263-2810: Advanced Compiler Design

6.0 Partial redundancy elimination

Thomas R. Gross

Computer Science Department
ETH Zurich, Switzerland

Outline

PRE with SSA in five (easy) steps

- Assumption: Program in SSA form for scalar variables

1. Insert Φ functions for expressions

- Introduce "temporary" to isolate expression

$$x = a_1 + b_1 \rightarrow E = a_1 + b_1$$
$$x = E$$

- Identify places where different versions merge

2. Identify/set version numbers for expressions

- Expressions on LHS: set version number
- Operands of functions: find correct version

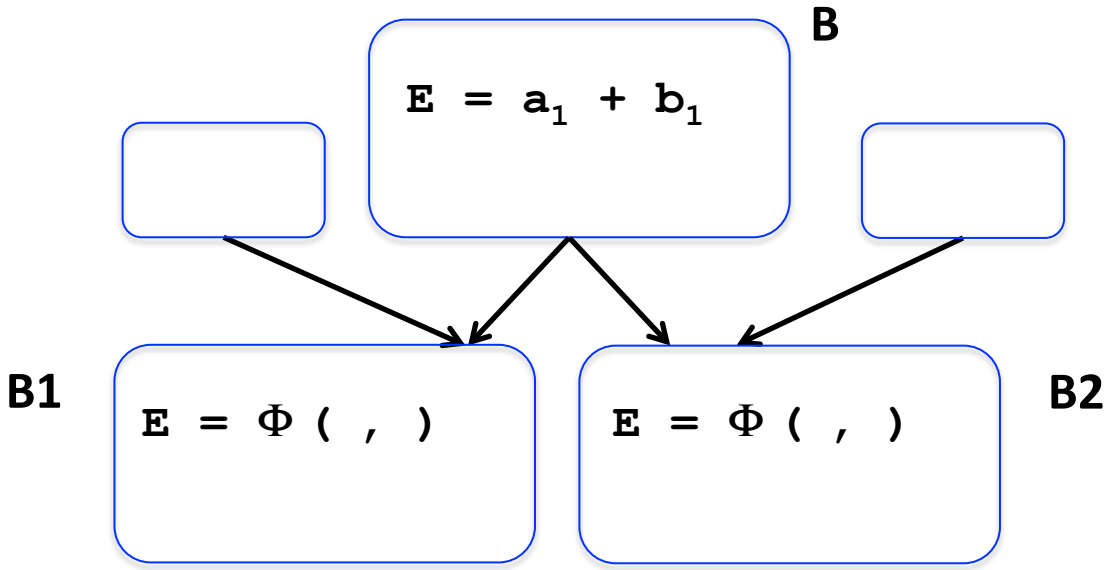
(5 steps continued)

3. **Identify places where is is *legal* to insert a copy of an expression**
   - "legal" is defined later
4. **Find places where the insertion on a copy is *profitable***
   - Only places that are legal can be considered
   - Expect removal of operations
   - Expect reduction in cycles
   - Metrics for profitability to be discussed
5. **Exploit full redundancy of expressions**
   - Transform program to reuse computed values

6.3 Insertion of Φ functions

- **Given the CFG of a program.**
  - SSA format for scalars needs dominator tree, dominance frontier.
  - Keep for insertion of Φ functions
- **Consider an expression E = `a + b` in some block B.**
- **Must insert a Φ function *somewhere* if**
  - E is computed explicitly *or*
  - One of the operands of E (i.e., `a` or `b`) is changed in block B.

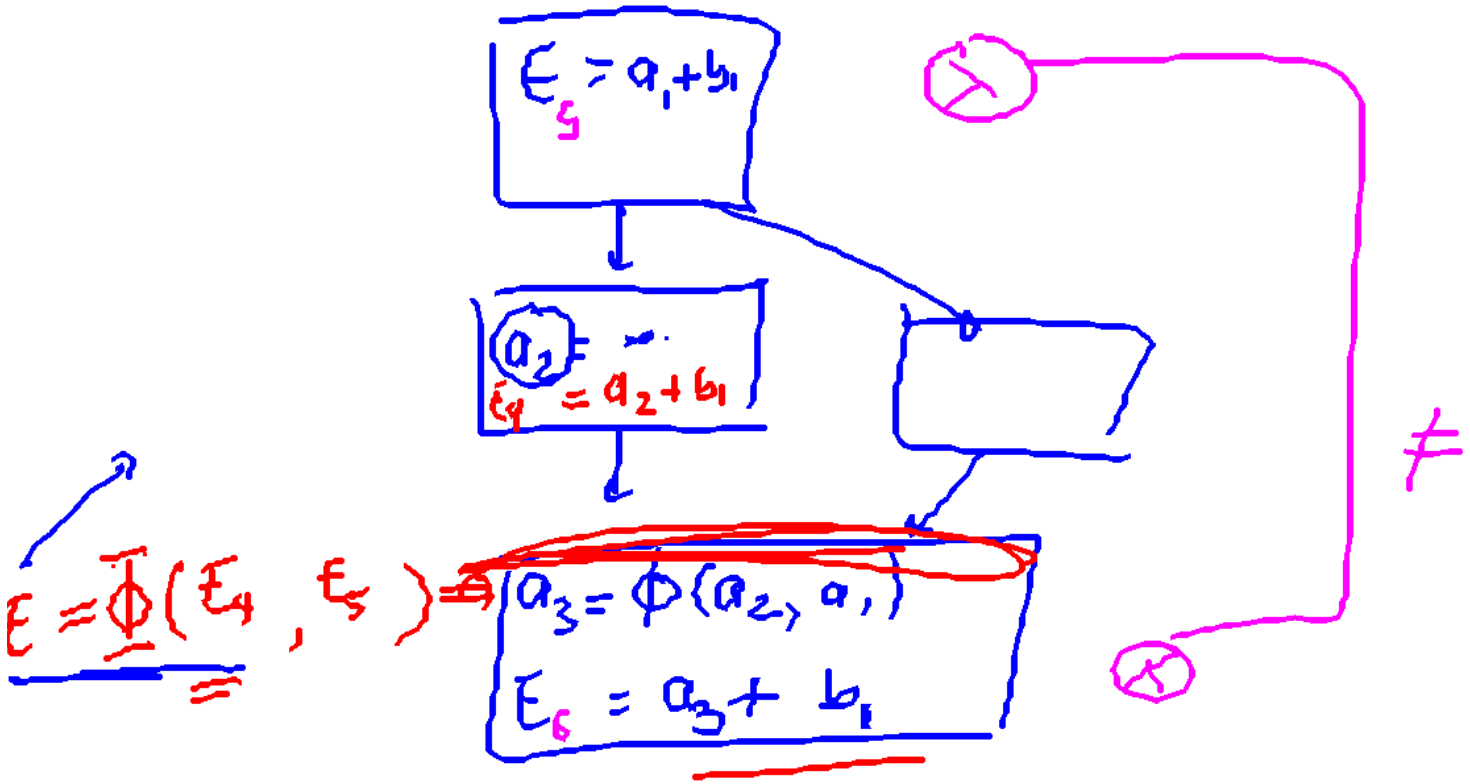
Part 1: E computed in B



- **Find dominance frontier DF(B)**
  - Here {B1, B2}
- **Insert a Φ function (unless already present)**
- **This step deals with all explicit computations of E in some block**

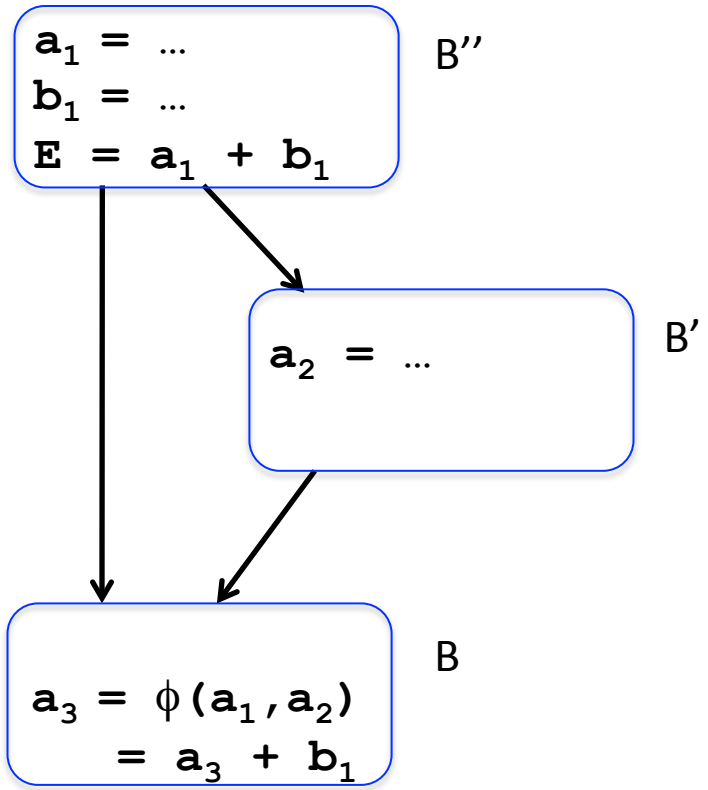
Part 2: Redefinition of operand(s)

- Consider E = a + b and a (or b) is defined in block B'



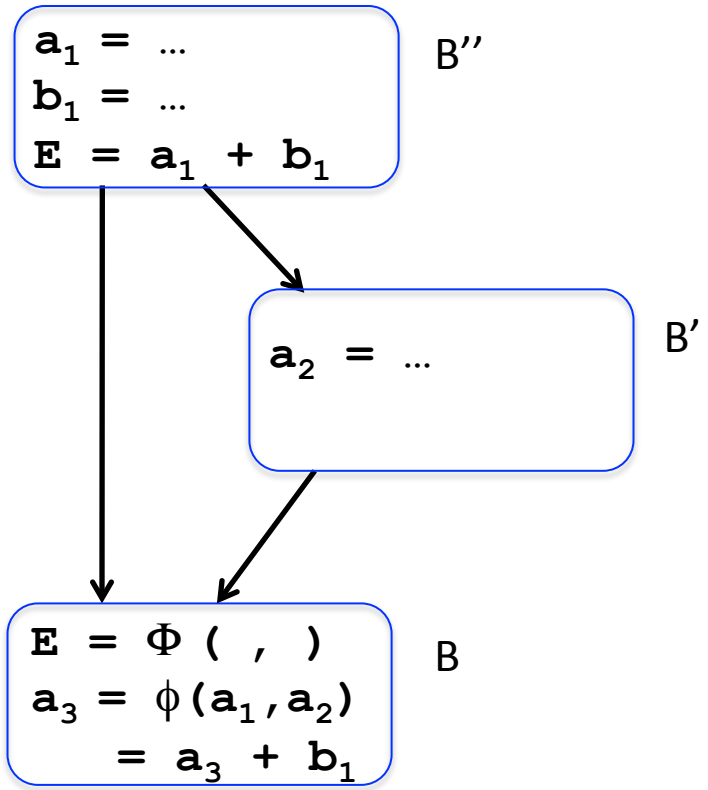
Part 2: Redefinition of operand(s)

- **Consider E = a + b and a (or b) is defined in block B'**
  - There must be a ϕ function in nodes in DF(B')
  - Consider B $\in$ DF(B')



Part 2: Redefinition of operand(s)

- Insert into B (∈DF(B')) a Φ function for all expressions E that contain a as an operand



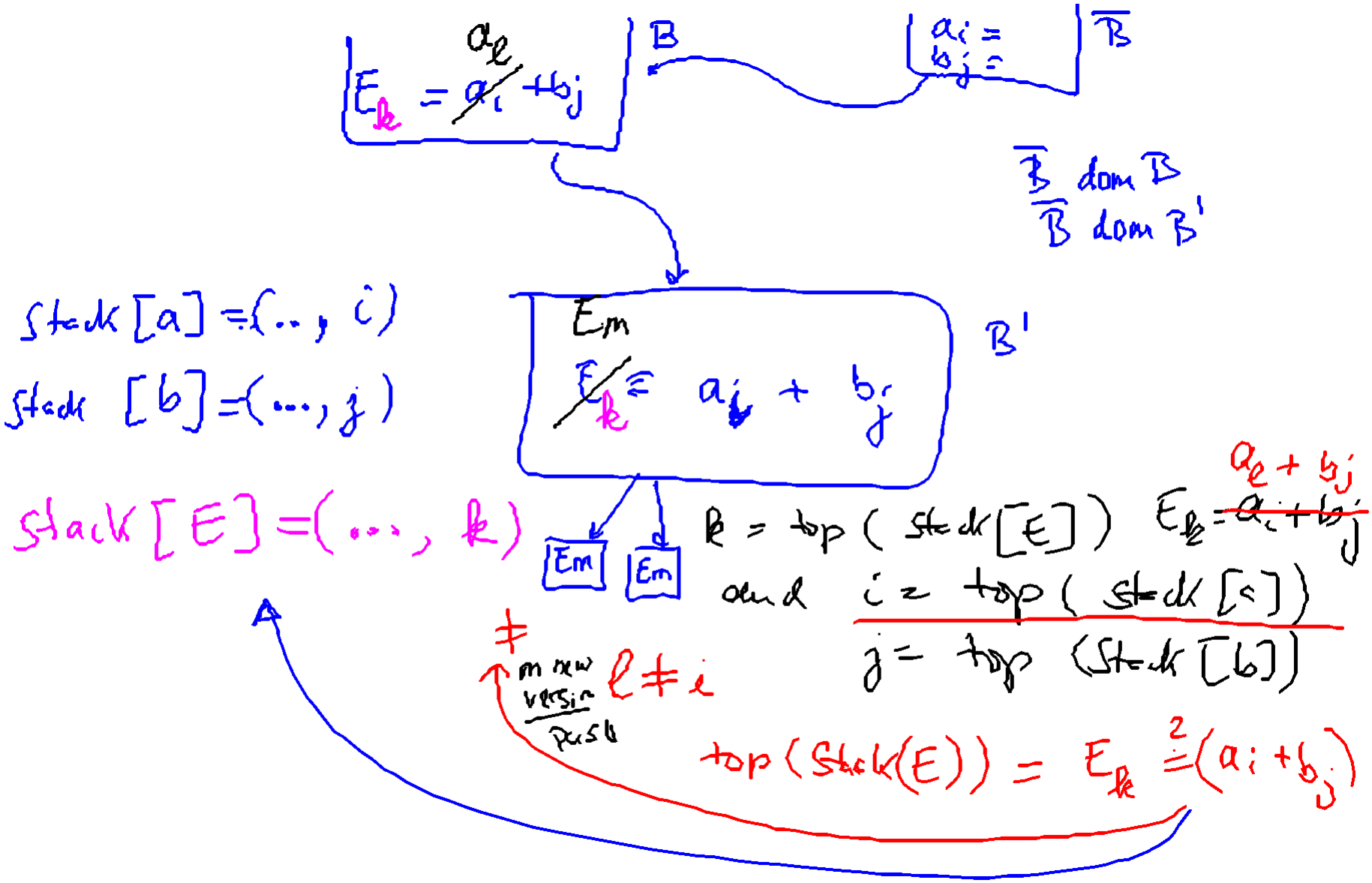
6.4 Version numbers for expressions

- **If E appears on the LHS: get a new version**
- **For the operands of a Φ function:**
  - Identify correct version

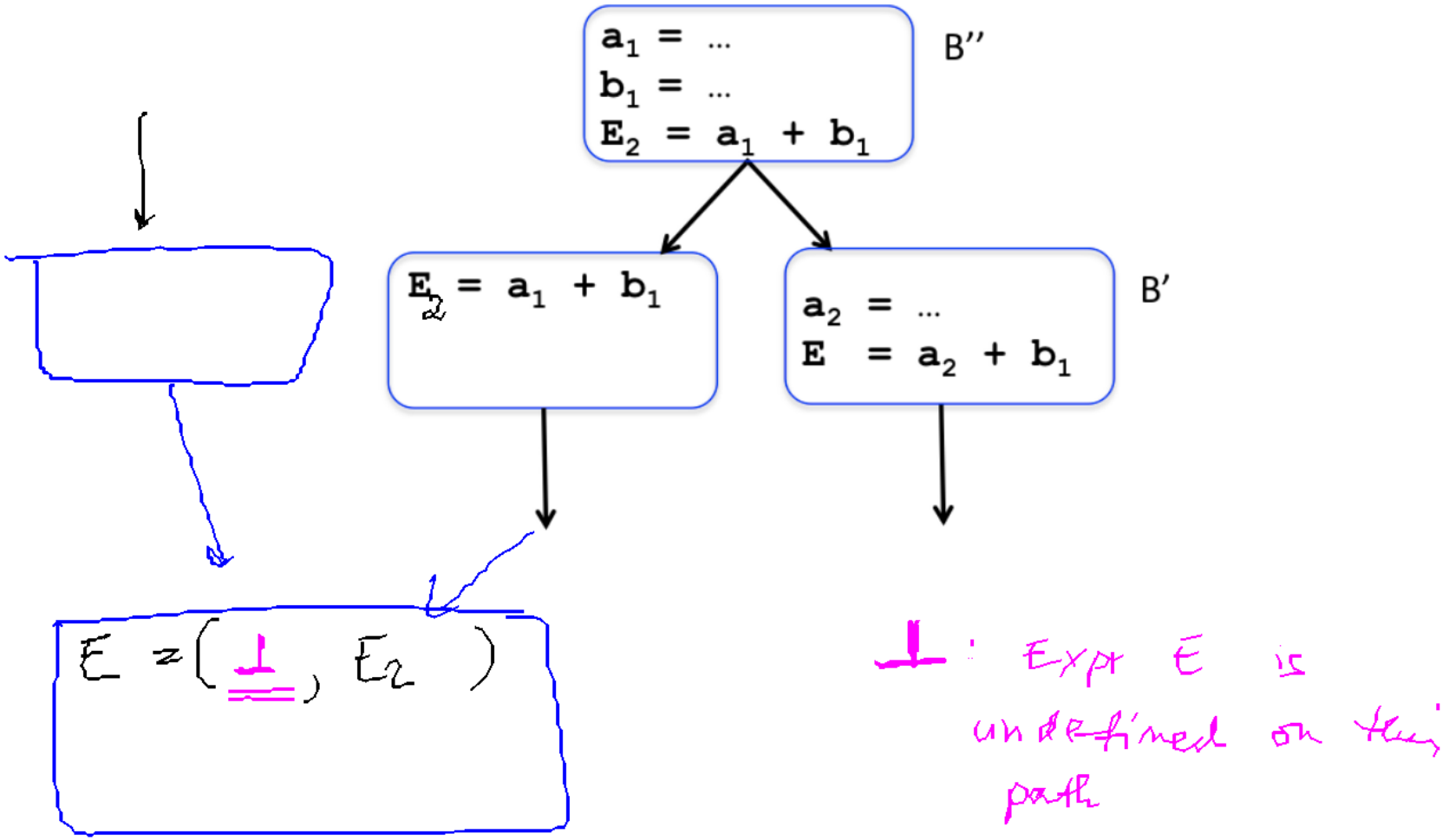
- **Recall: for functions (for scalars) we used a stack of versions**
  - Array [variable] of stacks
- **Use stack for operands of expression E to figure out which version is used**

- **Consider $\mathcal{E}$, the set of all expressions of interest in the program**
  - stack[$\mathcal{E}$]: one stack for each expression
- **E $\in$ $\mathcal{E}$ stack[E] : stack of versions (integers)**
- **Given an expression E = `a + b`, after turning program into SSA for scalars, versions of `a` and `b` are known**
  - E = a_i + b_j
- **If we have processed E = a_i + b_j before (with these versions of `a`, `b`) then we use the version *k* of E on top of stack[E]**
  - E_k = a_i + b_j
  - Must be the current version of E

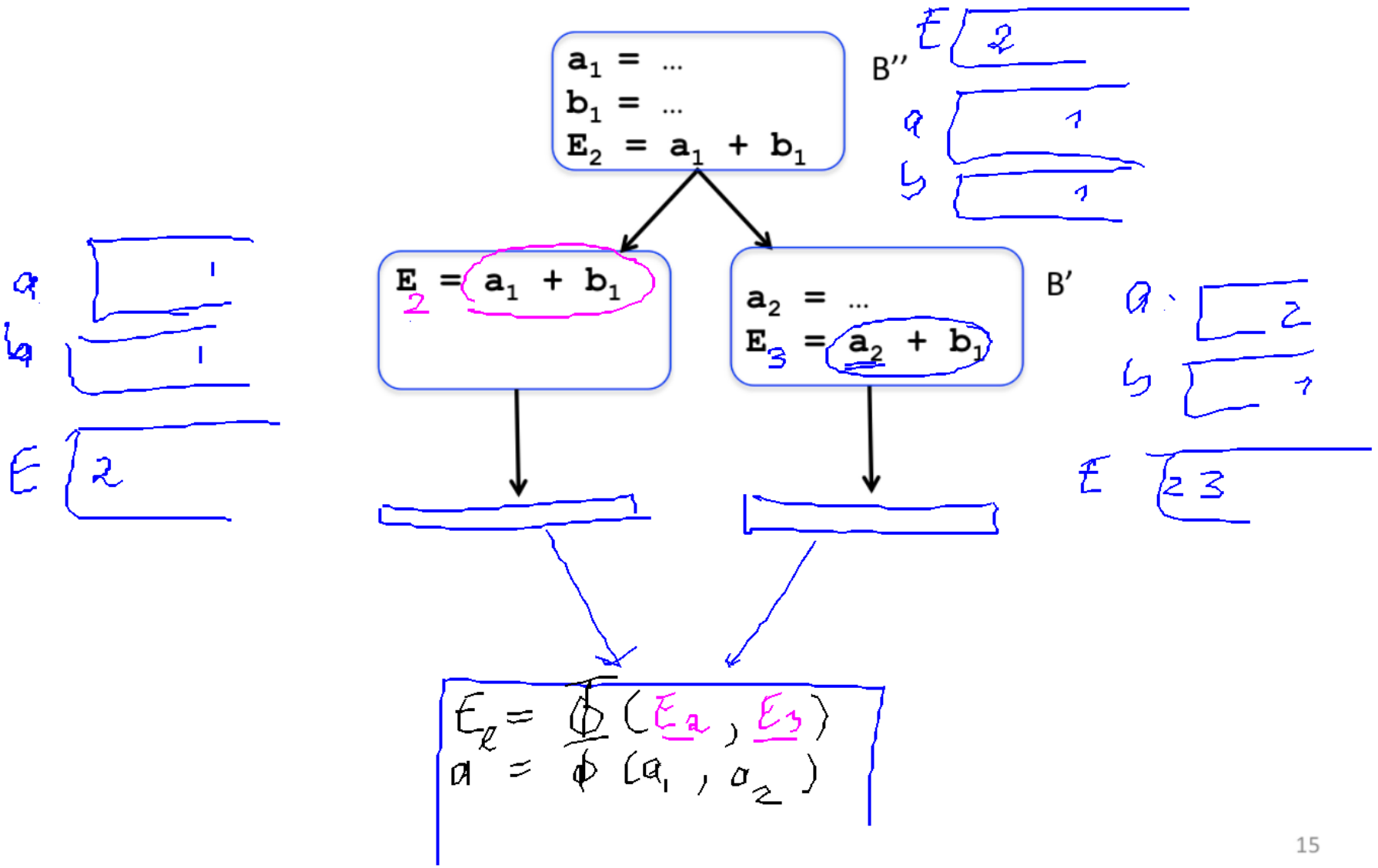
- **So use version *k* of E if**
  - stack(a).top = i
  - stack(b).top = j
  - and stack(E) = k with $E_k = a_i + b_j$
- **If $E = a_i + b_j$ has not been processed before, i.e.,**
  - stack(a).top = i
  - stack(b).top = j
  - and stack(E) = k with $E_k \neq a_i + b_j$ (i.e., operand versions changed but we did not find a new version for E) then
    - Get new version (say m)
    - stack(E).push(m)
    - Use version m for E


a_ℓ
$E_k = a_i + b_j$
B
$a_i =$
$b_j =$
$\overline{B}$
$\overline{B}$ dom B
$\overline{B}$ dom B'
stack [a] = (.., i)
stack [b] = (..., j)
E_m
$E_k = a_i + b_j$
B'
stack [E] = (..., k)
E_m
E_m
$k = top(stack[E])$
and
$i = top(stack[a])$
$j = top(stack[b])$
$E_k = a_i + b_j$
$a_\ell + b_j$
$\neq$
in new version push
$\ell \neq i$
$top(Stack(E)) = E_k \overset{?}{=} (a_i + b_j)$

Example


a_1 = ...
b_1 = ...
E_2 = a_1 + b_1
B''
E_2 = a_1 + b_1
a_2 = ...
E = a_2 + b_1
B'
E = (⊥ , E_2)
⊥ : Expr E is undefined on this path

Example


$a_1 = \ldots$
$b_1 = \ldots$
$E_2 = a_1 + b_1$
B''
$E_2 = a_1 + b_1$
$a_2 = \ldots$
$E_3 = a_2 + b_1$
B'
$E_\ell = \phi(E_2, E_3)$
$a = \phi(a_1, a_2)$

6.4 Version numbers for expressions

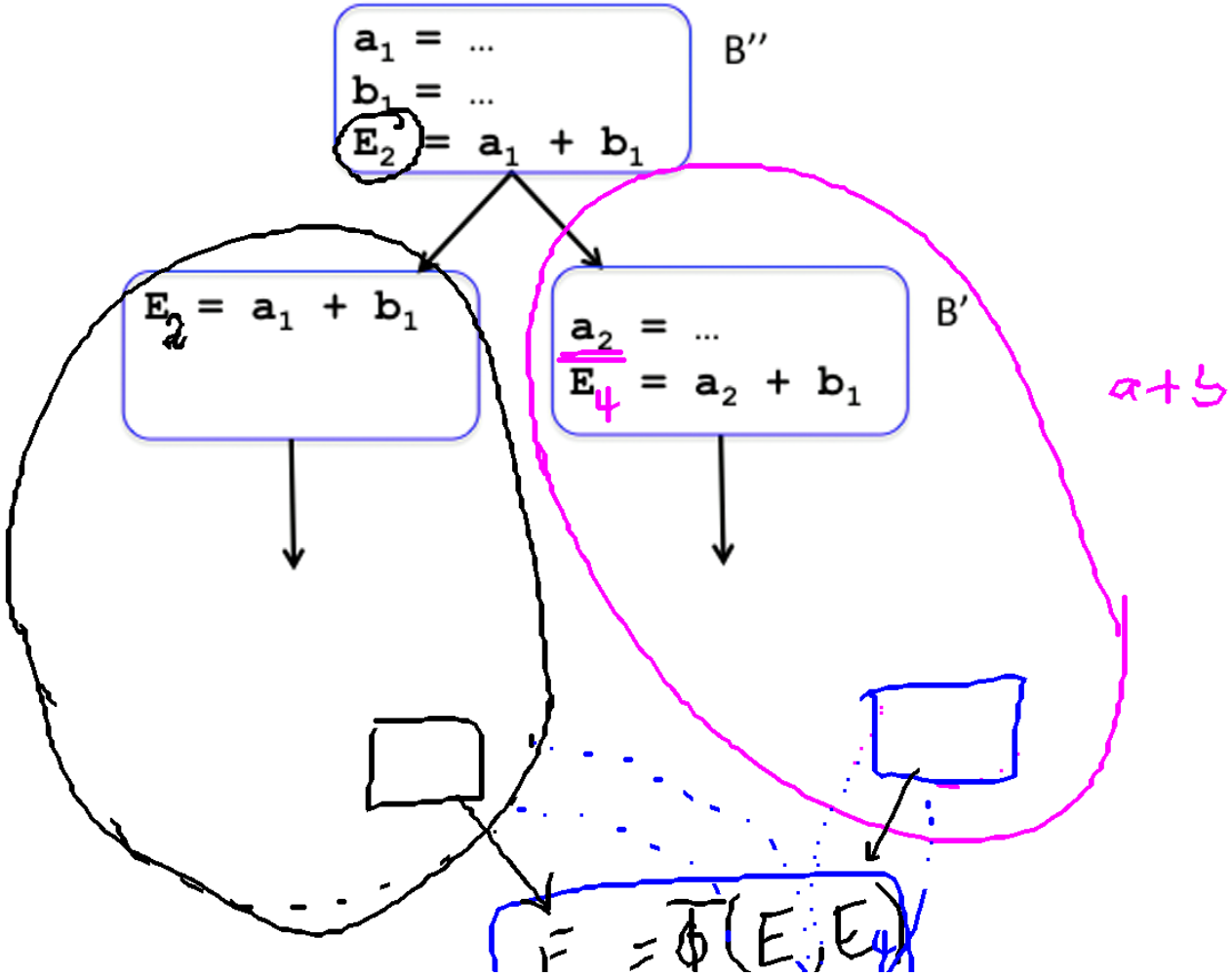
- **If E appears on the LHS: decide if a new version is needed or if the current version should be used**
  - "current": Version computed in a dominating node and no operand has been redefined
- **For the operands of a Φ function:**
  - Identify correct version

- **Recall: for functions (for scalars) we used a stack of versions**
  - Array [$\mathcal{V}$] of stacks, with $\mathcal{V}$ the set of all variables
- **For $\mathcal{E}$, the set of all expressions of interest in the program**
  - stack[$\mathcal{E}$]: one stack for each expression
  - E $\in \mathcal{E}$: stack[E] is a stack of versions (integers)
- **Given an expression E = `a + b`, after turning program into SSA for scalars, versions of `a` and `b` are known**
  - E = a_i + b_j
- **If we have processed E = a_i + b_j before (with these versions of `a`, `b`) then we use the version *k* of E on top of stack[E]**
  - E_k = a_i + b_j
  - a_i, b_j must be current versions of a, b
  - Must be the current version of E

- **So use version *k* of E if**
  - stack(a).top = i
  - stack(b).top = j
  - and stack(E) = k with $E_k = a_i + b_j$

- **If $E = a_i + b_j$ has not been processed before, i.e.,**
  - stack(a).top = i
  - stack(b).top = j
  - and stack(E) = k with $E_k \neq a_i + b_j$ (i.e., operand versions changed but we do not find a new version for E) then
    - Get new version (say m)
    - stack(E).push(m)
    - Use version m of E: E_m

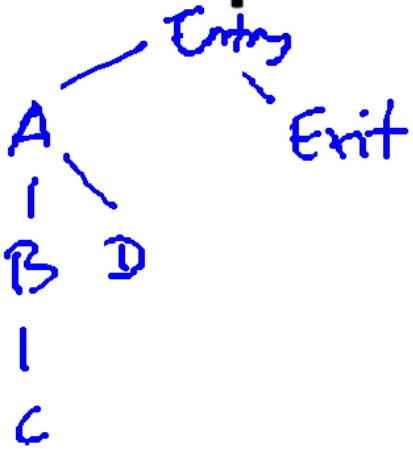
Example

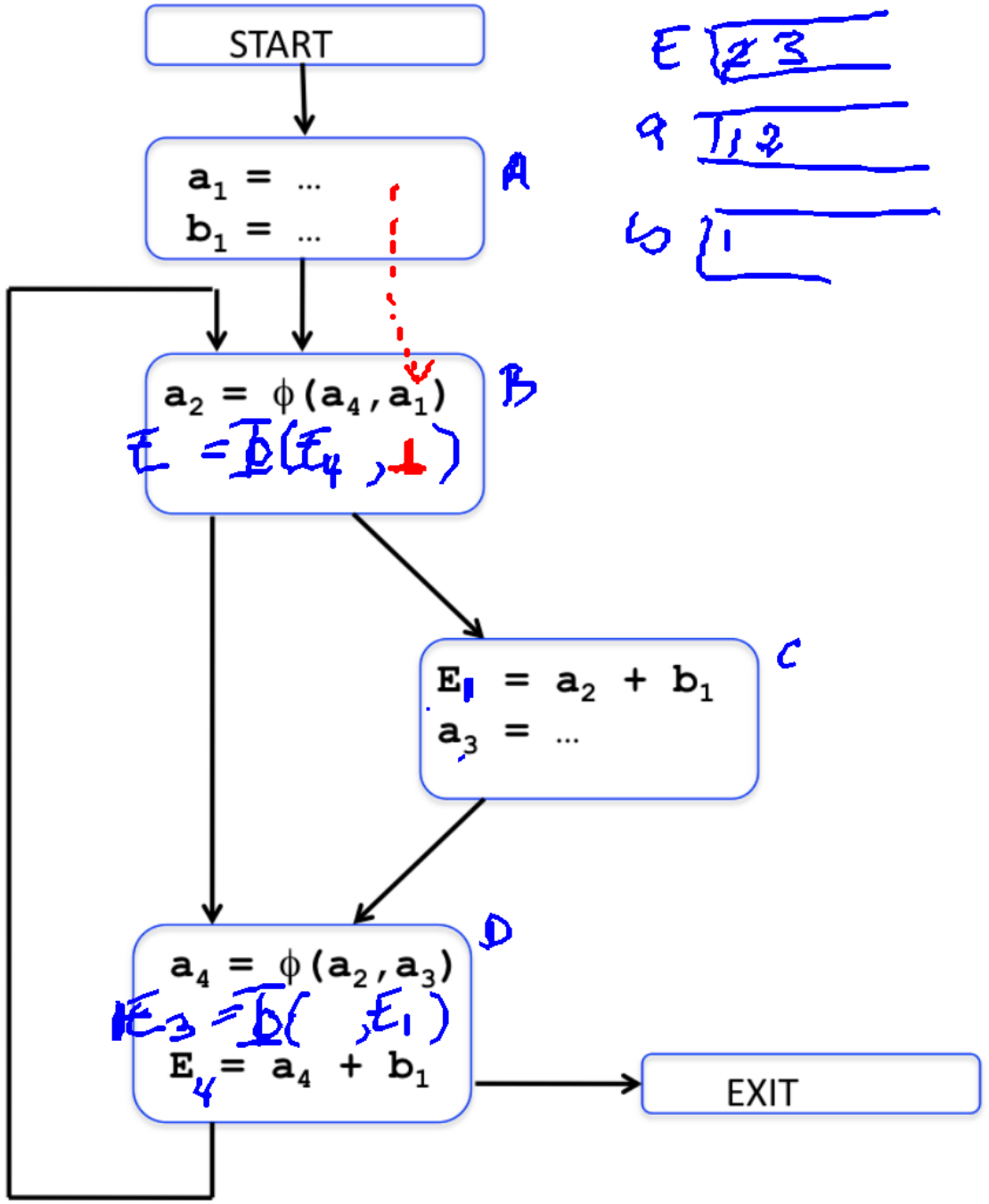


Operands of Φ function

- **Visit nodes in depth-first order**
  - Use dominator tree
  - Like when handling ϕ functions
- **After processing a block (node), record expression version for Φ function(s) in successor blocks that are not dominated**
  - Make sure argument to Φ function reflects expression set along corresponding path

Example


Entry
Exit
A
B
D
C


START
$a_1 = \ldots$
$b_1 = \ldots$
A
$a_2 = \phi(a_4, a_1)$
B
$E_2 = \Sigma(E_4, 1)$
$E_1 = a_2 + b_1$
$a_3 = \ldots$
C
$a_4 = \phi(a_2, a_3)$
D
$E_3 = \Sigma(\ , E_1)$
$E_4 = a_4 + b_1$
EXIT

6.5 Are copies legal at point P?

- **Given a program in SSA format with ϕ and Φ functions**
- **Versions of scalars and expressions have been determined**

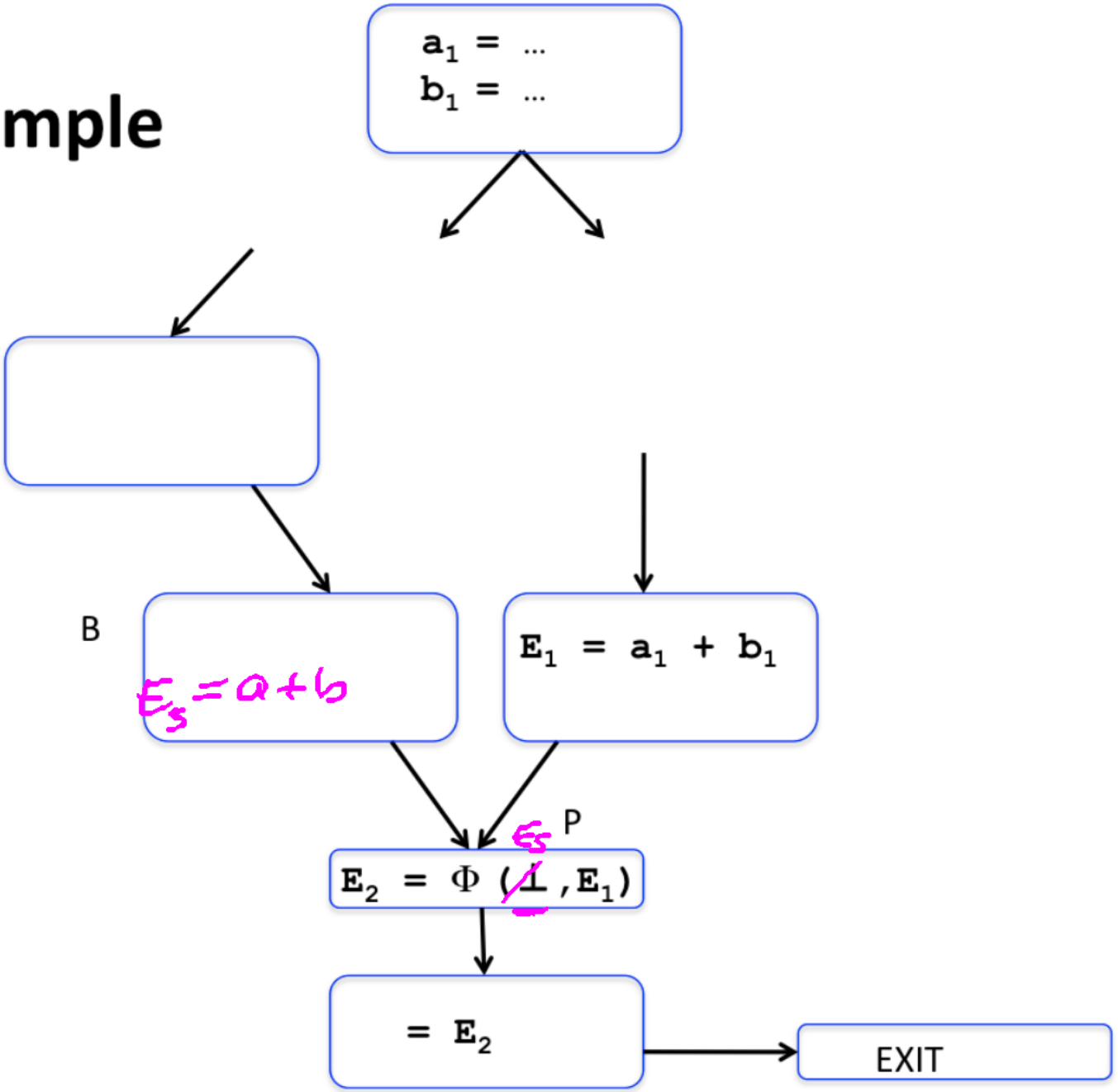
- **Given a basic block, P point at the start of block.**

  A Φ function for E at point P with (one or more) $\perp$ operands indicates that

  - Value of expression E is undefined if control reaches P along path that corresponds to $\perp$ operand
  - Expression E is defined if control reaches P along a path that corresponds to version E_k of E

- **Predecessor basic block that corresponds to $\perp$ operand is a candidate to place a copy of E**

Example



- **A copy of E can be inserted into B**
  - Or any of its predecessors
  - (As long as operands of E are available)

- **Is inserting a copy acceptable?**

- **A copy of E can be inserted into B**
  - Or any of its predecessors
  - (As long as operands of E are available)
- **Is inserting a copy (into B) legal?**
- **“Gold standard”: insertion of a copy must not change the program (results)**

Detour: “must not change the program”

- **Given program P, transformed program T(P) = P’**
- **Transformation is legal if**
  - **P** computes the same result as **P’**
    - Same output
  - Returns no new errors
  - Throws no new exceptions
  - Termination behavior unchanged
- **Allow that P encounters an error earlier or later**
- **Allow that P throws an exception earlier or later**
- **Note: values stored in memory by P may differ from values stored by P‘**
  - May accept that values after error/exception differ

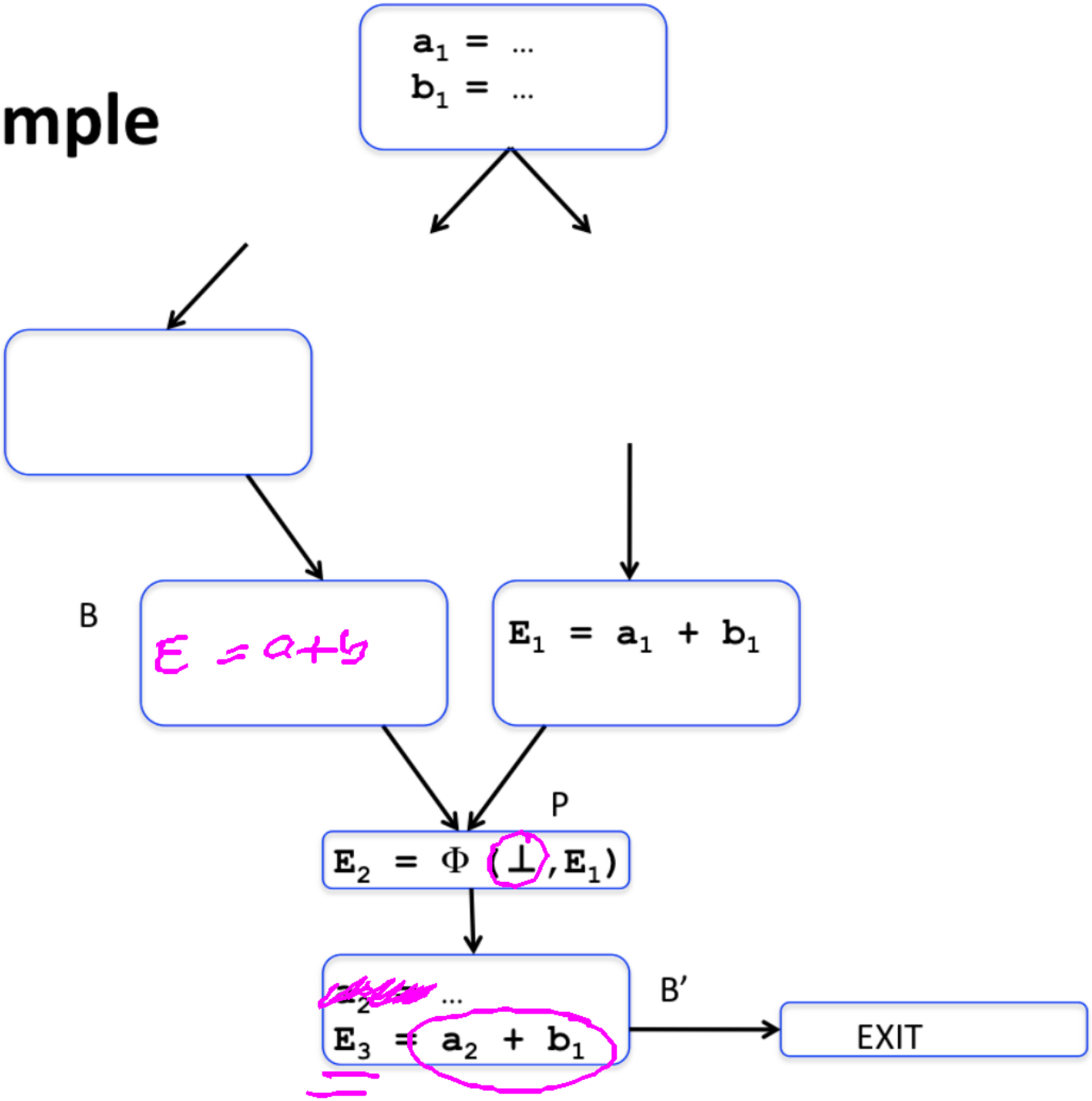
Legal copies

- **Could we insert a copy of E into B?**

- **We do not know anything about the effect of E.**
  - Might throw an exception
  - Might raise an error (overflow, memory protection error, …)

- **A copy of E can be inserted into B if E is evaluated along all paths from B to EXIT**

Example


$a_1 = \ldots$
$b_1 = \ldots$
B
$E = a+b$
$E_1 = a_1 + b_1$
P
$E_2 = \Phi(\perp, E_1)$
$\ldots$
$E_3 = a_2 + b_1$
B'
EXIT

- **Our model of legality: insert into B only if there is a copy of E on all paths from B to EXIT**
  - Blocks with this property are called “downsafe”
  - Earlier papers use the phrase “E anticipated in B”

- **Insertion legal: iff B is downsafe**

Downsafety

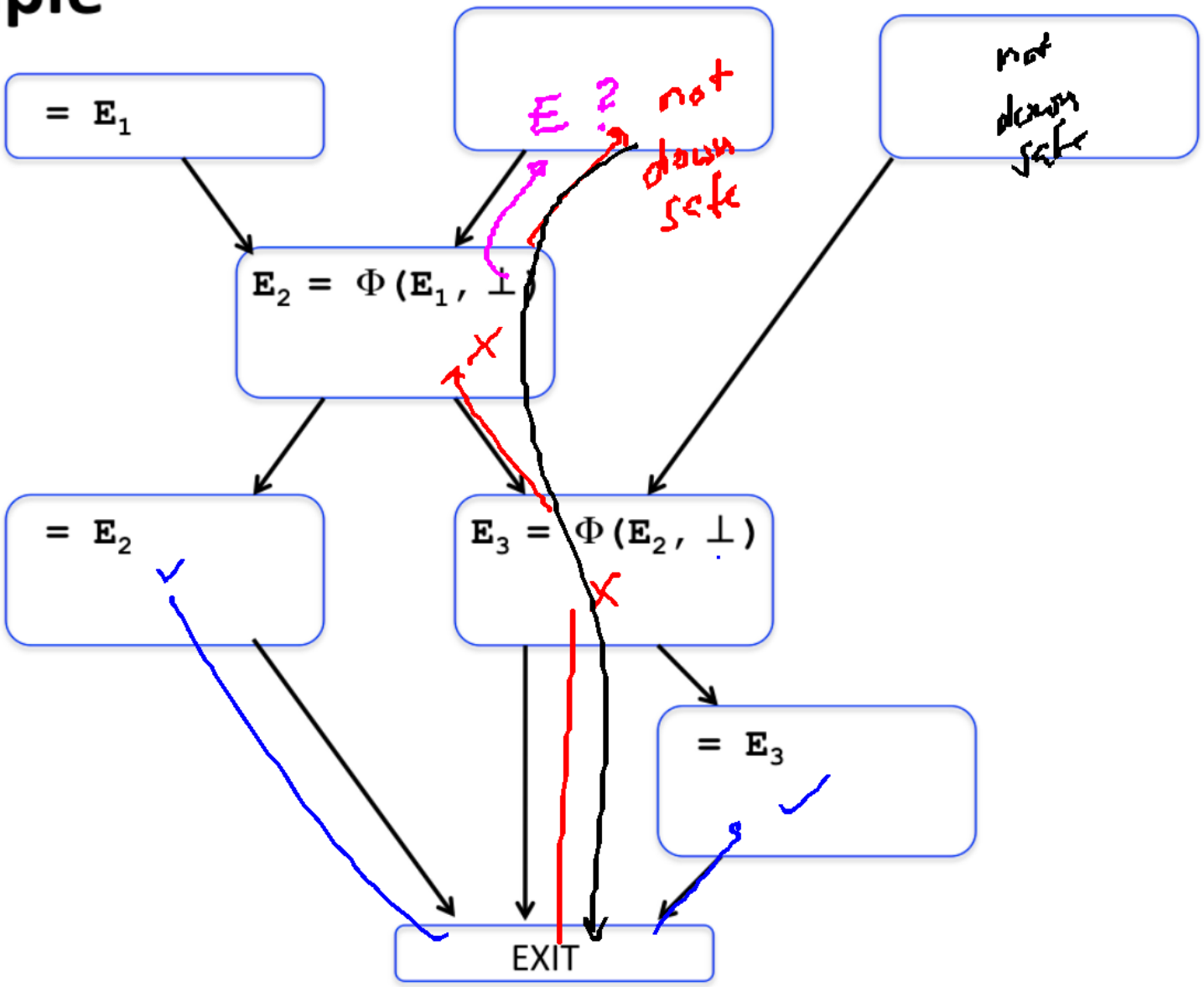
- **Check if there is a path from B to EXIT that does *not* include E**
- **If E occurs only as the operand to a Φ function in block B'. Check that E is used in B' or B' is downsafe.**

- **We say a Φ function in block B' is downsafe if E appears on the RHS of a statement in B' or B' is downsafe.**
  - Can insert copies for Φ functions that are downsafe

Downsafety

- **Need to check that along all paths from B to EXIT there is a *real* occurrence of E or a downsafe Φ function**
- **Simple algorithm:**
  - Start at EXIT
  - Visit recursively all predecessor nodes, until all nodes have been visited
  - Mark a Φ function as downsafe iff a real occurrence or a downsafe Φ function appears on all paths to EXIT
  - After visiting all nodes: downsafe Φ functions are marked
- **Downsafety is a necessary condition**

Example


= E_1
$E_2 = \Phi(E_1, \bot)$
= E_2
$E_3 = \Phi(E_2, \bot)$
= E_3
EXIT
E ?
not
down
safe
not
down
safe