Advanced Compiler Techniques

http://lamp.epfl.ch/teaching/advancedCompiler/

Erik Stenman

Virtutech

Introduction

- What is this course about?
- How will this be taught?
- Who is teaching the course?
- Where to find more information?
- Why is this course interesting?

Teachers

- Lecturer
  - Erik Stenman
    - Have been hacking for fun since 1980.
    - Have been hacking compilers for money since 1996.
    - Got a Ph.D. on “Efficient Implementation of Concurrent Programing Languages” (i.e. Erlang) from Uppsala University in 2002.
    - Was a member of LAMP and the Scala team 2003-2004.
    - Currently working at Virtutech implementing full system simulators.
- Assistant
  - Iulian Dragos
    - Office: INR321, 021-69 36864 .

Course Goals

- ♦ Give some theoretical framework for compiler optimizations.
- ♦ Give a general orientation on optimization techniques.
- ♦ Give an understanding of how some higher level constructs are implemented.

Non-Goals and Requirements

- This course will not try to teach you all possible optimizations, or even all common optimizations.
- We will not talk about parallel machines.
- You are supposed to be familiar with basic compiler concepts: scanning, parsing, semantic analysis, and simple code generation. (These topics will not be touched.)
- You are supposed to be used to programming in Java.

Course Content

- Optimization Techniques
- Implementation techniques for high level languages (HLL).

Course Content

- Optimization Techniques
  - Theory for analysis and optimization
  - Optimization algorithms
- Implementation techniques for high level languages (HLL).
  - Virtual Machines
  - Memory Management
  - High level constructs

Course Structure

- The course will be made up of lectures, articles, two projects, and an oral exam.
- The lectures will be given with slides like this one, and the slides will be available on the web: `http://lamp.epfl.ch/teaching/advancedCompiler/`
- I will try to have the final version of the slides on the web at least a day before the lecture.
- Since I am commuting from Sweden to give this course, the schedule is somewhat special.

Preliminary Schedule

1. Introduction, motivation, terminology,
   local optimizations: CSE, constant propagation, copy propagation, dead code elimination,
   (algebraic simplification, strength reduction)
   Introduction to global optimizations and dataflow analysis.
2. Foundations of dataflow analysis, introduction to abstract interpretation.
3. Analysis for global optimizations: reaching definitions, available expressions, and liveness analysis.
4. Static Single Assignment Form (SSA) & Dominators.
5. SSA-based Dead Code Elimination & Sparse Conditional Constant Propagation.
   Partial Redundancy Elimination.
6. Loop Optimizations.
   Lazy Code Motion.
7. Global Register Allocation
8. Code Scheduling
9. Introduction to part 2: "Implementation of high level languages"
   Implementation of Objects and FPL (higher order functions, laziness).
10. Implementation of Concurrency.
11. (Automatic) Memory Management
12. Virtual Machines, Interpretation Techniques, and Just-In-Time Compilers.
13. Bits and pieces, such as implementation of exceptions, linkers and loaders.
    Quiz.

Schedule

- 11/3 Lecture 1 & 2
- 18/3 Lecture 3 & 4
- 25/3 Easter Break
- 01/4 Easter break
- 08/4 Lecture 5 & 6
- 15/4 Project 1
- 22/4 Lecture 7 & 8
- 29/4 Project 2
- 06/5 Lecture 9 & 10
- 13/5 Lecture 11 & 12
- 20/5 Project 2
- 27/5 Project 2
- 03/6 Project 2
- 10/6 Project 2
- **13/6 – 17/6 Exams**
- 17/6 Lecture 13

The Projects

- There will be two projects in the course and you may work in groups of two persons.
- Project 1: A simple register allocator.
  - The main goal of the first project is to get familiar with the compiler framework that we will use for the second project.
  - The task is to implement a Sethi-Ullman tree-based register allocator for a given compiler.
- Project 2: Optimizations.
  - The goal of the second project is to get a concrete understanding of different optimization techniques.
  - The task will be to implement different optimizations in the given compiler in order to achieve a given speedup on a set of benchmarks.

Literature

Expect to read a lot for this class, especially in order to complete the projects.

- Course Book:
  - Keith Cooper and Linda Torczon,
    Engineering a Compiler, Morgan Kaufmann, October 2003.
- Alternative:
  - Andrew W. Appel,
    Modern compiler implementation in Java (second edition).
    Cambridge University Press, 2002, ISBN 052182060X.
- Reference:
  - Steven Muchnick,
    Advanced Compiler Design and Implementation,
    Morgan Kaufmann, August 1997.
- Additional articles that will be handed out.

The Slides

- Many of the slides are based on Konstantinos Sagonas set of slides for his **Advanced Compiler Techniques**, held at Uppsala University, January-February 2004.

The Exam

- There will be an oral exam during **the last week** of the course.
- The exam will concentrate on the understanding of the concepts taught in the course, and not on details of specific algorithms.

Why is this course interesting?

- ♦ Optimization is challenging — you can not write an optimal compiler: there is always room for improvements.
- ♦ The course will give you many techniques and tools that you can use in other areas.
- ♦ You will gain a better understanding of how a compiler works and what to expect of the code generated by compilers.
- ♦ It is fun!

Introduction to Compiler Optimization

- Compiler Optimization is hard.
- **The most important aspect of an optimization is that it is correct!**
- The subject is confusing:
  - The notion of optimality.
  - Huge number of possible optimizations.
  - Many intricate and NP-complete problems.
- The terminology is confusing:
  - Global optimization means function local.
  - Optimization means improvement.
- Compilation time vs. runtime speed is often a factor.

Introduction to Compiler Optimization

- Suggested method for (compiler) optimization:
  1. Look at the generated code – try to find sources of inefficient code. (Better yet profile.)
  2. Look in the literature for solutions to these inefficiencies. (Most likely someone has already solved the problem.)
  3. Implement the solution.
  4. Repeat from 1.
- Most optimizations consists of two activities: analysis and rewrite.
  - First you must know what the program does, then you can rewrite it so that it does it more efficiently.

Introduction to Compiler Optimization

- In this course we will describe a general framework for doing analysis of computer programs called *abstract interpretation*.
- This framework can be used in many different situations and for many different optimizations.

Introduction to Compiler Optimization

- When the analysis is done, the rewrite part is often easy.
- We need to be able to associate the results of the analysis with the actual code, and we need to have a representation of the code that will let us rewrite it easily.

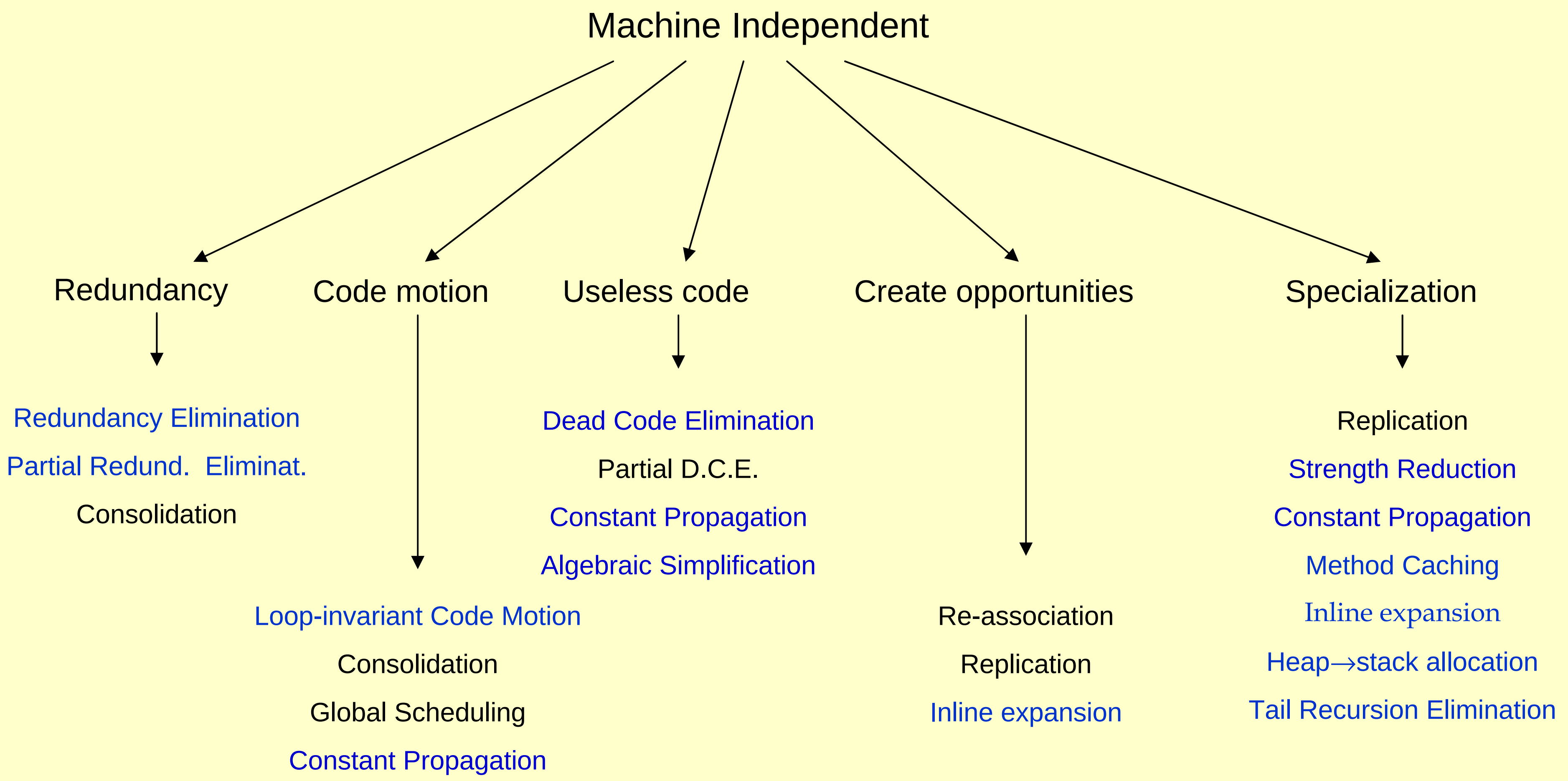
Optimization Techniques Taxonomy

- We can divide optimizations into:
  - Machine independent optimizations.
    - Decrease ratio of overhead to real work.
    - Example: dead code elimination.
  - Machine dependent optimizations.
    - Take advantage of specific machine properties.
    - Work around limitations of a specific machine.
    - Example: instruction scheduling.

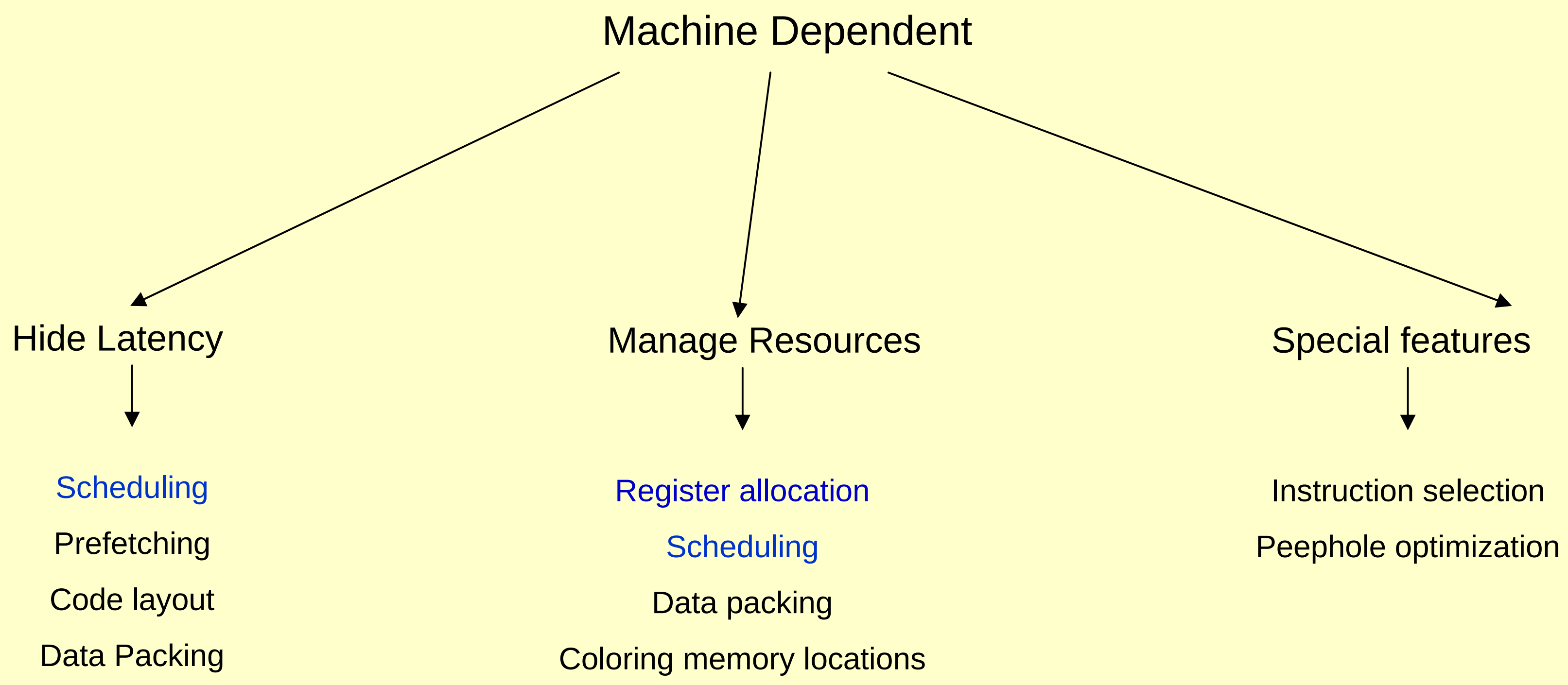
Optimization Techniques Taxonomy

- We can further divide the optimizations on their intended effect.
  - Machine independent optimizations.
    1. Eliminating redundant computations.
    2. Move code to execute it less.
    3. Eliminate dead code.
    4. Specialize on context.
    5. Enable other optimizations.
  - Machine dependent optimizations.
    1. Manage or hide latency.
    2. Take advantage of special hardware features.
    3. Manage finite resources.

Taxonomy of Global Compiler Optimizations



Taxonomy of Global Compiler Optimizations

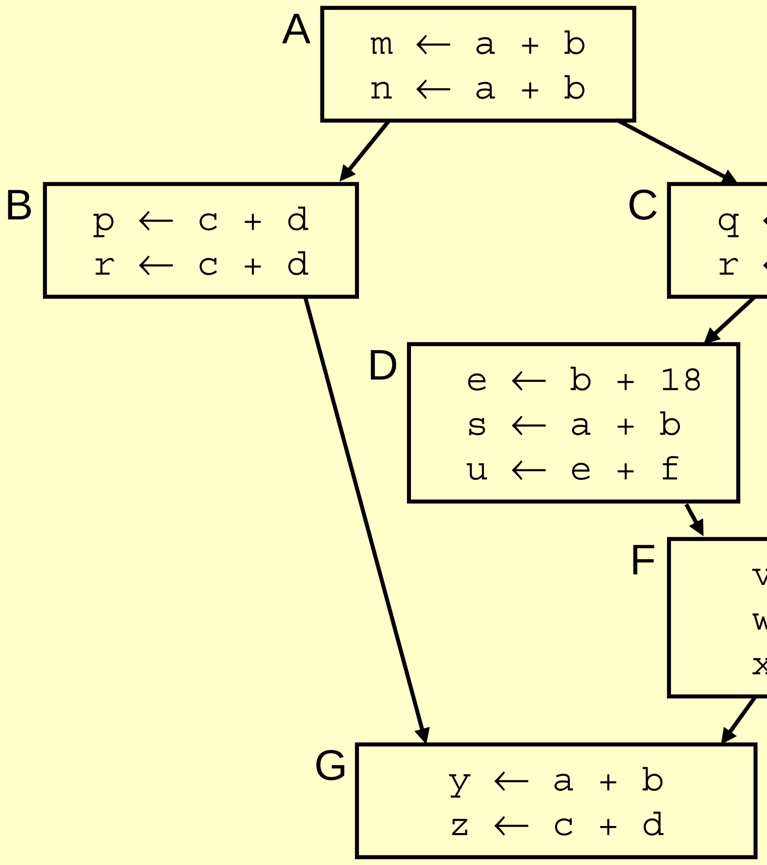


Terminology: Program Representation

Control Flow Graph (CFG):

- Nodes N – statements of program
- Edges E – flow of control
  - $pred(n)$ = set of all immediate predecessors of n
  - $succ(n)$ = set of all immediate successors of n
- Start node n_0
- Set of final nodes N_{final}

Terminology: Control-Flow Graph



Control-flow graph (CFG)

- Nodes for basic blocks
- Edges for branches
- Basis for much of program analysis & transformation

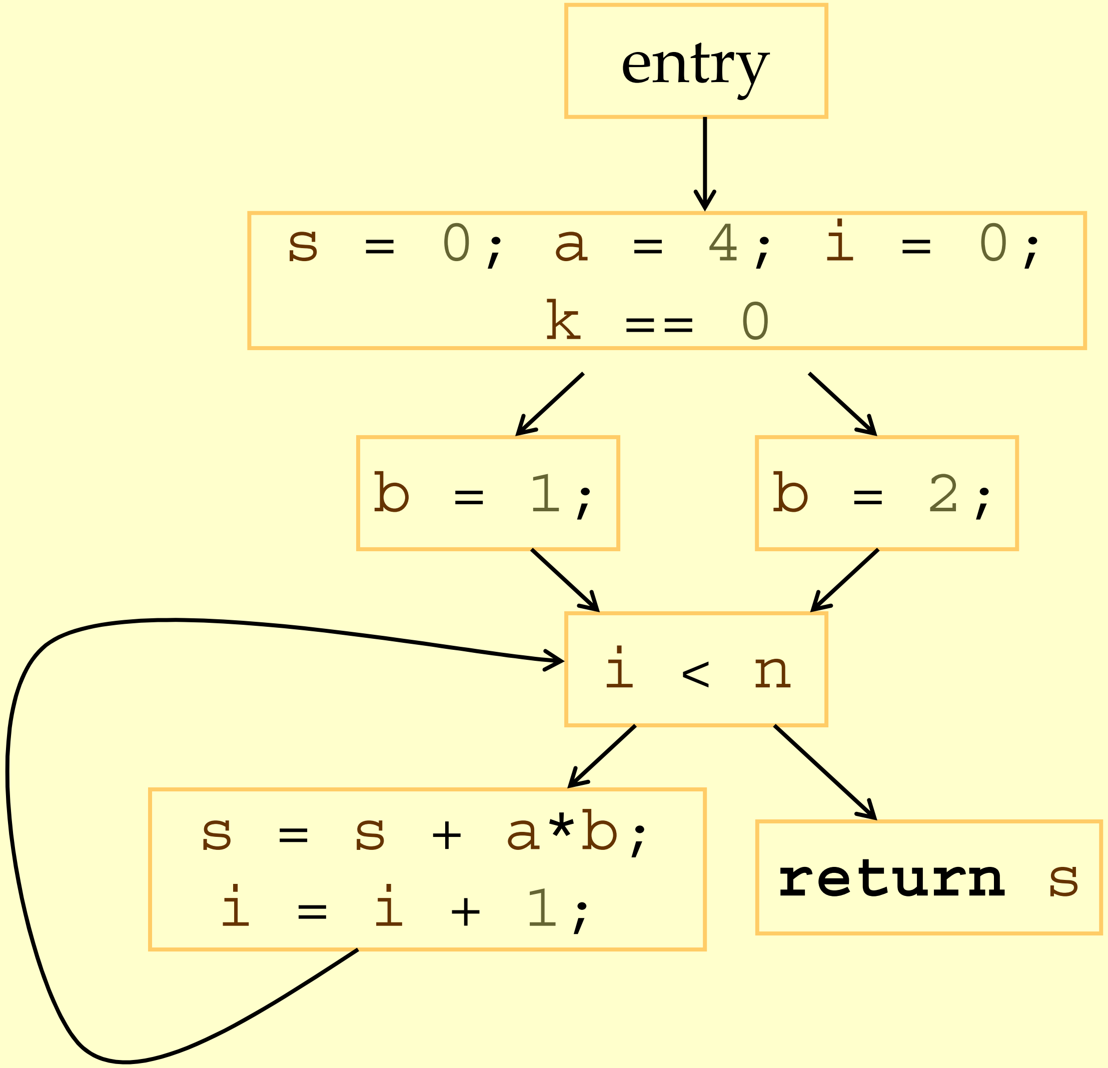
This CFG,

G = (N,E)

N = {A, B, C, D, E, F, G}
E = {(A, B), (A, C), (B, G), (C, D), (C, E), (D, F), (E, F),(F, G)}
|N| = 7
|E| = 8

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;
}
```



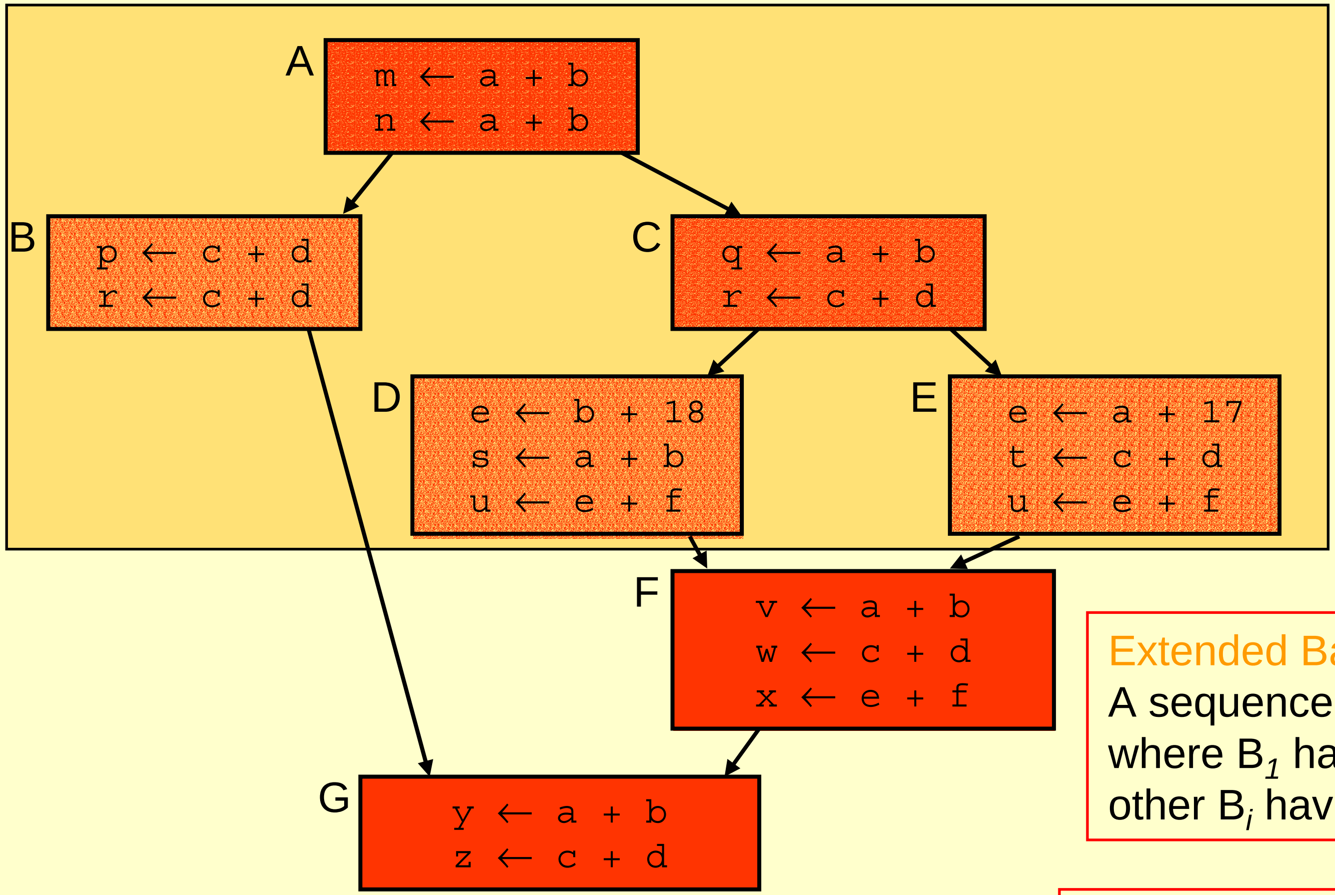
Control Flow Graph

- 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.

Terminology: Extended Basic Block



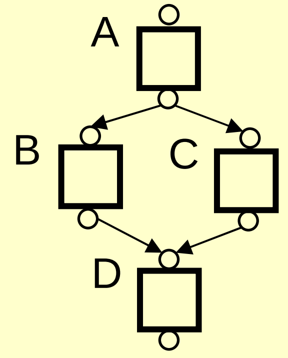
EBB: Conceptually it is a program sequence with only one entry point but possibly several exit points.

An EBB contains 1 or more paths. This EBB ({A,B,C,D,E}) contains the paths {A,B} {A,C,D} {A,C,E}

Extended Basic Block (EBB):
A sequence of basic blocks B_1, B_2, …, B_n where B_1 has more than 1 predecessor, all other B_i have a unique predecessor.

Path:
A sequence of basic blocks B_1, B_2, …, B_n where B_i is the predecessor of B_{i+1}.

Terminology: Program Points



- One program point before each node.
- One program point after each node.
- *Join point* – Program point with multiple predecessors.
- *Split point* – Program point with multiple successors.

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.

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

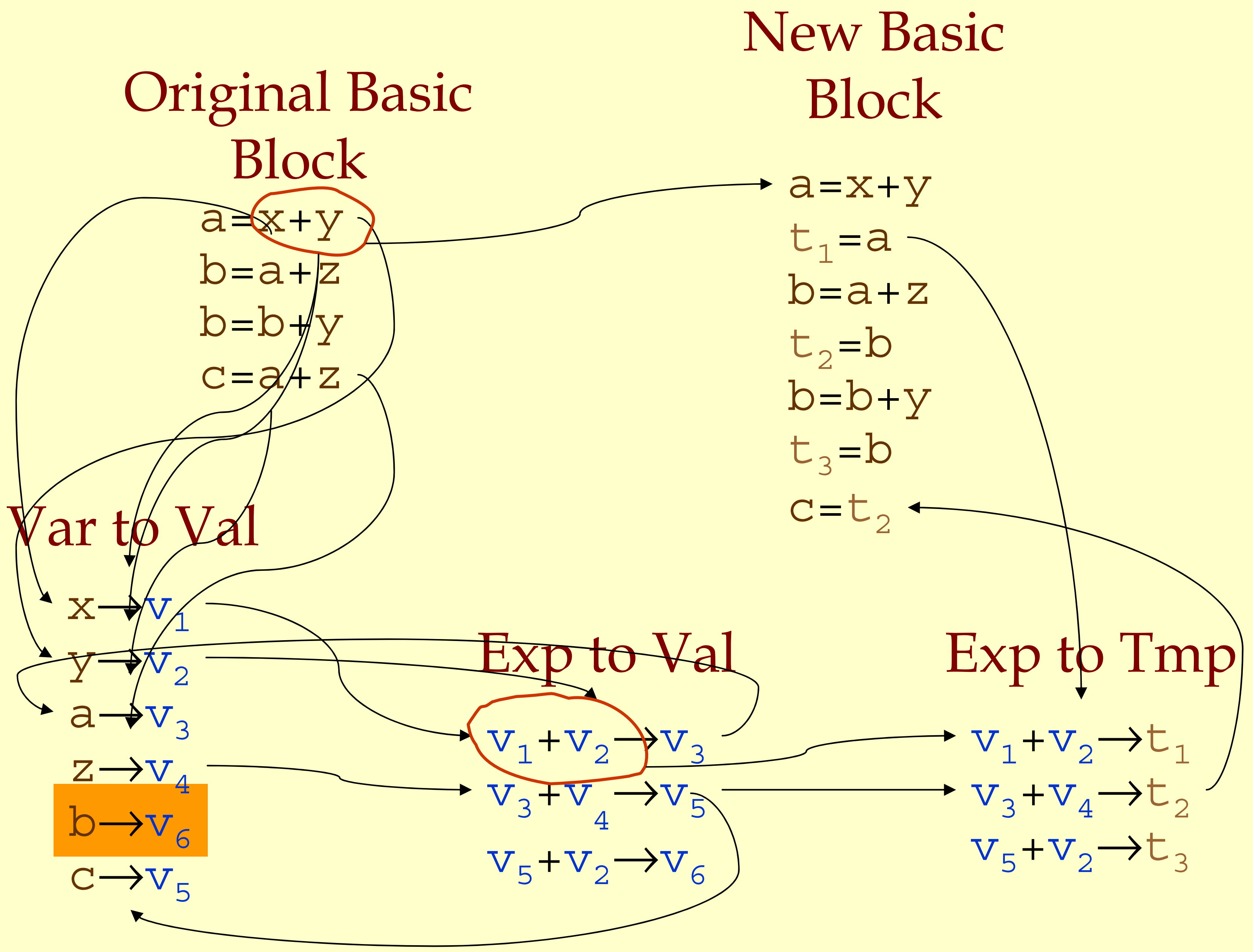
```
a=x+y
b=a+z
b=b+y
c=a+z
```

- 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

$x \rightarrow v_1$

$y \rightarrow v_2$

$a \rightarrow v_3$

$z \rightarrow v_4$

$b \rightarrow v_6$

$c \rightarrow v_5$

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

$v_1 + v_2 \rightarrow t_1$

$v_3 + v_4 \rightarrow t_2$

$v_5 + v_2 \rightarrow t_3$

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

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

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;` t_1`=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

$a=x+y$

$t_1=a$

$b=a+z$

$t_2=b$

$c=t_1$

$a=b$

- After CSE and Copy Propagation

$a=x+y$

$t_1=a$

$b=a+z$

$t_2=b$

c=a

$a=b$

Copy Propagation Example

Basic Block After CSE

```
a=x+y
```
t_1=a
```
b=a+z
```
t_2=b

c=t_1
```
a=b
```

tmp to var

$t_1 \rightarrow a$

$t_2 \rightarrow b$

Basic Block After CSE and Copy Prop

```
a=x+y
```
t_1=a
```
b=a+z
```
t_2=b
```
c=a
a=b
```

var to set

$a \rightarrow \{t_1\}$

$b \rightarrow \{t_2\}$

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;` $\Rightarrow$ `a<<2;`
  - `3*a;` $\Rightarrow$ `a+a+a;`
- Algebraic Simplification:
  - `a*1;` $\Rightarrow$ `a;`
  - `b+0;` $\Rightarrow$ `b;`
- Unified transformation framework.

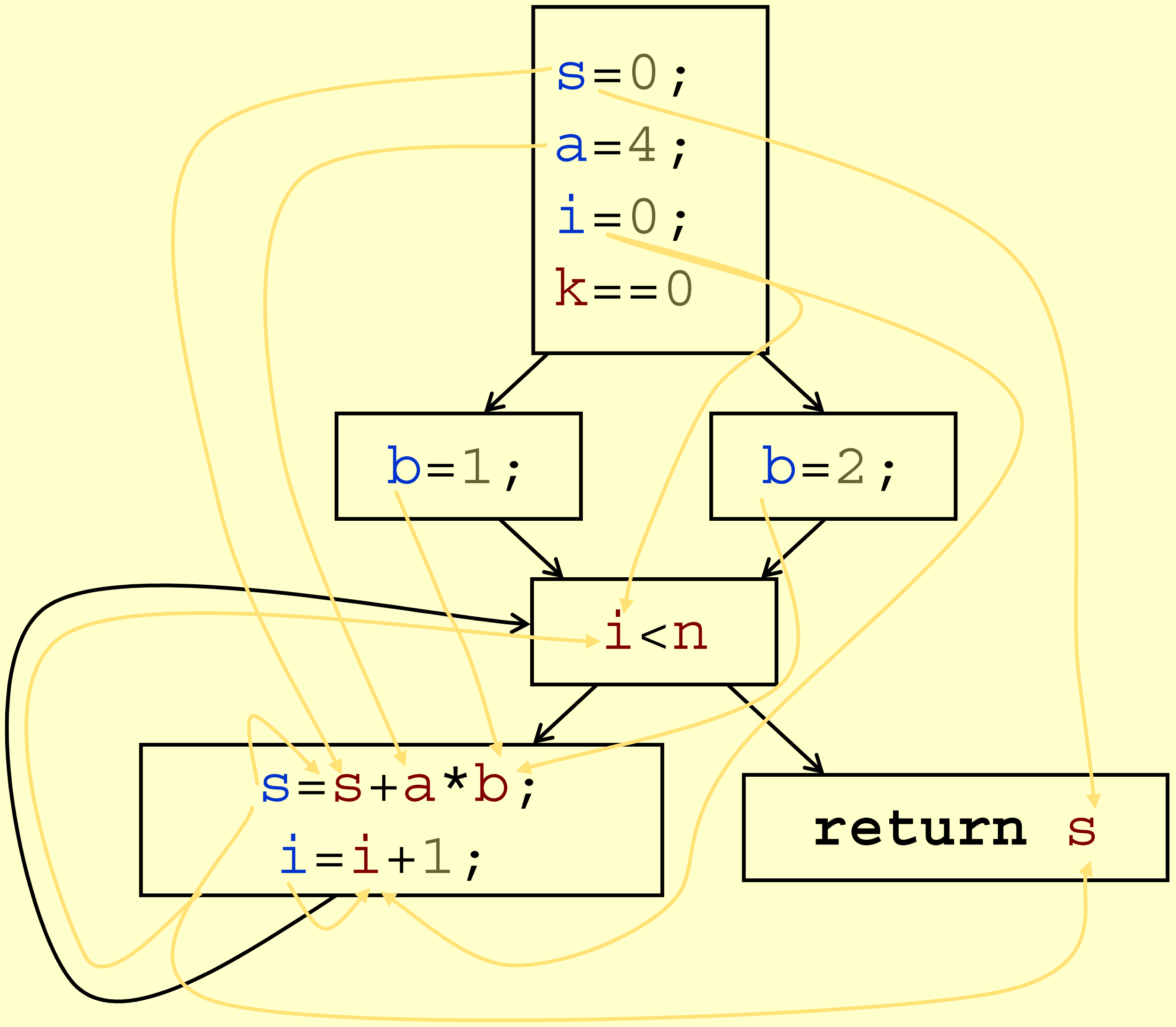
Dataflow Analysis
(Global Analysis)

- 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.

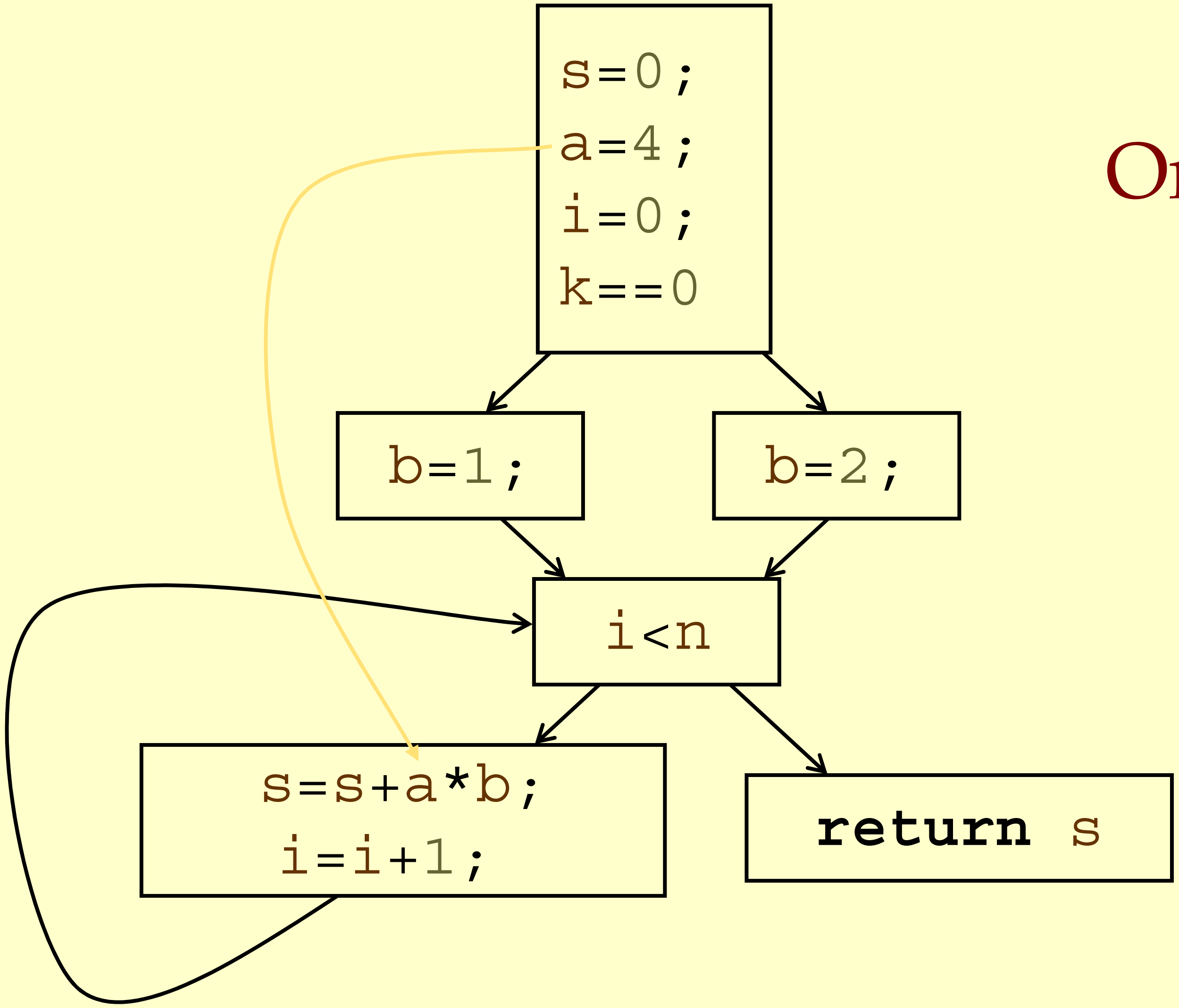
Reaching Definitions



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?

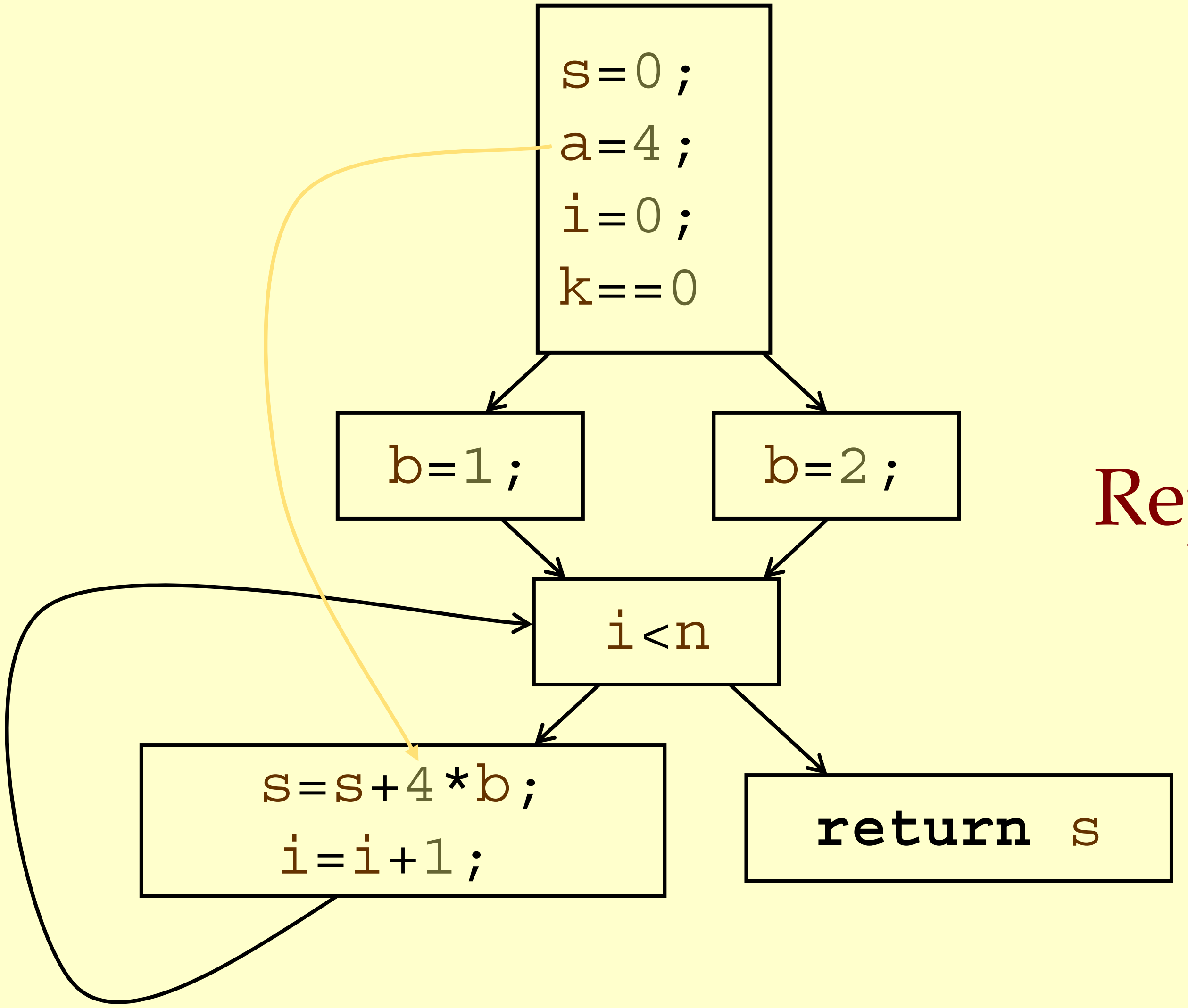


Yes!

On all reaching definitions

a=4

Constant Propagation Transform



Yes!

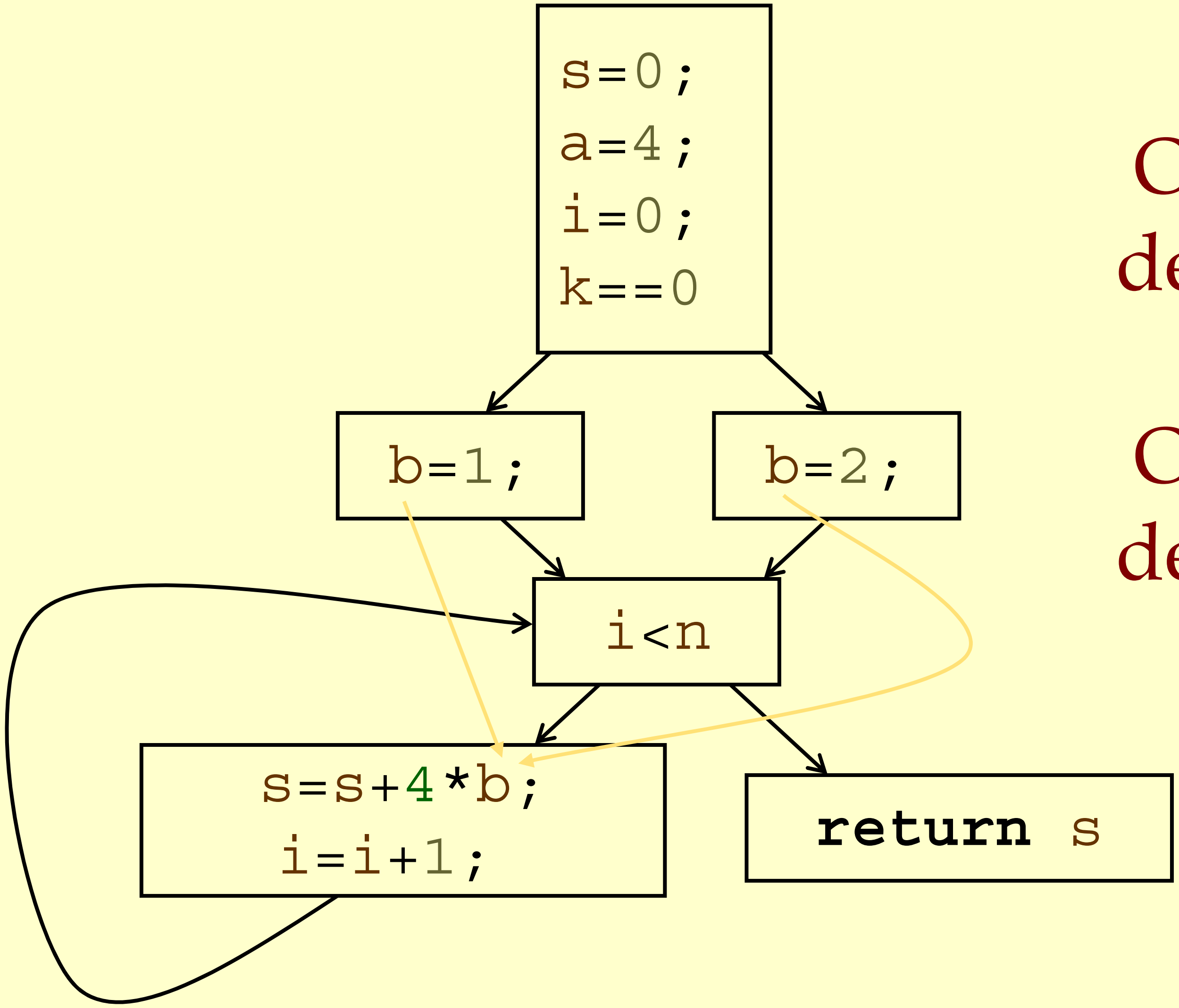
a=4

in

s=s+a*b

Replace use of a with 4.

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



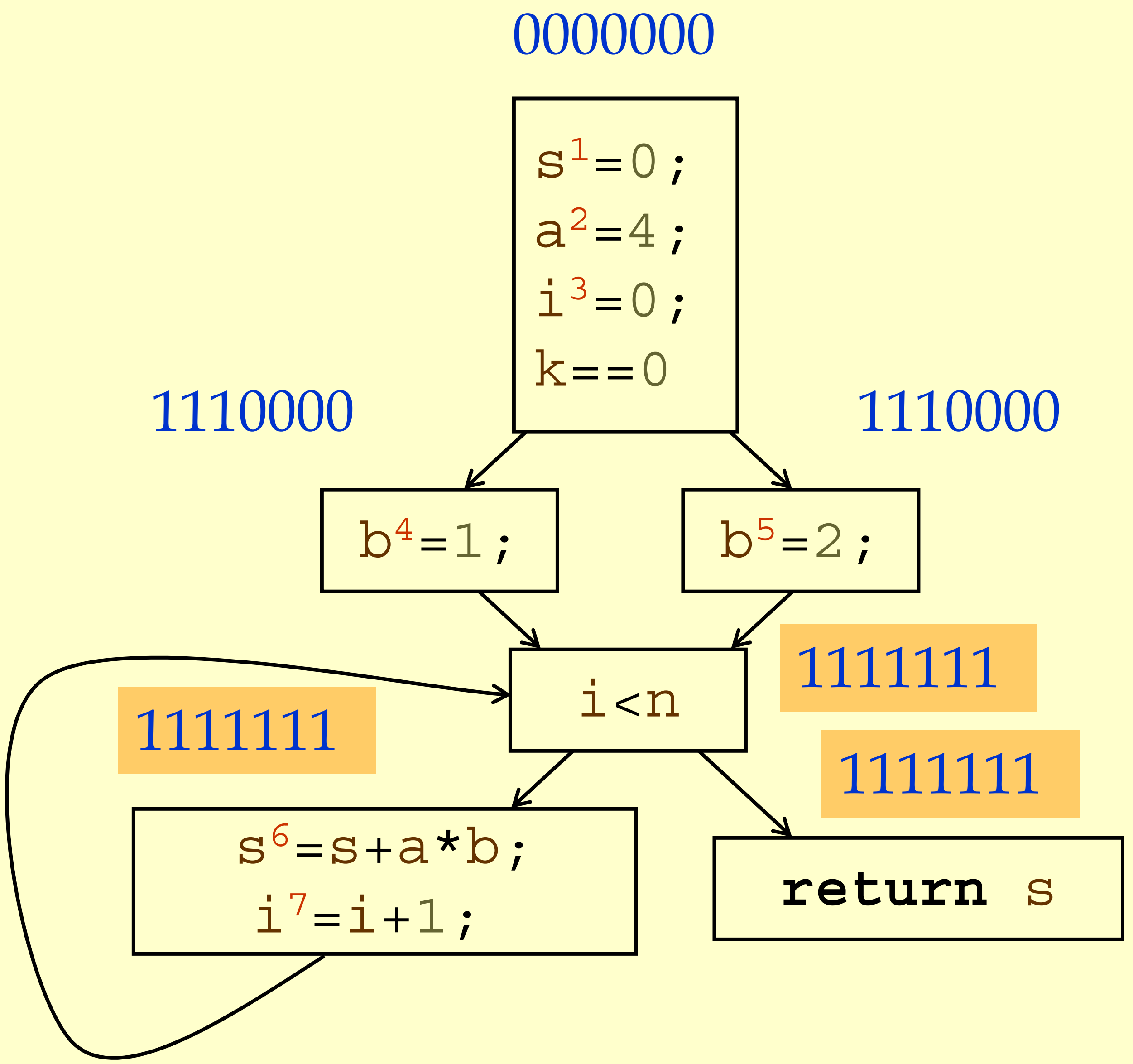
No!

One reaching definition with `b=1`

One reaching definition with `b=2`

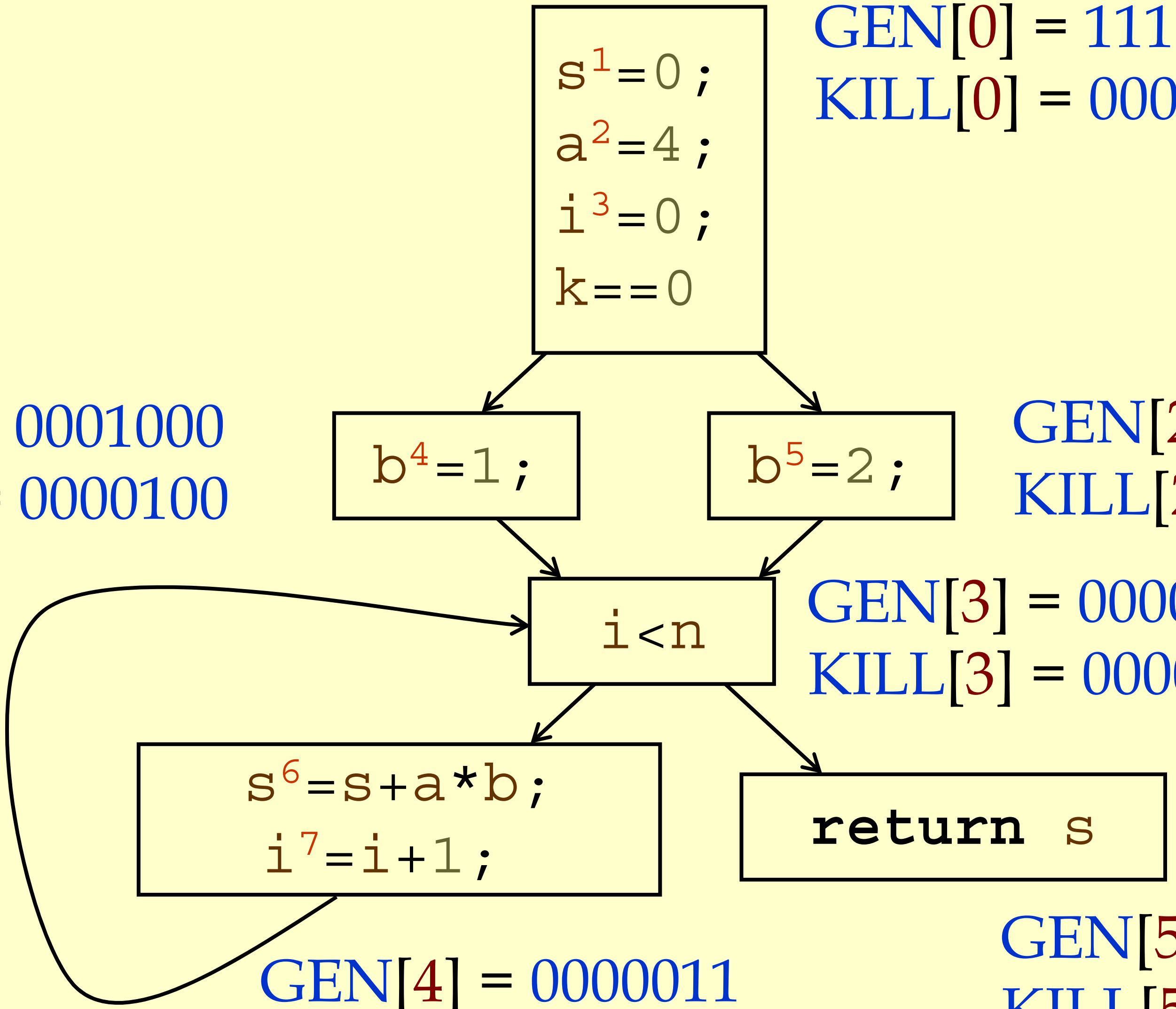
Computing Reaching Definitions

- 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.



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

GEN[1] = 0001000
KILL[1] = 0000100

GEN[2] = 0000100
KILL[2] = 0001000

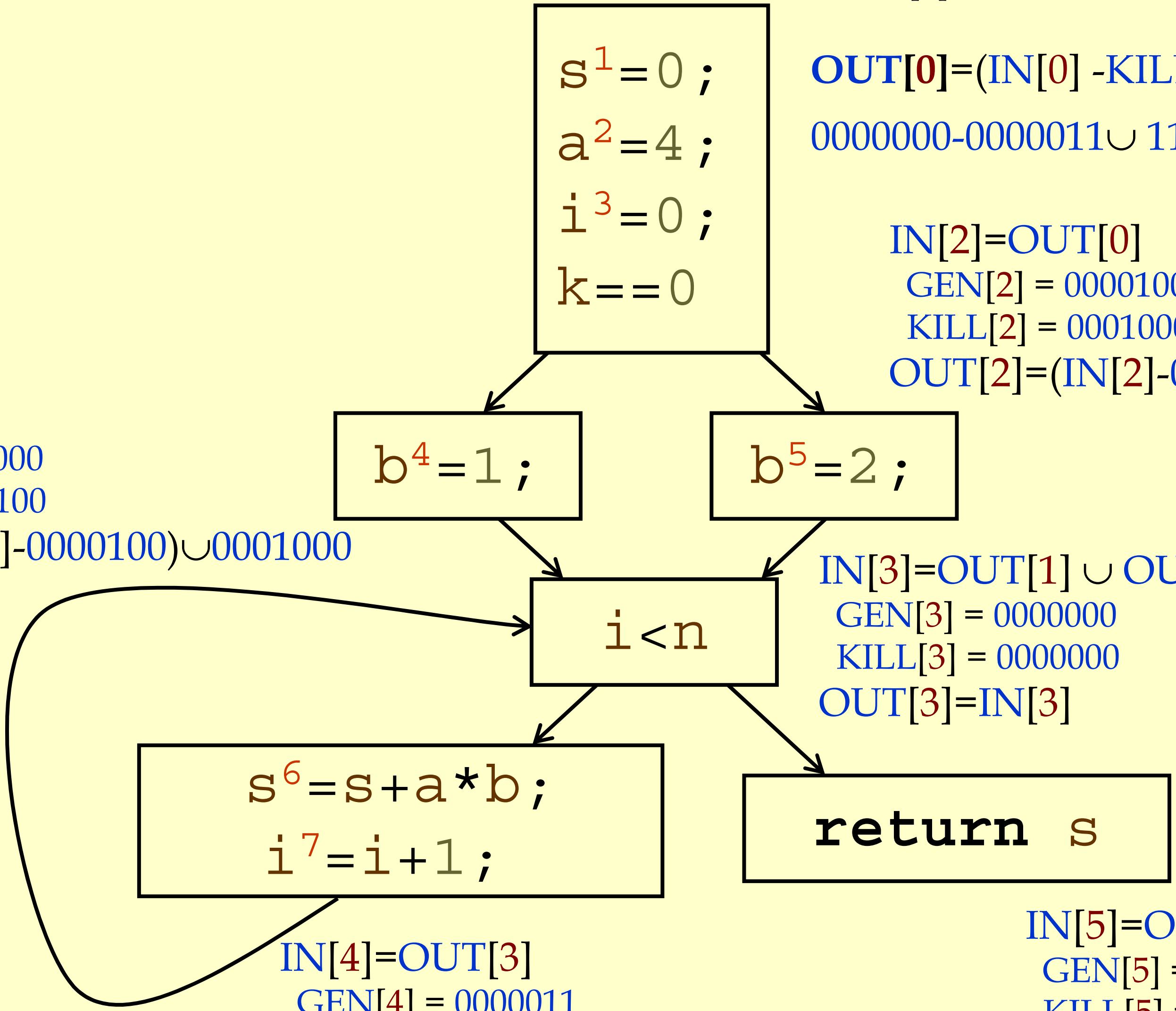
GEN[3] = 0000000
KILL[3] = 0000000

GEN[4] = 0000011
KILL[4] = 1010000

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 n2N
   OUT[n] = ;;                          // Or OUT[n] = GEN[n];
Changed = N;                            // N = all nodes in graph
while (Changed != ;)                    // Until fixed point reached.
  choose n2Changed;                     // Node from worklist
  Changed=Changed-{n};                  // Remove from worklist
   OldOut = OUT[n]                      // Remember old result
   IN[n] = ;;                           // Calculate IN as join
   for all nodes p2predecessors(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 s2successors(n)
         Changed=Changed∪{s};           //Add succs to worklist
```

Question

- Does the algorithm halt?
  - We need some theory to answer this with confidence.

Summary

- Optimization is hard but fun.
- Terminology: CFG, BB, EBB, Program points.
- 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.