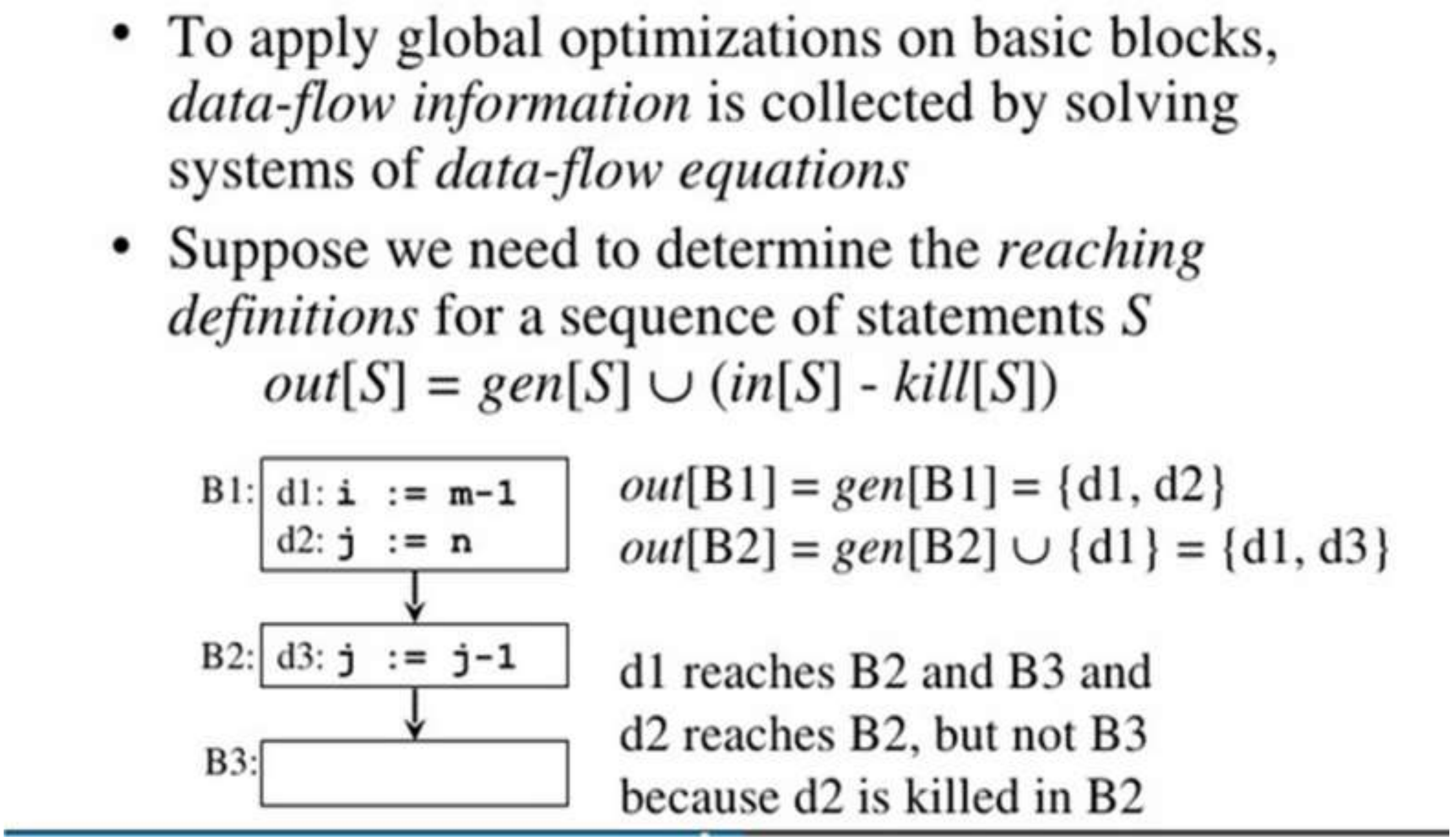
SNS COLLEGE OF TECHNOLOGY

(Autonomous)

COIMBATORE-35

Introduction to Global data flow Analysis & Code improving Transformations

Introduction to Global data flow Analysis



- To apply global optimizations on basic blocks, *data-flow information* is collected by solving systems of *data-flow equations*
- Suppose we need to determine the *reaching definitions* for a sequence of statements *S*

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

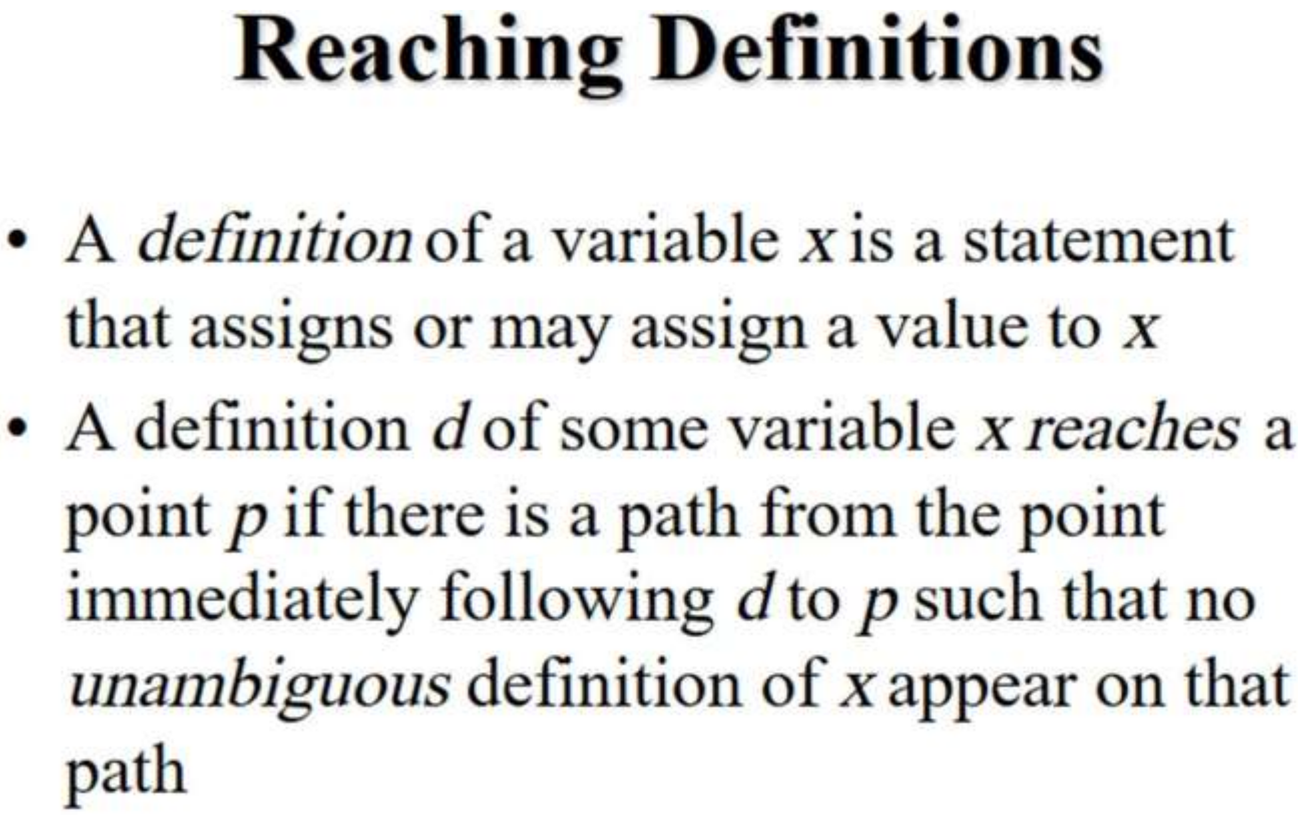
```
B1: d1: i  := m-1
    d2: j  := n

B2: d3: j  := j-1

B3:
```

$out[B1] = gen[B1] = \{d1, d2\}$

$out[B2] = gen[B2] \cup \{d1\} = \{d1, d3\}$

d1 reaches B2 and B3 and d2 reaches B2, but not B3 because d2 is killed in B2

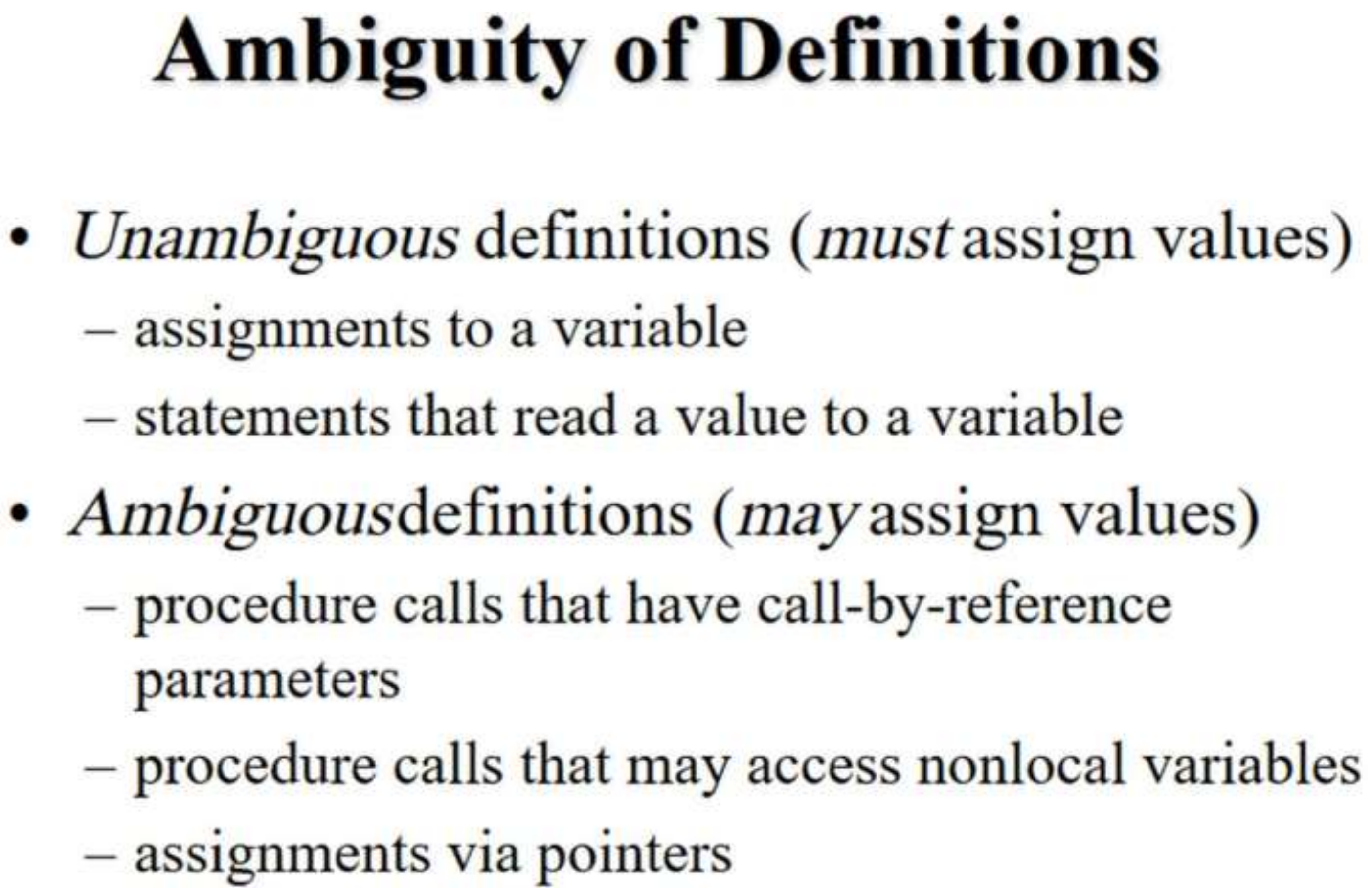
Introduction to Global data flow Analysis



Reaching Definitions

- A *definition* of a variable *x* is a statement that assigns or may assign a value to *x*
- A definition *d* of some variable *x* *reaches* a point *p* if there is a path from the point immediately following *d* to *p* such that no *unambiguous* definition of *x* appear on that path

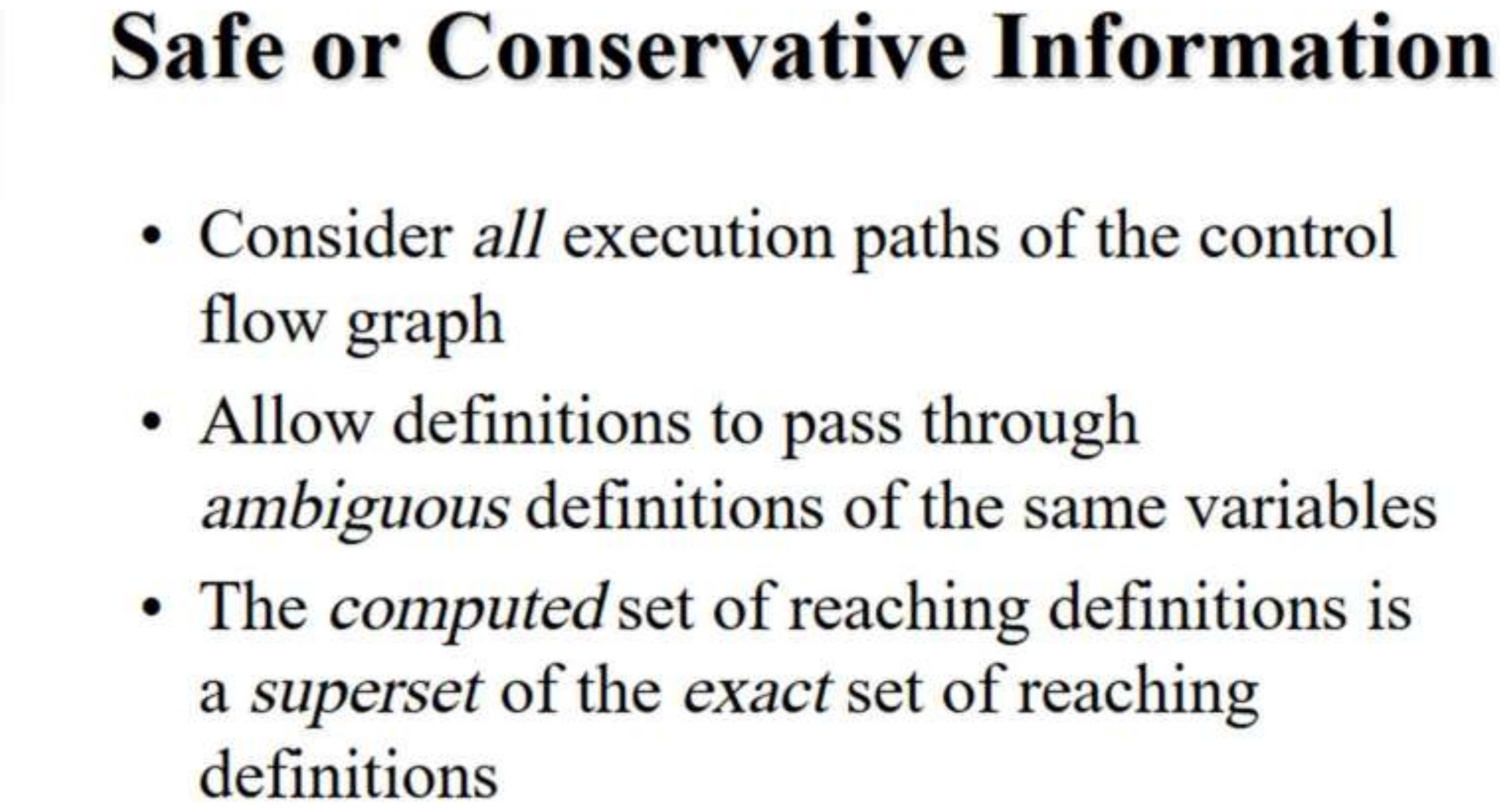
Introduction to Global data flow Analysis



Ambiguity of Definitions

- *Unambiguous* definitions (*must* assign values)
  - – assignments to a variable
  - – statements that read a value to a variable
- *Ambiguous* definitions (*may* assign values)
  - – procedure calls that have call-by-reference parameters
  - – procedure calls that may access nonlocal variables
  - – assignments via pointers

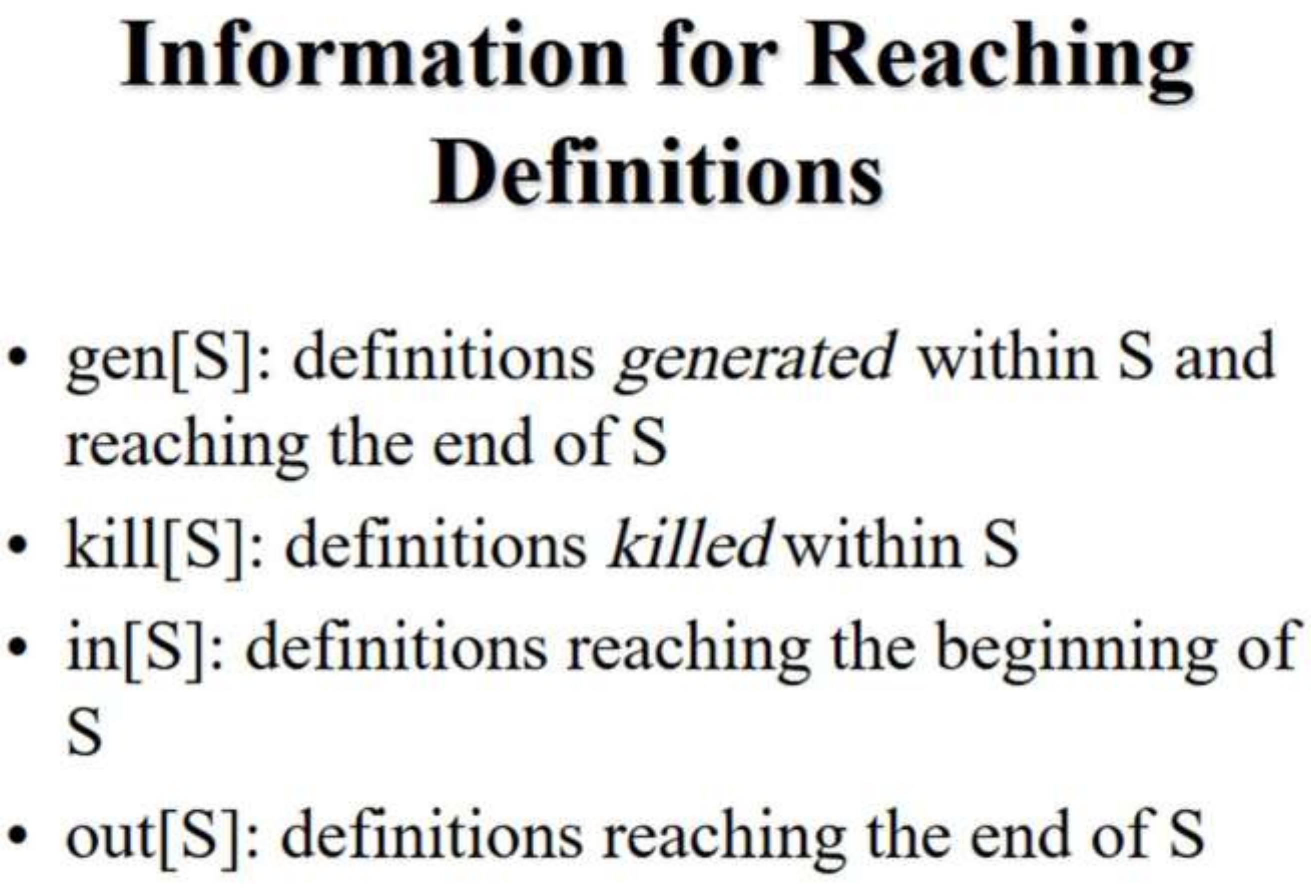
Introduction to Global data flow Analysis



Safe or Conservative Information

- Consider *all* execution paths of the control flow graph
- Allow definitions to pass through *ambiguous* definitions of the same variables
- The *computed* set of reaching definitions is a *superset* of the *exact* set of reaching definitions

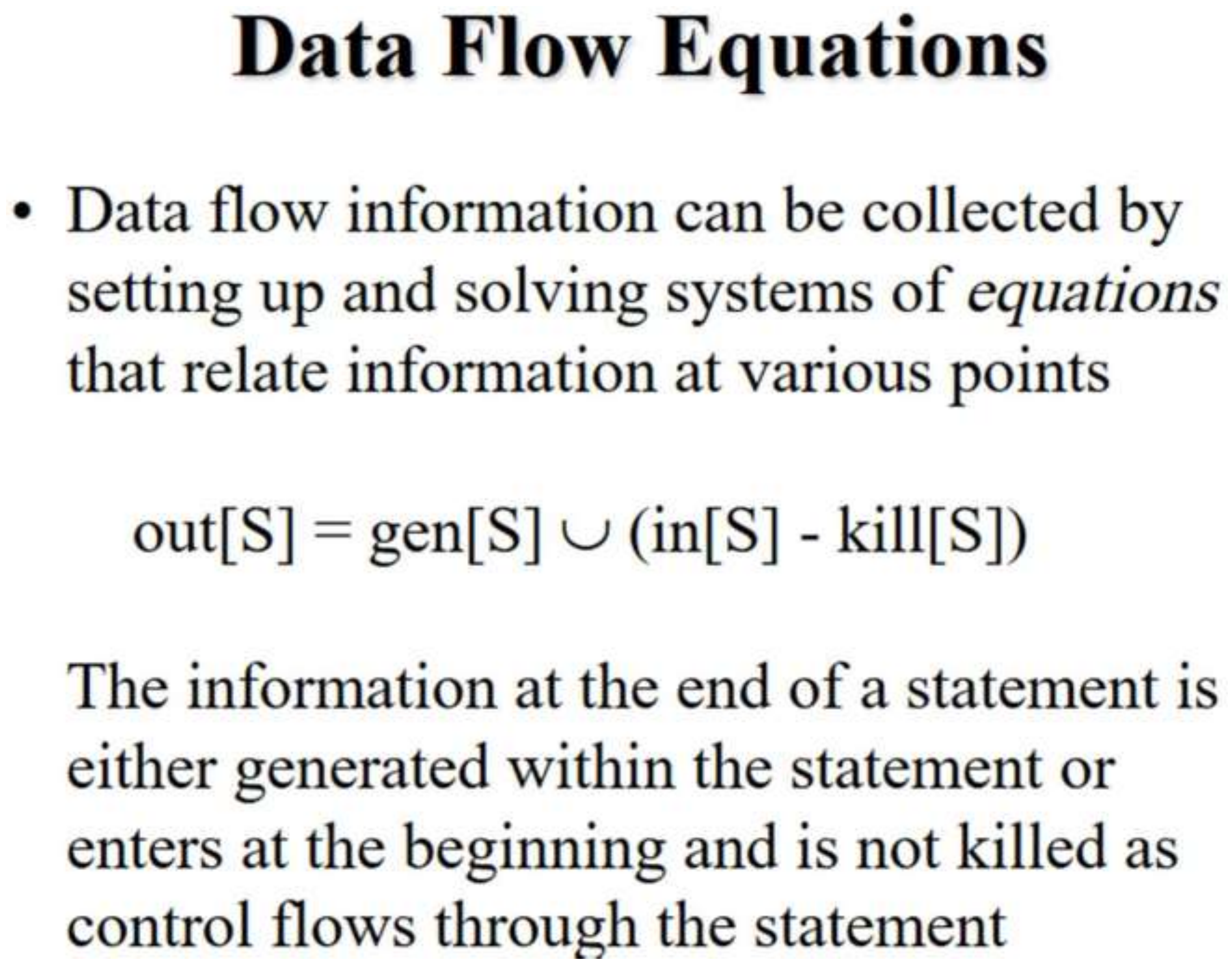
Introduction to Global data flow Analysis



Information for Reaching Definitions

- gen[S]: definitions *generated* within S and reaching the end of S
- kill[S]: definitions *killed* within S
- in[S]: definitions reaching the beginning of S
- out[S]: definitions reaching the end of S

Introduction to Global data flow Analysis



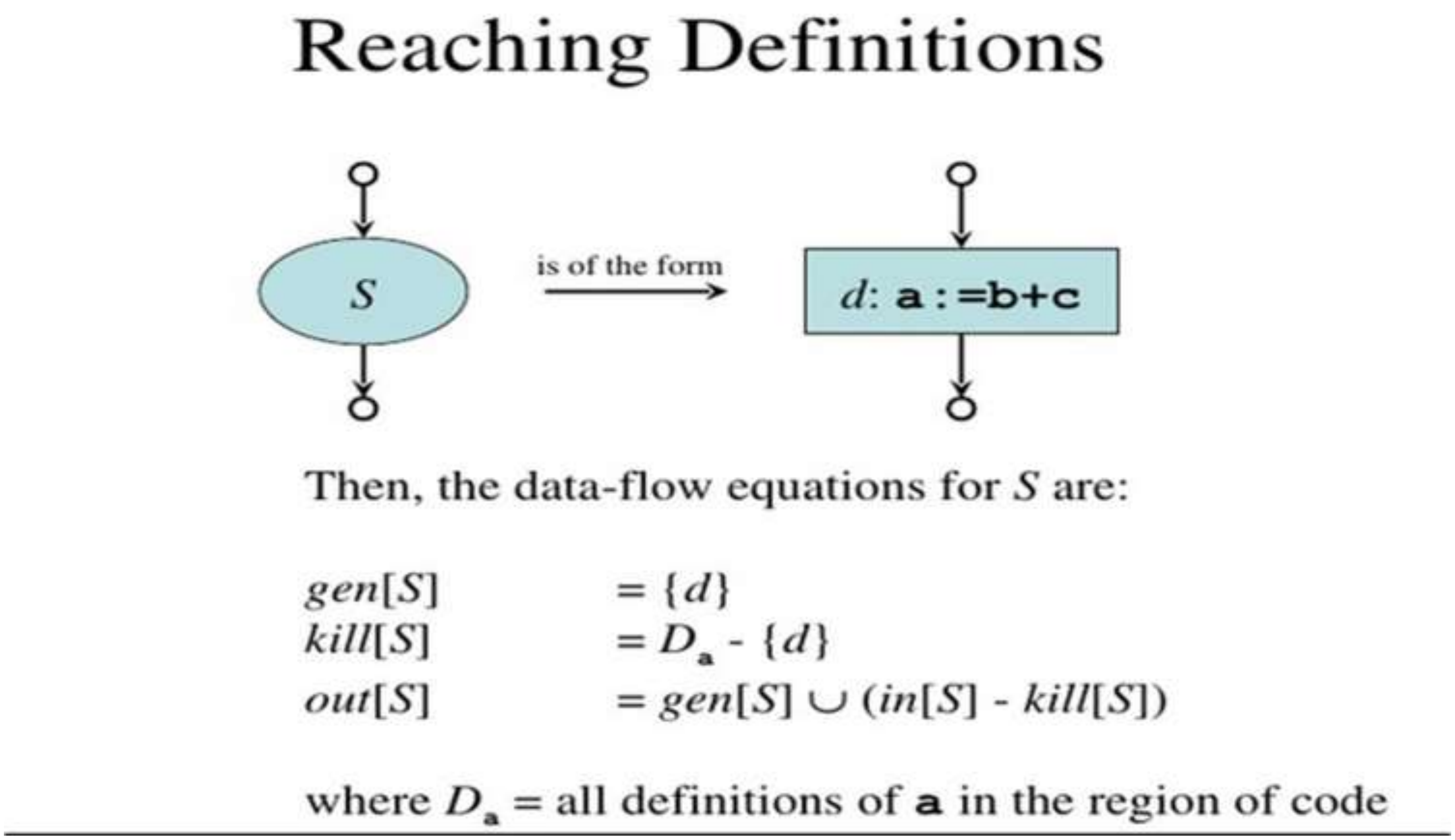
Data Flow Equations

- Data flow information can be collected by setting up and solving systems of *equations* that relate information at various points

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

The information at the end of a statement is either generated within the statement or enters at the beginning and is not killed as control flows through the statement

Introduction to Global data flow Analysis



Reaching Definitions

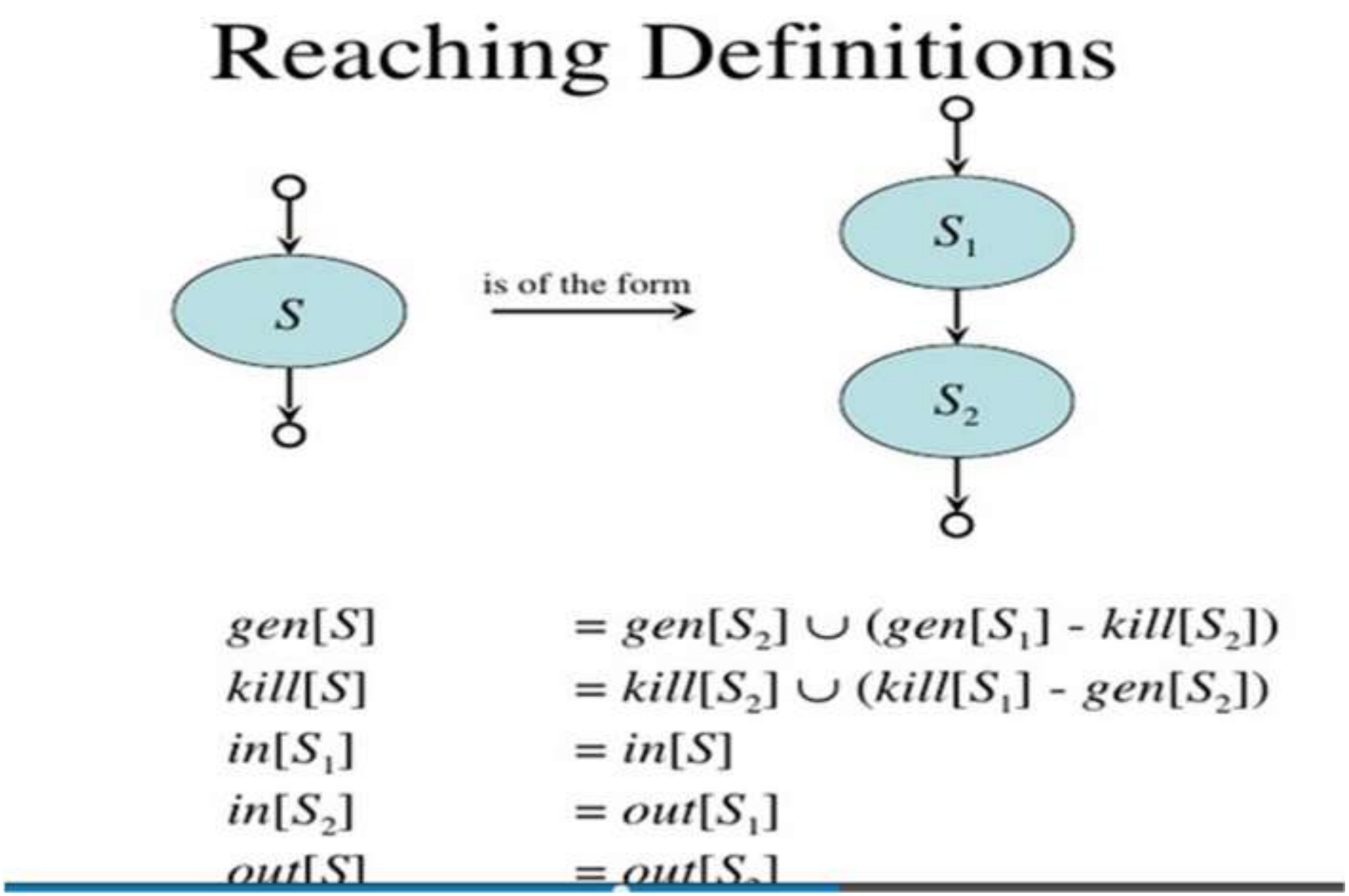
S
is of the form
d: a:=b+c

Then, the data-flow equations for S are:

$$\begin{aligned}
gen[S] &= \{d\} \\
kill[S] &= D_{\mathbf{a}} - \{d\} \\
out[S] &= gen[S] \cup (in[S] - kill[S])
\end{aligned}$$

where $D_{\mathbf{a}}$ = all definitions of **a** in the region of code

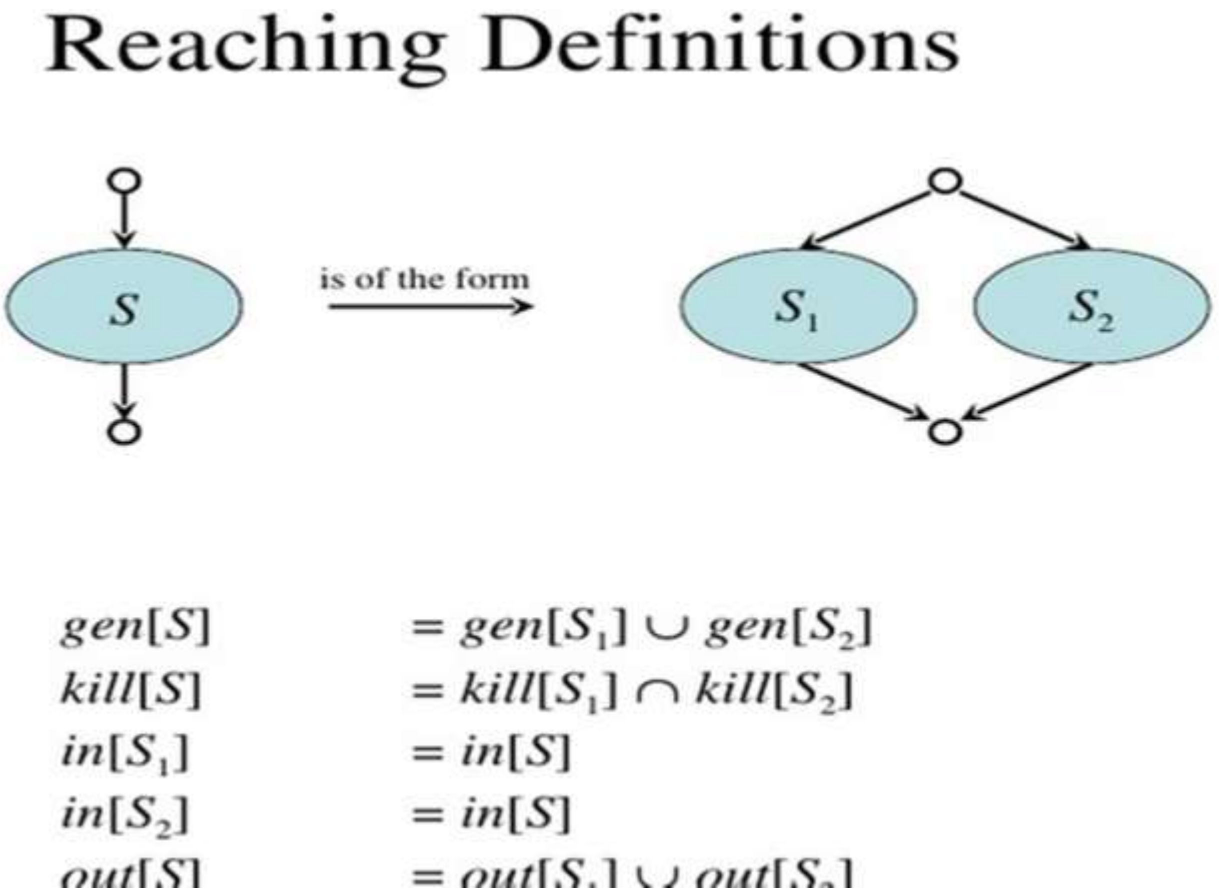
Introduction to Global data flow Analysis



Reaching Definitions

$$
\begin{aligned}
gen[S] &= gen[S_2] \cup (gen[S_1] - kill[S_2]) \\
kill[S] &= kill[S_2] \cup (kill[S_1] - gen[S_2]) \\
in[S_1] &= in[S] \\
in[S_2] &= out[S_1] \\
out[S] &= out[S_2]
\end{aligned}
$$

Introduction to Global data flow Analysis



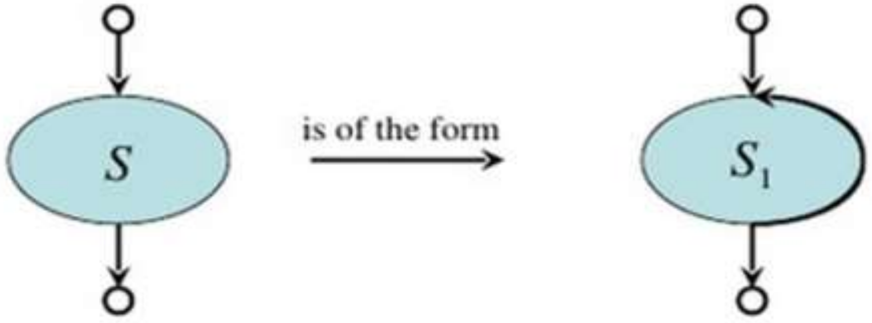
Reaching Definitions

S
is of the form
S_1
S_2

$$
\begin{aligned}
gen[S] &= gen[S_1] \cup gen[S_2] \\
kill[S] &= kill[S_1] \cap kill[S_2] \\
in[S_1] &= in[S] \\
in[S_2] &= in[S] \\
out[S] &= out[S_1] \cup out[S_2]
\end{aligned}
$$

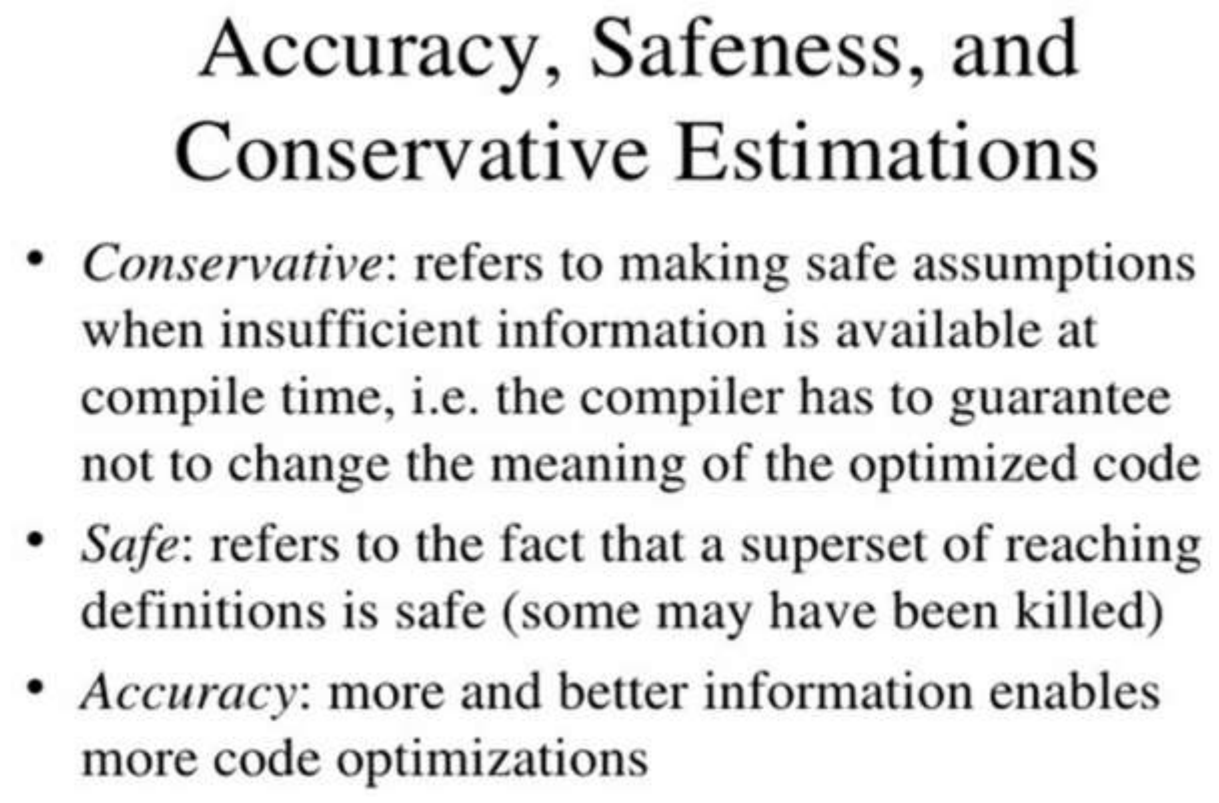
Introduction to Global data flow Analysis

Reaching Definitions



$$
\begin{aligned}
gen[S] &= gen[S_1] \\
kill[S] &= kill[S_1] \\
in[S_1] &= in[S] \cup gen[S_1] \\
out[S] &= out[S_1]
\end{aligned}
$$

Introduction to Global data flow Analysis

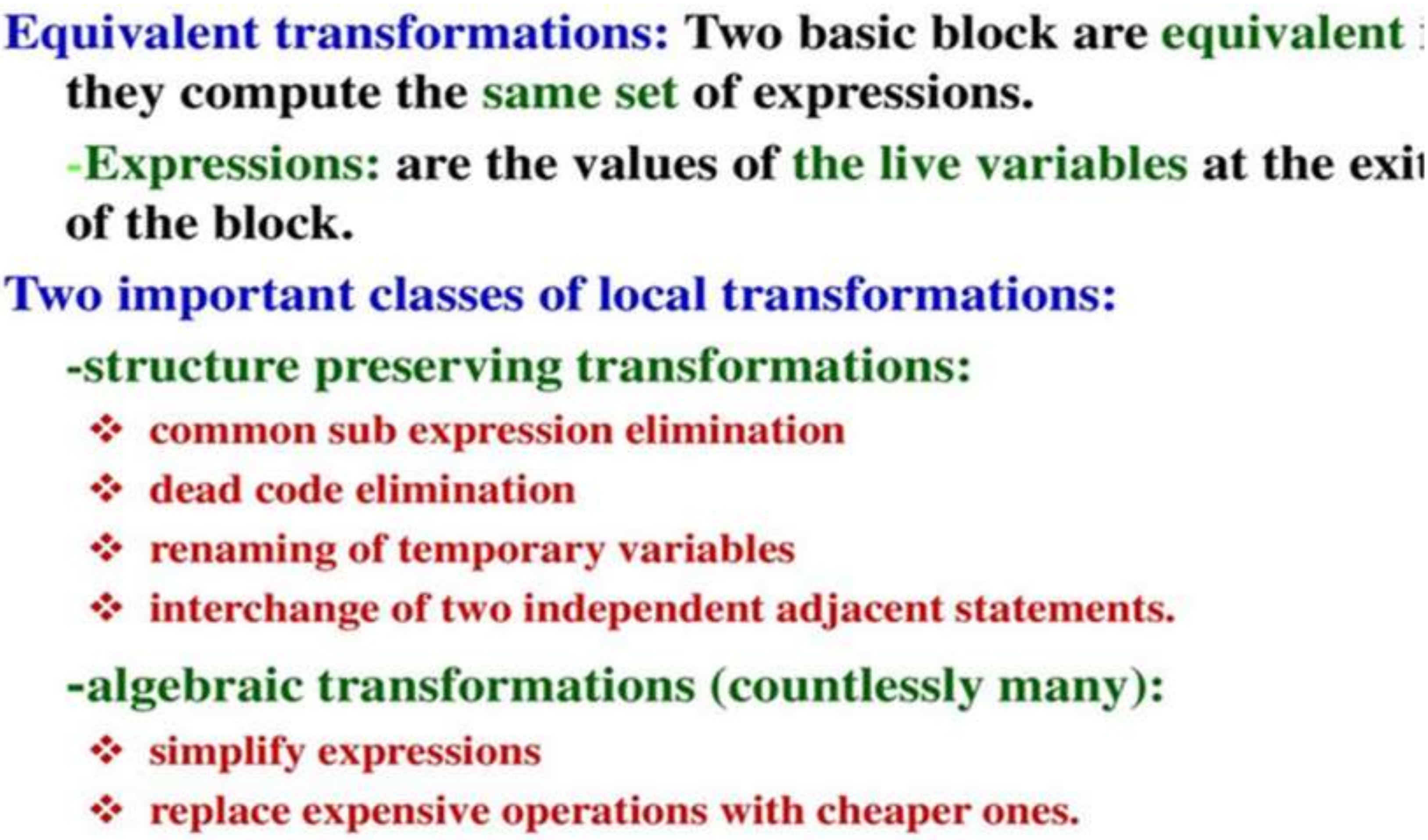


Accuracy, Safeness, and Conservative Estimations

- *Conservative*: refers to making safe assumptions when insufficient information is available at compile time, i.e. the compiler has to guarantee not to change the meaning of the optimized code
- *Safe*: refers to the fact that a superset of reaching definitions is safe (some may have been killed)
- *Accuracy*: more and better information enables more code optimizations

Code improving Transformations

Code improving Transformations



Equivalent transformations: Two basic block are equivalent they compute the same set of expressions.

-Expressions: are the values of the live variables at the exit of the block.

Two important classes of local transformations:

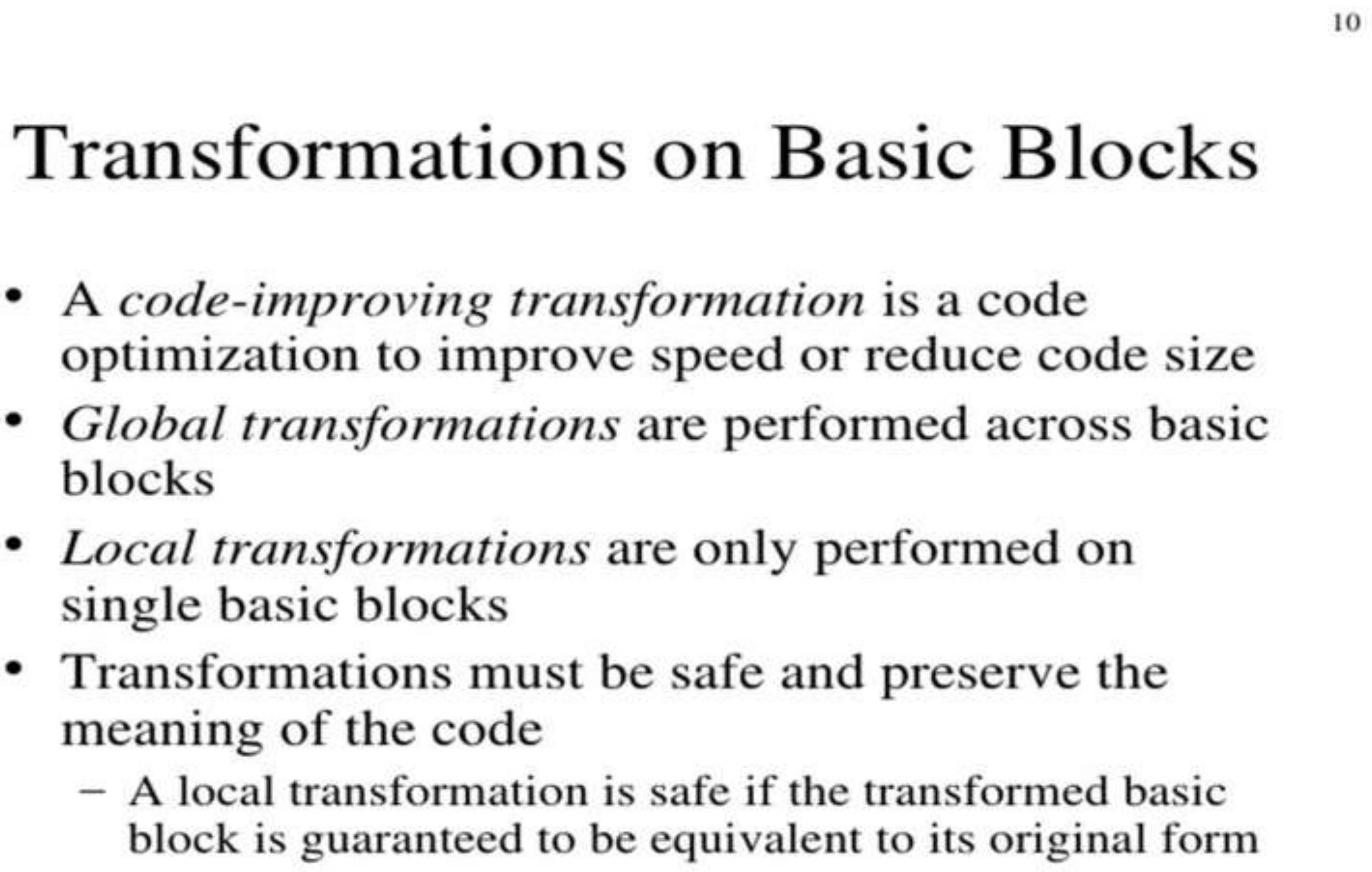
-structure preserving transformations:

- common sub expression elimination
- dead code elimination
- renaming of temporary variables
- interchange of two independent adjacent statements.

-algebraic transformations (countlessly many):

- simplify expressions
- replace expensive operations with cheaper ones.

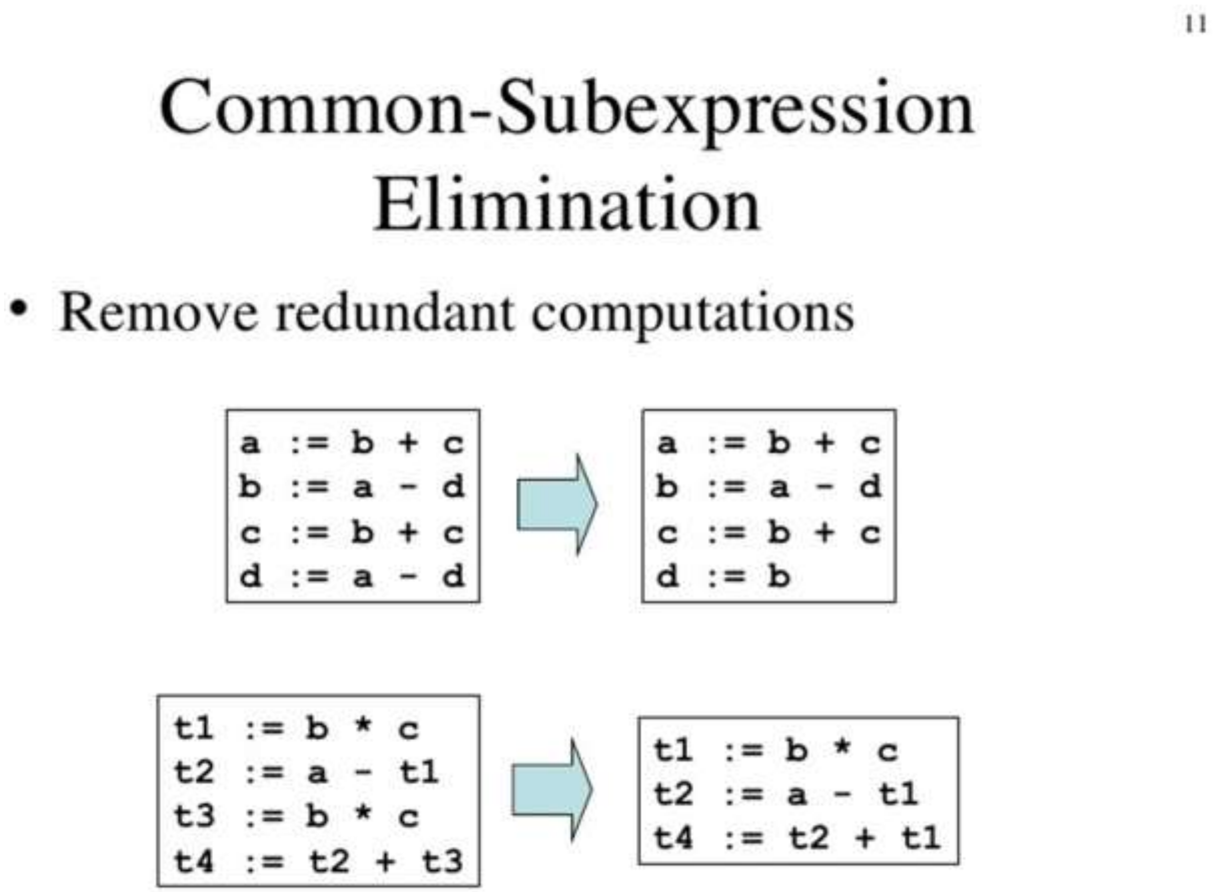
Code improving Transformations



Transformations on Basic Blocks

- A *code-improving transformation* is a code optimization to improve speed or reduce code size
- *Global transformations* are performed across basic blocks
- *Local transformations* are only performed on single basic blocks
- Transformations must be safe and preserve the meaning of the code
  - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form

Code improving Transformations



11

Common-Subexpression Elimination

- Remove redundant computations

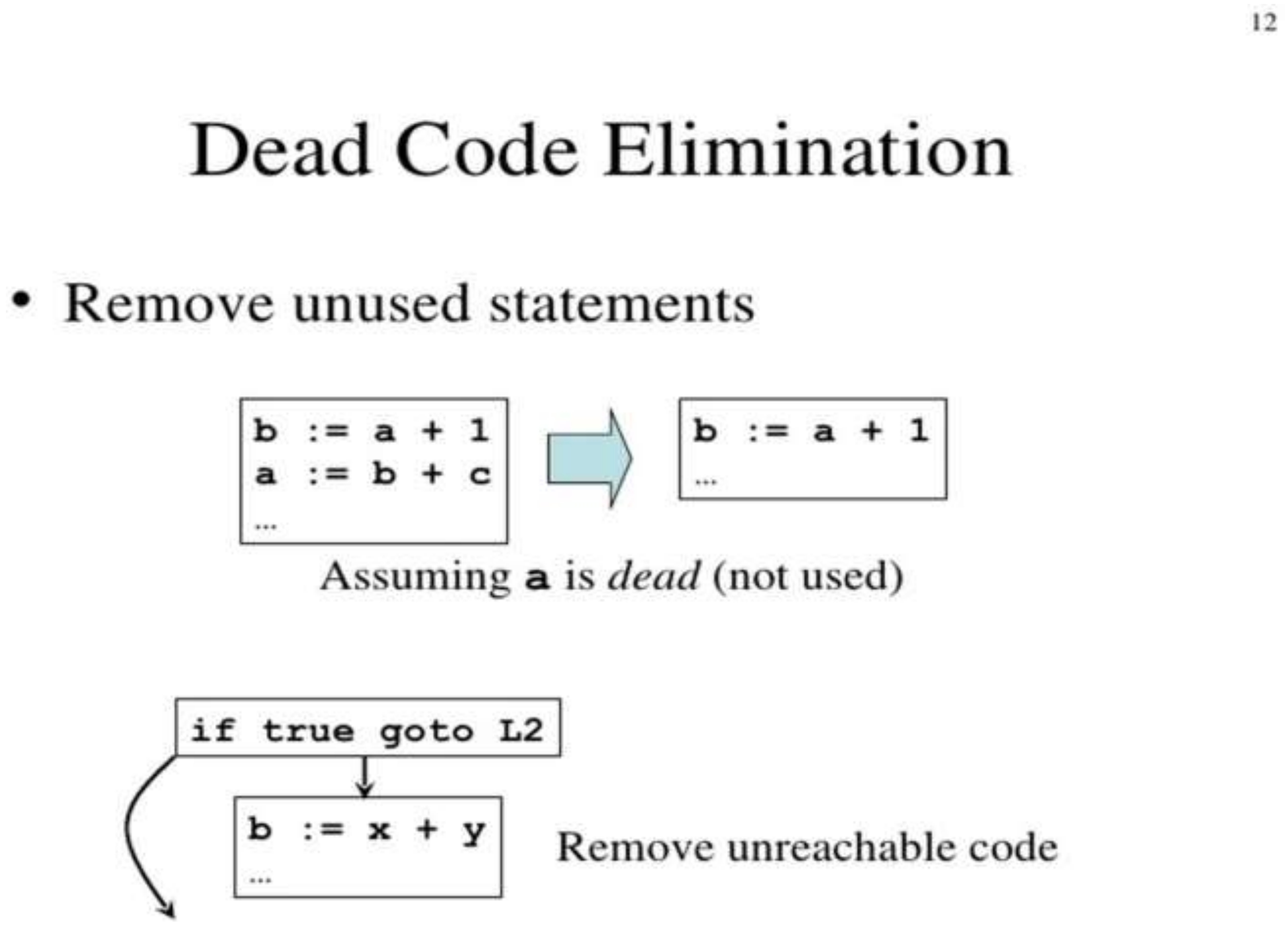
```
a  := b + c
b  := a - d
c  := b + c
d  := a - d
```

```
a  := b + c
b  := a - d
c  := b + c
d  := b
```

```
t1  := b * c
t2  := a - t1
t3  := b * c
t4  := t2 + t3
```

```
t1  := b * c
t2  := a - t1
t4  := t2 + t1
```

Code improving Transformations



12

Dead Code Elimination

- Remove unused statements

```
b := a + 1
a := b + c
...
```

```
b := a + 1
...
```

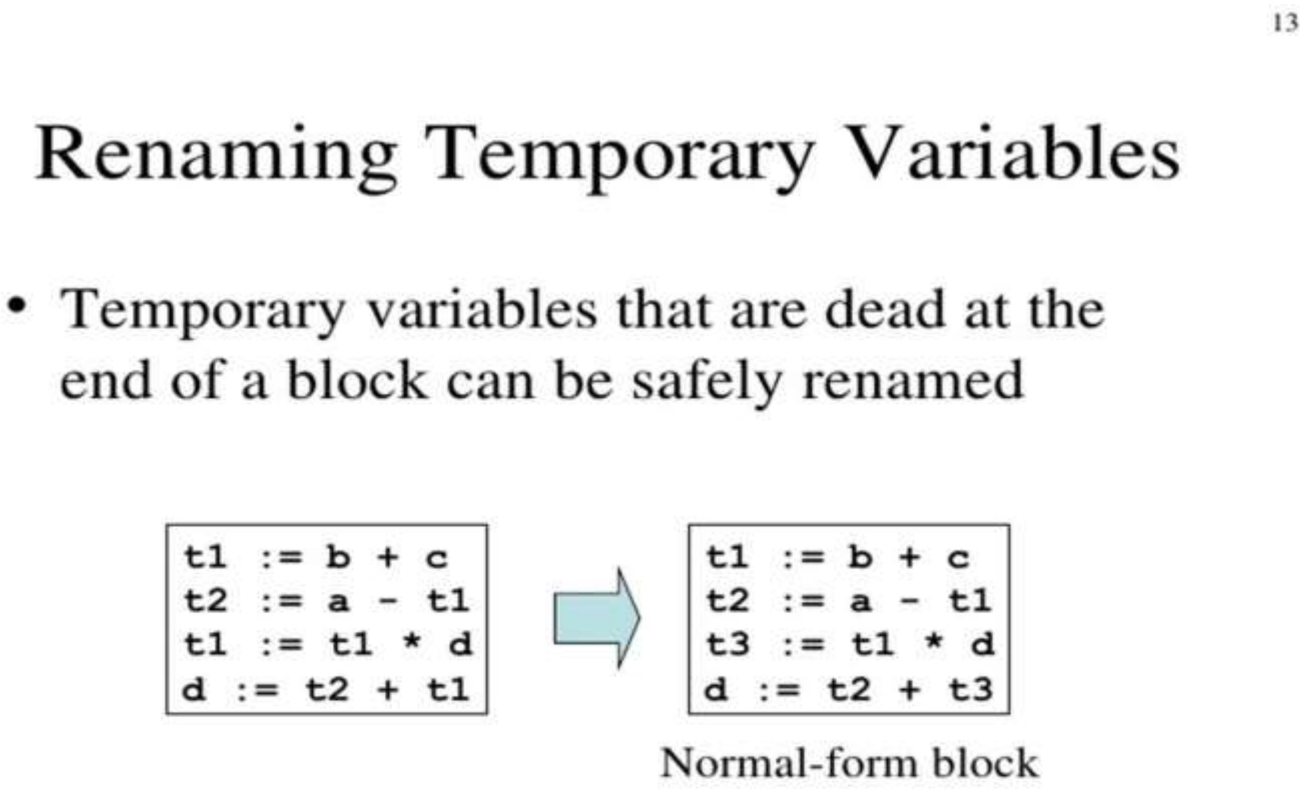
Assuming **a** is *dead* (not used)

```
if true goto L2
```

```
b := x + y
...
```

Remove unreachable code

Code improving Transformations



Renaming Temporary Variables

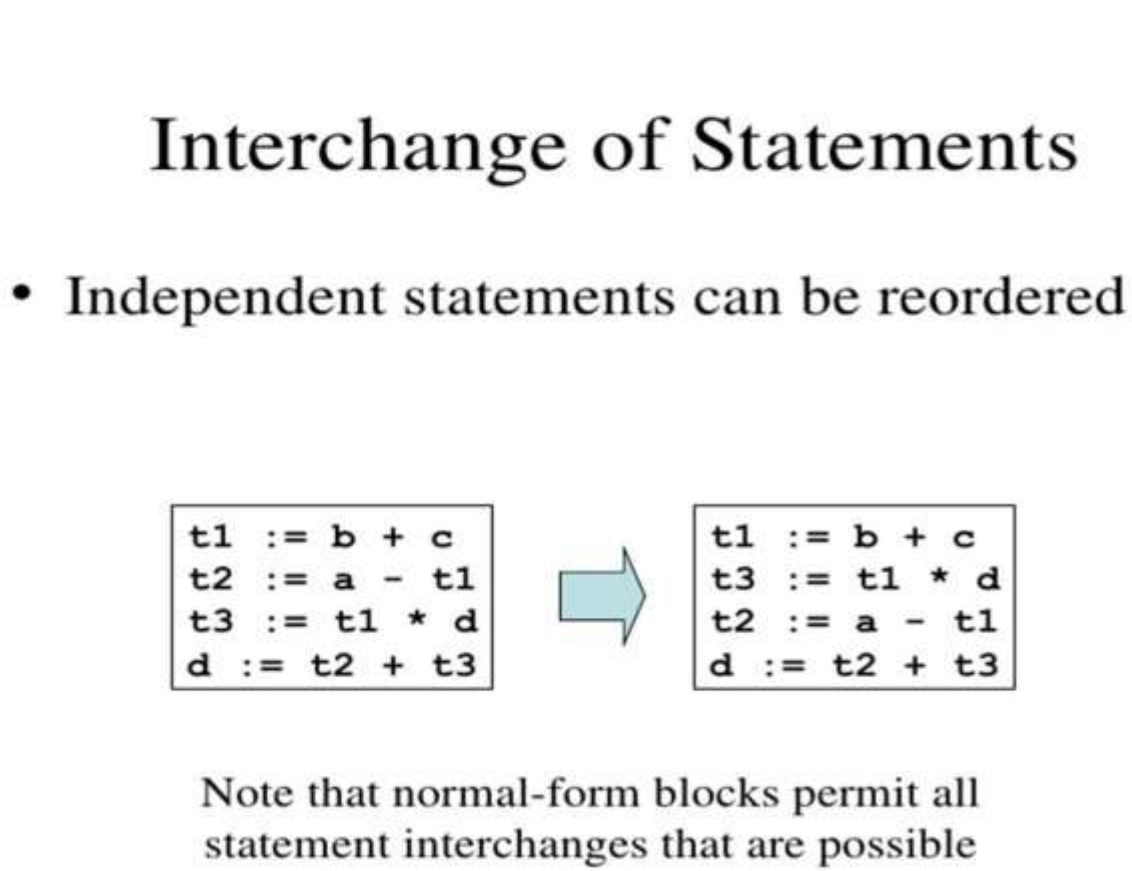
- Temporary variables that are dead at the end of a block can be safely renamed

```
t1  := b + c
t2  := a - t1
t1  := t1 * d
d  := t2 + t1
```

```
t1  := b + c
t2  := a - t1
t3  := t1 * d
d  := t2 + t3
```

Normal-form block

Code improving Transformations



14

Interchange of Statements

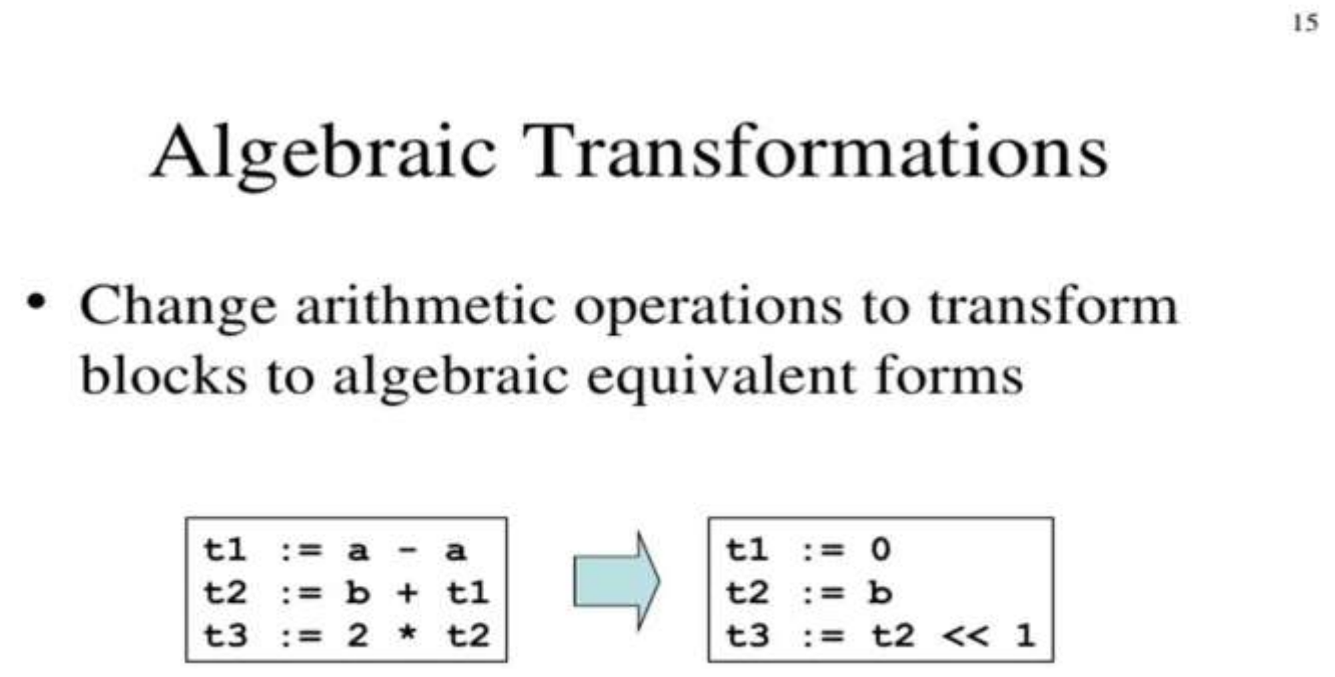
- Independent statements can be reordered

```
t1  := b + c
t2  := a - t1
t3  := t1 * d
d  := t2 + t3
```

```
t1  := b + c
t3  := t1 * d
t2  := a - t1
d  := t2 + t3
```

Note that normal-form blocks permit all statement interchanges that are possible

Code improving Transformations



15

Algebraic Transformations

- Change arithmetic operations to transform blocks to algebraic equivalent forms

```
t1  := a - a
t2  := b + t1
t3  := 2 * t2
```

```
t1  := 0
t2  := b
t3  := t2 << 1
```

Summarization