return s;
}
```



Control Flow Graph

- Nodes represent computation.
  - Each node is a Basic Block (BB).
  - Basic Block is a sequence of instructions with:
    - No branches out of middle of basic block.
    - No branches into middle of basic block.
    - Basic blocks should be maximal.
  - Execution of basic block starts with first instruction.
  - Includes all instructions in basic block.
- Edges represent control flow.

Two Kinds of Variables

- Temporaries (temps, a tmp):
  - Introduced by the compiler.
  - Transfer values only within basic block.
  - Introduced as part of instruction flattening.
  - Introduced by optimizations/transformations.
- Program variables (vars, a var):
  - Declared in original program.
  - May transfer values between basic blocks.

Basic Block Optimizations
(Local Optimizations)

- Common Sub-Expression Elimination (CSE)

```
a=(x+y)+z; b=x+y;
t=x+y; a=t+z; b=t;
```

- Constant Propagation

```
x=5; b=x+y;
b=5+y;
```

- Algebraic Simplification

```
a=x*1;
a=x;
```

- Copy Propagation

```
a=x+y; b=a; c=b+z;
a=x+y; b=a; c=a+z;
```

- Dead Code Elimination

```
a=x+y; b=a; c=a+z;
a=x+y; c=a+z
```

- Strength Reduction

```
t=i*4;
t=i<<2;
```

Value Numbering

- Normalize BB so that all statements are of the form:
  - var = var op var (where op is a binary operator)
  - var = op var (where op is a unary operator)
  - var = var
    (I.E., no complex statements like x=a+b*c.)
- Simulate execution of basic block:
  - Assign a virtual value to each variable.
  - Assign a virtual value to each expression.
  - Assign a temporary variable to hold value of each computed expression.

Value Numbering for CSE

As we simulate execution of program, generate a new version of program:

- Each new value assigned to temporary
  `a=x+y;` becomes
  `a=x+y;` t_1`=a;`
- Temporary preserves value for use later in program even if original variable rewritten
  `a=x+y; a=a+z;` becomes
  `a=x+y;` t_1`=a; a=a+z;` t_2`=a;`

CSE Example

- Original

```
a=x+y
b=a+z
b=b+y
c=a+z
```

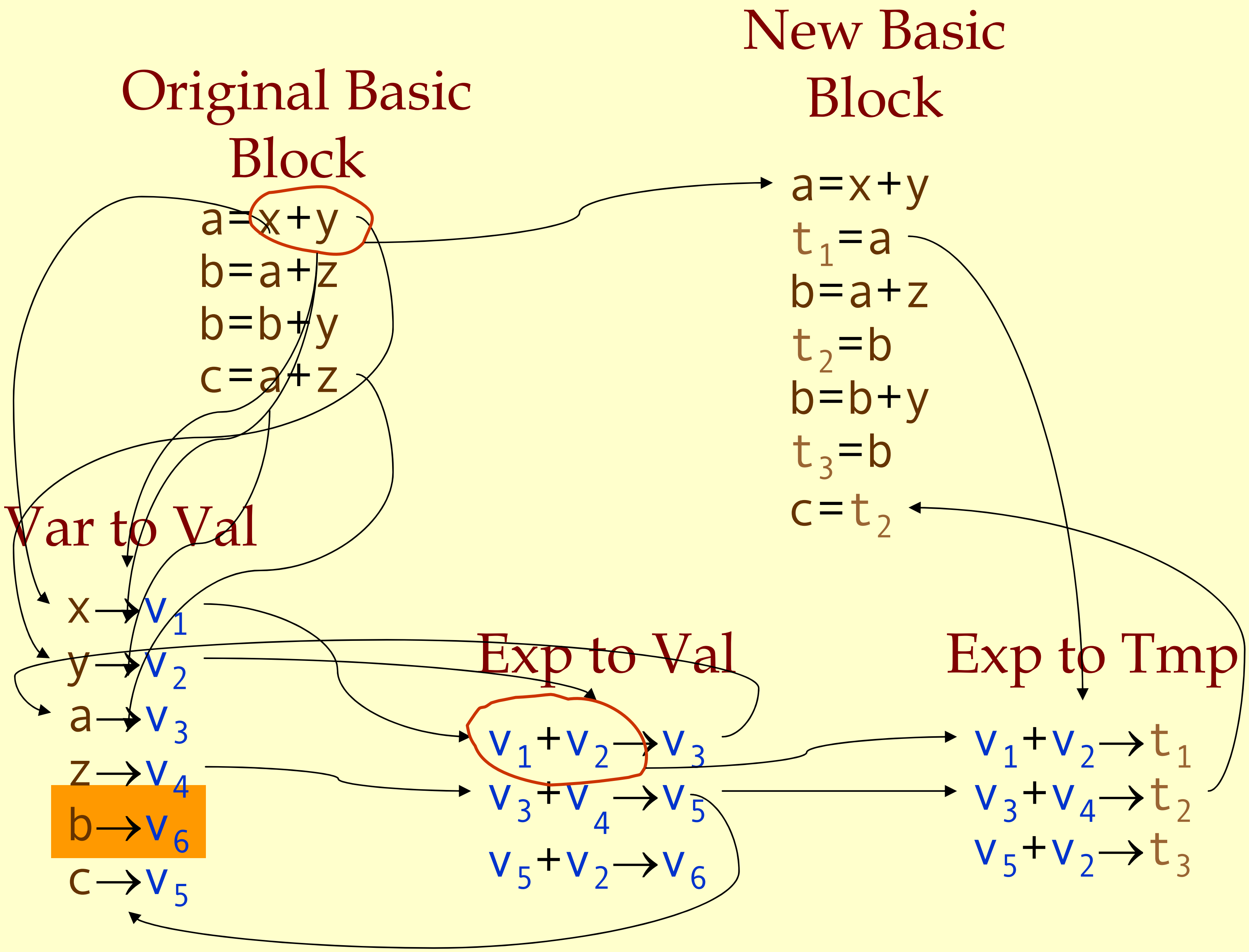
- After CSE

```
a=x+y
t1=a
b=a+z
t2=b
b=b+y
t3=b
c=t2
```

- Issues:
  - CSE with different names:

    `a=x; b=x+y; c=a+y;`

  - Excessive temp generation and use.


Original Basic Block
a=x+y
b=a+z
b=b+y
c=a+z
New Basic Block
a=x+y
t1=a
b=a+z
t2=b
b=b+y
t3=b
c=t2
Var to Val
x→v1
y→v2
a→v3
z→v4
b→v6
c→v5
Exp to Val
v1+v2→v3
v3+v4→v5
v5+v2→v6
Exp to Tmp
v1+v2→t1
v3+v4→t2
v5+v2→t3

Original Basic Block

```
a=x+y
b=a+z
b=b+y
c=a+z
```

New Basic Block

$a=x+y$
$t_1=a$
$b=a+z$
$t_2=b$
$b=b+y$
$t_3=b$
$c=t_2$

Var to Val

$x \rightarrow v_1$
$y \rightarrow v_2$
$a \rightarrow v_3$
$z \rightarrow v_4$
$b \rightarrow v_5$ $\quad b \rightarrow v_6$
$c \rightarrow v_5$

Exp to Val

$v_1+v_2 \rightarrow v_3$
$v_3+v_4 \rightarrow v_5$
$v_5+v_2 \rightarrow v_6$

Exp to Tmp

$v_1+v_2 \rightarrow t_1$
$v_3+v_4 \rightarrow t_2$
$v_5+v_2 \rightarrow t_3$

Problems

- Algorithm has a temporary for each value.

  ```
  a=x+y;  t1=a;
  ```
- Introduces
  - lots of temporaries.
  - lots of copy statements to temporaries.
- In many cases, temporaries and copy statements are unnecessary.
- So we eliminate them with copy propagation and dead code elimination.

Copy Propagation (CP)

- Once again, simulate execution of program
- If possible, use the original variable instead of a temporary
  - `a=x+y; b=x+y;`
  - After CSE becomes `a=x+y;` t_1`=a; b=`t_1`;`
  - After CP becomes `a=x+y; b=a;`
- **Key idea**: determine when original variables are **NOT** overwritten between computation of stored value and use of stored value.

Copy Propagation Maps

- Maintain two maps
  - tmp to var: tells which variable to use instead of a given temporary variable.
  - var to set: inverse of tmp to var. Tells which temps are mapped to a given variable by tmp to var.

Copy Propagation Example

- Original

```
a=x+y
b=a+z
c=x+y
a=b
```

- After CSE

```
a=x+y
t1=a
b=a+z
t2=b
c=t1
a=b
```

- After CSE and Copy Propagation

```
a=x+y
t1=a
b=a+z
t2=b
c=a
a=b
```

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a=x+y	a=x+y
t_1=a	t_1=a
b=a+z	b=a+z
t_2=b	t_2=b
c=t_1	c=a
a=b	a=b

tmp to var	var to set
$t_1 \to$ a	a $\to \{t_1\}$
$t_2 \to$ b	b $\to \{t_2\}$

Copy Propagation Example

Basic Block After CSE

```
a=x+y
t1=a
b=a+z
t2=b
c=t1
a=b
```

tmp to var

$t_1 \rightarrow t_1$

$t_2 \rightarrow b$

Basic Block After CSE and Copy Prop

```
a=x+y
t1=a
b=a+z
t2=b
c=a
a=b
```

var to set

$a \rightarrow \{\}$

$b \rightarrow \{t_2\}$

Dead Code Elimination

- Copy propagation keeps all temporaries.
- There may be temps that are never read.
- Dead Code Elimination removes them.

Basic block after
CSE and Copy Prop.

a=x+y
t1=a
b=a+z
t2=b
c=a
a=b

Basic block after
CSE, CP, &
Dead Code Elimination

a=x+y
b=a+z
c=a
a=b

Dead Code Elimination

- Basic idea:
  - Process code in **reverse** execution order.
  - Maintain a set of variables that are needed later in computation.
  - On encountering an assignment to a temporary that is not needed, we remove the assignment.

Basic Block After CSE and Copy Prop	Needed Set
a=x+y	{a,z}
t1=a	{a,z}
b=a+z	{a,b,z}
t2=b	{a,b}
c=a	{a,b}
⇒ a=b	{b}

Interesting Properties

- Analysis and optimization algorithms simulate execution of the program.
  - CSE and Copy Propagation go forward.
  - Dead Code Elimination goes backwards.
- Optimizations are stacked.
  - Group of basic transformations.
  - Work together to get good result.
  - Often, one transformation creates inefficient code that is cleaned up by following transformations.

Other Basic Block Transformations

- Constant Propagation.
- Strength Reduction:
  - `a*4;` ⇒ `a<<2;`
  - `3*a;` ⇒ `a+a+a;`
- Algebraic Simplification:
  - `a*1;` ⇒ `a;`
  - `b+0;` ⇒ `b;`
- Unified transformation framework.

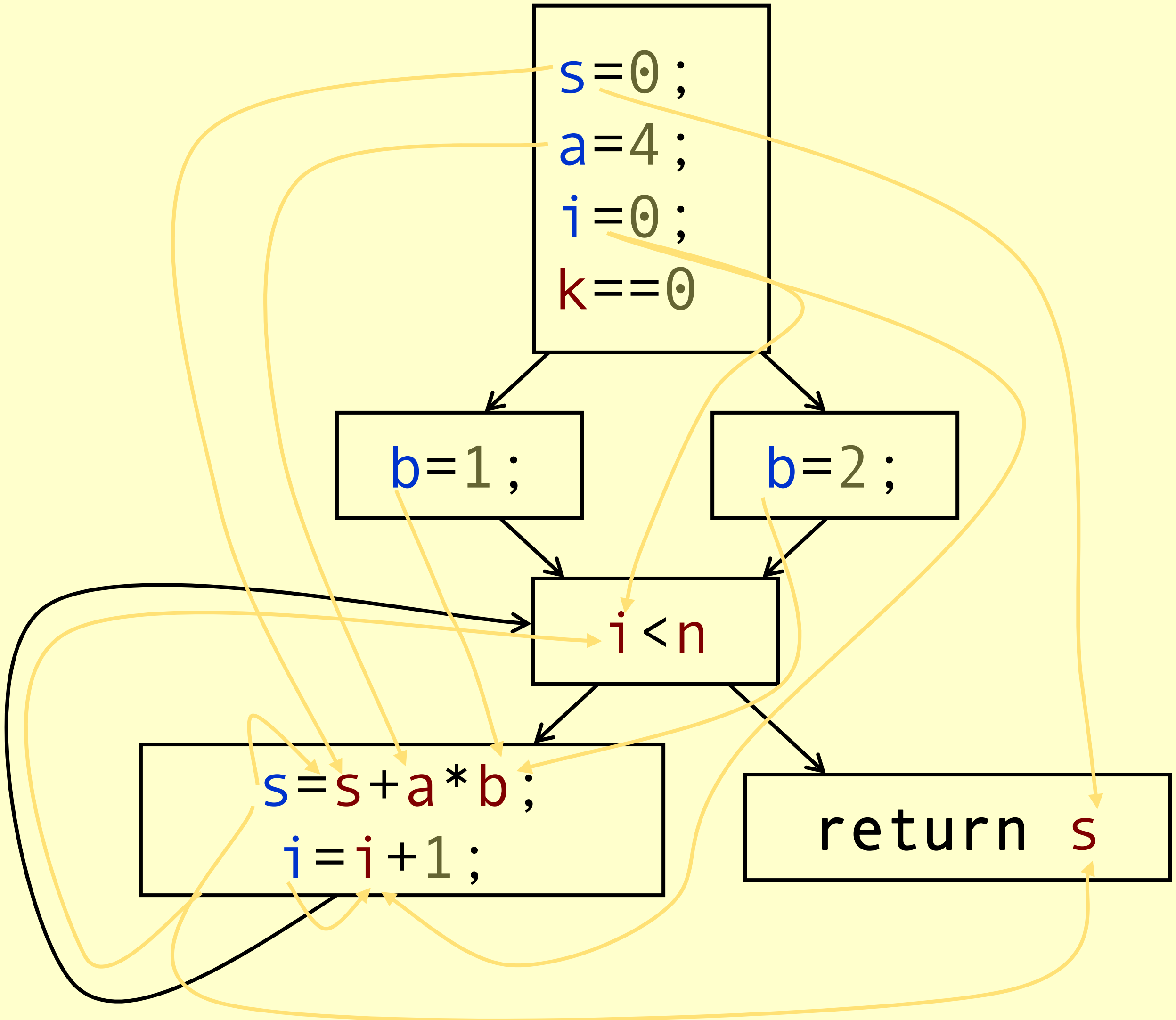
Dataflow Analysis
(Global Analysis)

- Used to determine properties of programs that involve multiple basic blocks.
- Typically used to enable transformations.
  - common sub-expression elimination.
  - constant and copy propagation.
  - dead code elimination.
- Analysis and transformation often come in pairs.

Reaching Definitions

- Concept of *definition* and *use*
  - `a=x+y`
    - is a definition of a.
    - is a use of x and y.
- A definition reaches a use if value written by definition may be read by use.

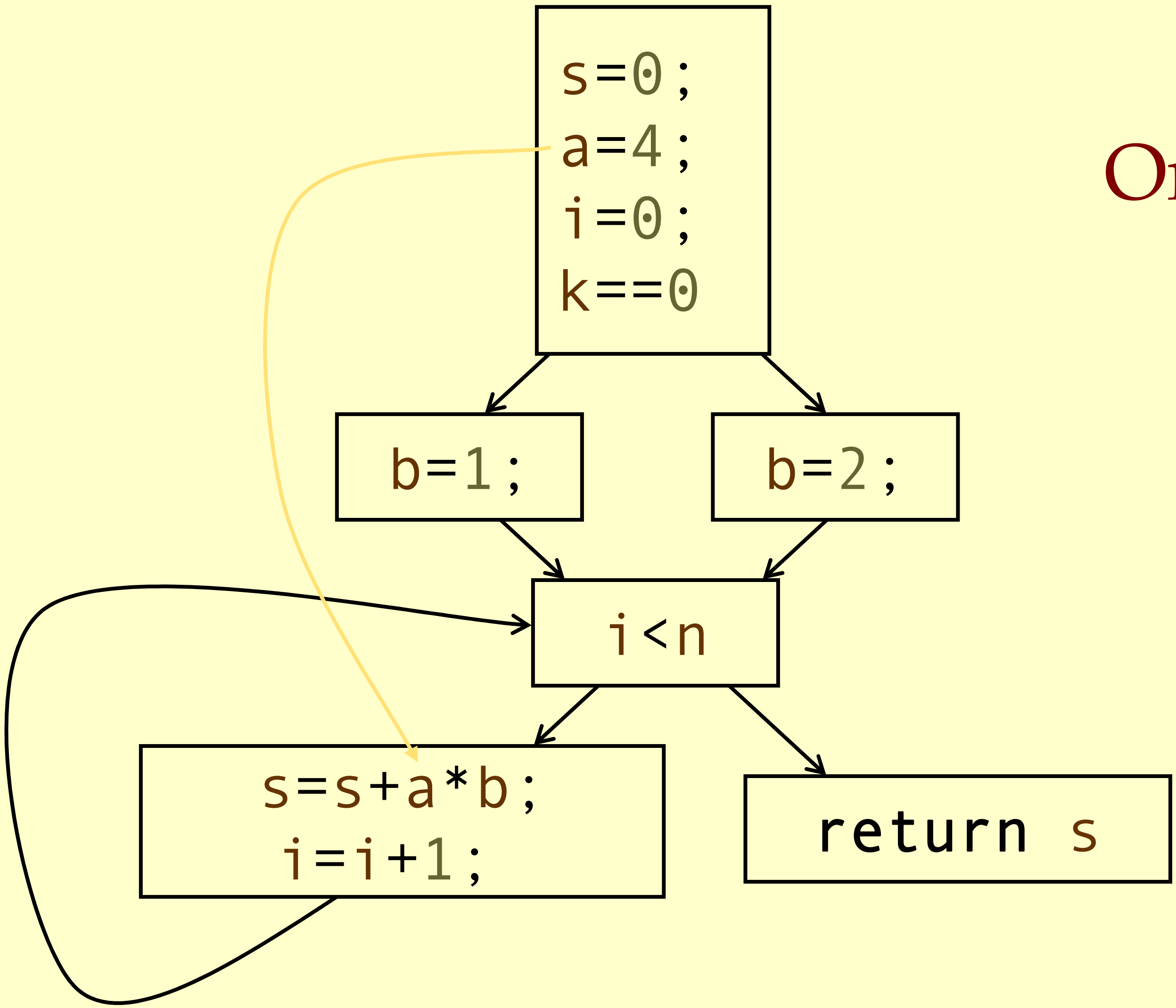
Reaching Definitions



Reaching Definitions and Constant Propagation

- Is a use of a variable a constant?
  - Check all reaching definitions.
  - If all assign variable to same constant.
  - Then use is in fact a constant.
- Can replace variable with constant.

Is a constant in s=s+a*b?

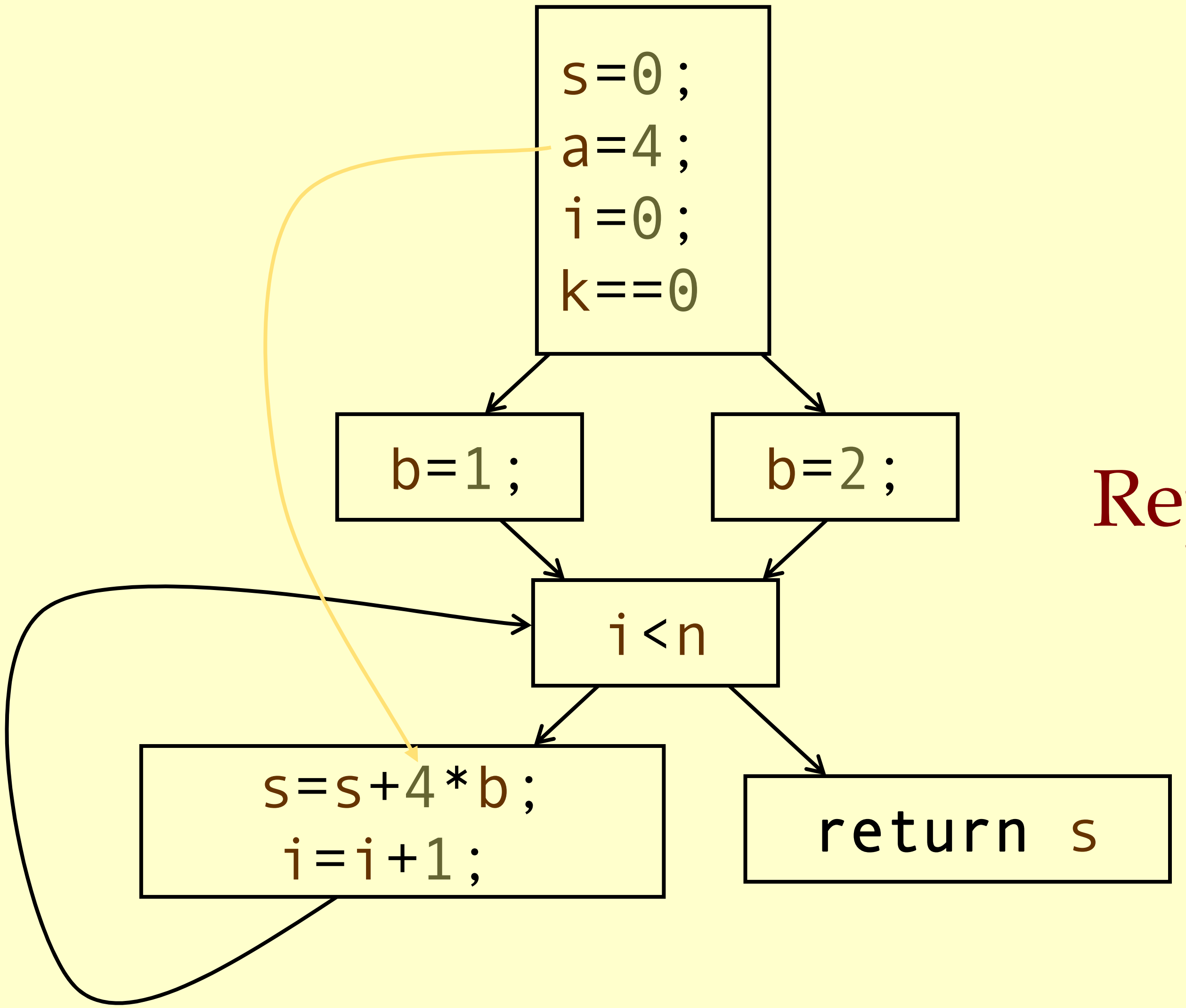


Yes!

On all reaching definitions

a=4

Constant Propagation Transform



Yes!

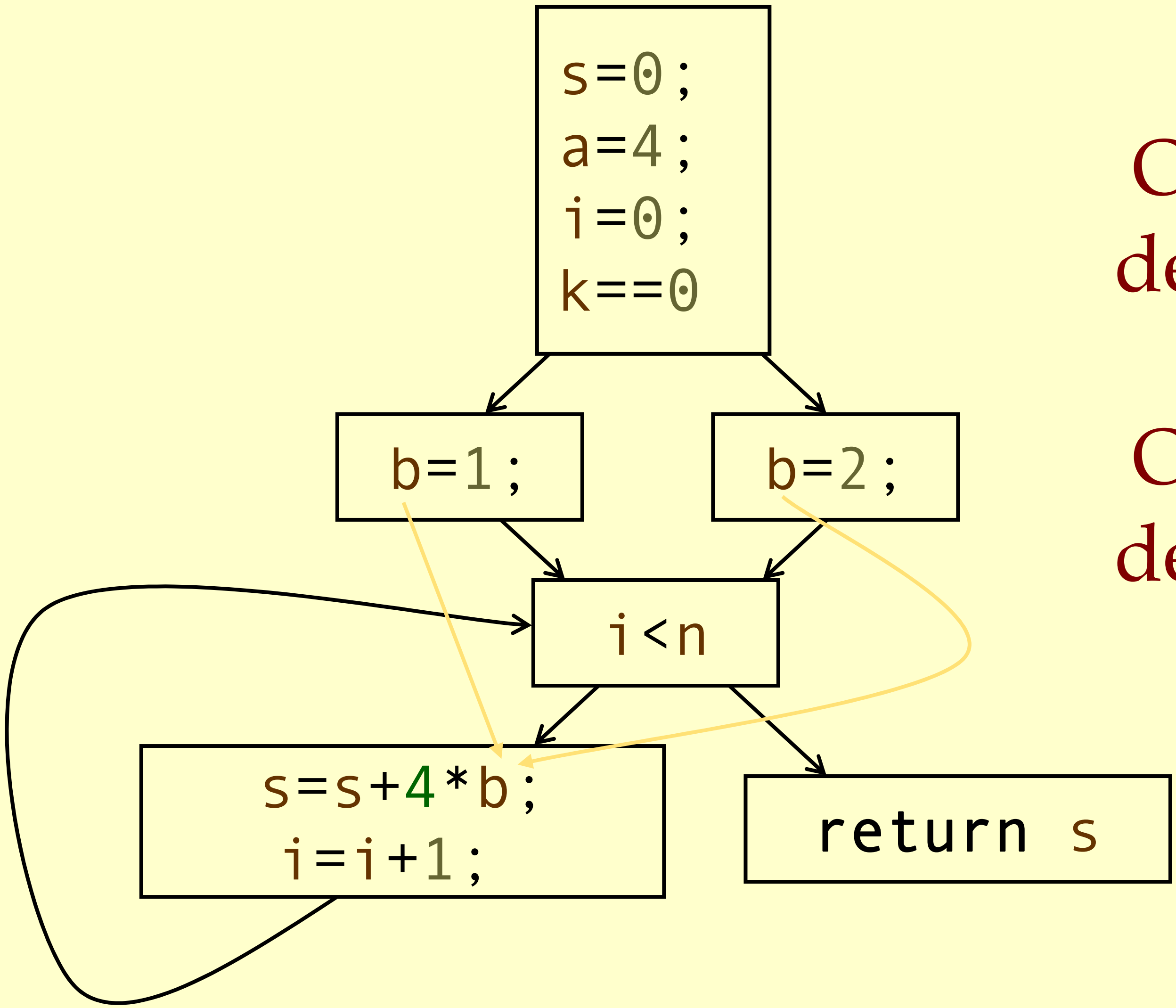
a=4

in

s=s+a*b

Replace use of a with 4.

Is b constant in s=s+4*b?



No!

One reaching definition with

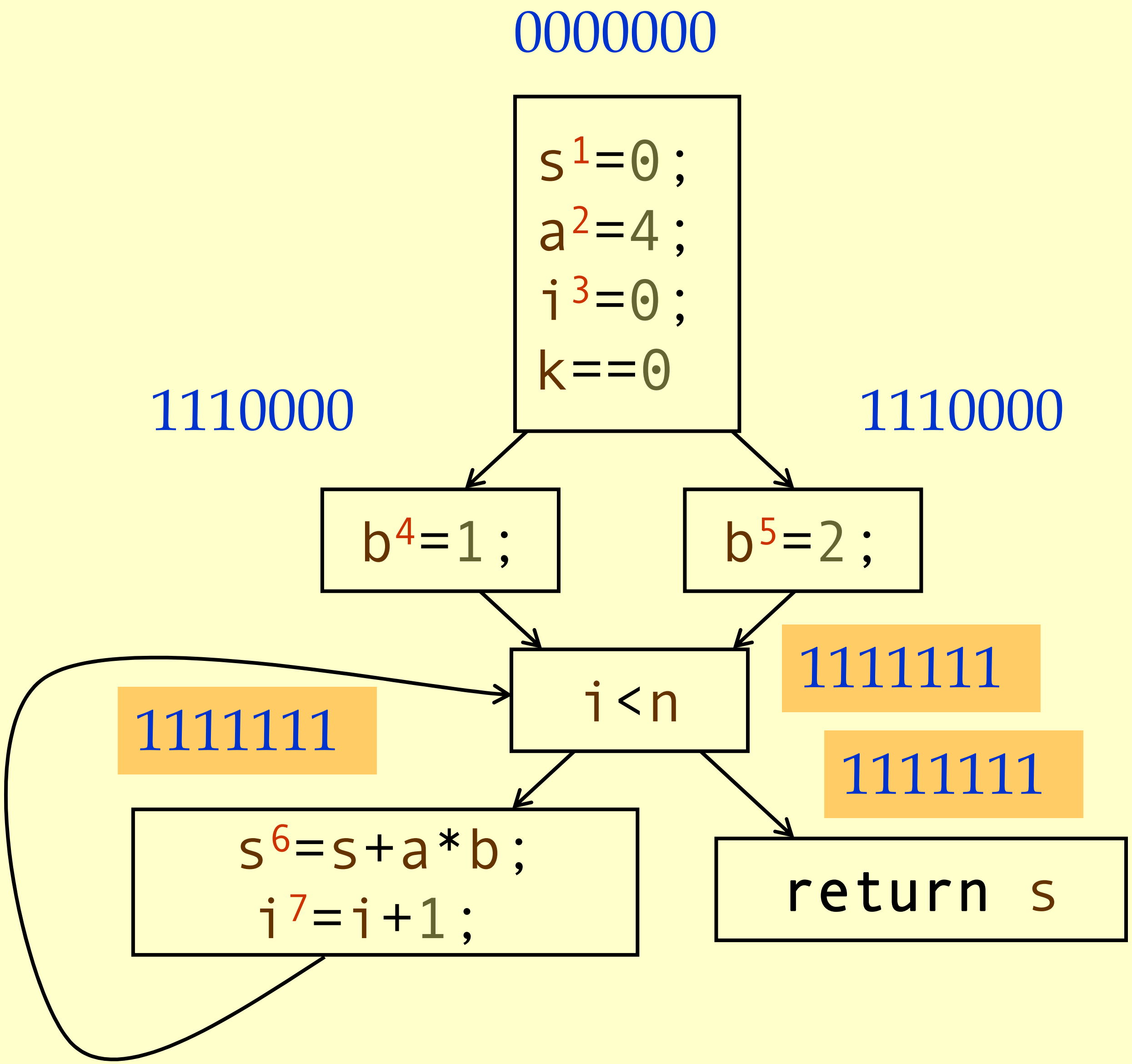
b=1

One reaching definition with

b=2

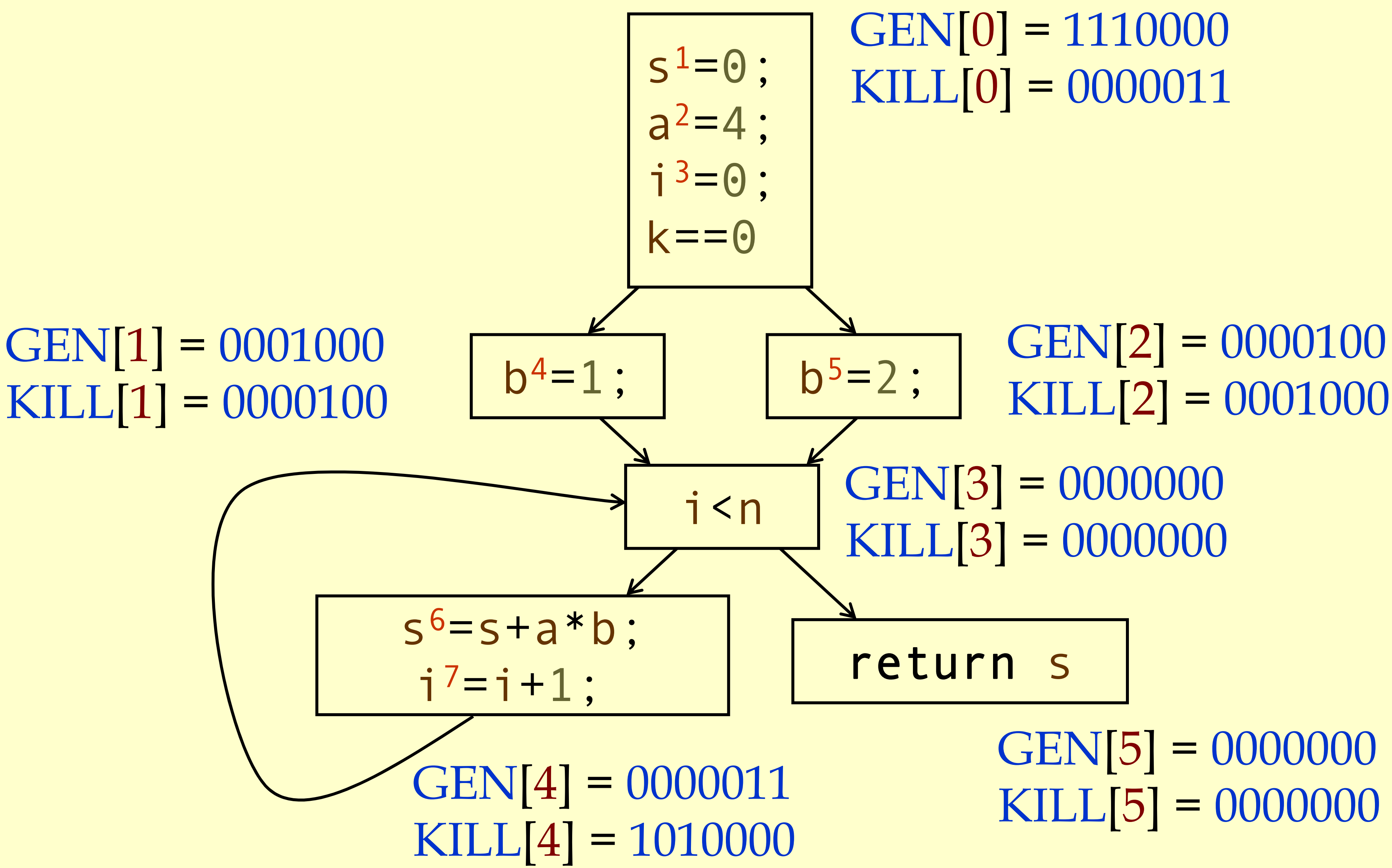
Computing Reaching Definitions

- Compute with sets of definitions:
  - Represent sets using bit vectors.
  - Each definition has a position in bit vector.
- At each basic block, compute:
  - Definitions that reach start of block.
  - Definitions that reach end of block.
- Do computation by simulating execution of program until the fixed point is reached.


0000000
s1=0;
a2=4;
i3=0;
k==0
1110000
1110000
b4=1;
b5=2;
i<n
1111111
1111111
1111111
s6=s+a*b;
i7=i+1;
return s

Formalizing Analysis

- Each basic block has
  - IN - set of definitions that reach beginning of block
  - OUT - set of definitions that reach end of block
  - GEN - set of definitions generated in block
  - KILL - set of definitions killed in the block
- GEN[`s`^{6}`=s+a*b;i`^{7}`=i+1;`] = 0000011
- KILL[`s`^{6}`=s+a*b;i`^{7}`=i+1;`] = 1010000
- Compiler scans each basic block to derive GEN and KILL sets.

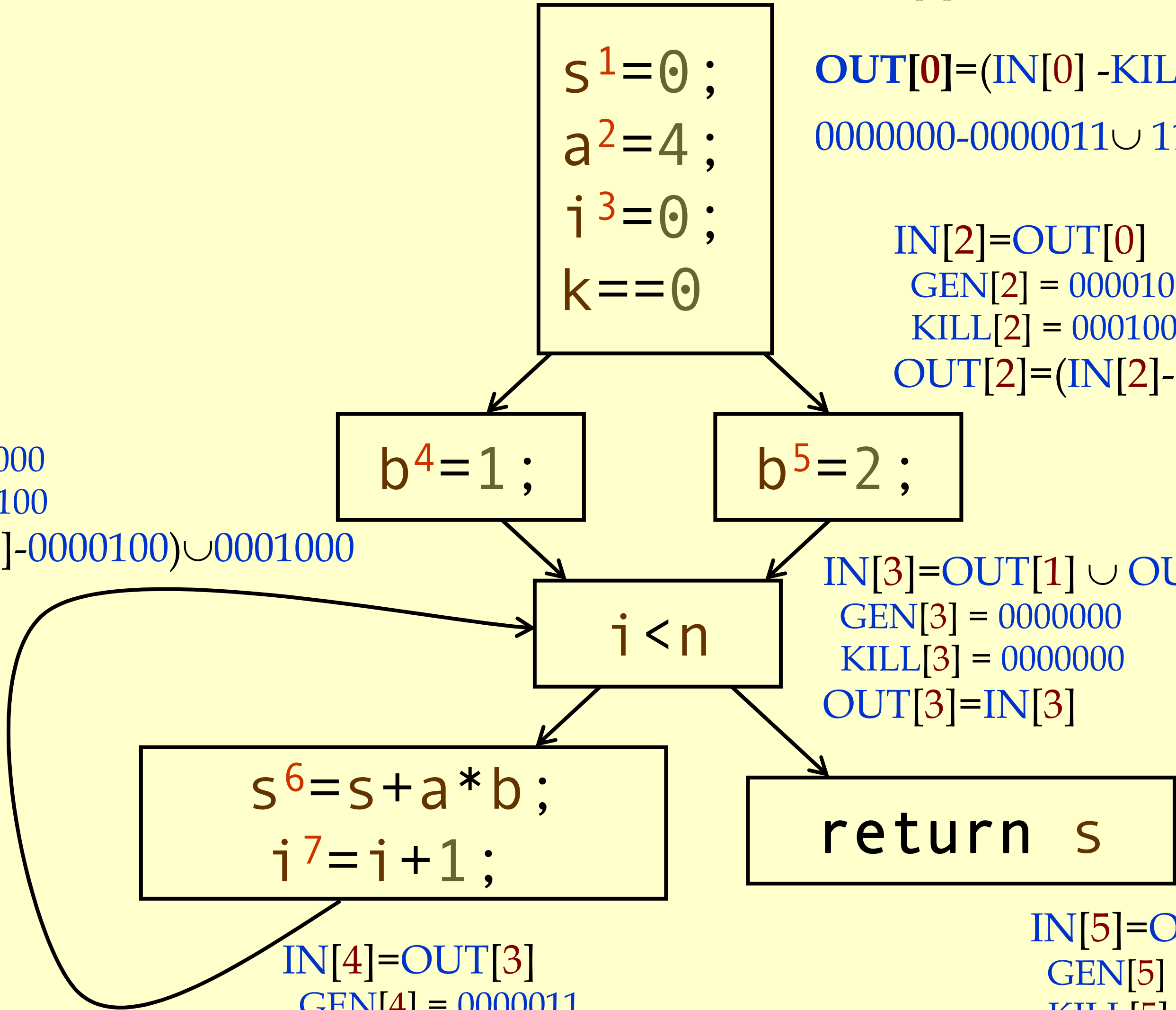

s1=0;
a2=4;
i3=0;
k==0
GEN[0] = 1110000
KILL[0] = 0000011
b4=1;
GEN[1] = 0001000
KILL[1] = 0000100
b5=2;
GEN[2] = 0000100
KILL[2] = 0001000
i<n
GEN[3] = 0000000
KILL[3] = 0000000
s6=s+a*b;
i7=i+1;
GEN[4] = 0000011
KILL[4] = 1010000
return s
GEN[5] = 0000000
KILL[5] = 0000000

Dataflow Equations

- $IN[b_i] = OUT[b_1] \cup \ldots \cup OUT[b_n]$
  where $b_1, \ldots, b_n$ are predecessors of b_i
- $OUT[b_i] = (IN[b_i] - KILL[b_i]) \cup GEN[b_i]$
- IN[entry] = 0000000
- Result: system of equations.

IN[0] = 0000000
GEN[0] = 1110000
KILL[0] = 0000011

OUT[0]=(IN[0] -KILL[0])∪GEN[0]=
0000000-0000011∪ 1110000=**1110000**



IN[2]=OUT[0]
GEN[2] = 0000100
KILL[2] = 0001000
OUT[2]=(IN[2]-0001000)∪0000100

IN[1]=OUT[0]
GEN[1] = 0001000
KILL[1] = 0000100
OUT[1]=(IN[1]-0000100)∪0001000

IN[3]=OUT[1] ∪ OUT[2]
GEN[3] = 0000000
KILL[3] = 0000000
OUT[3]=IN[3]

IN[4]=OUT[3]
GEN[4] = 0000011
KILL[4] = 1010000
OUT[4]=(IN[4]-1010000)∪0000011

IN[5]=OUT[3]
GEN[5] = 0000000
KILL[5] = 0000000
OUT[5]=IN[5]

Solving Equations

- Use fix point algorithm.
- Initialize with solution of $OUT[b_i] = 0000000$
- Repeatedly apply equations:
  - $IN[b_i] = OUT[b_1] \cup \ldots \cup OUT[b_n]$
  - $OUT[b_i] = (IN[b_i] - KILL[b_i]) \cup GEN[b_i]$
- Until reach fixed point, i.e., until equation application has no further effect.
- Use a worklist to track which equation applications may have further effect.

Reaching Definitions Algorithm

```
for all nodes n∈ℕ
   OUT[n] = ∅;                              // Or OUT[n] = GEN[n];
Changed = ℕ;                                // ℕ = all nodes in graph
while (Changed != ∅)                        // Until fixed point reached.
  choose n∈Changed;                         // Node from worklist
  Changed=Changed-{n};                      // Remove from worklist
   OldOut = OUT[n]                          // Remember old result
   IN[n] = ∅;                               // Calculate IN as join
   for all nodes p∈predecessors(n)          //    of predecessors.
         IN[n]=IN[n]∪OUT[p];
   OUT[n]=(IN[n]-KILL[n])∪GEN[n];           // Recalculate OUT
   if (OUT[n] != OldOut)                    // If OUT[n] changed
    for all nodes s∈successors(n)
         Changed=Changed∪{s};               //Add succs to worklist
```

Questions

- Does the algorithm halt?
  - yes, because transfer function is monotonic.
  - if increase IN, increase OUT.
  - in limit, all bits are 1.
- If bit is 1, is there always an execution in which corresponding definition reaches basic block?
- If bit is 0, does the corresponding definition ever reach basic block?
- Concept of conservative analysis.

Available Expressions

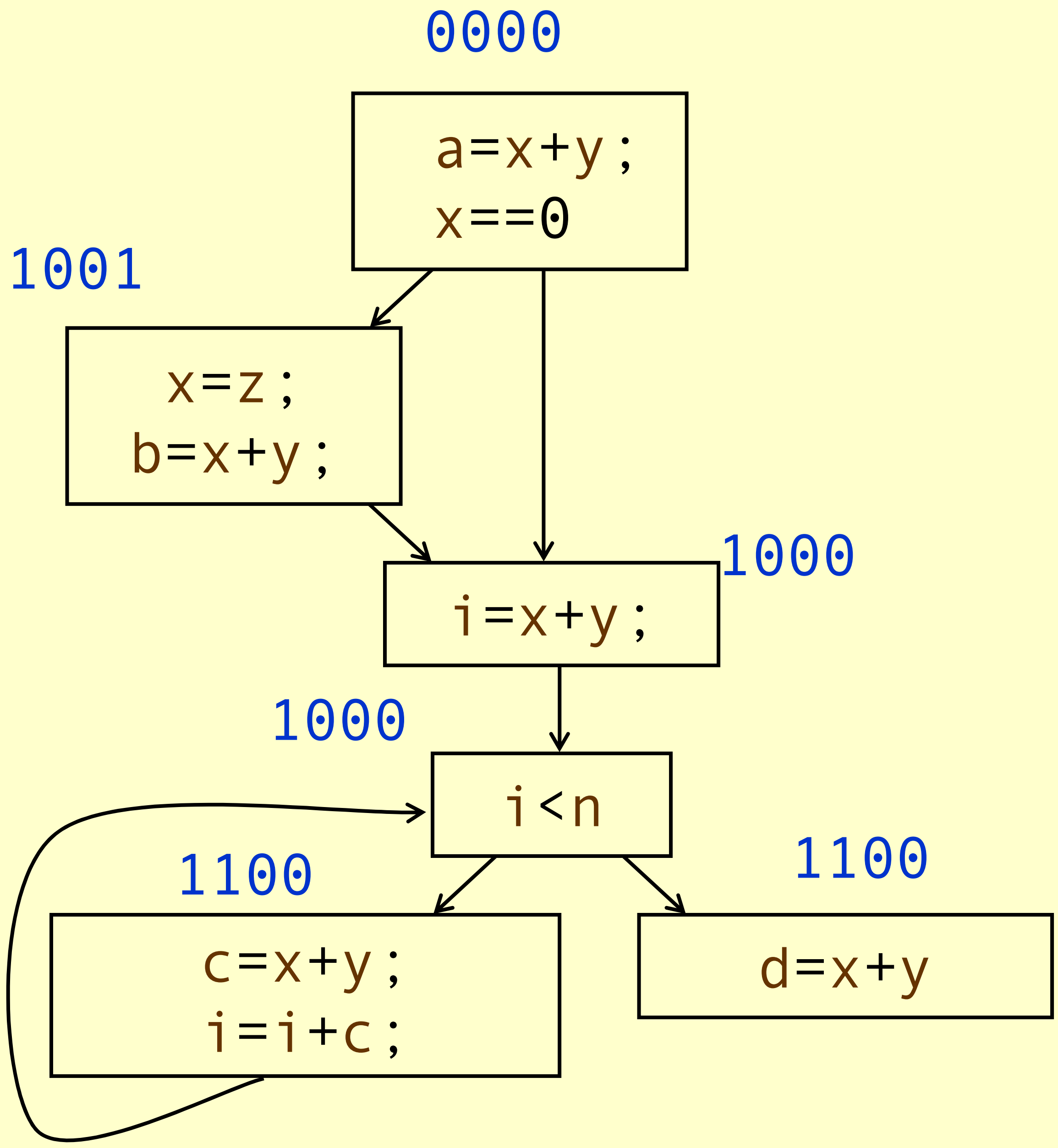
- An expression x+y is available at a point p if
  - **every** path from the initial node to p evaluates x+y before reaching p,
  - and there are **no assignments** to x or y after the evaluation but before p.
- Available Expression information can be used to do global (across basic blocks) CSE.
- If an expression is available at use, there is no need to re-evaluate it.

Computing Available Expressions

- Represent sets of expressions using bit vectors.
- Each expression corresponds to a bit.
- Run dataflow algorithm similar to reaching definitions.
- Big difference:
  - Definition reaches a basic block if it comes from **ANY** predecessor in CFG.
  - Expression is available at a basic block only if it is available from **ALL** predecessors in CFG.

Expressions

1: x+y
2: i<n
3: i+c
4: x==0

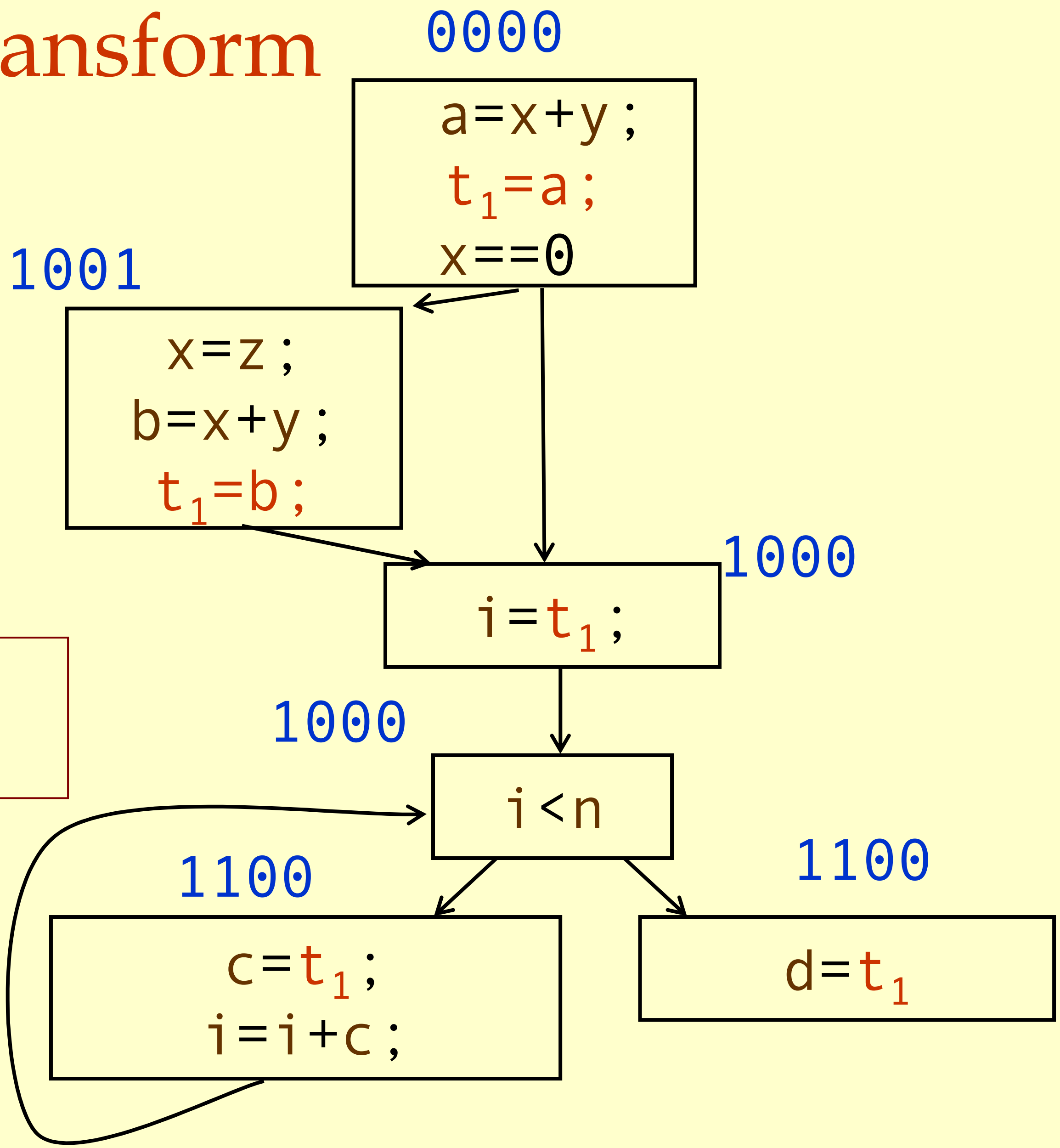


Global CSE Transform

Expressions

1: x+y
2: i<n
3: i+c
4: x==0

Must use same temp for CSE in all blocks



Formalizing Analysis

- Each basic block has
  - IN - set of expressions that reach beginning of block.
  - OUT - set of expressions that reach end of block.
  - GEN - set of expressions generated in block.
  - KILL - set of expressions killed in the block.
- GEN[`x=z; b=x+y`] = 1000
- KILL[`x=z; b=x+y`] = 1001
- Compiler scans each basic block to derive GEN and KILL sets.

Dataflow Equations

- $IN[b_i] = OUT[b_1] \cap \ldots \cap OUT[b_n]$
  - where $b_1, \ldots, b_n$ are predecessors of b_i
- $OUT[b_i] = (IN[b_i] - KILL[b_i]) \cup GEN[b_i]$
- $IN[entry] = 0000$
- Result: system of equations.

Solving Equations

- Use fix point algorithm.
- IN[entry]=0000
- Initialize with solution of OUT[b_i] = 1111
- Repeatedly apply equations:
  - IN[b_i] = OUT[b_1] $\cap$ ... $\cap$ OUT[b_n]
  - OUT[b_i] = (IN[b_i] - KILL[b_i]) $\cup$ GEN[b_i]
- Use a worklist to track which equation applications may have further effect.

Available Expressions Algorithm

```
for all nodes n∈ℕ                    // 𝔼 is set of all expressions.
   OUT[n] = 𝔼;                       // OUT[n] =𝔼 -KILL[n];
Changed = ℕ;                         // ℕ = all nodes in graph
while (Changed != ∅)
   choose n∈Changed;
   Changed=Changed-{n};
   IN[n] = 𝔼 ;
   OldOut = OUT[n]
   for all nodes p∈predecessors(n)
         IN[n]=IN[n]∩OUT[p];
  OUT[n]=(IN[n]-KILL[n])∪GEN[n];
  if (OUT[n] != OldOut)
   for all nodes s∈successors(n) Changed=Changed∪{s};
```

Questions

- Does algorithm always halt?
- If expression is available in some execution, is it always marked as available in analysis?
- If expression is not available in some execution, can it be marked as available in analysis?
- In what sense is the algorithm conservative?

Duality In Two Algorithms

- Reaching definitions
  - Confluence operation is set **union.**
  - OUT[b] initialized to **empty set**.
- Available expressions
  - Confluence operation is set **intersection.**
  - OUT[b] initialized to **set of available expressions.**
- General framework for dataflow algorithms.
- Build parameterized dataflow analyzer once, use for all dataflow problems.

Liveness Analysis

- A variable v is live at point p if
  - v is used along some path starting at p, and
  - no definition of v along the path before the use.
- When is a variable v dead at point p?
  - No use of v on any path from p to exit node, or
  - If all paths from p, redefine v before using v.

What Use is Liveness Information?

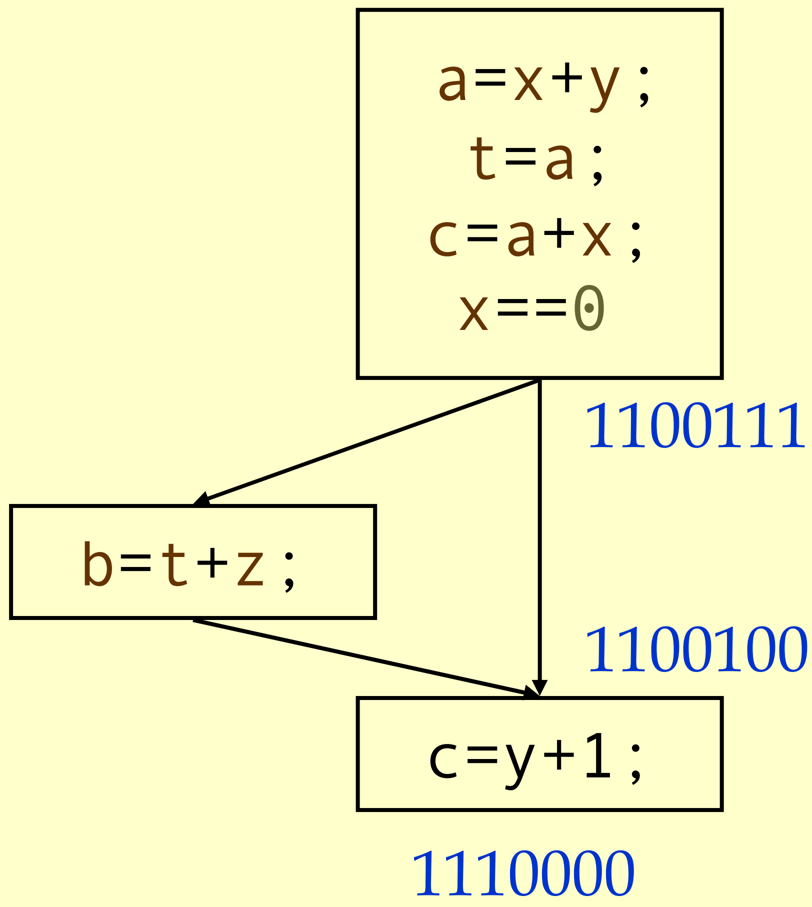
- Register allocation.
  - If a variable is dead, we can reassign its register.
- Dead code elimination.
  - Eliminate assignments to variables not read later.
  - But must not eliminate last assignment to variable (such as instance variable) visible outside CFG.
  - Can eliminate other dead assignments.
  - Handle by making all externally visible variables live on exit from CFG.

Conceptual Idea of Analysis

- Simulate execution.
- But start from exit and go **backwards** in CFG.
- Compute liveness information from end to beginning of basic blocks.

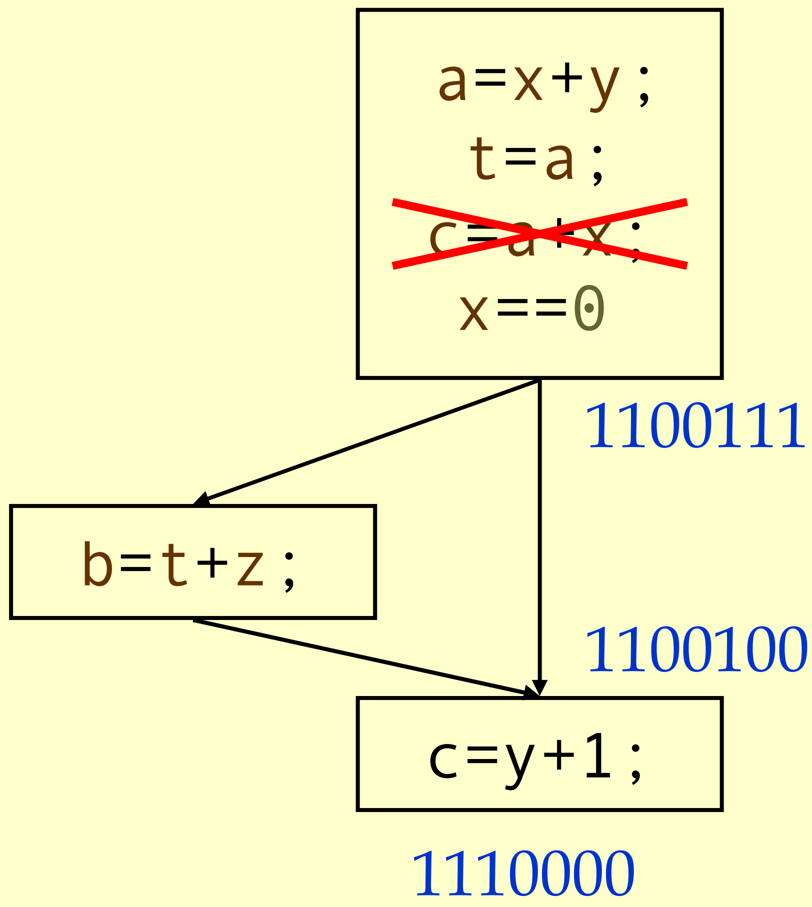
Liveness Example

- Assume `a, b, c` visible outside function. They are live on exit.
- Assume `x, y, z, t` are not visible.
- Represent liveness using a bit vector: order is `abcxyzt`.



Using Liveness Information for Dead Code Elimination

- Assume `a, b, c` visible outside function. They are live on exit.
- Assume `x, y, z, t` are not visible.
- Represent liveness using a bit vector:
  order is `abcxyzt`.



Formalizing Analysis

- Each basic block has
  - IN - set of variables live at start of block.
  - OUT - set of variables live at end of block.
  - USE - set of variables with upwards exposed uses in block. (**GEN**)
  - DEF - set of variables defined in block. (**KILL**)
- USE[`x=z ; x=x+1 ; y=1 ;`] = {z} (x not in USE)
- DEF[`x=z ; x=x+1 ; y=1 ;`] = {x, y}
- Compiler scans each basic block to derive USE and DEF sets.

Algorithm

```
OUT[Exit] = ∅;
IN[Exit] = USE[n];
 for all nodes n∈ℕ-{Exit}
   IN[n] = ∅;
Changed = ℕ-{Exit};
 while (Changed != ∅)
   choose n ∈ Changed;
    Changed = Changed-{n};
    OldIn=IN[n]
    OUT[n] = ∅;
    for all nodes s ∈ successors(n) OUT[n] = OUT[n] ∪ IN[p];
    IN[n] = USE[n] ∪ (OUT[n] - DEF[n]);
    if (IN[n] != OldIn)
      for all nodes p ∈ predecessors(n) Changed=Changed∪{p};
```

Similar to Other Dataflow Algorithms

- Backwards analysis, not forwards.
- Still have transfer functions.
- Still have confluence operators.
- Can generalize framework to work for both forwards and backwards analyses.

Analysis Information Inside Basic Blocks

- One detail:
  - Given dataflow information at IN and OUT of node.
  - Also need to compute information at each statement of basic block.
  - Simple propagation algorithm usually works fine.
  - Can be viewed as restricted case of dataflow analysis.

Summary

- Basic blocks and basic block optimizations.
  - Copy and constant propagation.
  - Common sub-expression elimination.
  - Dead code elimination.
- Dataflow Analysis
  - Control flow graph.
  - IN[b], OUT[b], transfer functions, join points.
- Pairs of analyses and transformations:
  - Reaching definitions/constant propagation.
  - Available expressions/common sub-expression elimination.
  - Liveness analysis/Dead code elimination.