SSA Construction

Daniel Grund & Sebastian Hack

Saarland University

CC Winter Term 09/10

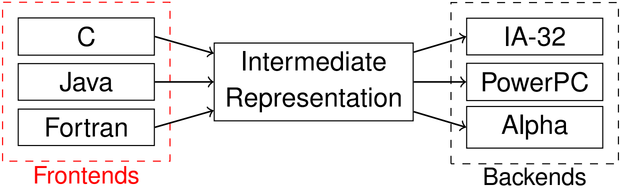
Outline

Overview

Intermediate Representations
- Why?
- How?
- IR Concepts

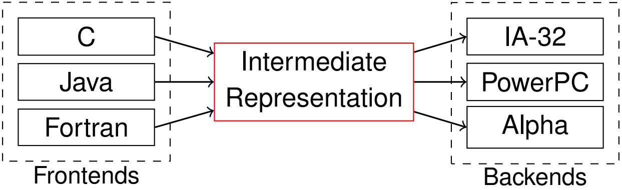
Static Single Assignment Form
- Introduction
- Theory
- SSA Construction

Frontend



- *Checks correctness* of source code wrt. a given language definition
- Transforms (valid) source into the intermediate representation

Intermediate Representation (IR)



C

Java

Fortran

Frontends

Intermediate Representation

IA-32

PowerPC

Alpha

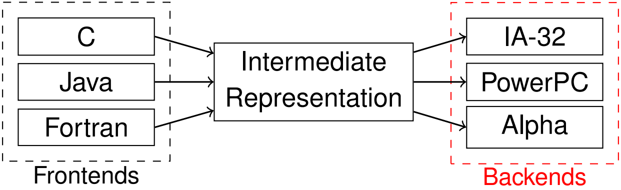
Backends

- Compiler internal data structures representing a program
- *Uniform abstraction* from source languages and target architectures

⇒ $n + m$ compiler components instead of $n \cdot m$ compilers

- *Optimizations* are performed on the IR

Backend



- Encapsulates all details of a target architecture
- *Code generation*
    - Instruction selection
    - Instruction scheduling
    - Register allocation

Outline

Overview

Intermediate Representations
- Why?
- How?
- IR Concepts

Static Single Assignment Form
- Introduction
- Theory
- SSA Construction

Outline

Overview

Intermediate Representations
- Why?
- How?
- IR Concepts

Static Single Assignment Form
- Introduction
- Theory
- SSA Construction

Motivating IRs

- Bridge the gap between abstract syntax tree and object code
- Provide data structures more suitable for analyses/optimizations
- Easier retargetability (reuse of IR for source-target pairs)
- Reuse of machine independent optimizations

Outline

Overview

Intermediate Representations
  - Why?
  - How?
  - IR Concepts

Static Single Assignment Form
  - Introduction
  - Theory
  - SSA Construction

Design Issues

- Consider source language and target
- Consider (type) of planned optimizations
- Choose the right “level”
  - Higher level means closer to source
  - Lower level closer to target loses some structure/information

- Procedure cloning, inlining, and loop optimizations need structural high-level information
- Branch optimization, software pipelining, and register allocation need representation close to machine

⇒ Possibly multiple levels in one IR (same generic data structures).
So called “lowering” transforms them from high to low.

Lowering

Typical constructs subject to lowering

- array accesses
- struct accesses
- calls (calling convention, ABI)
- instruction selection can be seen as lowering

```
t1 := a[i,j+2]
```

```
t1 := j+2
t2 := 10 * i
t3 := t1 + t2
t4 := 4 * t3
t5 := addr(a)
t6 := t4 + t5
t7 := *t6
```

Outline

Overview

Intermediate Representations

- Why?
- How?
- IR Concepts

Static Single Assignment Form

- Introduction
- Theory
- SSA Construction

Different IR Concepts

Representation of control flow

- Control-flow graph (CFG)
- Basic Block Graph (BBG)

Representation of computation

- Triple code
- Expression trees
- Data dependence graphs

Control Flow Graph (CFG)

Definition

In a CFG there is 1:1 correspondence of nodes to statements/instructions.
Edges represent possible control flow.

Basic Block Graph (BBG)

Definition

A basic block (BB) is a maximal sequence of statements/instructions such that if any is executed all are executed.

Definition

In a BBG nodes are BBs and control flow is represented only between basic blocks.
Inside a BB there are no control dependencies.

Remark: Most people call this CFG.

Triple Code and Expression Trees

Representation of computation/data flow.

What is inside the BBs?

- Triple code: List of elementary instructions
  (x = op a b)
- Expression trees: List of trees
  (x = a + b * c; y = call foo (3 * x);)

Data Dependence Graphs

- Nodes represent computation results (operators)
- Edges represent data dependencies (data flow)
- Problem with concept of variables (state)
- No problem with side-effect-free operators (functional programming)
- Suitable representation for SSA form

Outline

Overview

Intermediate Representations

- Why?
- How?
- IR Concepts

Static Single Assignment Form

- Introduction
- Theory
- SSA Construction

Outline

Motivation

Main goal:

- Make data-flow analyses more efficient
- Make optimizations more effective

Nice “side-effects”:

- Some analyses/optimizations happen implicitly for free
- SSA-construction can implicitly perform CSE
- Use-Def chains are explicit in representation
- Def-Use chains are cheaper to represent

Definition

Static Single Assignment is a property of an IR regarding variables.

Definition

A program is in SSA form if every variable is statically assigned at most once.
I.e. there are no two program locations assigning the same variable.

Intuition Behind Construction

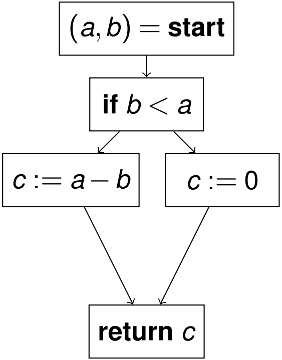
- Replace concept of variable by concept of abstract values
- The entity statically referred to is a value

- For each assignment to a variable v a new abstract value v_i is defined
  v is replaced by $v_1, v_2, \ldots$
- For each use of v the definition v_i valid at that location is used instead

Merge Problem and Phi-Functions

- Problem: What to do when control flow merges?
- Here: Which *c* to use at the return?

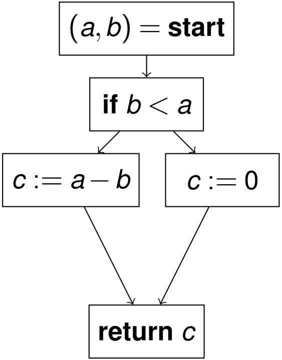
non-SSA



Merge Problem and Phi-Functions

- Problem: What to do when control flow merges?
- Here: Which c to use at the return?
- Solution: Introduce pseudo operation, ϕ-functions
- ϕs select the correct value dependent on control flow

non-SSA



$(a, b) = \textbf{start}$

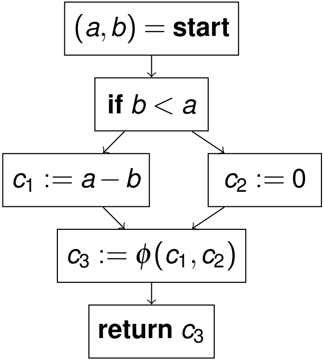
$\textbf{if } b < a$

$c := a - b$

$c := 0$

$\textbf{return } c$

SSA



$(a, b) = \textbf{start}$

$\textbf{if } b < a$

$c_1 := a - b$

$c_2 := 0$

$c_3 := \phi(c_1, c_2)$

$\textbf{return } c_3$

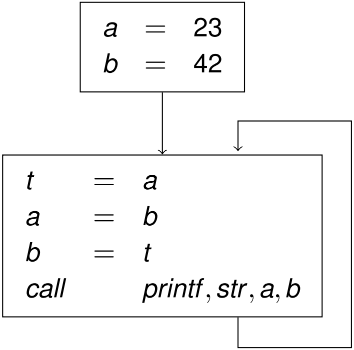
Outline

Phi-Functions

- ϕs have as many operands as the corresponding BB has predecessors
- Each operand is uniquely associated with one of these predecessors
- The result of a ϕ is the operand associated to the predecessor through which the BB was reached

- ϕs always are the first "instructions" in a BB
- all ϕs in a BB must be evaluated simultaneously

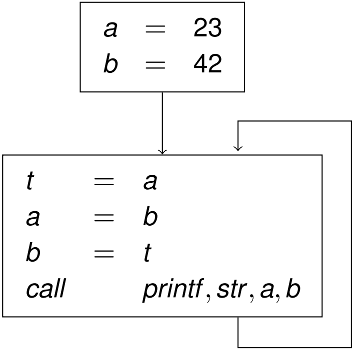
Why Simultaneously? Swap Example



$$\begin{aligned} a &= 23 \\ b &= 42 \end{aligned}$$

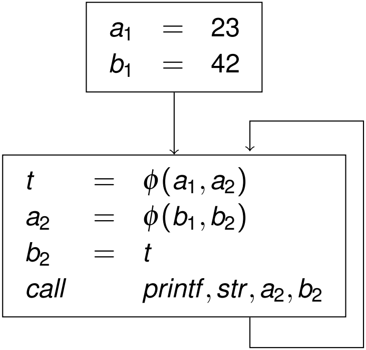
$$\begin{aligned} t &= a \\ a &= b \\ b &= t \\ call & \quad printf, str, a, b \end{aligned}$$

Why Simultaneously? Swap Example



$a = 23$
$b = 42$

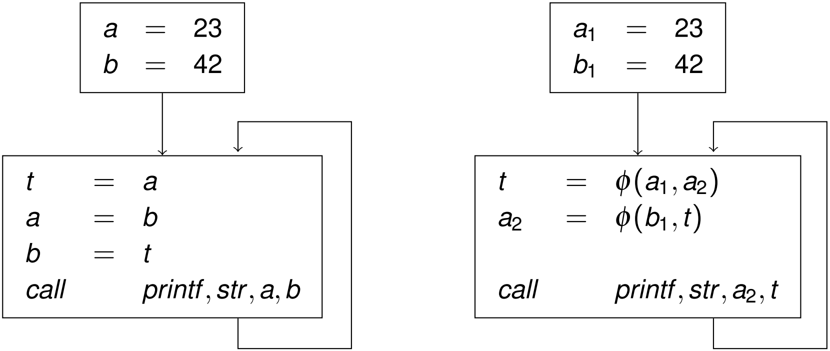
$t = a$
$a = b$
$b = t$
$\mathit{call} \quad \mathit{printf}, \mathit{str}, a, b$



$a_1 = 23$
$b_1 = 42$

$t = \phi(a_1, a_2)$
$a_2 = \phi(b_1, b_2)$
$b_2 = t$
$\mathit{call} \quad \mathit{printf}, \mathit{str}, a_2, b_2$

Why Simultaneously? Swap Example



$a = 23$
$b = 42$

$t = a$
$a = b$
$b = t$
$call \quad printf, str, a, b$

$a_1 = 23$
$b_1 = 42$

$t = \phi(a_1, a_2)$
$a_2 = \phi(b_1, t)$

$call \quad printf, str, a_2, t$

Dominance

Given a CFG with basic blocks X, Y, Z, and S, where S is the start block.

- Dominance: $X \geq Y$
  Each path from S to Y goes through X

- Strict dominance: $X > Y$
  $X > Y$ if $X \geq Y \wedge X \neq Y$

- Dominance is a tree order

- Immediate dominator: $\text{idom}(X)$
  $X = \text{idom}(Y)$ if $X > Y \wedge \nexists Z : X > Z > Y$

SSA Program

A CFG is in SSA form iff

- every variable has exactly one program point where it is defined
- for every use of a variable *x*

  $$\ell : \cdots \leftarrow \tau(\ldots, x, \ldots)$$

  the definition of *x* either
  - dominates ℓ if $\tau \neq \phi$
  - dominates the *i*-th predecessor of ℓ if $\tau = \phi$ and *x* is the *i*-th argument

(Iterated) Join Points

- Consider two paths $p : p_1, \ldots, p_n$, $q : q_1, \ldots q_m$ of nodes in the CFG
- Say p and q *converge* at z if

$$\exists k \leq n, l \leq m. (p_k = q_l = z) \land (\forall 1 \leq i < k, 1 \leq j < l. p_i \neq q_j)$$

- Let $\mathscr{J}(x,y)$ be the set of convergence/join points of x and y:

$$\mathscr{J}(x,y) := \{z \mid \exists p. x \rightarrow^{+} z, q : y \rightarrow^{+} z. p, q \text{ converge at } z\}$$

- $\mathscr{J}(x,y)$ can be extended to sets of nodes:

$$\mathscr{J}(\{x_1, \ldots, x_n\}) := \bigcup_{1 \leq i < j \leq n} \mathscr{J}(x_i, x_j)$$

- When putting a program to SSA form, ϕ-functions have to be inserted for a variable v at all $\mathscr{J}(\mathit{defs}(v))$
- But ϕ-functions constitute new definitions of SSA variables related to v
- Hence $\mathscr{J}$ needs to be iterated:

$$\begin{array}{rcl} \mathscr{J}^1(X) & := & \mathscr{J}(X) \\ \mathscr{J}^{i+1}(X) & := & \mathscr{J}(\mathscr{J}^i(X) \cup X) \\ \mathscr{J}^{+} & := & \mathscr{J}^n \text{ for } n > 1 \text{ and } \mathscr{J}^n = \mathscr{J}^{n+1} \end{array}$$

Placement of Phi-Functions

Theorem (ϕ placement)

Given a non-SSA CFG and a variable x. Let $defs(x)$ be the set of program points where x is defined. A correct SSA construction algorithm has to place a ϕ for x at all program points in

$$\mathscr{J}^+(defs(x)) \cap live(x)$$

Proof sketch:

- Let X and Y contain definitions of v and Z be a join point of two paths $X \rightarrow^+ Z$ and $Y \rightarrow^+ Z$
- ϕ can not be placed before Z
- ϕ must not be placed after Z, e.g. in Z' with $Z \rightarrow^+ Z'$ Disambiguation of paths in a Z' would be impossible
- *Iterated* join points are necessary, since inserted ϕs are new definitions of the variable

Outline

SSA Construction

- In the worst case each BB has a ϕ for each variable.
  - complexity $O(n^2)$
  - linear in practice
- Join criterion only says where to place ϕs. What are the correct arguments?
- Idea by Click 1995:
  - don't compute join sets explicitly
  - perform global value numbering during construction
  - place ϕs on the fly

Value Numbering

- Find congruent variables
- Reuse instead of recomputation
- Two computations are congruent if
  - identical operators w/o side-effects (includes constants)
  - congruent operands
- Normalize expressions. More congruence detectable.
- In $c = a + 1$ and $d = 1 + b$
  c and d are congruent if a and b are congruent

SSA Construction with VN (1)

Starting point:

- AST or BBG
- w.l.o.g. computations are in form $x = \tau(y, z)$

Proceeding:

- in each BB store valid value number VN(τ, y, z) for each variable
  - store value number: setVN(x, vn)
  - get value number: getVN(x)
- getVN(x) possibly inserts ϕs if VN not defined in current BB

Nice:

- ϕs are only inserted if variable is live

SSA Construction with VN (2)

For each $x = \tau(y, z)$ do:

- getVN(y), getVN(z)
- compute VN(τ, y, z)
- if value number is new insert
  $\mathrm{VN}(\tau, y, z) = \widehat{\tau}(\mathrm{getVN}(y), \mathrm{getVN}(z))$
  into the basic block
- store value number of x: setVN(x, VN(τ, y, z))

Nice:

- computation of VN implicitly performs CSE

SSA Construction with VN (3)

Details of getVN(v):

- if value v_i is valid for variable v in current BB return v_i
- else if BB has exactly one predecessor call getVN(v) there
- else (more predecessors):
    - call getVN(v) for all predecessors
    - let the values $v_1, v_2, \ldots$ be the results
    - insert $\text{VN}(\phi, v, v) = \phi(v_1, v_2, \ldots)$ into BB
    - avoid unnecessary ϕs
    - store new value of v: $\text{setVN}(v, \text{VN}(\phi, v, v))$
    - return this new value

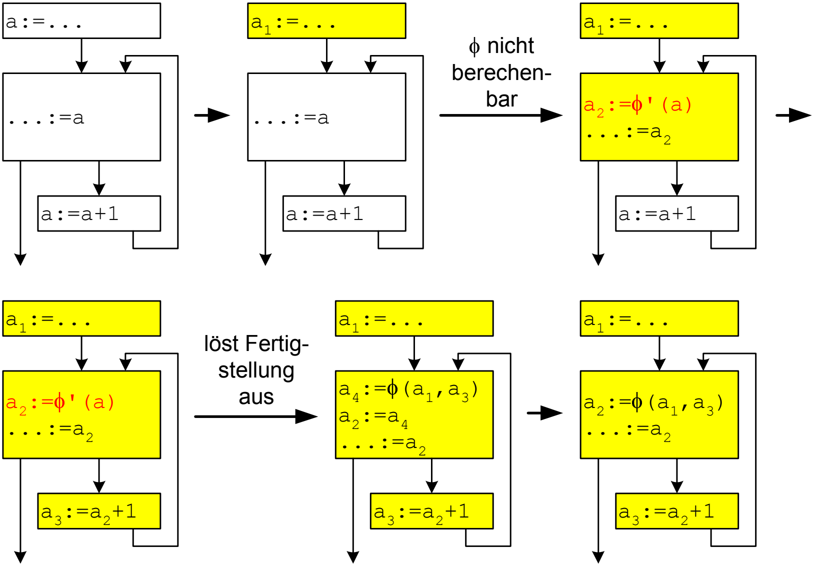
Unknown Predecessors: Problem

Observation: getVN might be undefined for some predecessors (loops!)

Solution: Two-stage approach

- mark a BB as ready when it is in SSA form
- if all predecessors are ready proceed as described
- else insert ϕ' and remember operand for finishing later
- when marking a BB as ready check successors and possibly finish them

Unknown Predecessors: Example



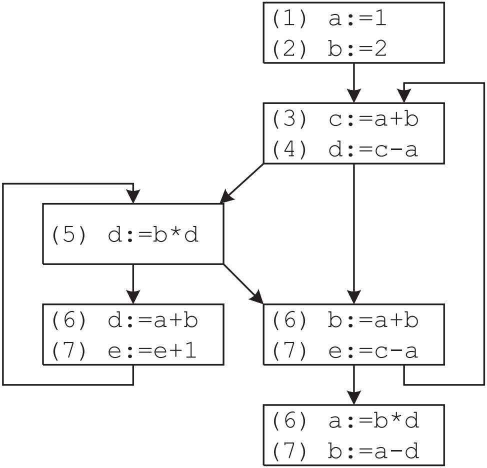
Unknown Predecessors: Consequences

Consequence: Do construction in control-flow order (as much as possible)

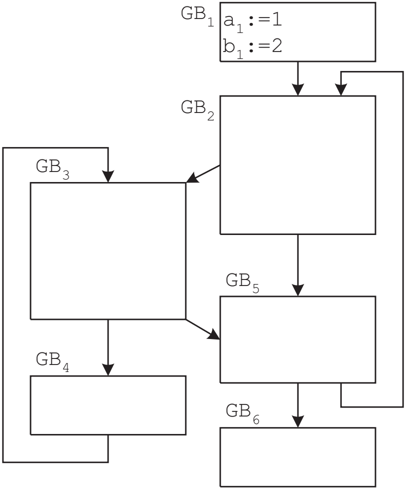
- Use post-order of a reverse depth-first search
- keeps number of ϕ's low
- dominating BBs are constructed before dominated BBs
- this makes the implicit CSE more effective

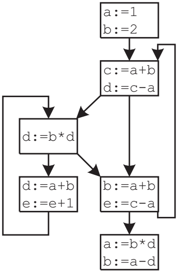
Larger Example

```
 (1)  a:=1;
 (2)  b:=2;
      while (true){
 (3)    c:=a+b;
 (4)    if (d:=c-a)
 (5)      while (d:=b*d){
 (6)        d:=a+b;
 (7)        e:=e+1;
          }
 (8)    b:=a+b;
 (9)    if (e:=c-a)
          break;
      }
(10)  a:=b*d;
(11)  b:=a-d;
```



SSA Construction Block 1

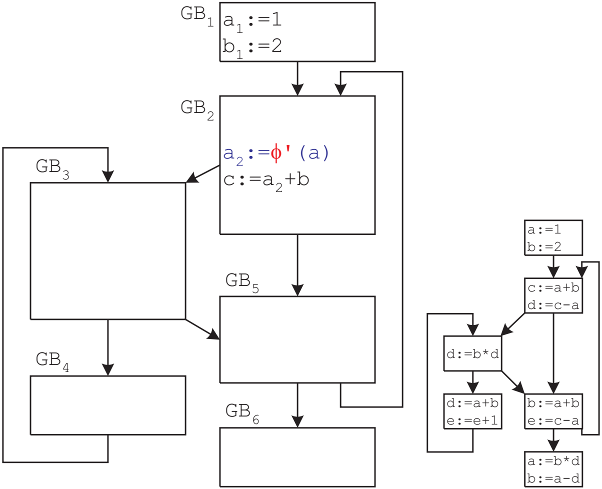




SSA Construction Block 2

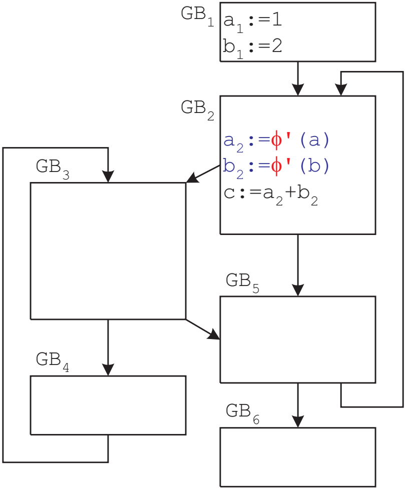
Get value number for *a* first places ϕ' for *a*

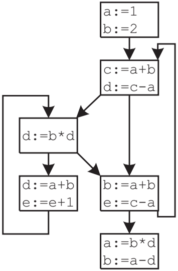
. . .



SSA Construction Block 2

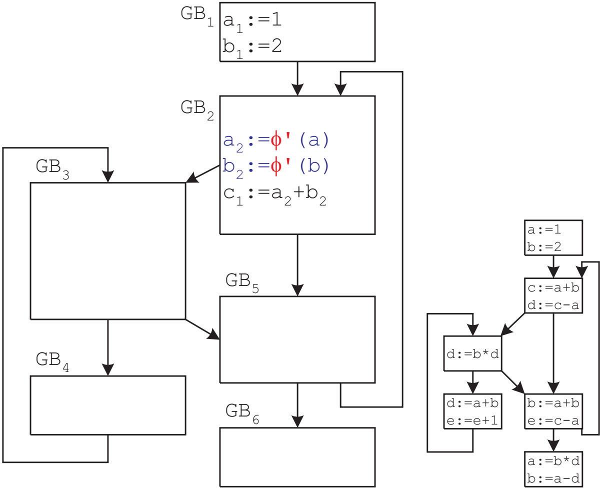
…then for *b*…





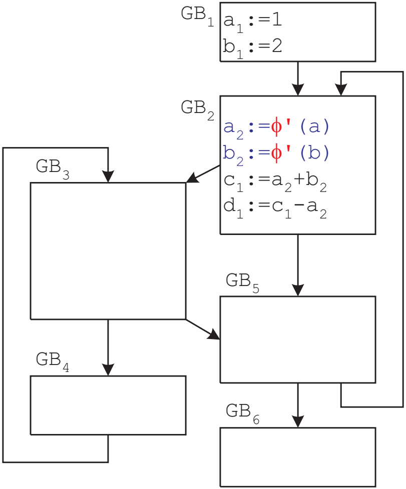
SSA Construction Block 2

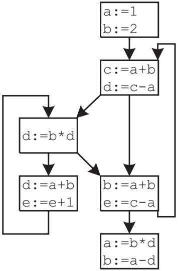
…and eventually a VN for *c*.



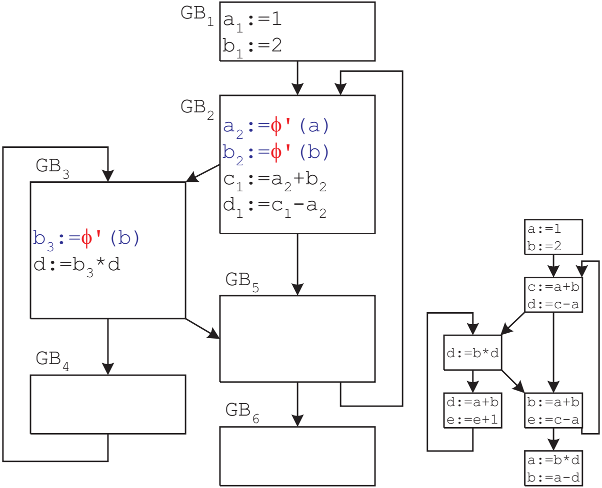
SSA Construction Block 2

Inserting $d := c - a$ works like normal value numbering.





SSA Construction Block 3



GB_1

a_1:=1

b_1:=2

GB_2

a_2:=ϕ'(a)

b_2:=ϕ'(b)

c_1:=a_2+b_2

d_1:=c_1-a_2

GB_3

b_3:=ϕ'(b)

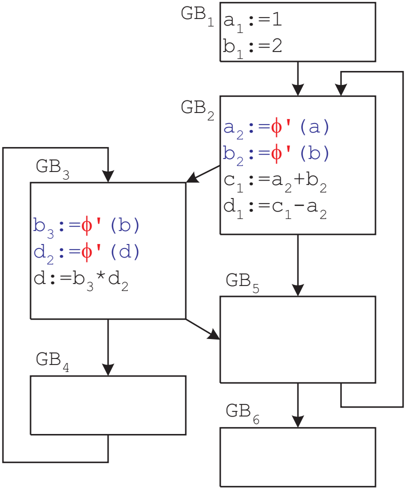
d:=b_3*d

GB_4

GB_5

GB_6

SSA Construction Block 3



GB$_1$

$a_1 := 1$

$b_1 := 2$

GB$_2$

$a_2 := \phi'(a)$

$b_2 := \phi'(b)$

$c_1 := a_2 + b_2$

$d_1 := c_1 - a_2$

GB$_3$

$b_3 := \phi'(b)$

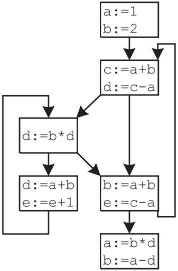
$d_2 := \phi'(d)$

$d := b_3 * d_2$

GB$_4$

GB$_5$

GB$_6$



```
a:=1
b:=2
```

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

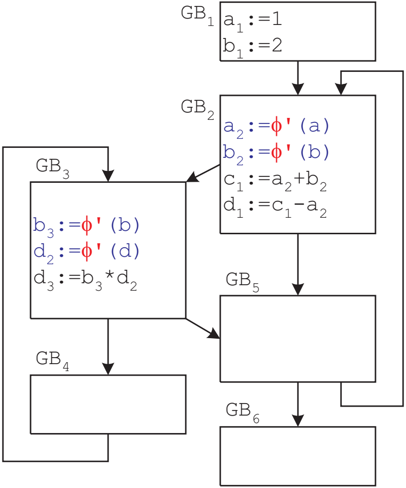
```
d:=b*d
```

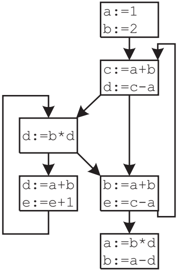
```
d:=a+b
e:=e+1
```

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

```
a:=b*d
b:=a-d
```

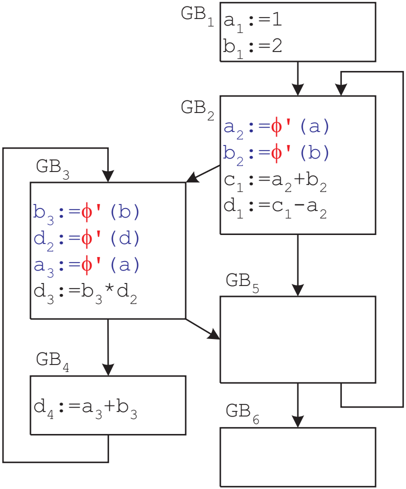
SSA Construction Block 3

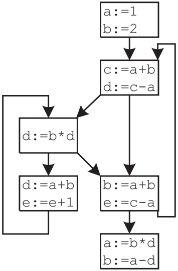




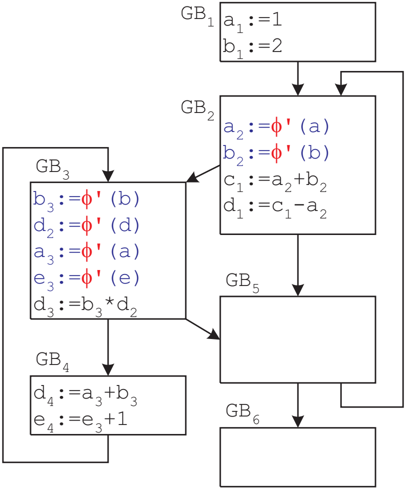
SSA Construction Block 4

Call to getVN(a) in 4 lead to recursive call getVN(a) in 3. This in turn produces a ϕ' for a in 3.





SSA Construction Block 4



GB_1

$a_1 := 1$

$b_1 := 2$

GB_2

$a_2 := \phi'(a)$

$b_2 := \phi'(b)$

$c_1 := a_2 + b_2$

$d_1 := c_1 - a_2$

GB_3

$b_3 := \phi'(b)$

$d_2 := \phi'(d)$

$a_3 := \phi'(a)$

$e_3 := \phi'(e)$

$d_3 := b_3 * d_2$

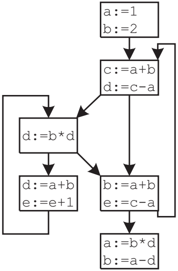
GB_4

$d_4 := a_3 + b_3$

$e_4 := e_3 + 1$

GB_5

GB_6



```
a:=1
b:=2
```

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

```
d:=b*d
```

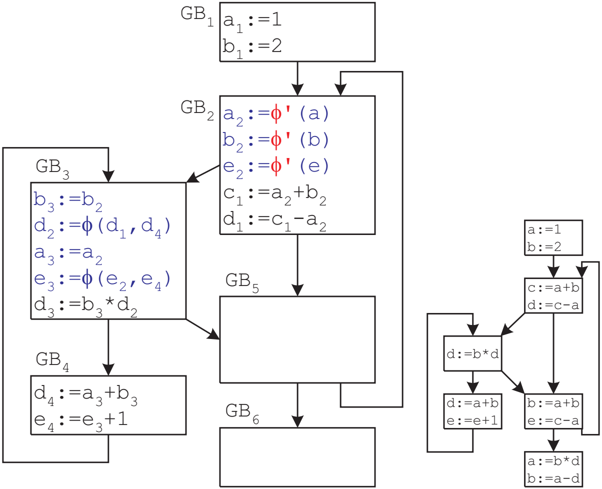
```
d:=a+b
e:=e+1
```

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

```
a:=b*d
b:=a-d
```

SSA Construction Block 4

All predecessors of 3 are now in SSA form: ϕs are placed. In block 2 a ϕ' is recursively placed for *e*.



GB$_1$

$a_1 := 1$

$b_1 := 2$

GB$_2$

$a_2 := \phi'(a)$

$b_2 := \phi'(b)$

$e_2 := \phi'(e)$

$c_1 := a_2 + b_2$

$d_1 := c_1 - a_2$

GB$_3$

$b_3 := b_2$

$d_2 := \phi(d_1, d_4)$

$a_3 := a_2$

$e_3 := \phi(e_2, e_4)$

$d_3 := b_3 * d_2$

GB$_4$

$d_4 := a_3 + b_3$

$e_4 := e_3 + 1$

GB$_5$

GB$_6$

a:=1
b:=2

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

d:=b*d

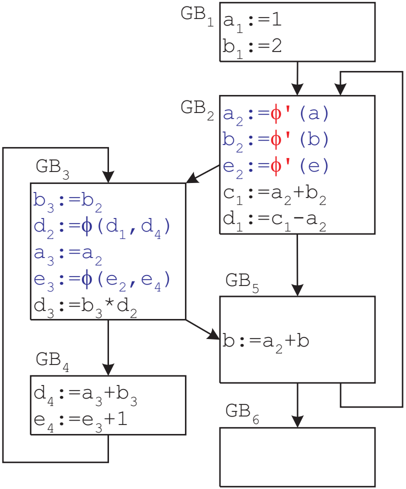
d:=a+b
e:=e+1

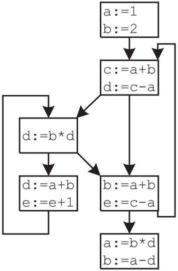
b:=a+b
e:=c-a

a:=b*d
b:=a-d

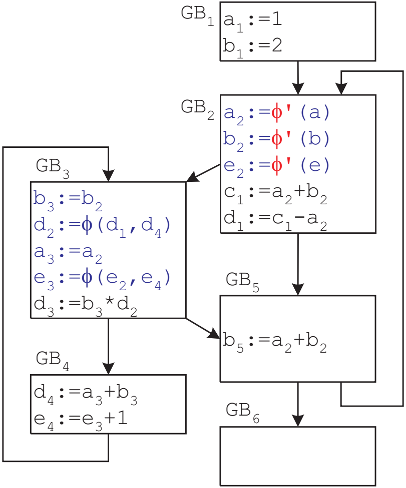
SSA Construction Block 5

getVN(a) in 5 recognizes copies, finds unique definition: no ϕ is necessary





SSA Construction Block 5



GB_1

$a_1 := 1$

$b_1 := 2$

GB_2

$a_2 := \phi'(a)$

$b_2 := \phi'(b)$

$e_2 := \phi'(e)$

$c_1 := a_2 + b_2$

$d_1 := c_1 - a_2$

GB_3

$b_3 := b_2$

$d_2 := \phi(d_1, d_4)$

$a_3 := a_2$

$e_3 := \phi(e_2, e_4)$

$d_3 := b_3 * d_2$

GB_4

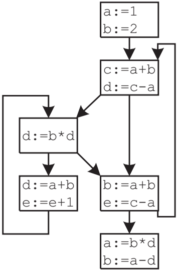
$d_4 := a_3 + b_3$

$e_4 := e_3 + 1$

GB_5

$b_5 := a_2 + b_2$

GB_6



a:=1
b:=2

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

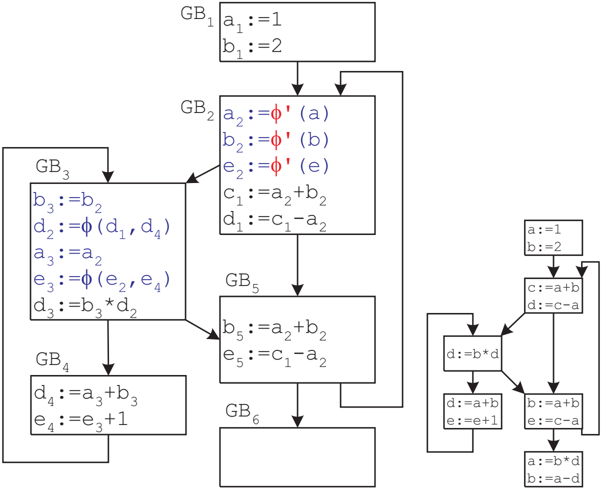
d:=b*d

d:=a+b
e:=e+1

b:=a+b
e:=c-a

a:=b*d
b:=a-d

SSA Construction Block 5



GB_1

$a_1 := 1$

$b_1 := 2$

GB_2

$a_2 := \phi'(a)$

$b_2 := \phi'(b)$

$e_2 := \phi'(e)$

$c_1 := a_2 + b_2$

$d_1 := c_1 - a_2$

GB_3

$b_3 := b_2$

$d_2 := \phi(d_1, d_4)$

$a_3 := a_2$

$e_3 := \phi(e_2, e_4)$

$d_3 := b_3 * d_2$

GB_4

$d_4 := a_3 + b_3$

$e_4 := e_3 + 1$

GB_5

$b_5 := a_2 + b_2$

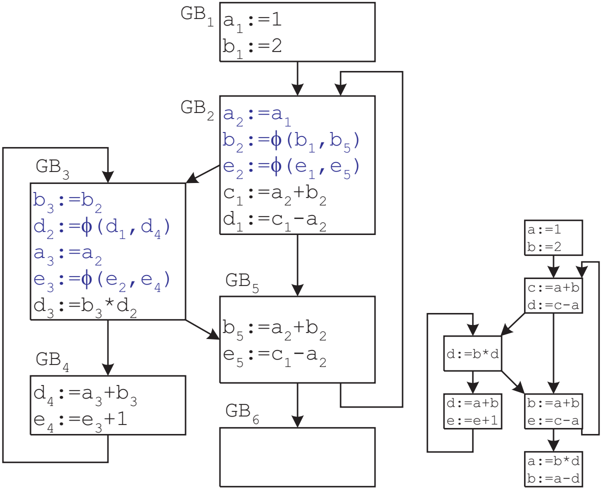
$e_5 := c_1 - a_2$

GB_6

SSA Construction Block 5

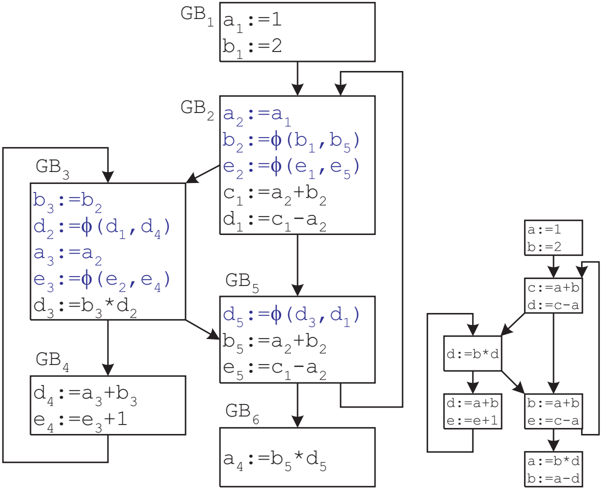
All predecesors of 2 are now in SSA form: ϕs are placed.

Algorithm recognices: *e* is uninitialized! Insert undefined value e_1



SSA Construction Block 6

Recursive call to getVN(d) in 5 places complete ϕ function d_5



GB_1

$a_1 := 1$
$b_1 := 2$

GB_2

$a_2 := a_1$
$b_2 := \phi(b_1, b_5)$
$e_2 := \phi(e_1, e_5)$
$c_1 := a_2 + b_2$
$d_1 := c_1 - a_2$

GB_3

$b_3 := b_2$
$d_2 := \phi(d_1, d_4)$
$a_3 := a_2$
$e_3 := \phi(e_2, e_4)$
$d_3 := b_3 * d_2$

GB_4

$d_4 := a_3 + b_3$
$e_4 := e_3 + 1$

GB_5

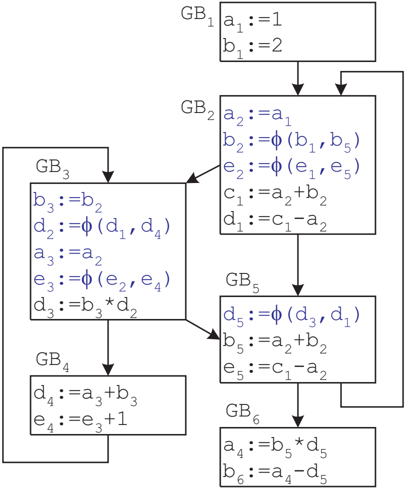
$d_5 := \phi(d_3, d_1)$
$b_5 := a_2 + b_2$
$e_5 := c_1 - a_2$

GB_6

$a_4 := b_5 * d_5$

```
a:=1
b:=2

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

d:=b*d

d:=a+b
e:=e+1

b:=a+b
e:=c-a

a:=b*d
b:=a-d
```

SSA Construction Block 6



GB_1

$a_1 := 1$
$b_1 := 2$

GB_2

$a_2 := a_1$
$b_2 := \phi(b_1, b_5)$
$e_2 := \phi(e_1, e_5)$
$c_1 := a_2 + b_2$
$d_1 := c_1 - a_2$

GB_3

$b_3 := b_2$
$d_2 := \phi(d_1, d_4)$
$a_3 := a_2$
$e_3 := \phi(e_2, e_4)$
$d_3 := b_3 * d_2$

GB_4

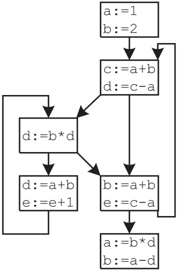
$d_4 := a_3 + b_3$
$e_4 := e_3 + 1$

GB_5

$d_5 := \phi(d_3, d_1)$
$b_5 := a_2 + b_2$
$e_5 := c_1 - a_2$

GB_6

$a_4 := b_5 * d_5$
$b_6 := a_4 - d_5$



```
a:=1
b:=2
```

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

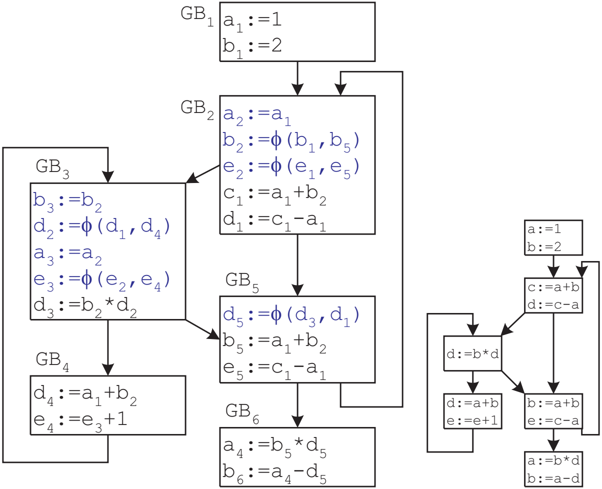
```
d:=b*d
```

```
d:=a+b
e:=e+1
```

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

```
a:=b*d
b:=a-d
```

Optimization: Copy Propagation



GB_1

$a_1 := 1$
$b_1 := 2$

GB_2

$a_2 := a_1$
$b_2 := \phi(b_1, b_5)$
$e_2 := \phi(e_1, e_5)$
$c_1 := a_1 + b_2$
$d_1 := c_1 - a_1$

GB_3

$b_3 := b_2$
$d_2 := \phi(d_1, d_4)$
$a_3 := a_2$
$e_3 := \phi(e_2, e_4)$
$d_3 := b_2 * d_2$

GB_4

$d_4 := a_1 + b_2$
$e_4 := e_3 + 1$

GB_5

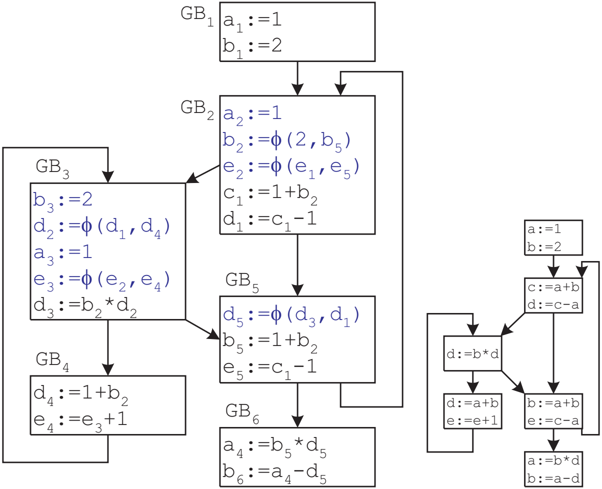
$d_5 := \phi(d_3, d_1)$
$b_5 := a_1 + b_2$
$e_5 := c_1 - a_1$

GB_6

$a_4 := b_5 * d_5$
$b_6 := a_4 - d_5$

```
a:=1
b:=2

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

d:=b*d

d:=a+b
e:=e+1

b:=a+b
e:=c-a

a:=b*d
b:=a-d
```

Optimization: Constant Propagation



GB_1

$a_1 := 1$

$b_1 := 2$

GB_2

$a_2 := 1$

$b_2 := \phi(2, b_5)$

$e_2 := \phi(e_1, e_5)$

$c_1 := 1 + b_2$

$d_1 := c_1 - 1$

GB_3

$b_3 := 2$

$d_2 := \phi(d_1, d_4)$

$a_3 := 1$

$e_3 := \phi(e_2, e_4)$

$d_3 := b_2 * d_2$

GB_4

$d_4 := 1 + b_2$

$e_4 := e_3 + 1$

GB_5

$d_5 := \phi(d_3, d_1)$

$b_5 := 1 + b_2$

$e_5 := c_1 - 1$

GB_6

$a_4 := b_5 * d_5$

$b_6 := a_4 - d_5$

a:=1
b:=2

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

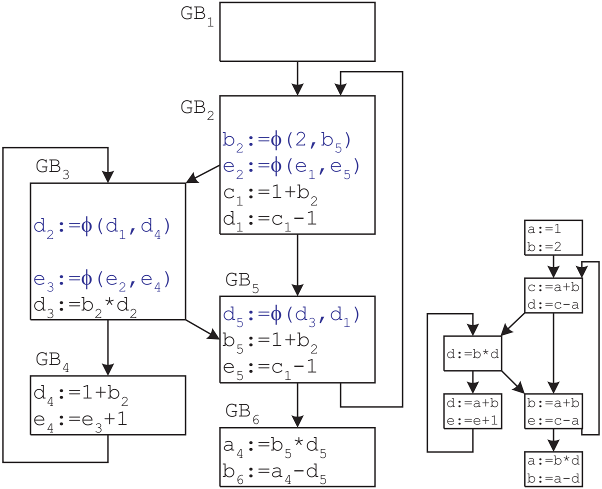
d:=b*d

d:=a+b
e:=e+1

b:=a+b
e:=c-a

a:=b*d
b:=a-d

Optimization: Dead Code Elimination



GB_1

GB_2

$b_2 := \phi(2, b_5)$
$e_2 := \phi(e_1, e_5)$
$c_1 := 1 + b_2$
$d_1 := c_1 - 1$

GB_3

$d_2 := \phi(d_1, d_4)$

$e_3 := \phi(e_2, e_4)$
$d_3 := b_2 * d_2$

GB_4

$d_4 := 1 + b_2$
$e_4 := e_3 + 1$

GB_5

$d_5 := \phi(d_3, d_1)$
$b_5 := 1 + b_2$
$e_5 := c_1 - 1$

GB_6

$a_4 := b_5 * d_5$
$b_6 := a_4 - d_5$

```
a:=1
b:=2
```

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

```
d:=b*d
```

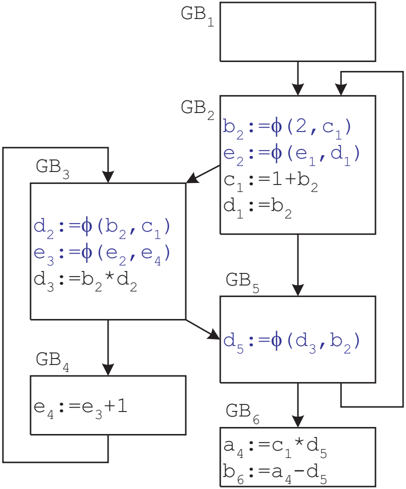
```
d:=a+b
e:=e+1
```

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

```
a:=b*d
b:=a-d
```

Further Optimizations

- common subexpressions
- reassociation
- evaluation of constant expressions
- copy propagation
- dead code elimination



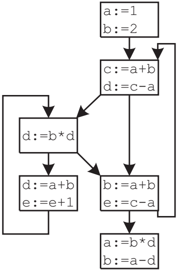
GB_2:
$b_2 := \phi(2, c_1)$
$e_2 := \phi(e_1, d_1)$
$c_1 := 1 + b_2$
$d_1 := b_2$

GB_3:
$d_2 := \phi(b_2, c_1)$
$e_3 := \phi(e_2, e_4)$
$d_3 := b_2 * d_2$

GB_4:
$e_4 := e_3 + 1$

GB_5:
$d_5 := \phi(d_3, b_2)$

GB_6:
$a_4 := c_1 * d_5$
$b_6 := a_4 - d_5$



a:=1
b:=2

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

d:=b*d

d:=a+b
e:=e+1

b:=a+b
e:=c-a

a:=b*d
b:=a-d

1. S. Muchnick: Advanced Compiler Design and Implementation
   (On IR issues and SSA)
2. C. Click et al.: His papers from 1995. Confer to DBLP
   (On practical SSA construction and an SSA-IR proposal)
3. R. Cytron et al.: An efficient method of computing SSA form
   (Original work on SSA. POPL 1989, similar article in TOPLAS 1991)
4. www.libfirm.org (optimizing graph-based SSA IR)