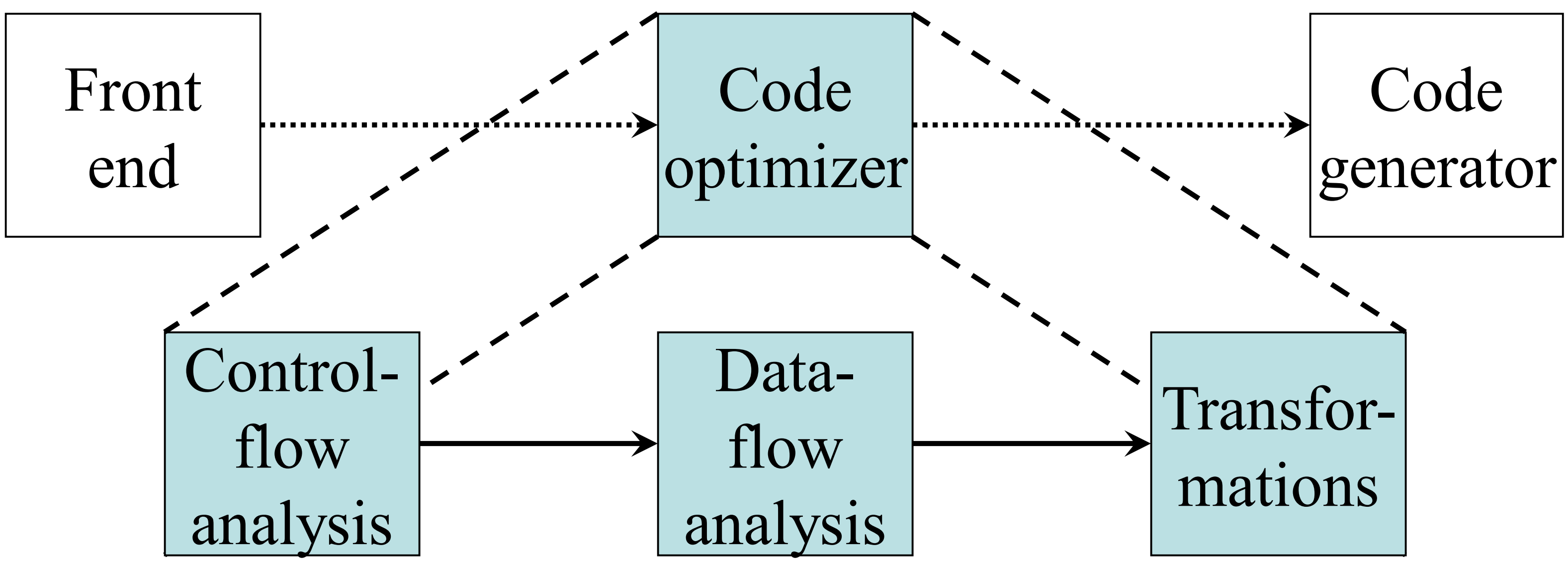
Code Optimization

Chapter 10

The Code Optimizer

- Control flow analysis: CFG (Ch. 9)
- Data-flow analysis
- Transformations



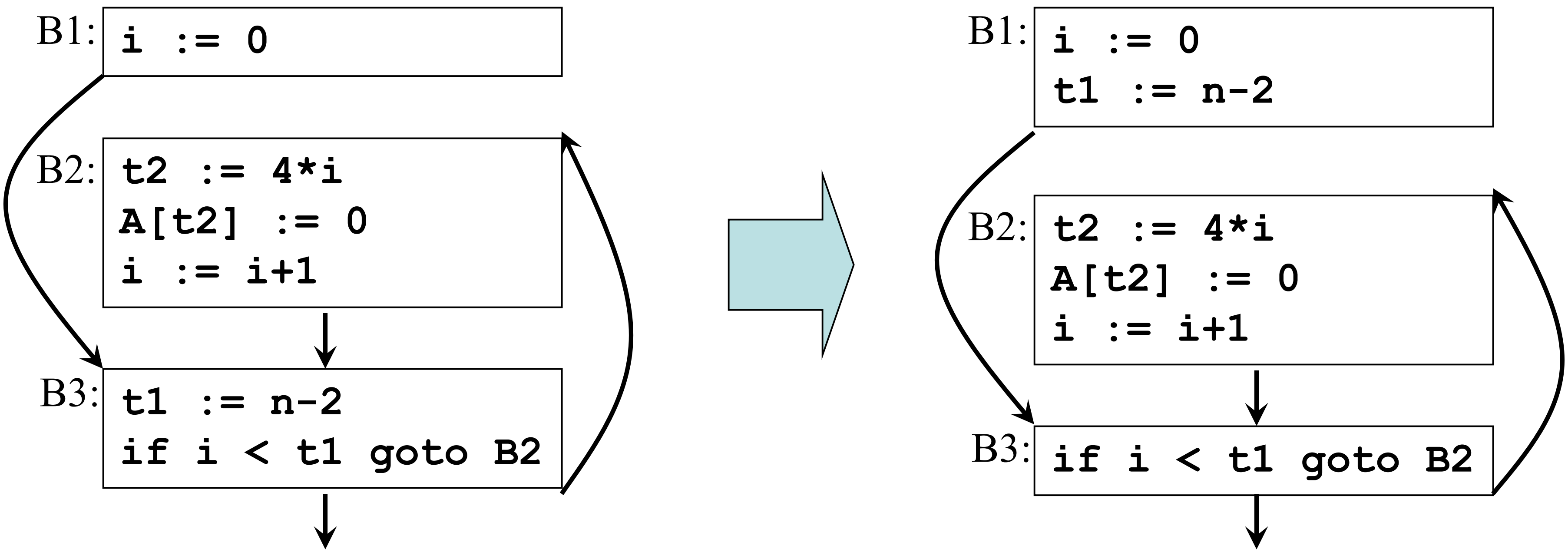
Code Optimizations

- Local/global common subexpression elimination
- Dead-code elimination
- Instruction reordering
- Constant folding
- Algebraic transformations
- Copy propagation
- *Loop optimizations*

Loop Optimizations

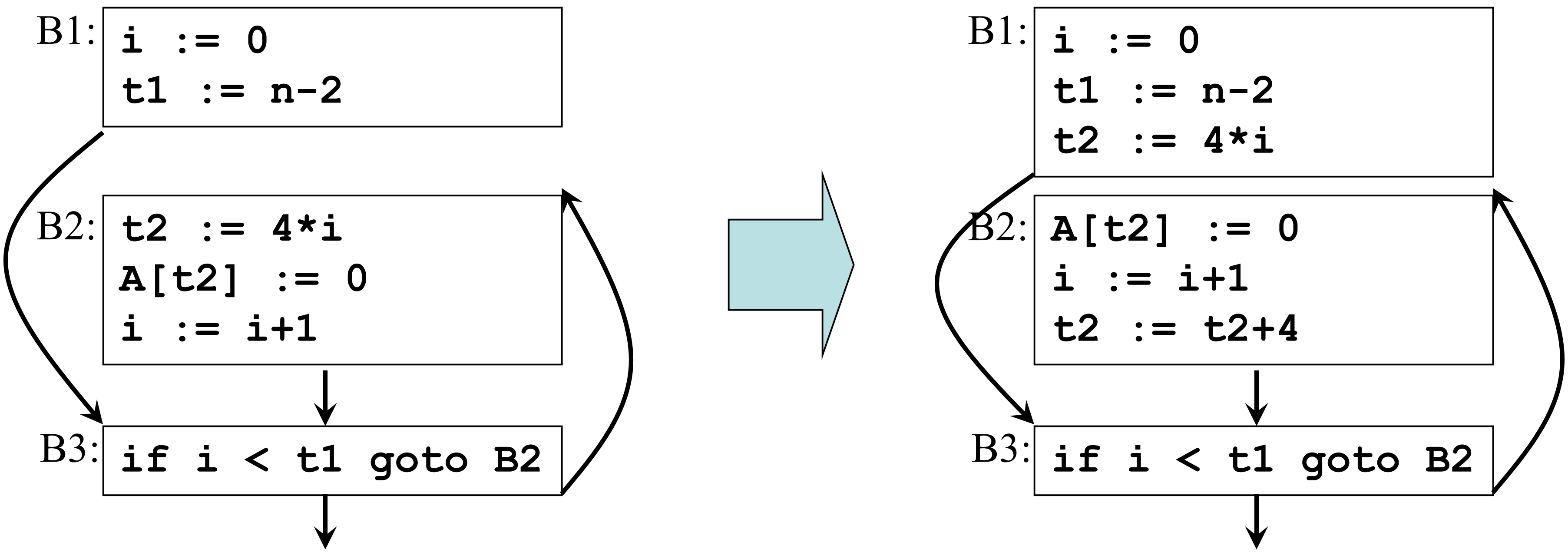
- Code motion
- Induction variable elimination
- Reduction in strength
- … lots more

Code Motion



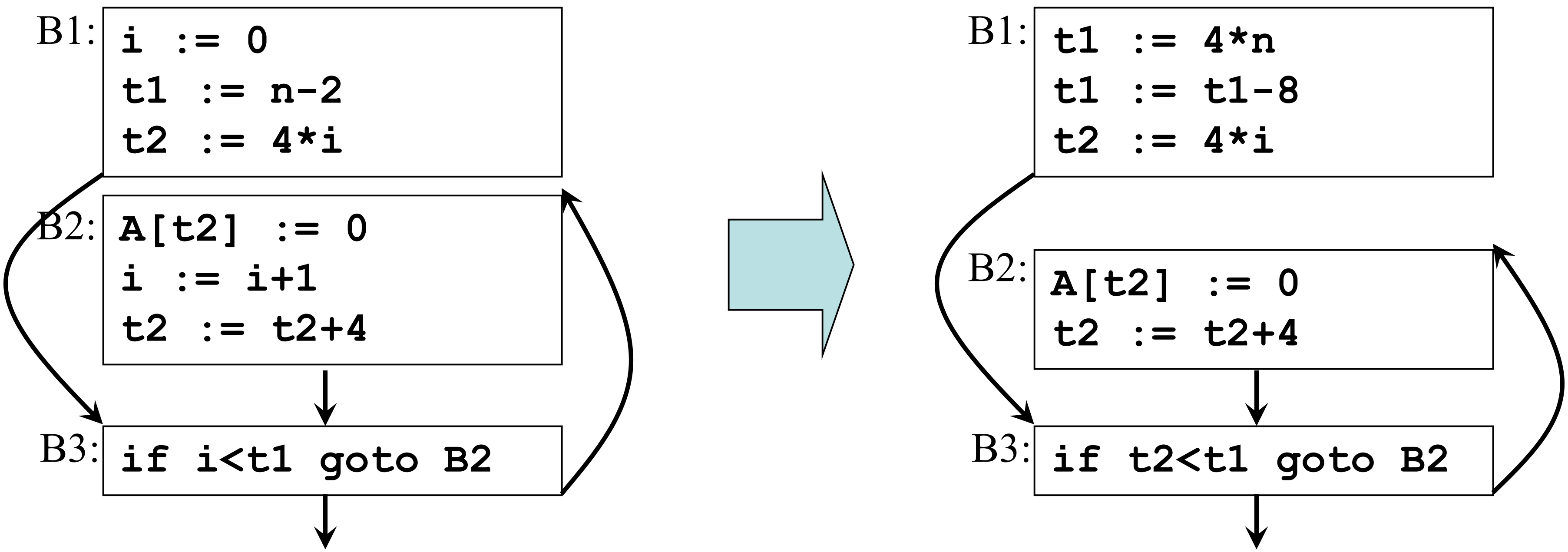
Move *loop-invariant computations* before the loop

Strength Reduction



Replace expensive computations with *induction variables*

Reduction Variable Elimination

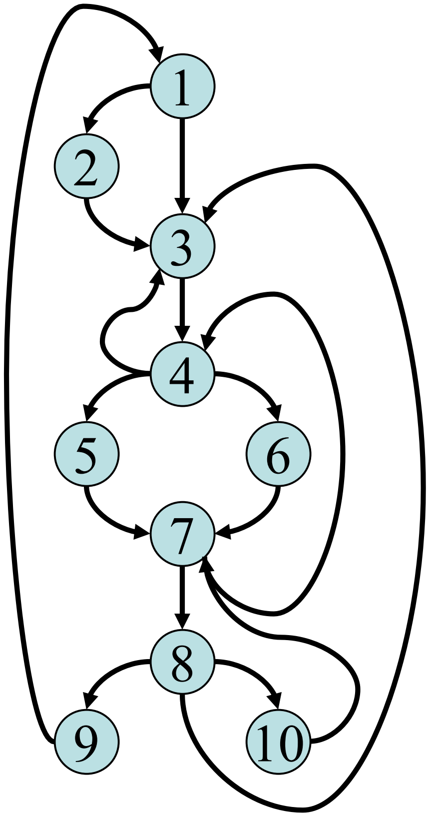


Replace induction variable in expressions with another

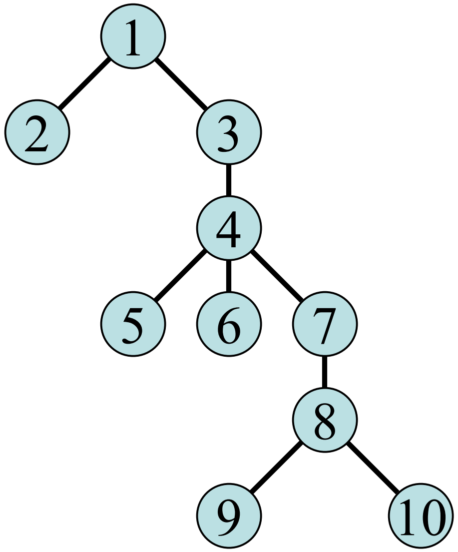
Determining Loops in Flow Graphs: Dominators

- Dominators: *d dom n*
  - Node *d* of a CFG *dominates* node *n* if *every* path from the initial node of the CFG to *n* goes through *d*
  - The loop entry dominates all nodes in the loop
- The *immediate dominator m* of a node *n* is the last dominator on the path from the initial node to *n*
  - If $d \neq n$ and *d dom n* then *d dom m*

Dominator Trees



CFG

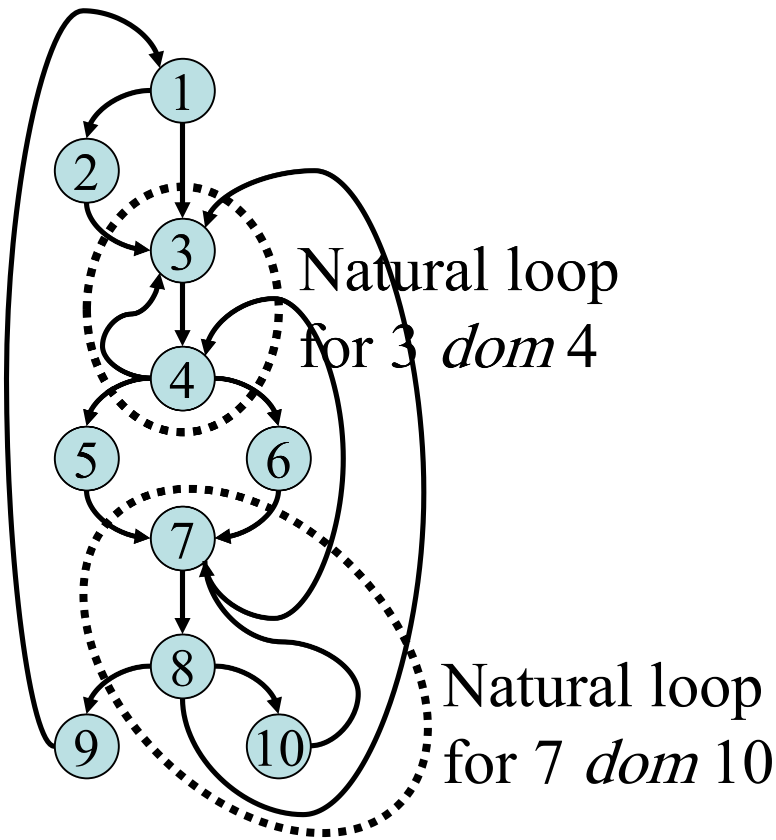


Dominator tree

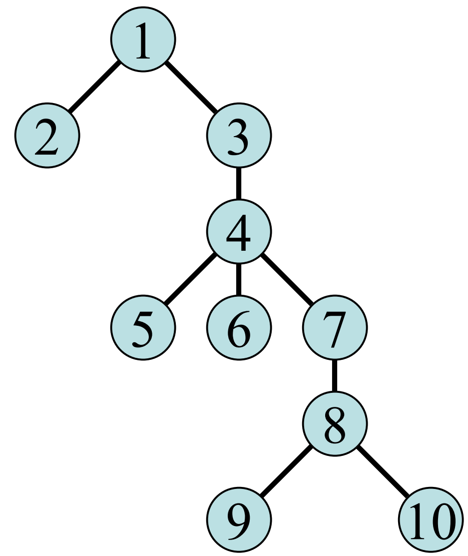
Natural Loops

- A *back edge* is is an edge $a \to b$ whose head b dominates its tail a
- Given a back edge $n \to d$
  - The *natural loop* consists of d plus the nodes that can reach n without going through d
  - The *loop header* is node d
- Unless two loops have the same header, they are disjoint or one is nested within the other
  - A nested loop is an *inner loop* if it contains no other loops

Natural (Inner) Loops Example



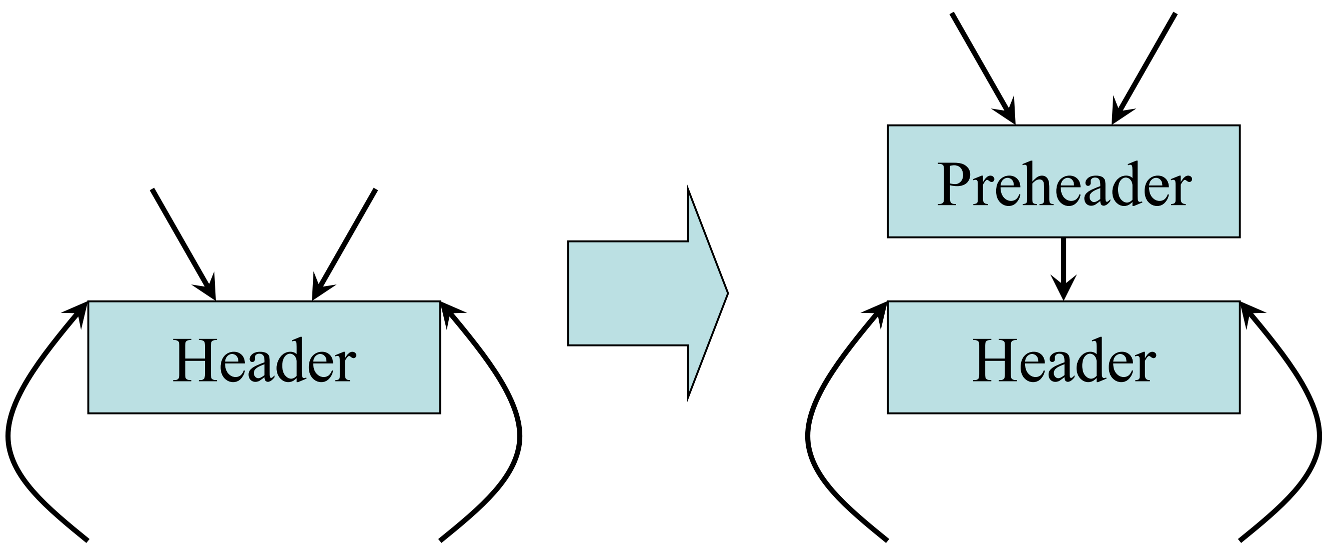
CFG



Dominator tree

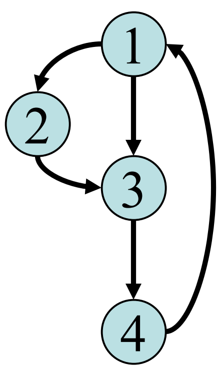
Pre-Headers

- To facilitate loop transformations, a compiler often adds a *preheader* to a loop
- Code motion, strength reduction, and other loop transformations populate the preheader

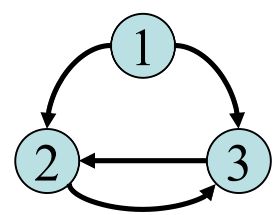


Reducible Flow Graphs

- *Reducible graph* = disjoint partition in forward and back edges such that the forward edges form an acyclic (sub)graph



Example of a reducible CFG

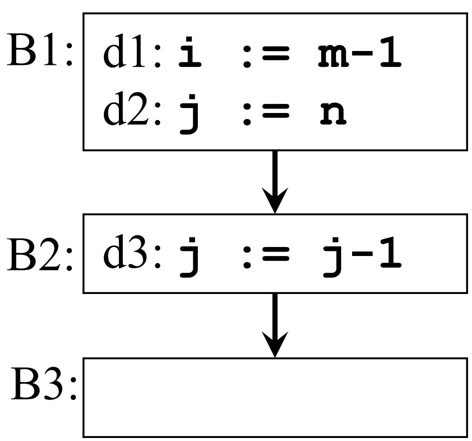


Example of a nonreducible CFG

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*

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

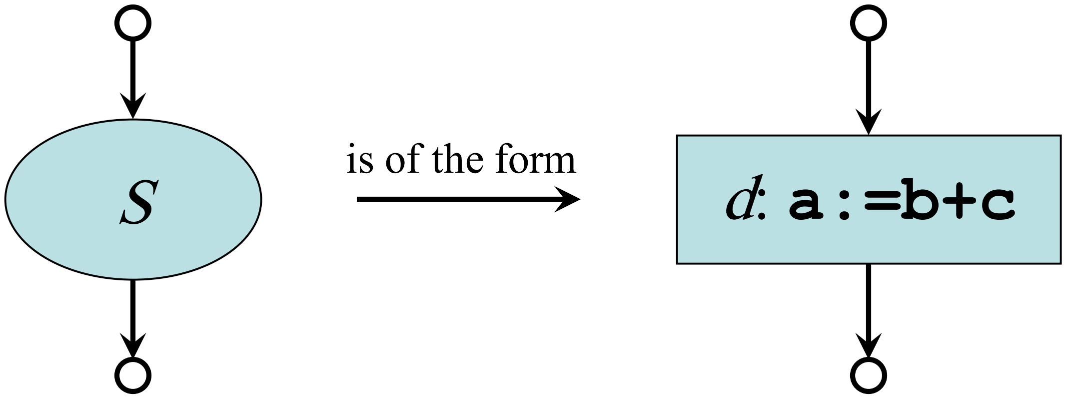


$out[\text{B1}] = gen[\text{B1}] = \{\text{d1, d2}\}$

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

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

Reaching Definitions

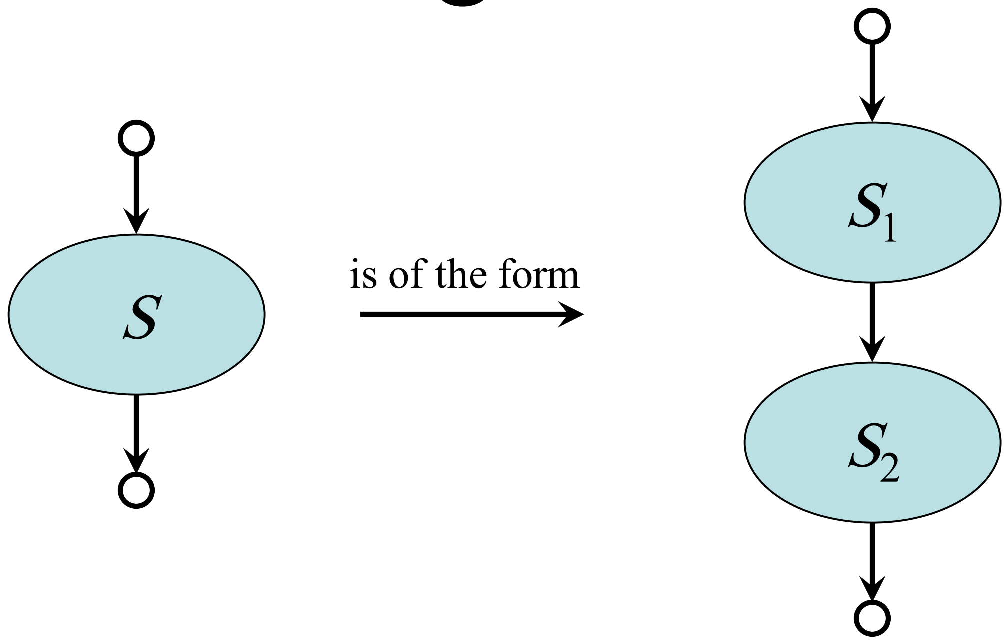


Then, the data-flow equations for S are:

$$gen[S] = \{d\}$$
$$kill[S] = D_{\mathtt{a}} - \{d\}$$
$$out[S] = gen[S] \cup (in[S] - kill[S])$$

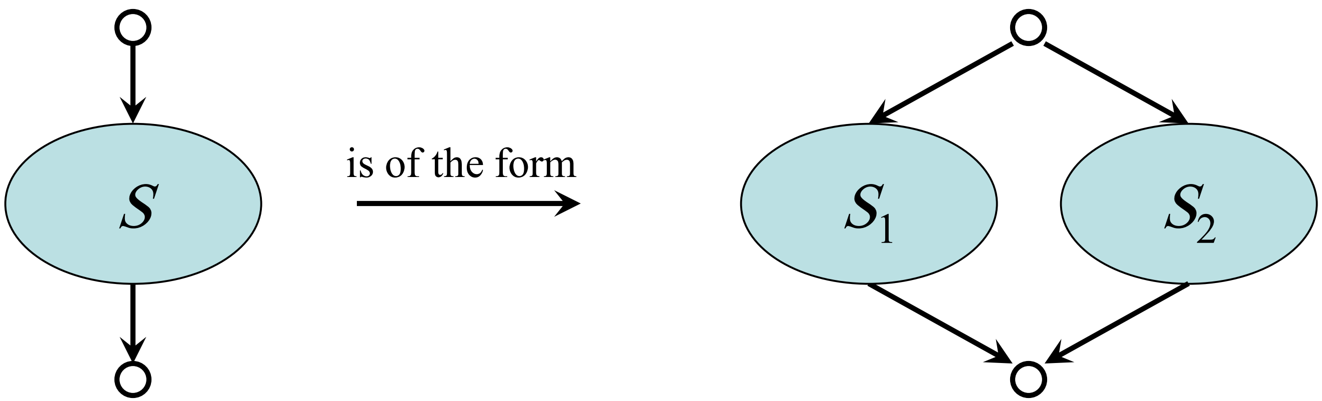
where $D_{\mathtt{a}}$ = all definitions of **`a`** in the region of code

Reaching Definitions



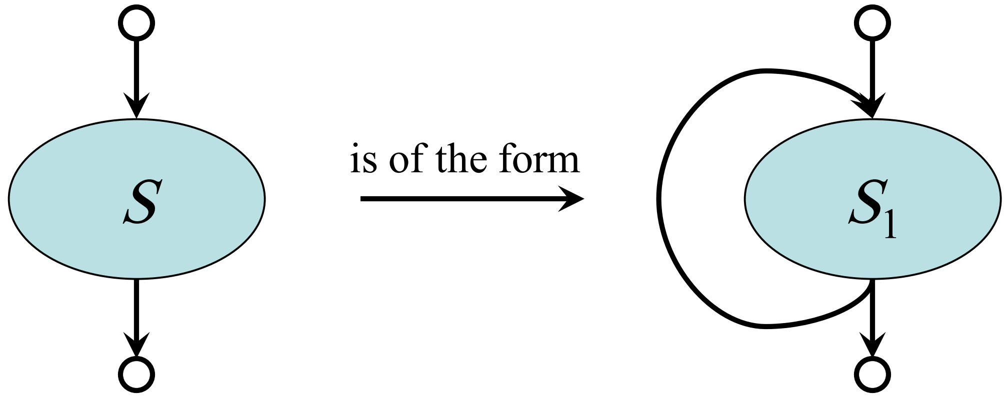
$$
\begin{aligned}
gen[S] &= gen[S_2] \cup (gen[S_1] - kill[S_2]) \\
kill[S] &= kill[S_2] \cup (kill[S_1] - gen[S_2]) \\
in[S_1] &= in[S] \\
in[S_2] &= out[S_1] \\
out[S] &= out[S_2]
\end{aligned}
$$

Reaching Definitions



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

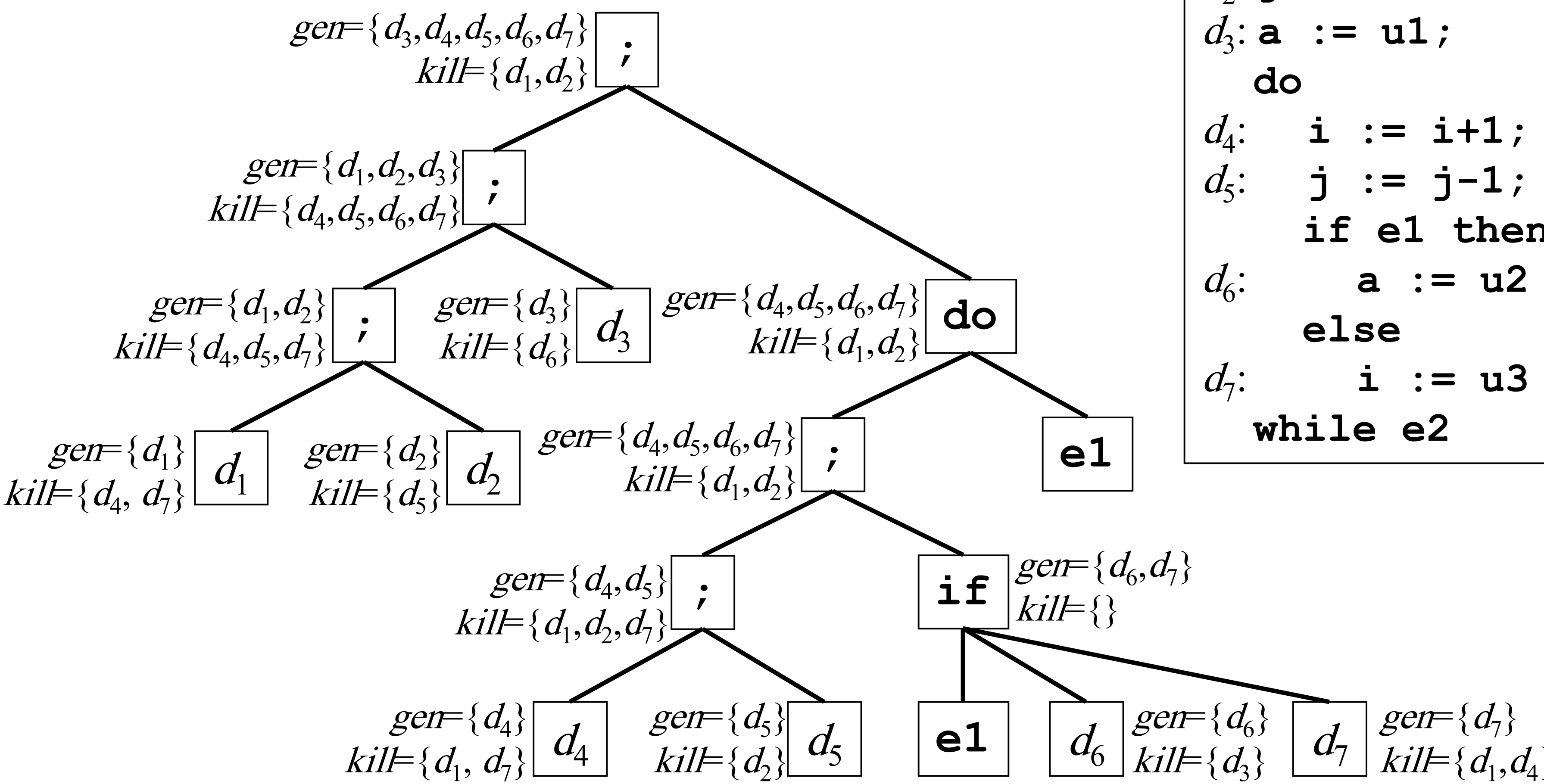
Reaching Definitions



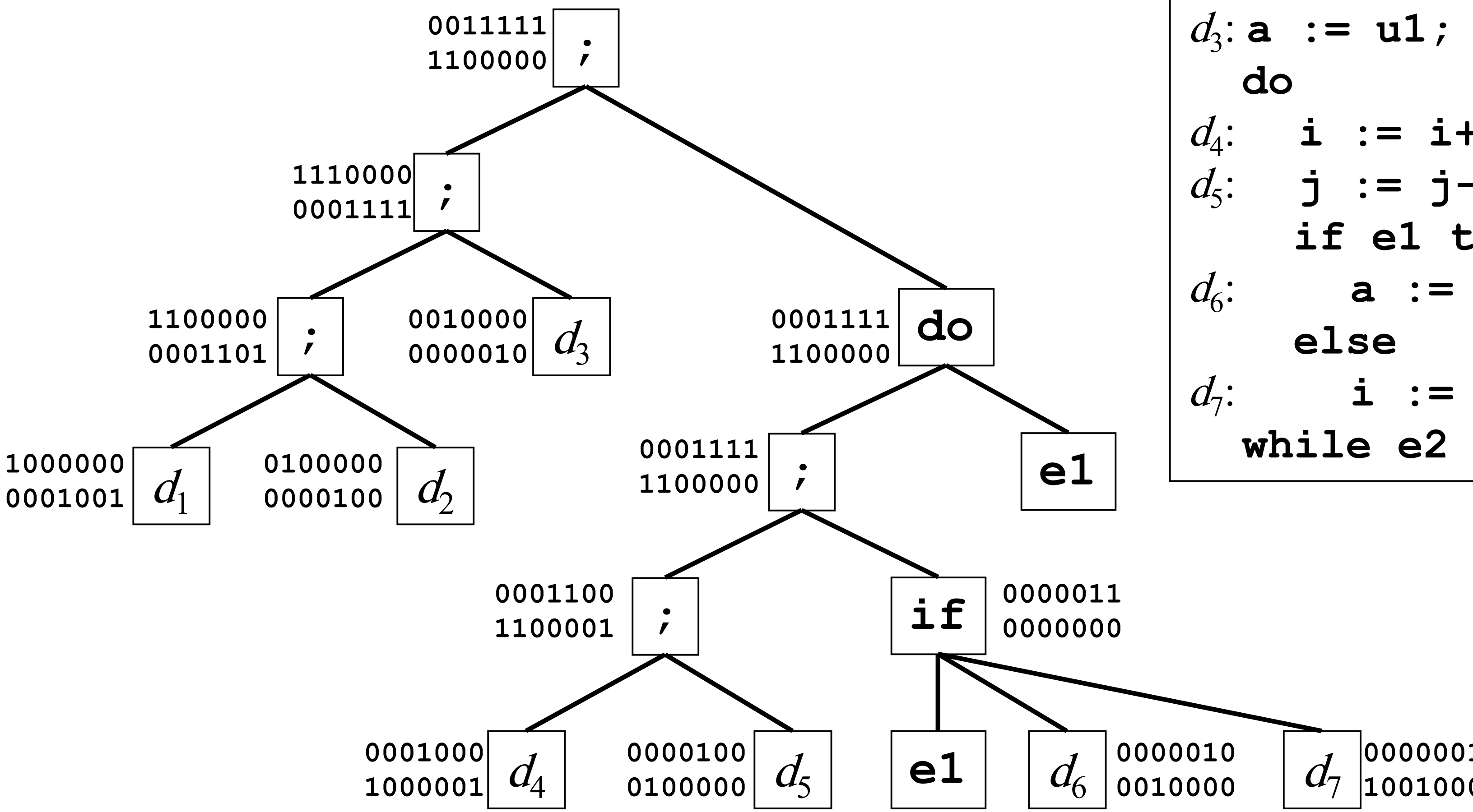
$$
\begin{aligned}
gen[S] &= gen[S_1] \\
kill[S] &= kill[S_1] \\
in[S_1] &= in[S] \cup gen[S_1] \\
out[S] &= out[S_1]
\end{aligned}
$$

Example Reaching Definitions

```
d1: i := m-1;
d2: j := n;
d3: a := u1;
  do
d4:   i := i+1;
d5:   j := j-1;
      if e1 then
d6:     a := u2
      else
d7:     i := u3
  while e2
```



Using Bit-Vectors to Compute Reaching Definitions

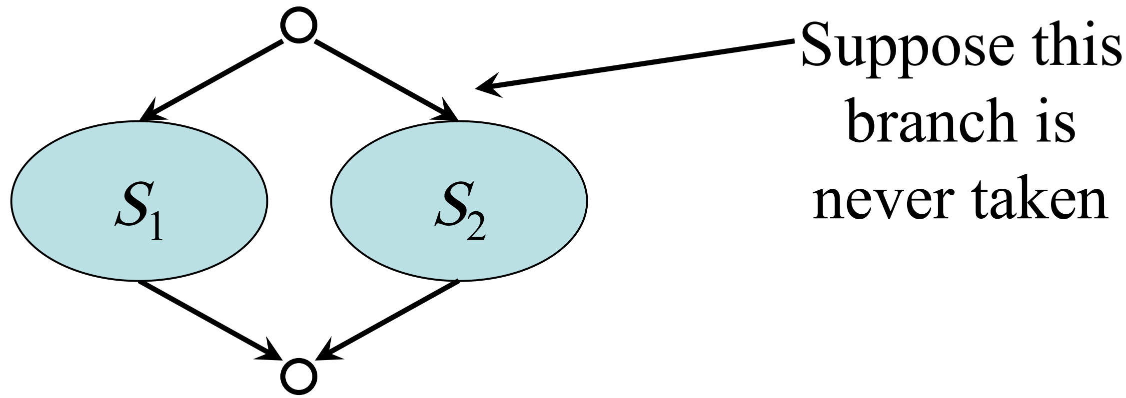


d_1: `i := m-1;`
d_2: `j := n;`
d_3: `a := u1;`
`do`
d_4: `i := i+1;`
d_5: `j := j-1;`
`if e1 then`
d_6: `a := u2`
`else`
d_7: `i := u3`
`while e2`

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 be have been killed)
- *Accuracy*: the larger the superset of reaching definitions, the less information we have to apply code optimizations

Reaching Definitions are a Conservative (Safe) Estimation



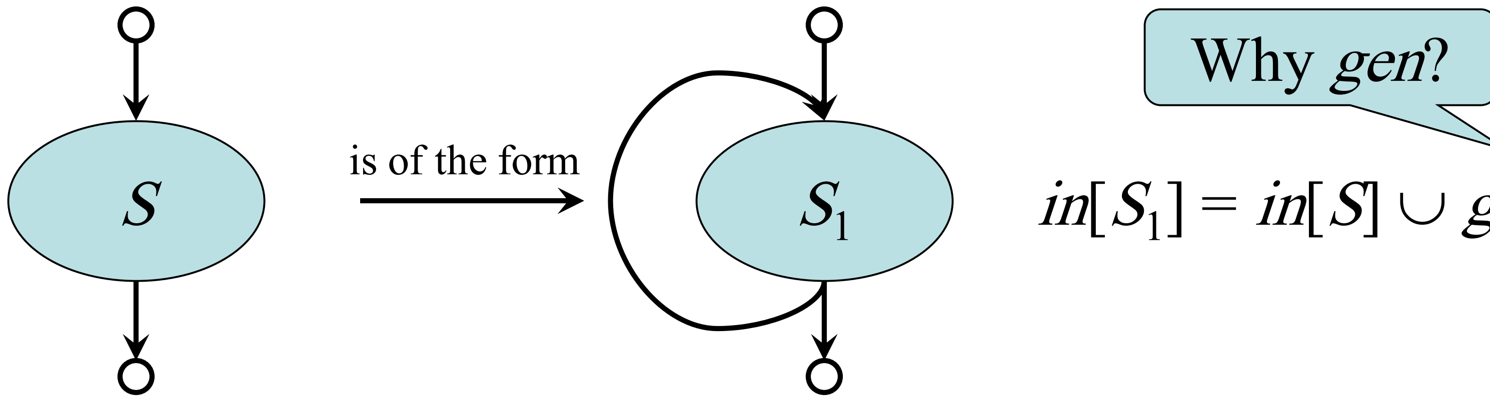
Estimation:

$$gen[S] = gen[S_1] \cup gen[S_2]$$
$$kill[S] = kill[S_1] \cap kill[S_2]$$

Accurate:

$$gen'[S] = gen[S_1] \subseteq gen[S]$$
$$kill'[S] = kill[S_1] \supseteq kill[S]$$

Reaching Definitions are a Conservative (Safe) Estimation



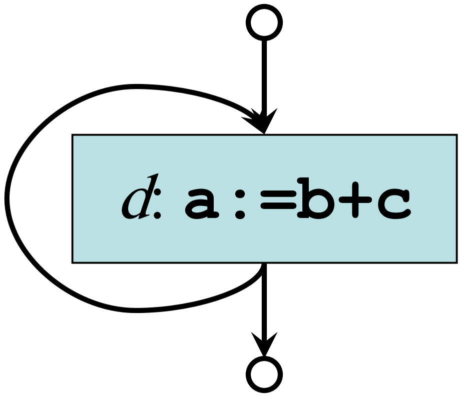
$$in[S_1] = in[S] \cup gen[S_1]$$

The problem is that

$$in[S_1] = in[S] \cup out[S_1]$$

makes more sense, but we cannot solve this directly, because $out[S_1]$ depends on $in[S_1]$

Reaching Definitions are a Conservative (Safe) Estimation



We have:

(1) $in[S_1] = in[S] \cup out[S_1]$

(2) $out[S_1] = gen[S_1] \cup (in[S_1] - kill[S_1])$

Solve $in[S_1]$ and $out[S_1]$ by estimating $in^1[S_1]$ using safe but approximate $out[S_1]=\emptyset$, then re-compute $out^1[S_1]$ using (2) to estimate $in^2[S_1]$, etc.

$in^1[S_1] \;=_{(1)}\; in[S] \cup out[S_1] = in[S]$

$out^1[S_1] \;=_{(2)}\; gen[S_1] \cup (in^1[S_1] - kill[S_1]) = gen[S_1] \cup (in[S] - kill[S_1])$

$in^2[S_1] \;=_{(1)}\; in[S] \cup out^1[S_1] = in[S] \cup gen[S_1] \cup (in[S] - kill[S_1]) = in[S] \cup gen[S_1]$

$out^2[S_1] \;=_{(2)}\; gen[S_1] \cup (in^2[S_1] - kill[S_1]) = gen[S_1] \cup (in[S] \cup gen[S_1] - kill[S_1])$

$= gen[S_1] \cup (in[S] - kill[S_1])$

Because $out^1[S_1] = out^2[S_1]$, and therefore $in^3[S_1] = in^2[S_1]$, we conclude that

$in[S_1] = in[S] \cup gen[S_1]$