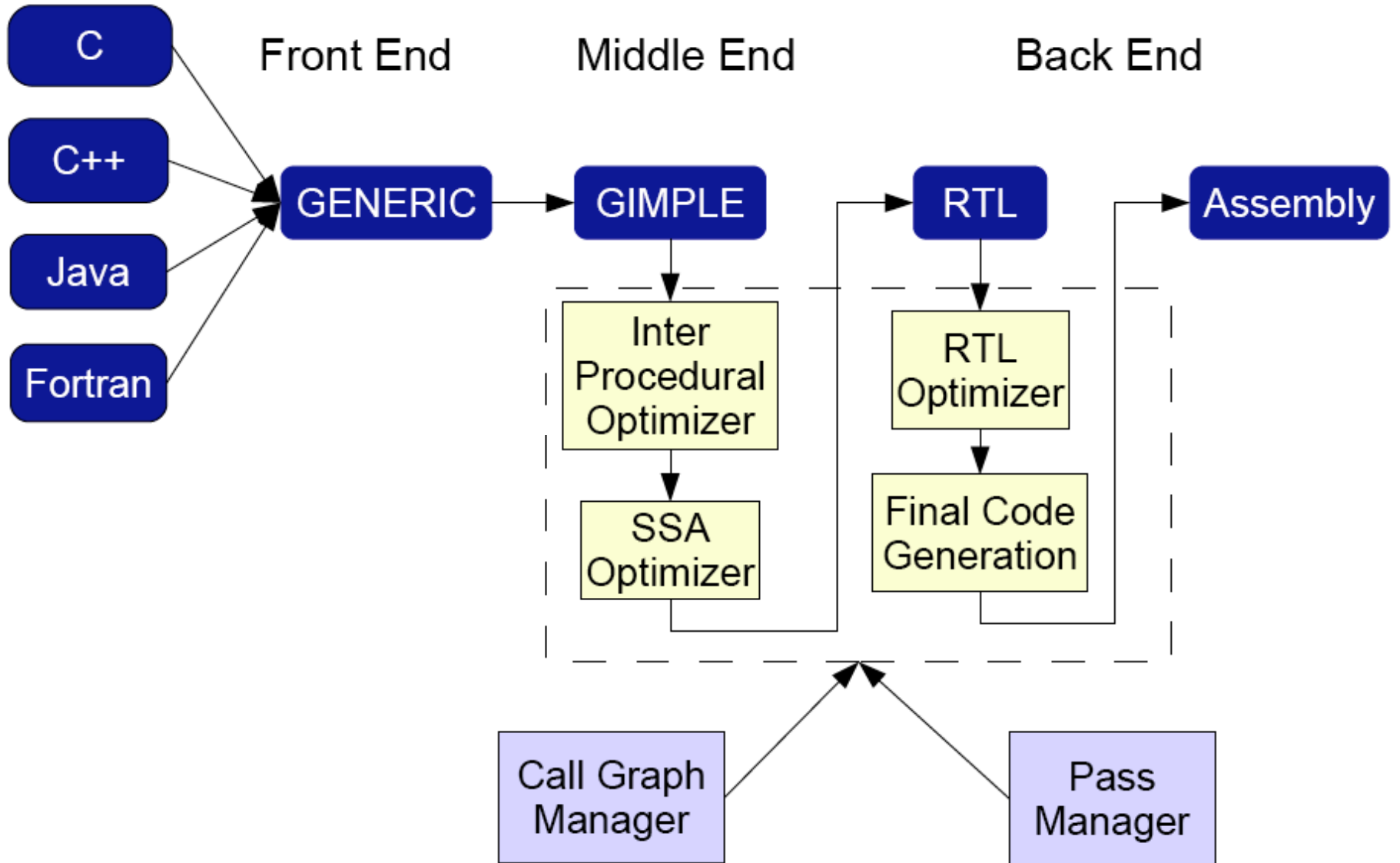


GCC Compiler Overview



GCC Overview cont...

SSA Optimizers

- vectorization
- loop optimizations
- scalar optimizations: CCP, DCE, DSE, FRE, PRE, VRP, SRA
- field-sensitive, points-to alias analysis

RTL Optimizers

- RTL has infinite registers
- register allocation
- scheduling, SW pipelining, CSE, ...

From **GCC - An Architectural Overview**, D. Novillo, September 2006.

Static Single Assignment Form

Last Time

- Static single assignment (SSA) form

Today

- Continue conversion to and from SSA
- Applications of SSA

Transformation to SSA Form

Two steps

- Insert ϕ -functions
- Rename variables

Inserting Phi Nodes

Calculate the dominator tree

- a lot of research has gone into calculating this quickly

Computing dominance frontier from dominator tree

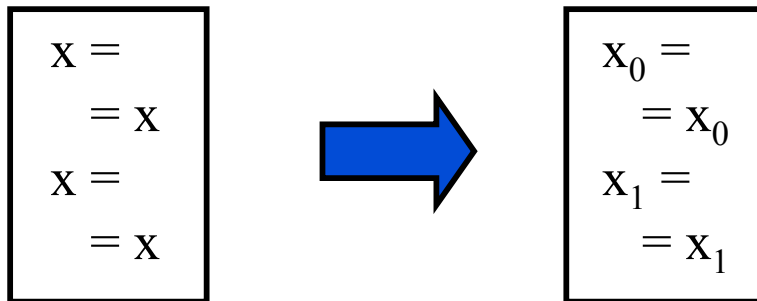
- $DF_{\text{local}}[n]$ = successors of n (in CFG) that are not strictly dominated by n
- $DF_{\text{up}}[n]$ = nodes in the dominance frontier of n that are not strictly dominated by n 's immediate dominator
- $DF[n] = DF_{\text{local}}[n] \cup \bigcup_{c \in \text{children}[n]} DF_{\text{up}}[c]$

Variable Renaming

Basic idea

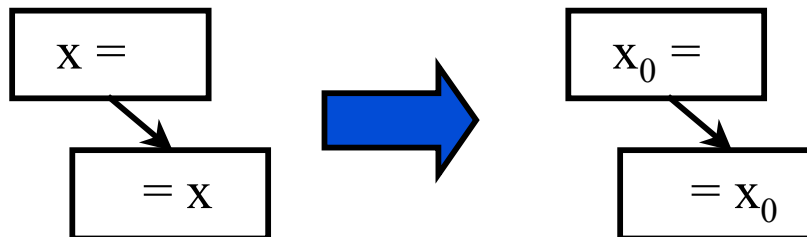
- When we see a variable on the LHS, create a new name for it
- When we see a variable on the RHS, use appropriate subscript

Easy for straightline code



Use a stack when there's control flow

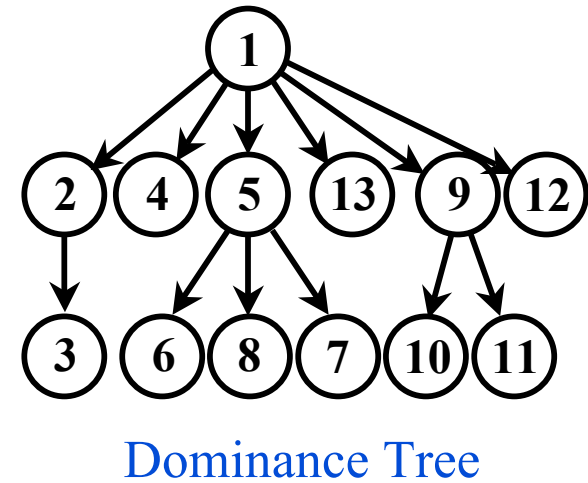
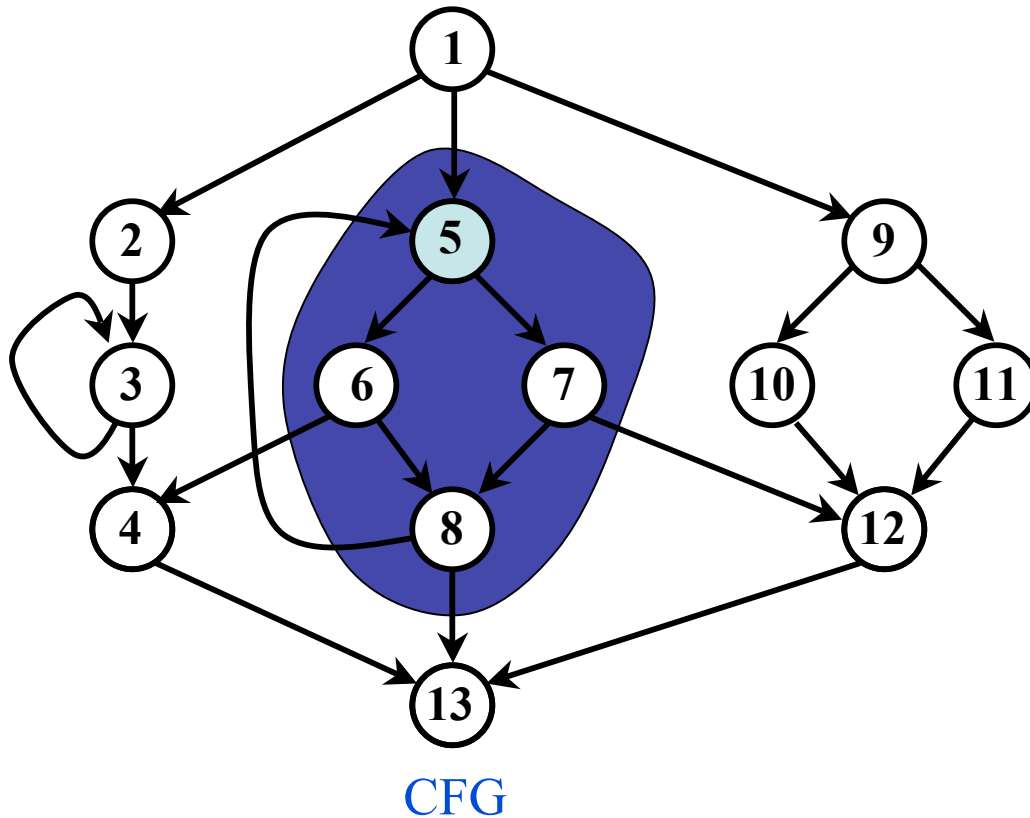
- For each use of x , find the definition of x that dominates it



Traverse the dominance tree

Dominance Tree Example

The dominance tree shows the dominance relation



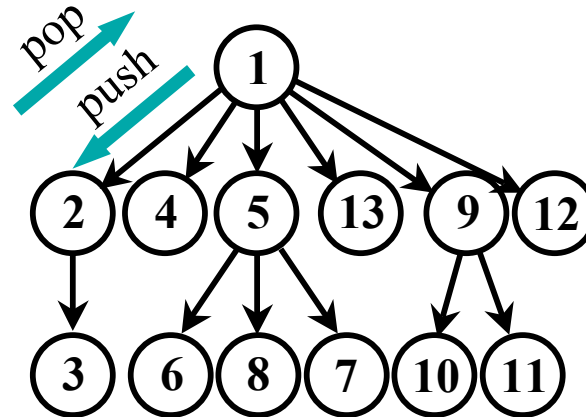
Variable Renaming (cont)

Data Structures

- $\text{Stacks}[v] \forall v$
Holds the subscript of most recent definition of variable v , initially empty
- $\text{Counters}[v] \forall v$
Holds the current number of assignments to variable v ; initially 0

Auxiliary Routine

```
procedure GenName(variable v)
  i := Counters[v]
  push i onto Stacks[v]
  Counters[v] := i + 1
```



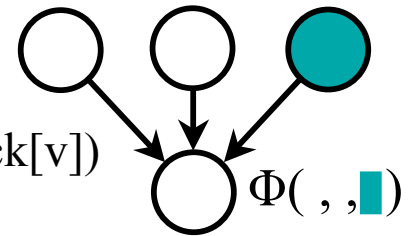
Use the Dominance Tree to remember the most recent definition of each variable

Variable Renaming Algorithm

procedure Rename(block b)
 if b previously visited return

Call Rename(entry-node)

for each statement s in b (in order)
for each variable $v \in \text{RHS}(s)$ (except for ϕ -functions)
 replace v by v_i , where $i = \text{Top}(\text{Stacks}[v])$
for each variable $v \in \text{LHS}(s)$
 GenName(v) and replace v with v_i , where $i = \text{Top}(\text{Stack}[v])$
for each $s \in \text{succ}(b)$ (in CFG)
 $j \leftarrow$ position in s's ϕ -function corresponding to block b
for each ϕ -function p in s
 replace the j^{th} operand of $\text{RHS}(p)$ by v_i , where $i = \text{Top}(\text{Stack}[v])$



for each $s \in \text{child}(b)$ (in DT)
 Rename(s)

for each ϕ -function or statement t in b
for each $v_i \in \text{LHS}(t)$
 Pop(Stack[v])

} Recurse using Depth First Search
 } Unwind stack when done with this node

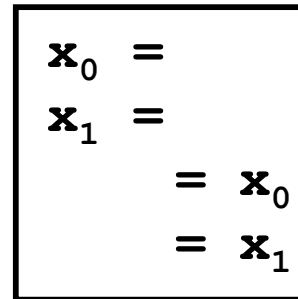
Transformation from SSA Form

Proposal

- Restore original variable names (*i.e.*, drop subscripts)
- Delete all ϕ -functions

Complications

- What if versions get out of order?
(simultaneously live ranges)



Alternative

- Perform dead code elimination (to prune ϕ -functions)
- Replace ϕ -functions with copies in predecessors
- Rely on register allocation coalescing to remove unnecessary copies

Dead Code Elimination for SSA

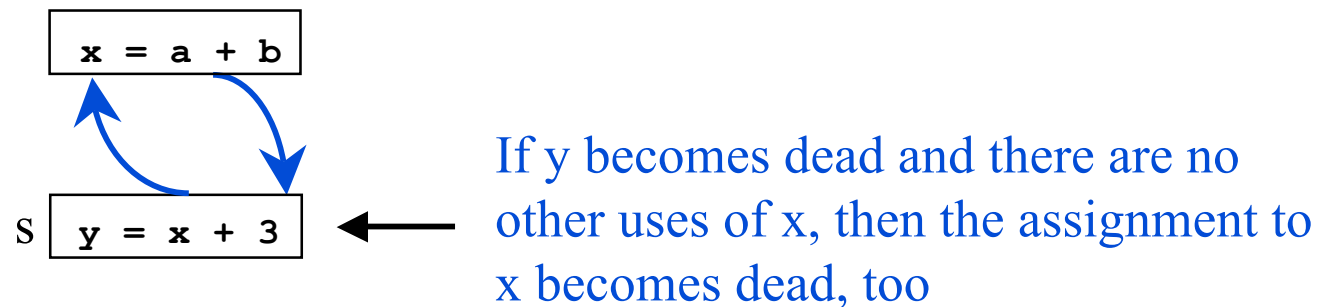
Dead code elimination

while \exists a variable v with no uses and whose def has no other side effects

Delete the statement s that defines v

for each of s 's uses w

Delete the use from list of uses of variable w



- Contrast this approach with one that uses liveness analysis
 - This algorithm updates information incrementally
 - With liveness, we need to invoke liveness and dead code elimination iteratively until we reach a fixed point

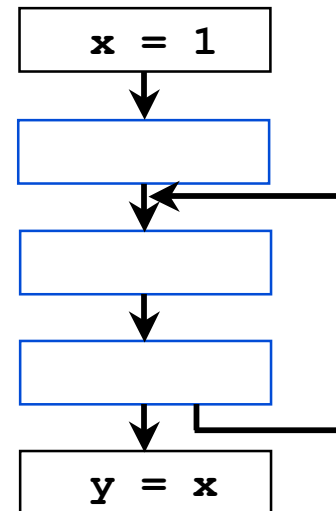
Implementing Simple Constant Propagation

Standard worklist algorithm

- Identifies simple constants
- For each program point, maintains one constant value for each variable

Problem

- Inefficient, since constants may have to be propagated through irrelevant nodes



Solution

- Exploit a sparse dependence representation (*e.g.*, SSA)

Sparse Simple Constant Propagation

Reif and Lewis algorithm

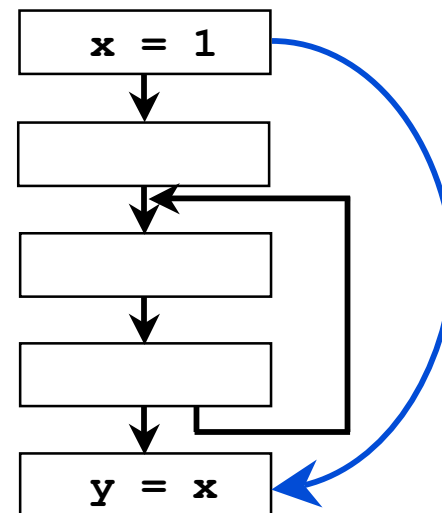
- Identifies simple constants
- Faster than Simple Constants algorithm

SSA edges

- Explicitly connect defs with uses
- How would you do this?

Main Idea

- Iterate over SSA edges instead of over all CFG edges



Sparse Simple Constants Algorithm (Ch. 19 in Appel)

worklist = all statements in SSA

while worklist $\neq \emptyset$

 Remove some statement S from worklist

 if S is $x = \text{phi}(c, c, \dots, c)$ for some constant c

 replace S with $v = c$

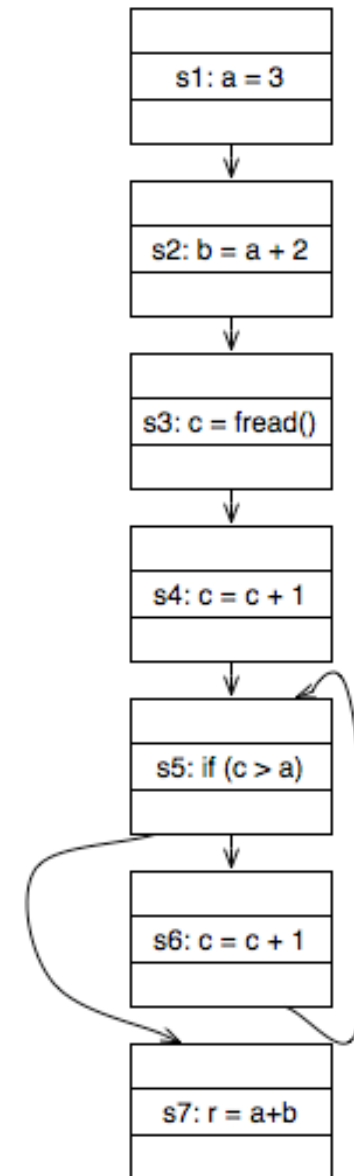
 if S is $x = c$ for some constant c

 delete S from program

 for each statement T that uses x

 substitute c for x in T

 worklist = worklist union {T}



Copy Propagation

Algorithm

worklist = all statements in SSA

while worklist $\neq \emptyset$

 Remove some statement S from worklist

 if S is $x = \text{phi}(y)$ or $x = y$

 for each statement T that uses x

 replace all use of x with y

 worklist = worklist union {T}

 delete S

Concepts

SSA construction

- Place phi nodes
- Variable renaming

Transformation from SSA to executable code depends on the optimizations copy propagation, dead-code elimination, and coalescing

Some optimizations that are simpler and more efficient with SSA

- dead-code elimination
- constant propagation
- copy propagation

Others that weren't covered

- induction variable detection, strength reduction, and elimination
- register allocation
- ...

Next Time

Assignments

- Read Alpern and Zadeck paper on value numbering

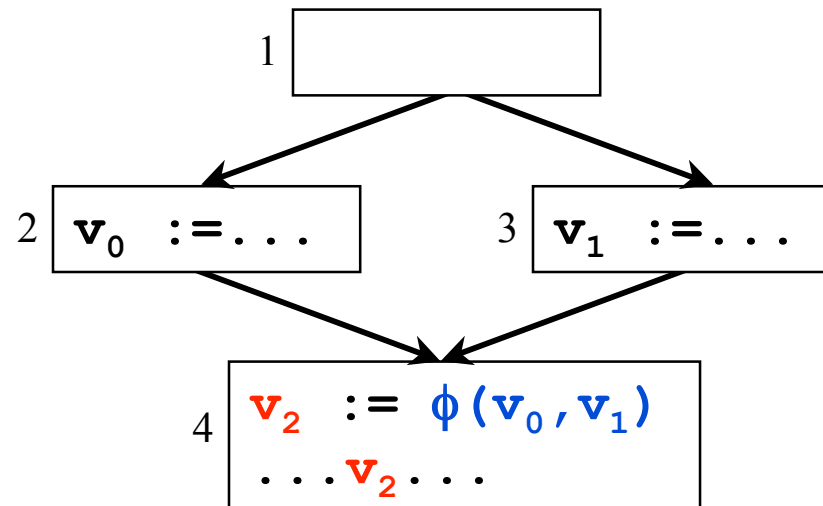
Lecture

- Using SSA for value numbering

Backward Analyses vs. Forward Analyses

For forward data-flow analysis, at phi node apply meet function

For backward data-flow analysis?



Static Single Information Form (SSI)

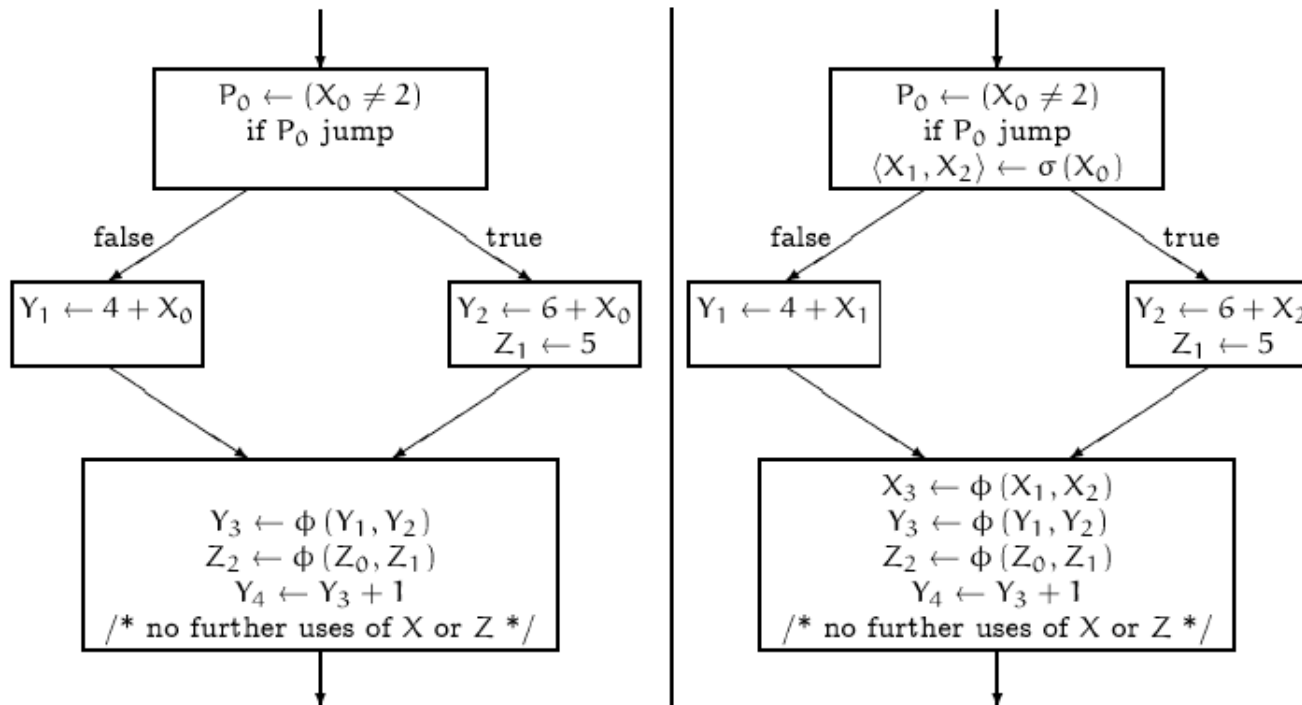


Figure 5.1: A comparison of SSA (left) and SSI (right) forms.

Ananian's Masters Thesis, 1997 MIT