Compilers and Code Optimization

EDOARDO FUSELLA

Intermediate Representations

Contents

Motivation

- The Front/Back Split
  - Front-end: lexical and syntactic analysis, semantic checks
  - Back-end: code generation, register allocation, target-dependent optimization
- Advantages of Front/Back Split
  - Portability: one front-end for each source language, one back-end for each machine ($N + M$ instead of $N \times M$)
  - Modularization and separation of concerns:
    - Design back-ends without taking into account each source language
    - Conversely, for front ends with respect to machine properties

Varieties of IRs

- Structure
  - Intermediate Language: functional representation of the source
  - Metadata: information from the source code useful for optimization
- Types of IRs
  - IR Trees: expression trees
  - Three-address instructions: pseudo-assembly languages (register based)
  - VM bytecode: Java Bytecode, DotNet CIL (usually stack based)

Desirable qualities

- Convenient to be produced during semantic analysis
- Convenient to translate into machine language for a broad range of architectures
- Each construct must have a clear and simple meaning, so that optimizing transformations can be easily specified and implemented
- For interpreted IRs, other requirements may arise

IR Trees

An example from A. Appel

- Design philosophy
  - IR constructs are nodes of a tree
  - Each IR construct must describe very simple operations such as a single fetch, store, add or jump
  - Child nodes contain the operands of the parent node
- IR Node classes
  - Expressions: perform computation of some value
  - Statements: perform side effects and control flow

IR Trees

Expressions

node type	meaning	note
CONST(*i*)	integer constant i	
NAME(*n*)	symbolic constant n	corresponds to assembly label
TEMP(*t*)	a temporary t	the temporary will be later mapped on a register
BINOP(*o,e1,e2*)	binary operator *o* applied to operands obtained by evaluating expr *e*1,*e*2 in order	ops: PLUS, MINUS, MUL, DIV; bitwise logical ops AND, OR, NOT; shift operators . . .
MEM(*e*)	the contents of wordSize bytes, starting at addr *e*	as left operand of MOVE, denotes a store, otherwise a fetch
CALL(*f,l*)	function name *f*, argument list *l*	
ESEQ(*s,e*)	Enforce an order: *stmt* is evaluated, then *exp* is evaluated.	The value of the ESEQ node are that of exp.

IR Trees

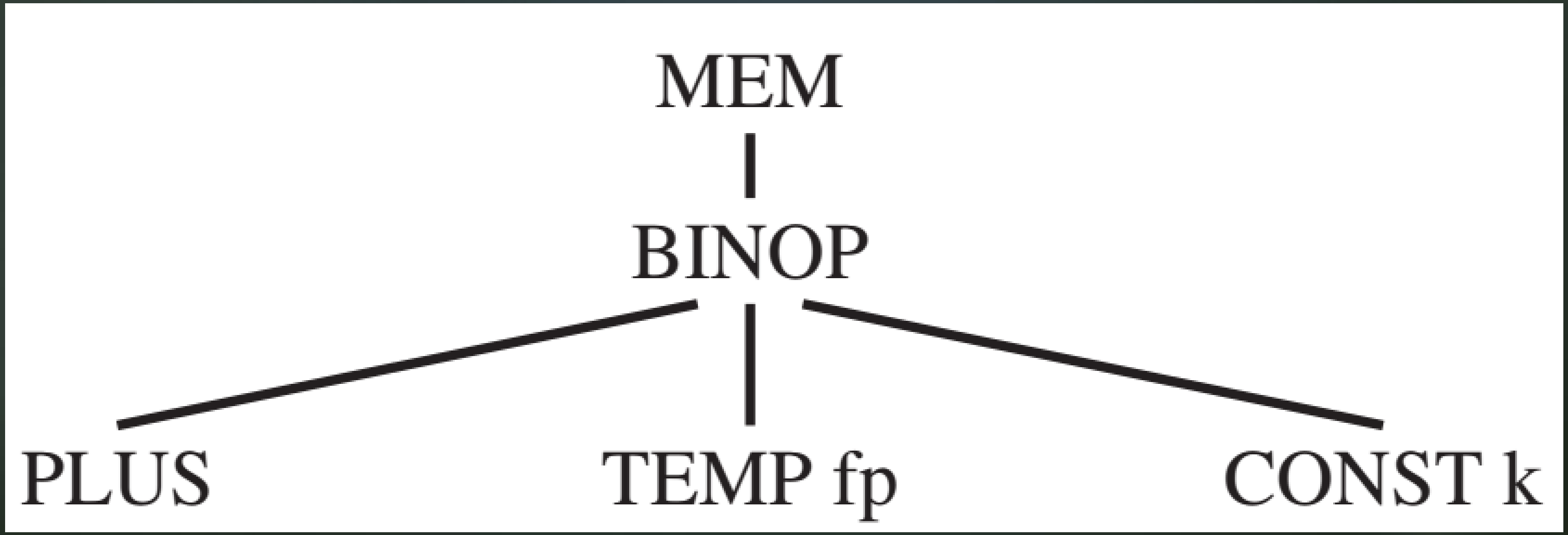
Statements

node type	meaning	note
MOVE(*TEMP t,e*)	evaluate expression e and move the result into temporary t	
MOVE(*MEM(e1), e2*)	eval expr *e*1 yielding addr *a*; eval *e*2 and store result in *k* bytes of memory, starting at *a*	
JMP(*l*)	jump to label l	
CJUMP(*o,e1,e2,t,f*)	eval expr *e*1 then *e*2, yielding values *a*,*b*; compare *a* and *b* with relational operator *o*; If true jump to *t* else to *f*	the relational operators are: EQ, NE, LT, GT, LE, GE, ULT, ULE, UGT, UGE.
SEQ(*s1, s2*)	The statement *s*1 followed by *s*2	
LABEL(*n*)	defines symbolic constant *n* as the current code address	the value NAME(*n*) may be target of jumps/calls.

IR Trees

Translation of simple variables

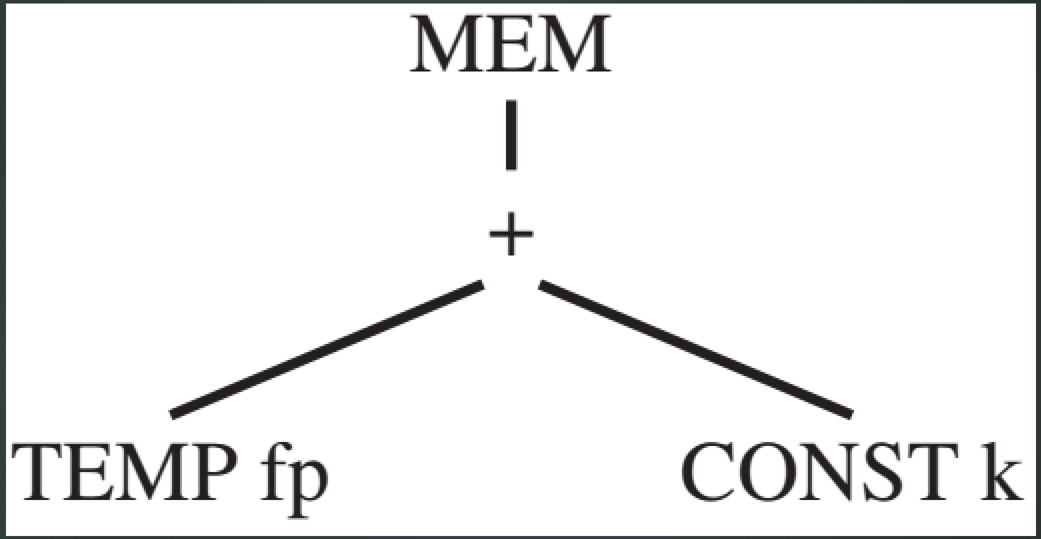
- For a simple variable declared in the current procedure’s stack frame, we translate it as **MEM(BINOP(PLUS, TEMP fp, CONST *k*))**
- The content of the memory cell at the address computed by the expression fp + k, where fp is the frame pointer (a register) and k is the constant offset of the variable within the frame



IR Trees

Translation of simple variables

- The representation can be shortened to: **MEM(+(TEMP fp, CONST k))**



IR Trees

Array access

- Array in C:

  int a [12];

  a[3] = …

- Array in Pascal:

  var a : array[1..12] of integer;

  a[4] = …

- Twelve 4-bytes cells are assigned to a in the frame. Base address a[0] (C) or a[1] (Pascal).

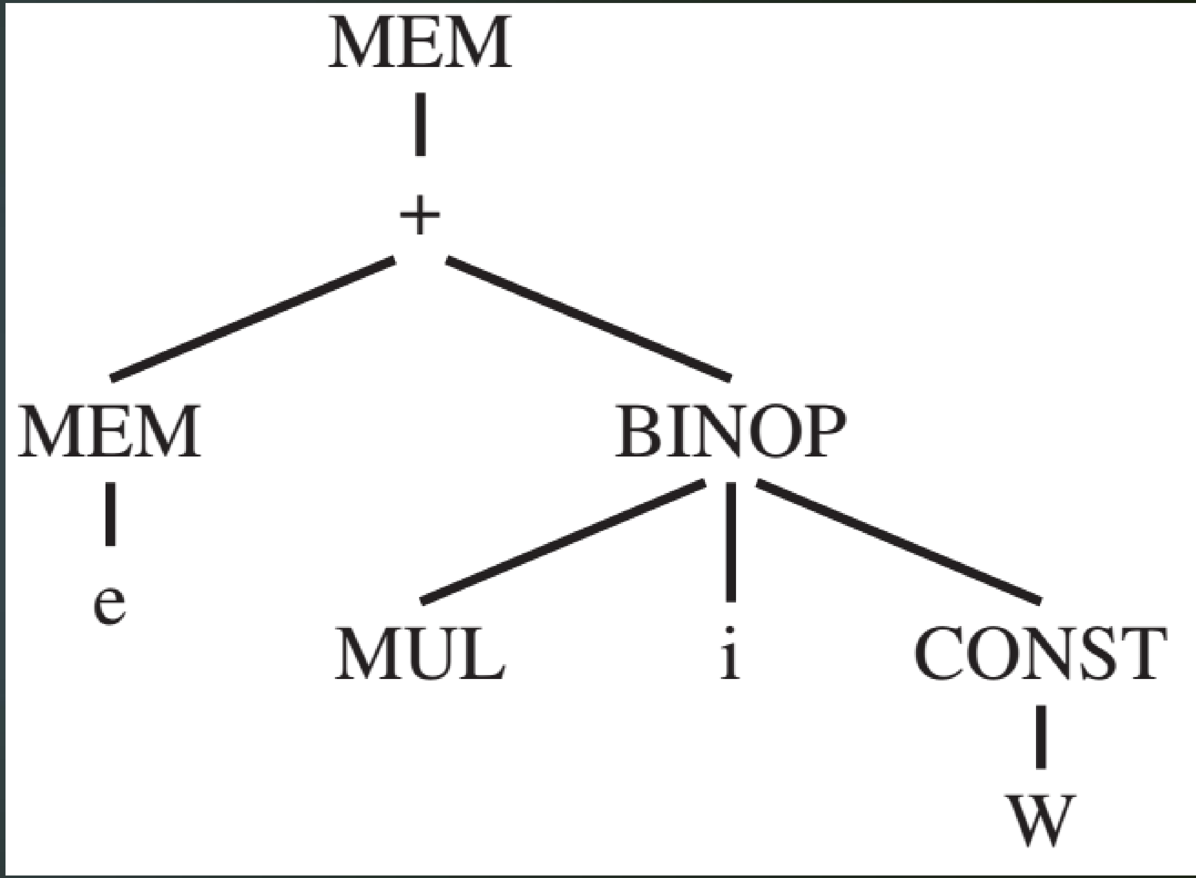
- To subscript an array in C or Pascal (to compute a[i]), just calculate the address of the *i*-th element of a: $(i - l) \times s + a$
  - *l* is the lower bound of the index range
  - s is the size (in bytes) of each array element
  - a is the base address of the array elements

IR Trees

Translation of array variables

- In C, to calculate the array reference *a[i]*
  - the lower bound is zero: *l = 0*
  - Assuming all elements are one word long: *s = w*
  - The base address of the array is the contents of a pointer variable, so MEM is required to fetch this base address
  - Thus:

MEM(+(MEM(e), BINOP(MUL, i, CONST W)))



IR Trees

Arithmetic

- Easy to translate
  - Each arithmetic operator corresponds to a binary operator
  - No unary arithmetic operators
    - Unary negation of integers can be implemented as subtraction from zero
    - unary complement can be implemented as XOR with all ones.
  - Unary floating-point operators are hard to translate
    - Unary floating-point negation requires a new operator

IR Trees

Conditional Instructions

- Easy to translate with the CJUMP operator

x<5

will be translated as

s1(t, f **) = CJUMP(**LT, x, CONST(5), t, f **)**

for any labels *t* and *f* .

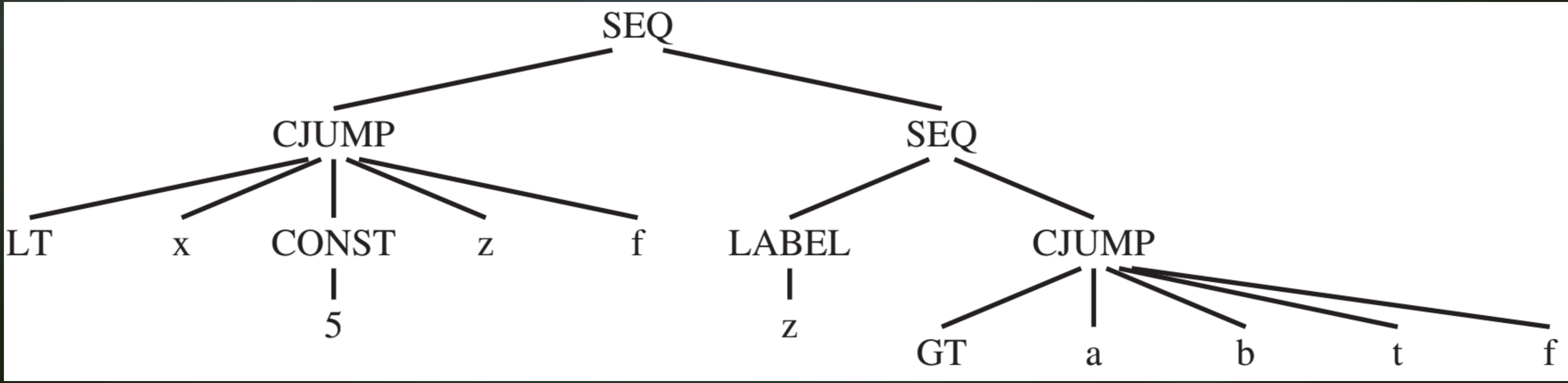
IR Trees

Conditional Instructions (more complex)

x < 5 and a > b

will be translated as

SEQ(s1(z, f), SEQ(LABEL z, s2(t, f)))



IR Trees

Loops

While loops

test:

if not (*condition*) goto *done*

body

goto *test*

done:

- Lowered to conditionals and jumps

For loops

```
for (i=lo; i<=hi; i++;) {
    body
}
```

equal to

```
i=lo;
limit=hi;
while (i<=limit) {
    body
    i++;
}
```

IR Trees

Limitis

- IR trees need to be translated into assembly or machine language
  - Operators of the Tree language are chosen carefully to match the capabilities of most machines
  - However, certain aspects of the tree language do not correspond exactly with machine languages. Some examples:
    - CJUMP instruction can jump to either of two labels, but real machines' conditional jump instructions fall through to the next instruction if the condition is false
    - ESEQ nodes enforces an order when evaluating subexpressions
    - CALL puts returned value in specific register

From IR Trees to Canonical Trees

- Take IR trees and rewrite it into an equivalent tree without any of the cases listed above, such as ESEQ and two-way CJUMP
- Three stages:
  - a tree is rewritten into a list of *canonical trees* without *SEQ* or *ESEQ* nodes
  - this list is grouped into a set of basic blocks which contain no internal jumps or labels
  - the basic blocks are ordered into a set of traces in which every CJUMP is immediately followed by its false label

Basic block

Definition

- A sequence of statements that is always entered at the beginning and exited at the end
  - First statement: LABEL
  - Last statement: JUMP or CJUMP
  - No other LABEL, JUMP or CJUMP statement

Basic block

Construction of Basic Blocks from sequences of statements

- Linear scan of the statement sequence
  - When a LABEL is found, start new BB and close previous one
  - When a JUMP or CJUMP is found, end current BB and start new one
  - Any BB opened with no LABEL gets a new one
  - Any BB closed with no JUMP or CJUMP receives a new JUMP to next BB LABEL
  - A “final” BB with just a LABEL statement is inserted at the end

Trace

Definition

- Basic blocks:
  - can be arranged in any order
  - end with a jump to the appropriate place
- A trace is a sequence of statements that could be consecutively executed during the execution of the program.
  - Can include conditional branches
  - Each block must be in exactly one trace
- A program has many different, overlapping traces

Trace

Construction of a set of traces

- GOAL: to make a set of traces that exactly covers the program:
  - Put as few traces as possible in our covering set
  - Order BBs satisfying the condition that each CJUMP is followed by its false label
  - Many of the unconditional JUMPs are immediately followed by their target label
- Possible algorithm:
  - Start with some block and follows a chain of jumps
  - Mark each block and appending it to the current trace
  - In case of a block whose successors are all marked, end the trace
  - Pick an unmarked block to start the next trace

Control Flow Graph

Definition

- Control Flow Graph (CFG) of a program is a directed graph $G(B, E)$ such that
  - There is one node $i \in B$ for each IR statement ($stat_i$)
  - There are two additional nodes i_{in}, i_{out}
  - There is one edge $(i, i') \in E$ if the statement $stat_{i'}$ is executed immediately after the statement $stat_i$
  - Each node must have at most two immediate successors
  - For the first statement ($stat_0$) there is an arc (i_{in}, i_0)
  - An arc (j, i_{out}) is added for each node j bound to a statement ($stat_j$) preceding an exit point of the program

Control Flow Graph

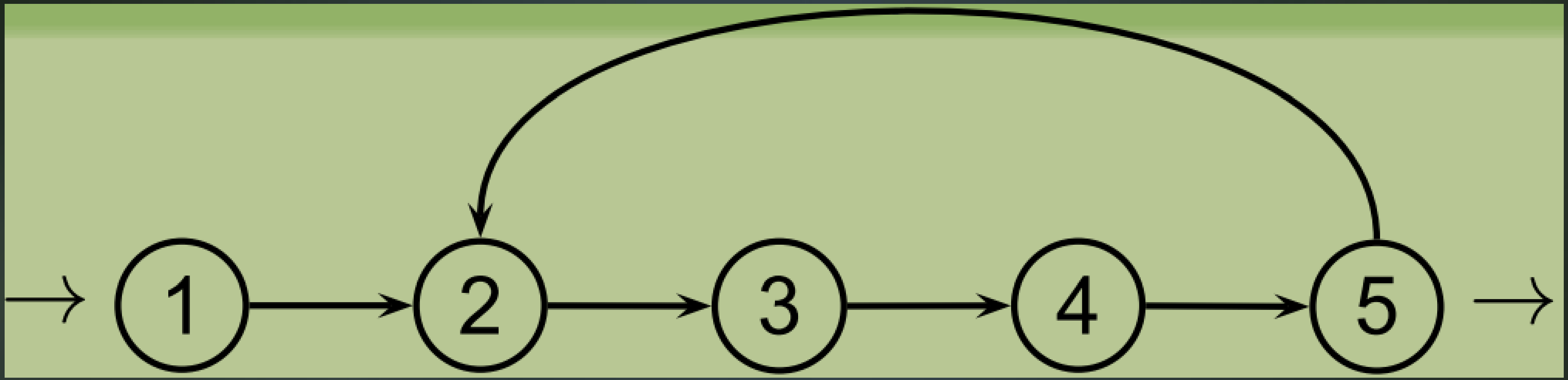
Basic Blocks in the CFG

- A BB is recognized as a sequence of nodes $i_0, \ldots, i_n$ such that:
  - There are arcs (i_0, i_1) … (i_{n-1}, i_n)
  - No other arc in the CFG has any node of the sequence except i_0 as its target
  - No other arc in the CFG has any node of the sequence except i_n as its starting point
- It is often useful to simplify the CFG by collapsing each BB in a single node

Control Flow Graph

Loops: a simple graph-theoretical definition

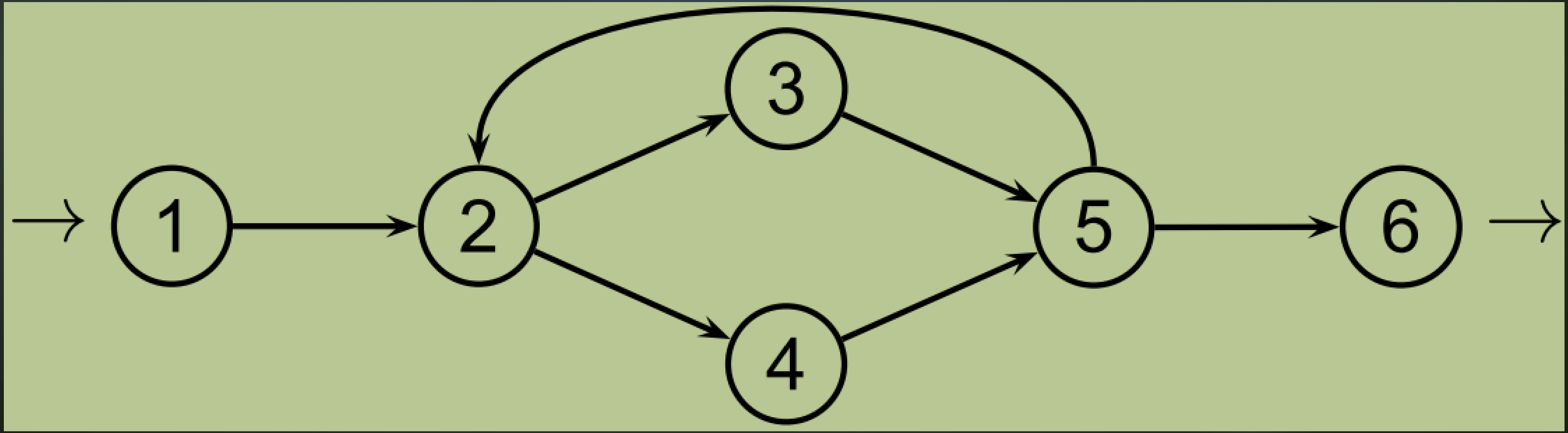
- A loop is a directed cycle (or circuit) in the CFG



Control Flow Graph

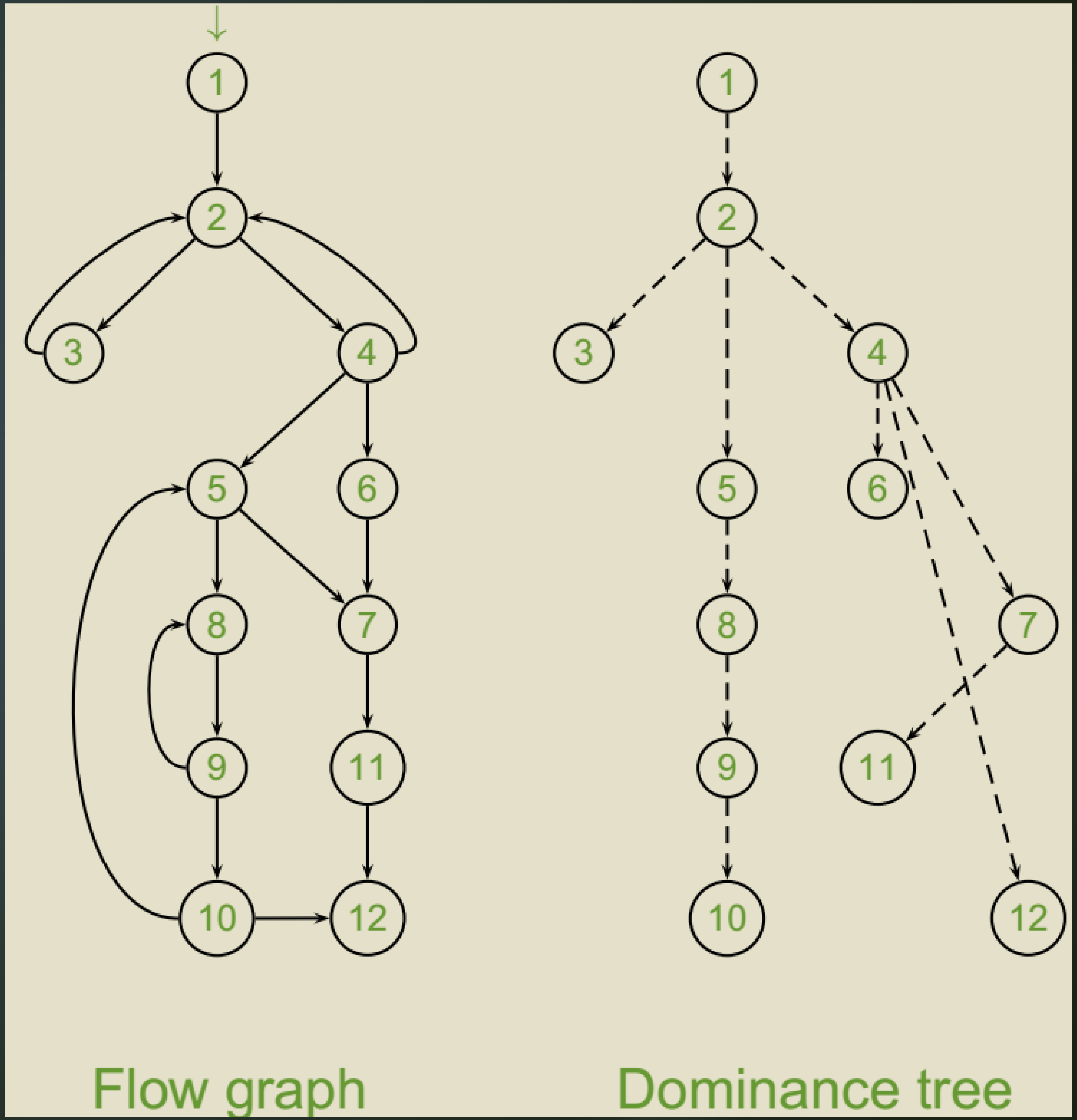
Loops: a second definition

- A loop is a strongly connected component (SCC) of the CFG: a set of nodes such that each one is reachable from every other node in the set
  - The picture contains three SCC’s: {2,3,4,5}, {2,3,5} and {2,4,5}
  - The first is a maximal SCC, while the others are strongly connected but not maximal



Control Flow Graph
Dominance relation

- A node d dominates a node n if d occurs before n on every directed path from the start node s_0 to n.
  - Every node dominates itself.
- In other words an execution trace reaching n cannot avoid executing d before.
- Properties
  - *Reflexivity:* $\forall a : \ a \ dom \ a$
  - *Transitivity :* $a \ dom \ b \ and \ b \ dom \ c \ \Rightarrow \ a \ dom \ c$
  - *Anti-symmetry :* $a \ dom \ b \ and \ b \ dom \ a \ \Rightarrow \ a \ = \ b$
- Therefore the dominance relation is a partial order



Control Flow Graph

Dominator Tree

- In a connected graph, suppose a node n has two dominators d, e
  - Then either d dominates e, or e dominates d
- Immediate Dominator
  - A node $m \neq n$ is the immediate dominator of n if
  - $m\ dom\ n$ and $\forall d$ such that $d\ dom\ n : d\ dom\ m$
  - The entry node has no immediate dominator, for every other node the immediate dominator is unique
- <u>Computing dominance is an essential step for program analysis</u> (and also for many other applications of graphs)

Static Single Assignment
Generalities

- Many optimizations require to represent definitions and uses of variables
- One possibility is to maintain an explicit structure representing this information
- An improvement is **static single assignment form (SSA)**
  - An IR where each variable has only one definition in the program text
    - The single definition can be in a loop

Static Single Assignment

Motivation

- Advantages of SSA
  1. Simplify data-flow analysis and program optimizations
     - each variable has only one definition
  2. The size of the SSA form is linear in the size of the original program
  3. It is easier to perform register allocation
  4. Unrelated uses of the same variable disappear, e.g.:

     for i ← 1 to 100 do A[i] ← 0

     for i ← 1 to 20 do s ← s + B[i]

     No need to use the same register to hold the control variables of the two loops.

Static Single Assignment

Construction: SSA of Basic Blocks

- Algorithm
  - Each new definition of a variable a is modified to define a fresh variable a1,a2,...
  - Each use of the variable is changed to use the most recently defined version

Original	SSA form
$a \leftarrow x + y$	$a_1 \leftarrow x + y$
$b \leftarrow a - 1$	$b_1 \leftarrow a_1 - 1$
$a \leftarrow y + b$	$a_2 \leftarrow y + b_1$
$b \leftarrow x \times 4$	$b_2 \leftarrow x \times 4$
$a \leftarrow a + b$	$a_3 \leftarrow a_2 + b_2$

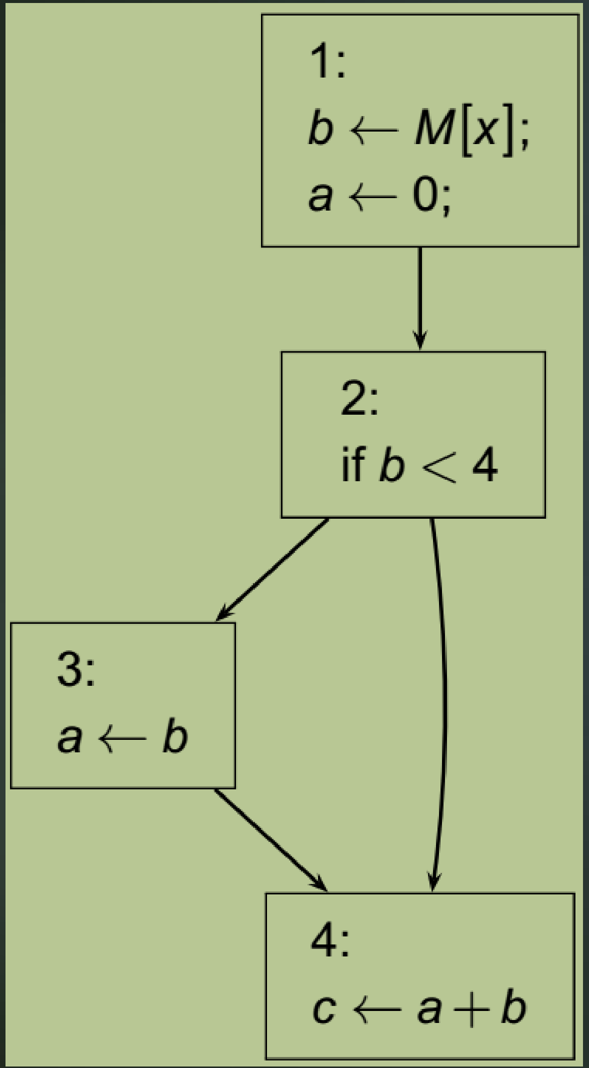
Static Single Assignment

Construction: SSA outside Basic Blocks

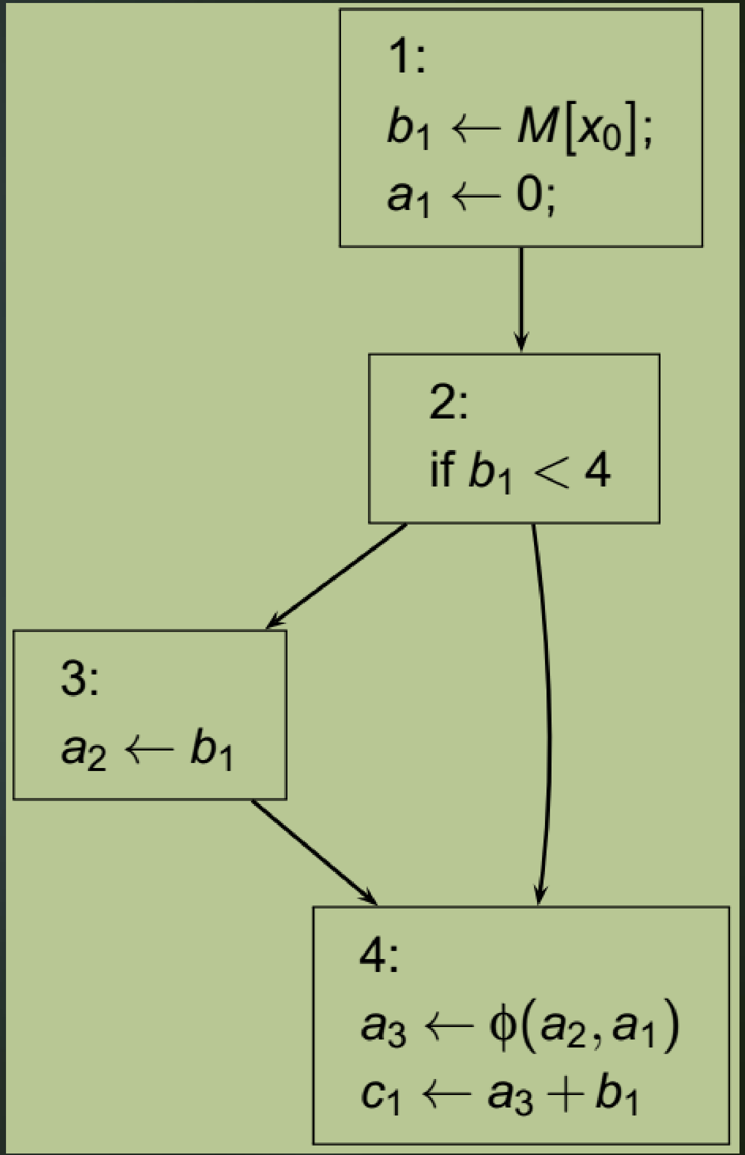
- When a statement has more than one predecessor, the idea of “most recent definition” becomes nonsense
- The definition used to assign the new value depends on the control flow (determined at runtime)
- How to choose the variable version on a confluence point?
- Add a special notation to support this selection operation: the φ-function
- A φ-function has as many arguments as the number of predecessor nodes

Static Single Assignment

Construction: SSA outside Basic Blocks

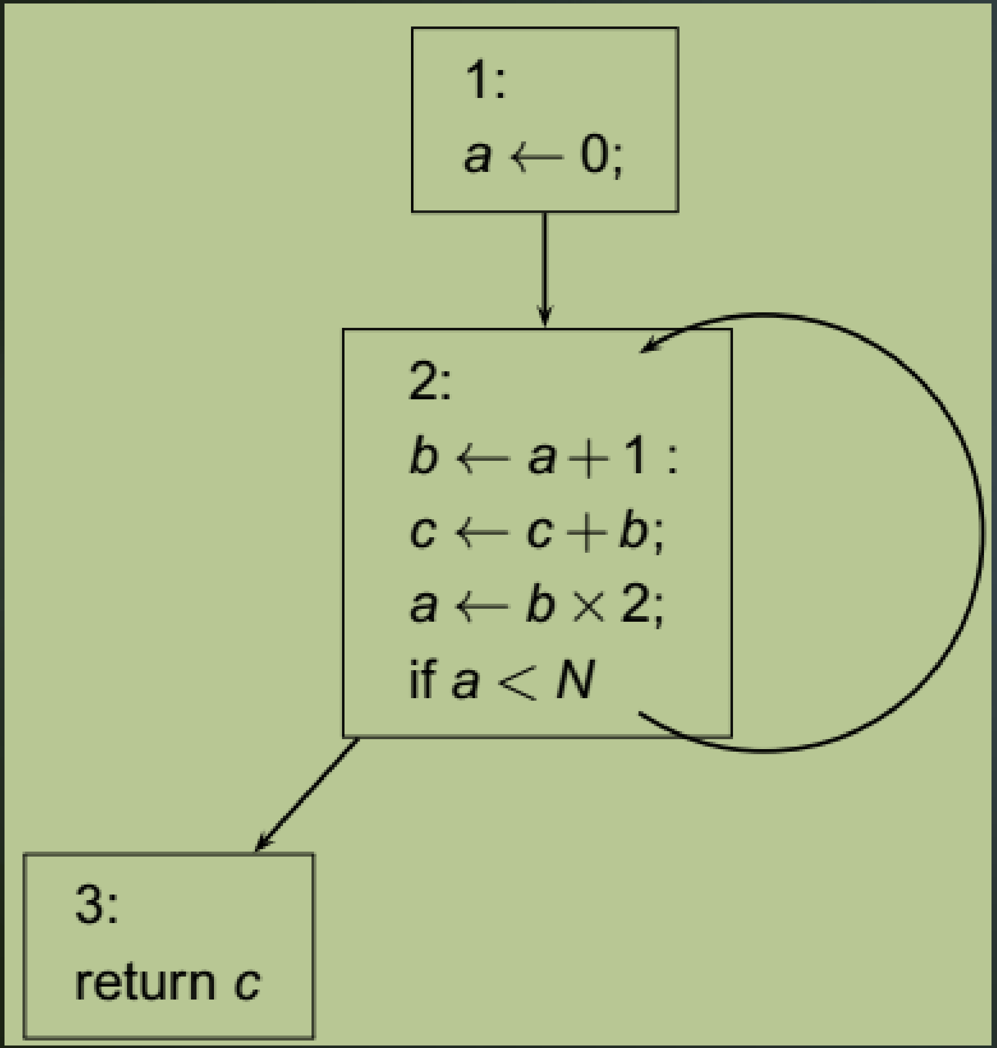


which version of *a* should be used in block 4?

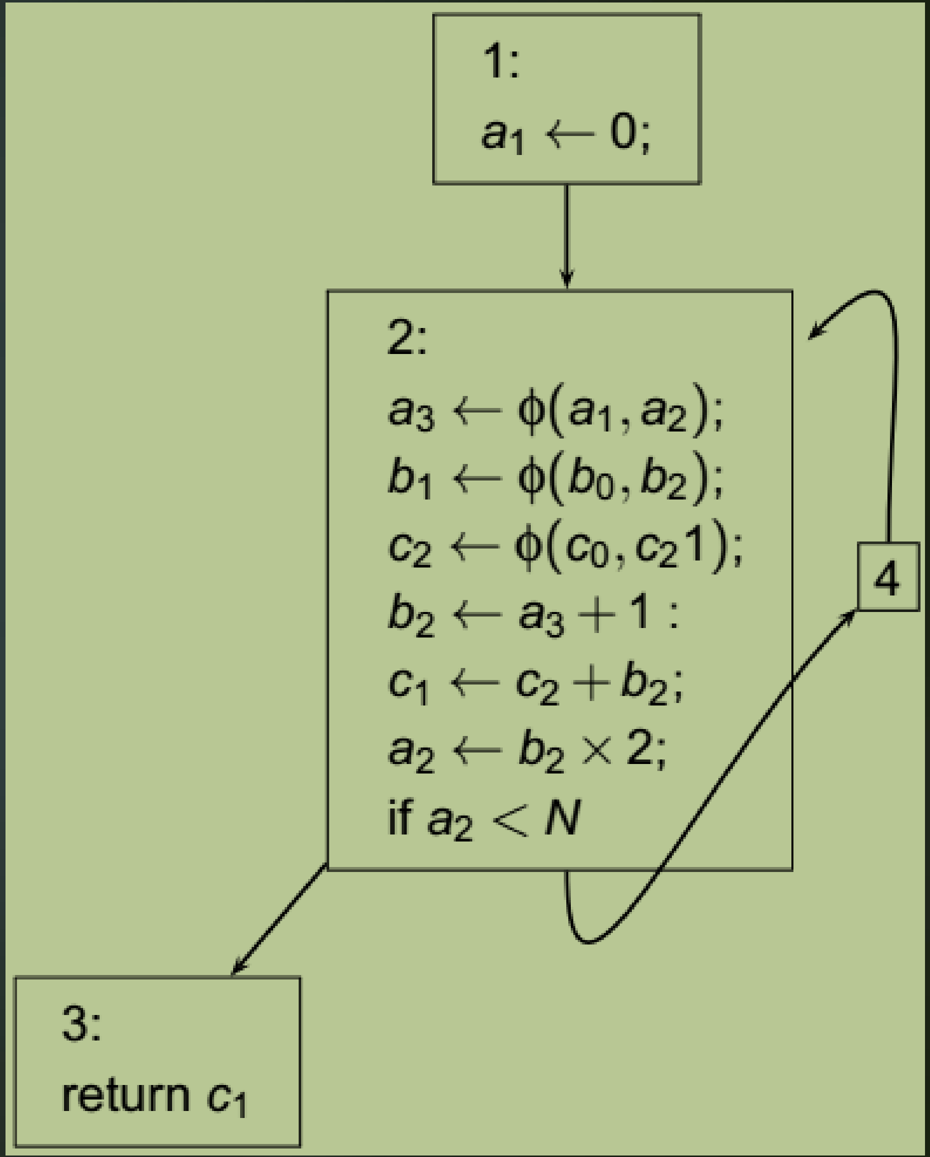


Static Single Assignment

Construction: SSA with loops



which version of *a*, *b* and *c* should be used in block 2?



Static Single Assignment

Construction: Φ-functions

- How does the φ-function know which edge was taken?
  1. If the program has to be made executable, place a MOVE instruction on each incoming edge.
  2. In compilation, data-flow analysis will provide the connection between a use a_i and a definition a_i of a variable
- Dominance properties
  1. if x is the *i*-th argument of a φ-function in block *n*, then the definition of x is always executed before (pre-dominates) the *i*-th predecessor of *n*.
  2. if x is used in a non-φ-statement in block *n*, then the definition of x dominates node *n*.

Static Single Assignment

Dominance properties

- **Path-convergence criterion (PCC)**. There should be a φ-function for variable a at node z of the flow graph exactly when all of the following are true:
  1. There is a block x containing a definition of a
  2. There is a block y (with $y \neq x$) containing a definition of a
  3. There is a nonempty path P_{xz} of edges from x to z
  4. There is a nonempty path P_{yz} of edges from y to z
  5. Paths P_{xz} and P_{yz} do not have any node in common other than z
  6. The node z does not appear within both P_{xz} and P_{yz} prior to the end

Static Single Assignment

Construction: Converting to SSA

- Hypothesis the start node contains an implicit definition of every variable

  (a ← unitialized or incoming argument)
- Conversion algorithm
  1. Inserting the φ functions
  2. Assigning subscripts to variables.
     - No need of φ-function for every variable at each junction
     - In loop example c ← a + b in BB4 is reached b for all edges
     - Need to insert a φ-function in a node that is reached by different def's of the same variable

Static Single Assignment

Construction: Inserting the φ functions

- Iterated path-convergence algorithm
  - to compute the nodes that need φ-functions

while ∃ blocks x, y, z satisfying PCC and z does not contain a φ function for variable a

do insert $a \leftarrow \varphi(a, a, \ldots, a)$ at node z

- Where the number of arguments of φ equals the number of predecessors of node z.
- After inserting a φ function the loop is reentered

Static Single Assignment

Construction: Placing φ-functions with Dominance Frontier

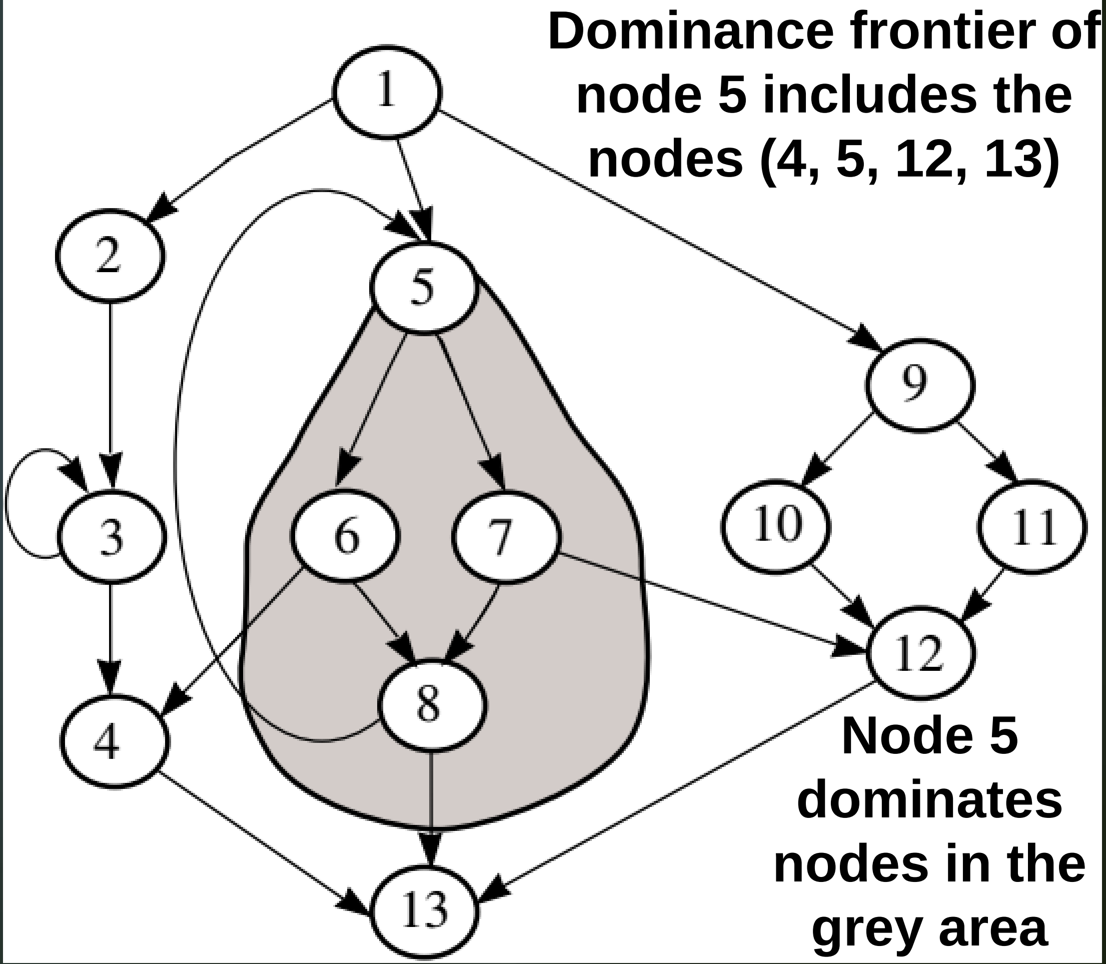
- Iterated path-convergence algorithm
  - Not practical
  - very costly to examine every triple of nodes x, y, z and every path leading from x and y
- Preliminaries:
  - Node x strictly dominates node w if x dominates w and $x \neq w$
  - Predecessor / Successor denote CFG relations.
  - Parent / child denote dominance tree relations.
  - In a tree, an ancestor of node n is the parent of n or the parent of an ancestor of n

Static Single Assignment

Construction: Placing φ-functions with Dominance Frontier

Dominance Frontier

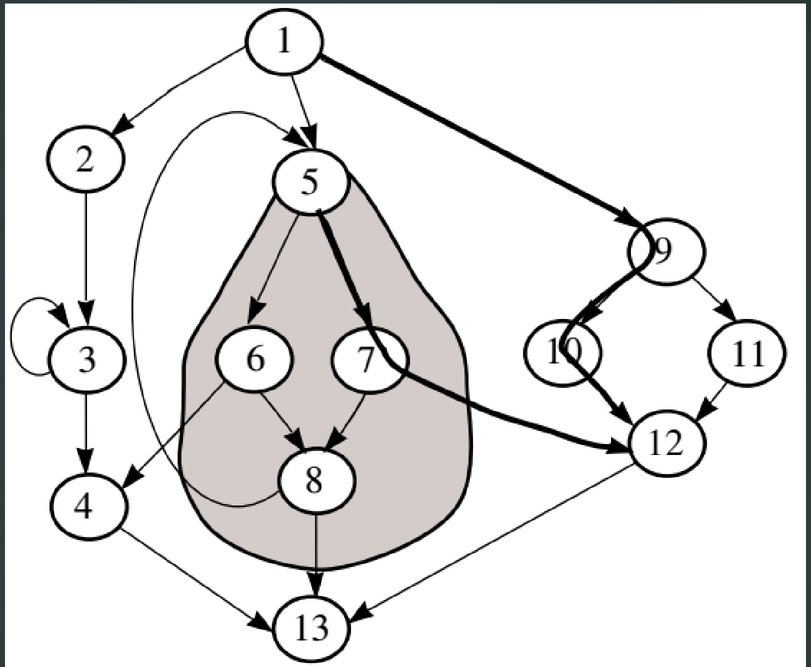
- The dominance frontier $DF(x)$ of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w
- If x dominated all predecessors of w, it would dominate w too
- Intuitively, DF is the border between dominated and non-dominated nodes, therefore it is a point of convergence of disjoint paths



Static Single Assignment

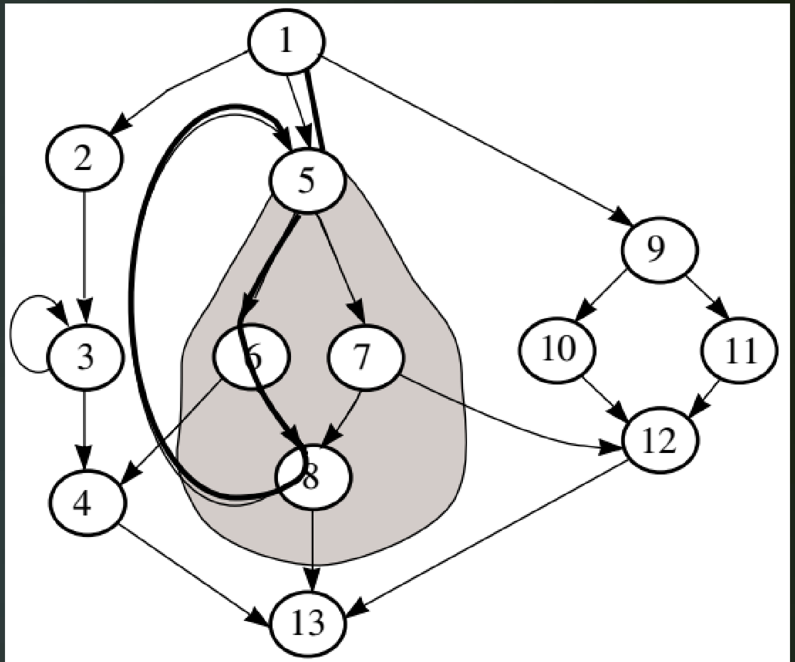
Construction: Placing φ-functions with Dominance Frontier

Any node in the dominance frontier of n is also a point of convergence of nonintersecting paths, one from n and one from the root node.



For node 12

$P_{5,12}$ and $P_{1,12}$



For node 5

$P_{1,5}$ and $P_{5,5}$

Static Single Assignment

Construction: Placing φ-functions with Dominance Frontier

Dominance Frontier Criterion

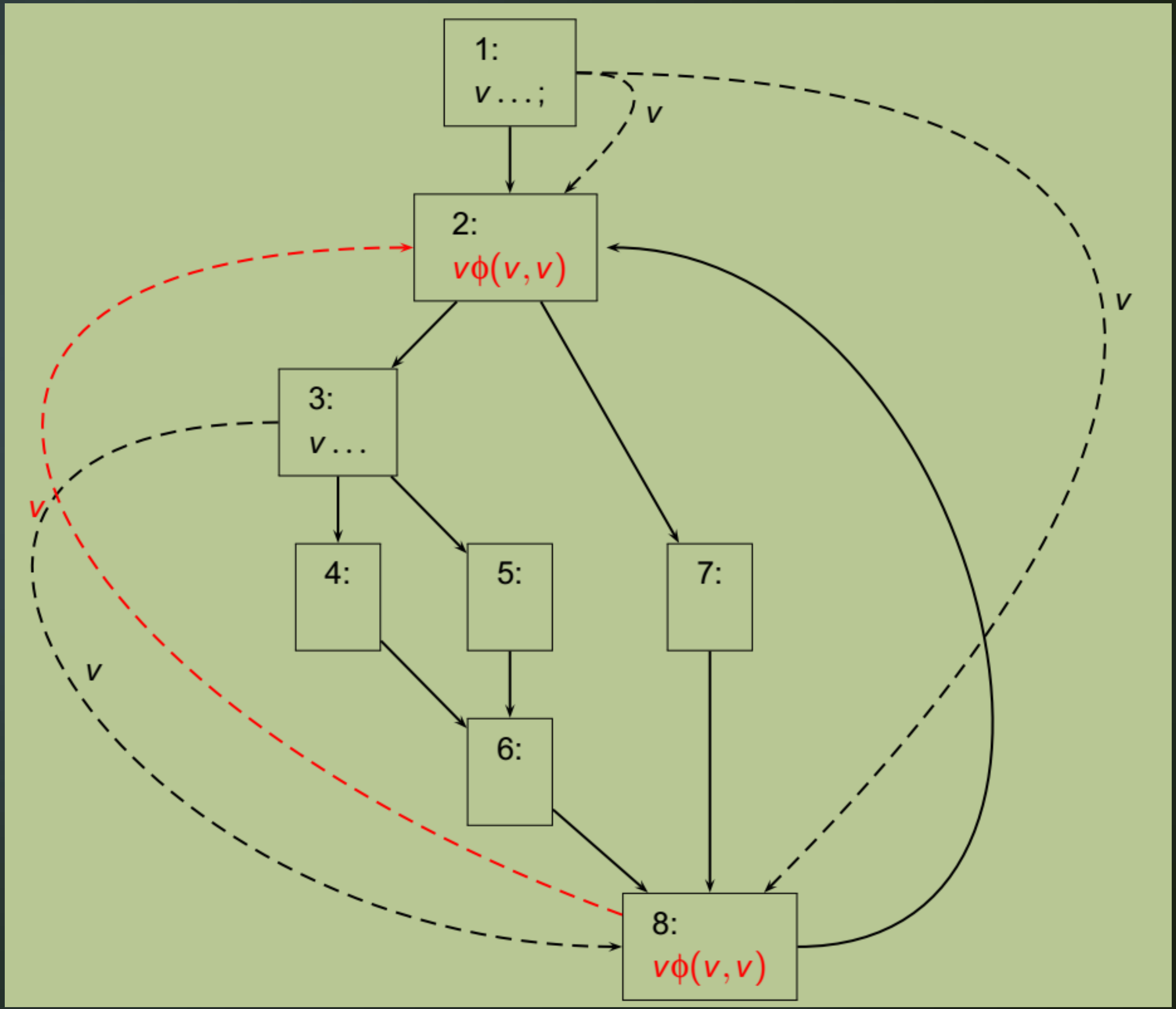
- For each variable a defined in node x, we insert a φ-function in every node that is in $DF(x)$.
- This criterion is equivalent to the Path Convergence criterion, since:
  1. If a_x is the *i*-th argument of a φ function in block z, then the definition $x: a \leftarrow \ldots$ dominates the *i*-th predecessor of z.
  2. If a is used in a non-φ statement in block n, then the definition of a dominates n.
  - Since a φ-function $a \leftarrow$ φ(...) is a kind of definition of a, we must apply the criterion to each newly introduced φ-functions
  - This requires an iterative procedure that terminates when no nodes need φ-functions.

Static Single Assignment

Construction: Placing φ-functions PCC vs Dominance Frontier

Example using **PCC**

- blocks 1 and 3 define v
- block 2 needs vφ(v,v) since ∃ paths P12 = 12 and P32 = 34682 which are disjoint



Static Single Assignment

Construction: Placing φ-functions PCC vs Dominance Frontier

Example using **DF**

- Block 1 contains a definition of v and $DF(1) = \emptyset$: no φ function needed
- Block 3 contains a definition of v and $DF(3) = \{8\}$: node 8 needs φ function
- Now Block 8 contains a definition of v and $DF(8) = \{2\}$: node 2 needs φ:
- Since $DF(2) = \{2\}$ and 2 contains already the φ function for v, terminate.