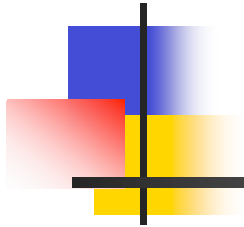
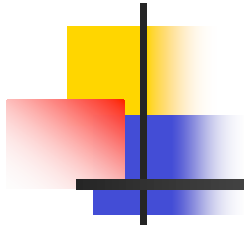


# Static Single Assignment Form



Masataka Sassa  
(Tokyo Institute of Technology)



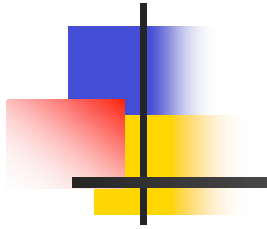
# Background

**Static single assignment (SSA) form** facilitates compiler optimizations.

## Outline

1. SSA form
2. Translation into SSA form
3. Back translation from SSA form
4. SSA form optimizations

# 1 Static single assignment form (SSA form)



# Static single assignment (SSA) form

```
1: a = x + y
2: a = a + 3
3: b = x + y
```

(a) Normal (conventional) form (source program or internal form)

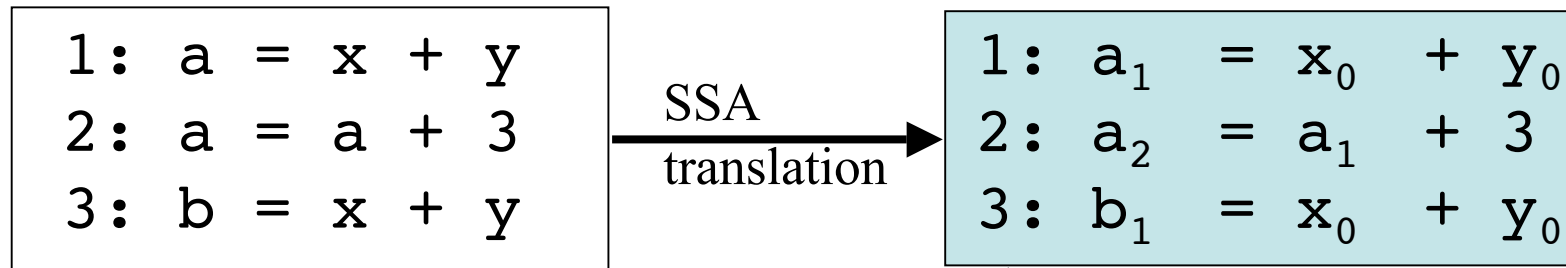
```
1: a1 = x0 + y0
2: a2 = a1 + 3
3: b1 = x0 + y0
```

(b) SSA form

SSA form is a recently proposed internal representation where each use of a variable has a single definition point.

Indices are attached to variables so that their definitions become unique.

# Optimization in static single assignment (SSA) form



(a) Normal form

(b) SSA form

Optimization in SSA form (common subexpression elimination)

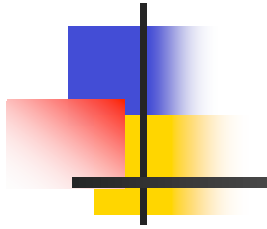


(c) After SSA form optimization

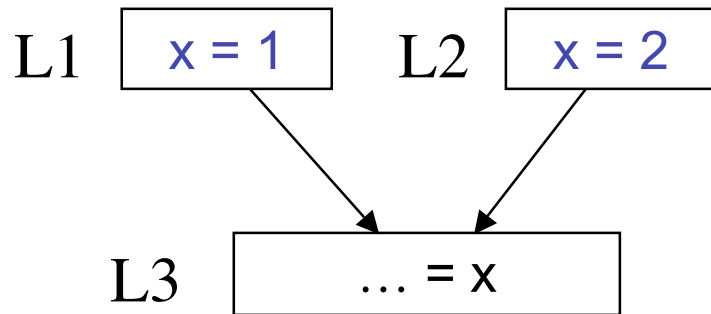
(d) Optimized normal form

SSA form is becoming increasingly popular in compilers, since it is suited for clear handling of dataflow analysis and optimization.

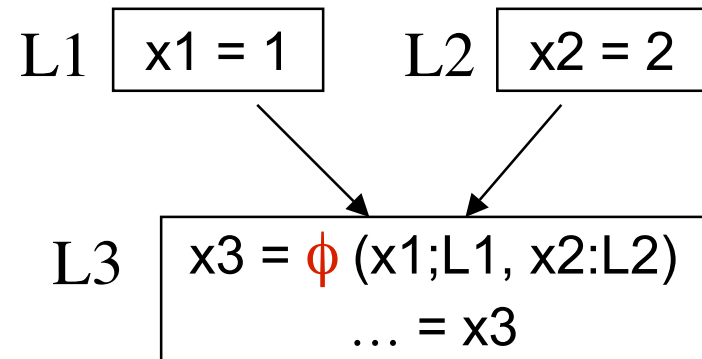
## 2 Translation into SSA form (SSA translation)



# Translation into SSA form (SSA translation)



(a) Normal form



(b) SSA form

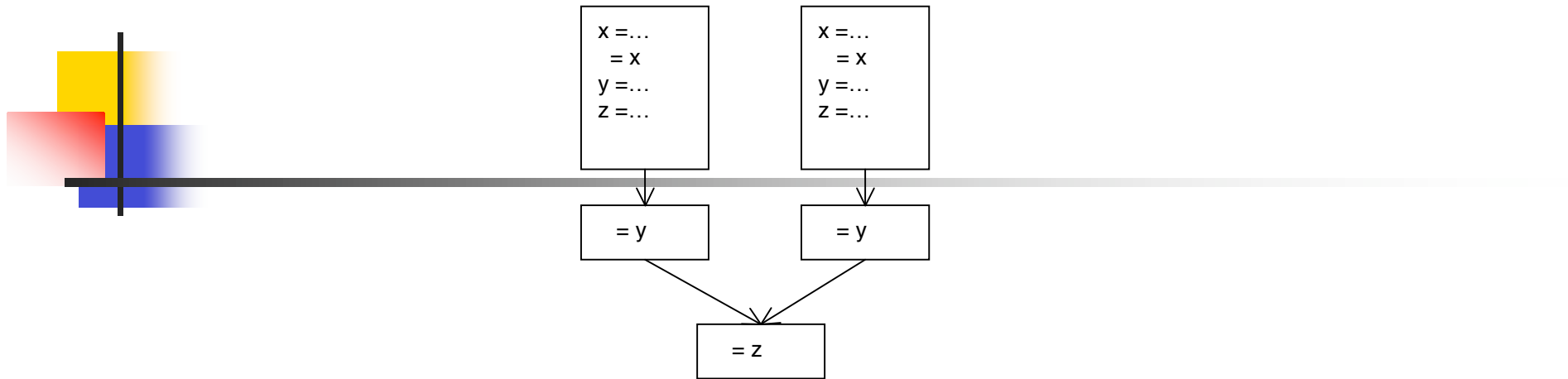


## Three variations have been proposed as the SSA form

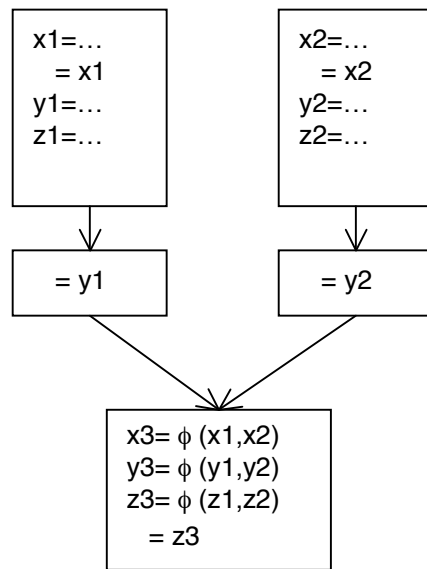
---

- minimal SSA [Cytron et al. 91]
- semi-pruned SSA [Briggs et al. 98]
- pruned SSA [Choi et al. 91]

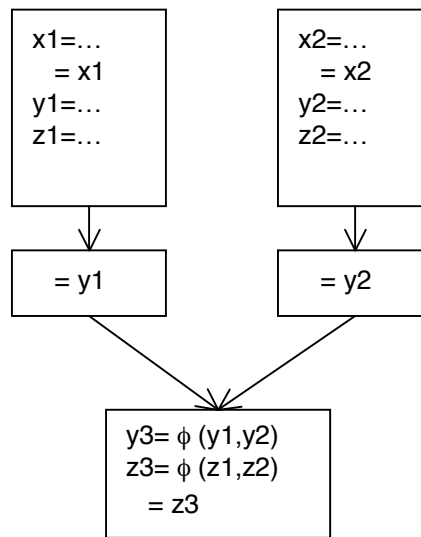
# Three variations of SSA form



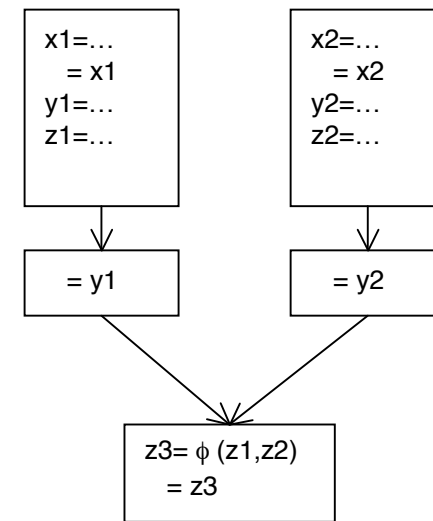
Normal form



Minimal SSA form



Semi-pruned SSA form



Pruned SSA form

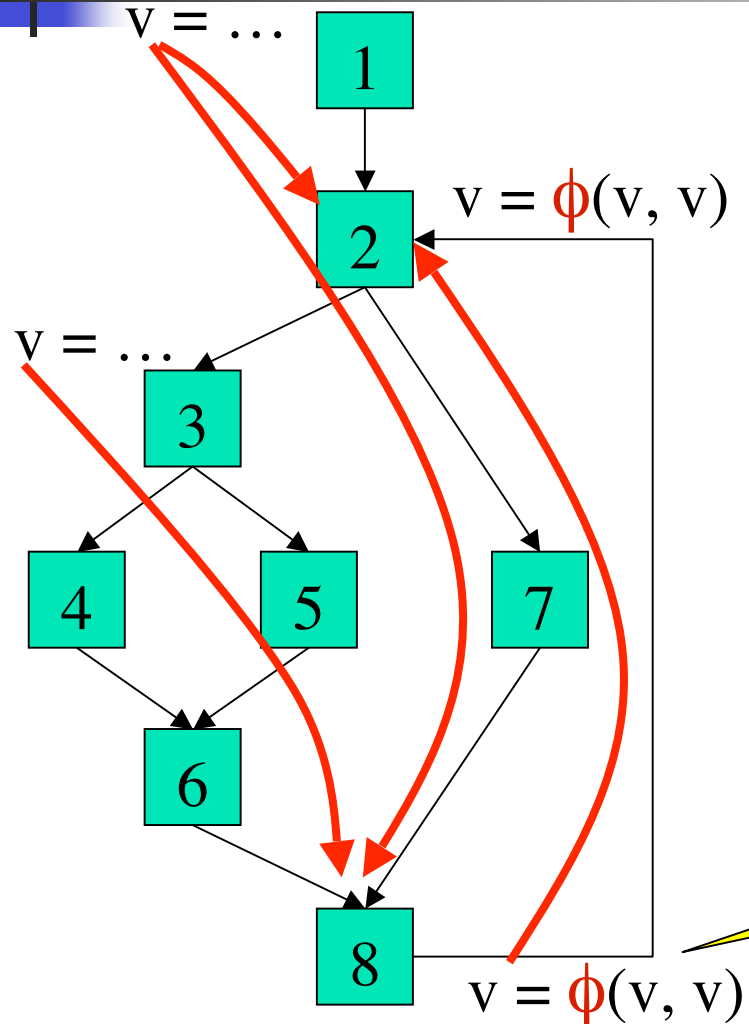


# Two major methods for inserting $\phi$ -functions

---

- (i) Method by Cytron et al.  
[Cytron et al. 91]
- (ii) Method by Sreedhar et al.  
[Sreedhar and Gao 95]

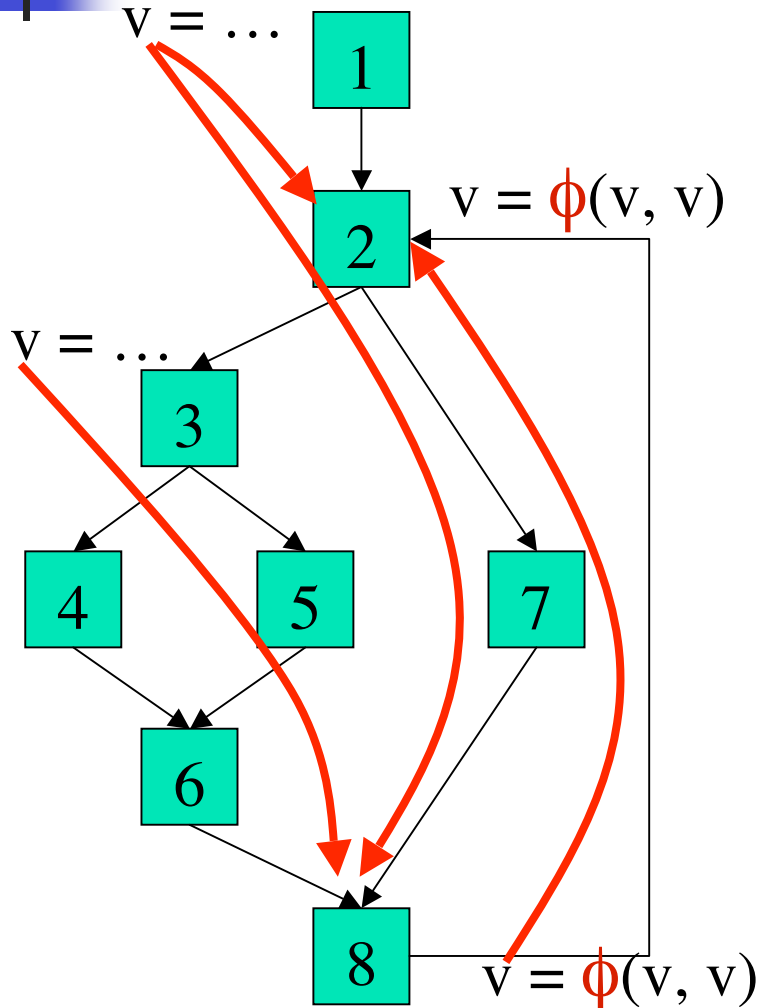
# SSA translation: inserting $\phi$ -functions



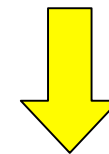
Outline:  
Insert  $\phi$ -functions to nodes  
which different definitions  
reach to.

New definition occurs  
where  $\phi$ -function is  
inserted.

# SSA translation: inserting $\phi$ -functions

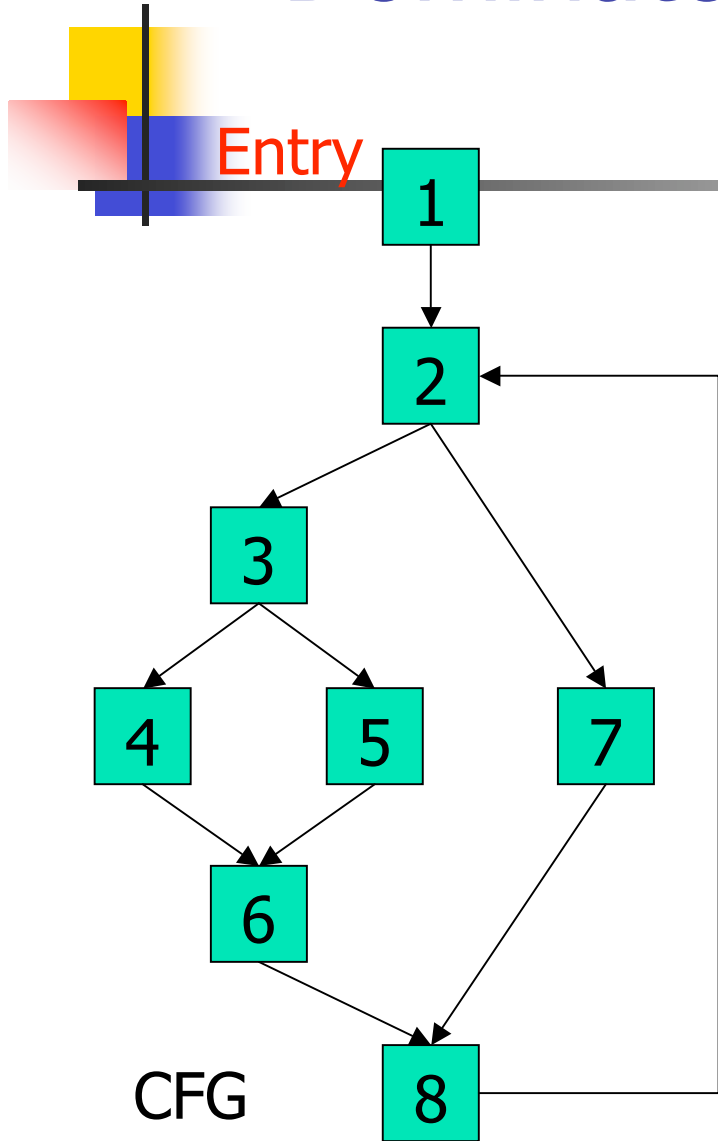


Nodes where  $\phi$ -functions are inserted:  
"dominance frontier" of node where variable  $v$  is defined.

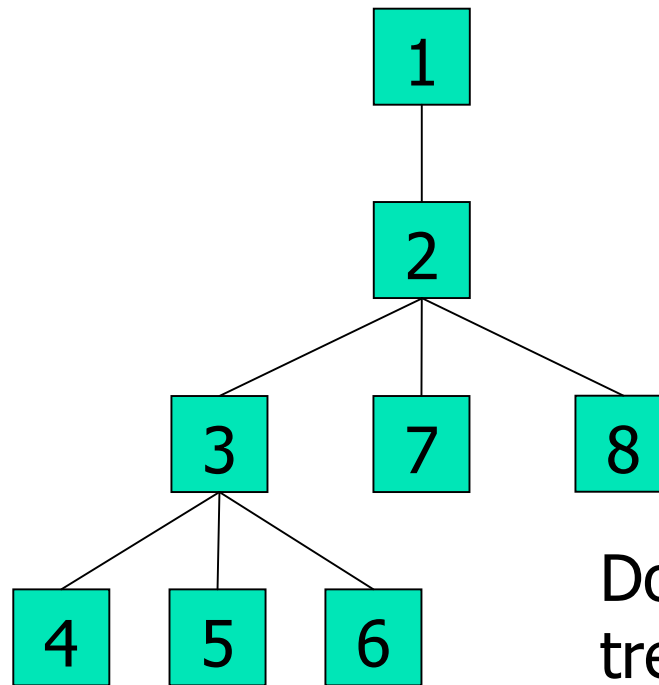


Insertion of  $\phi$ -functions:  
same as finding the dominance frontier

# Dominator tree

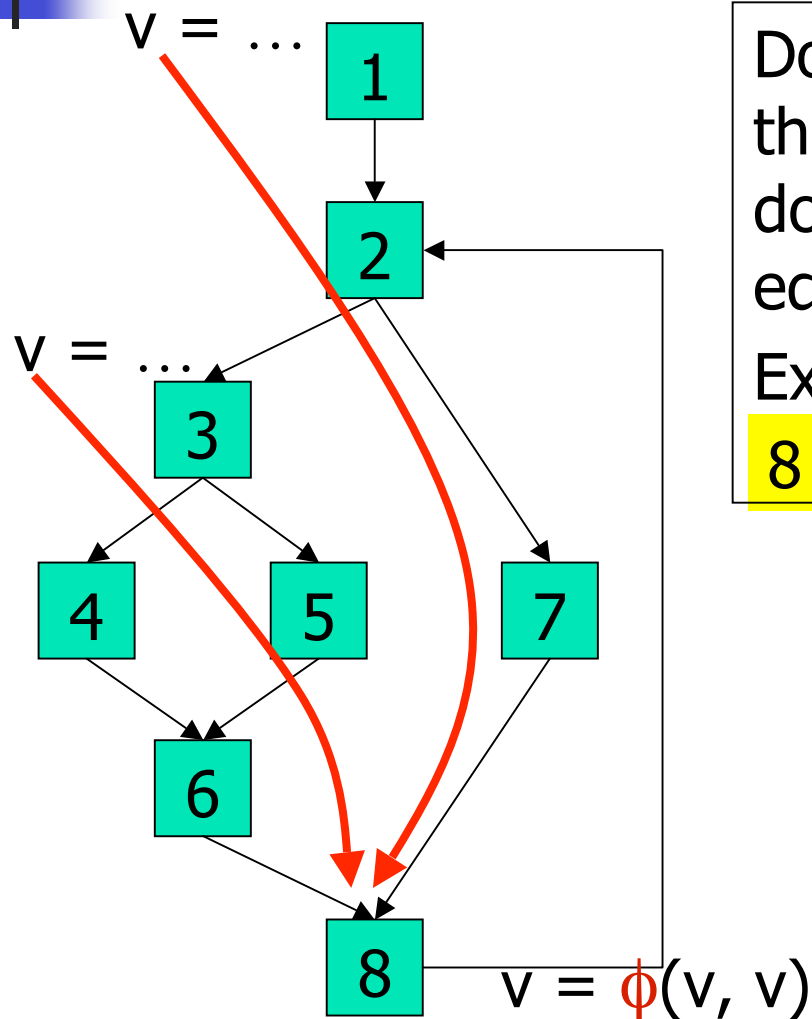


To go from 1 (Entry) to 3, always pass through 2 → 2 dominates 3



Dominator tree

# Dominance frontier

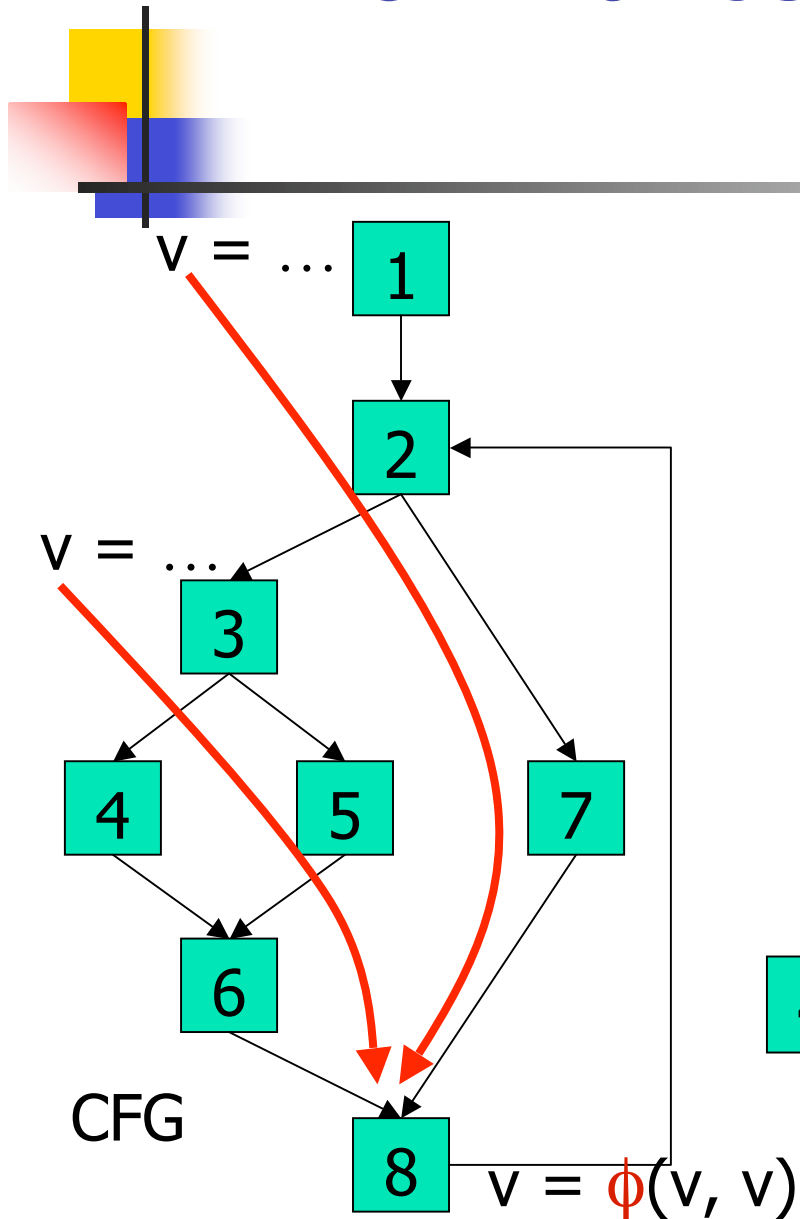


Dominance frontier of node X:  
the first node that X does not  
dominate, traversing the CFG  
edges starting from X.

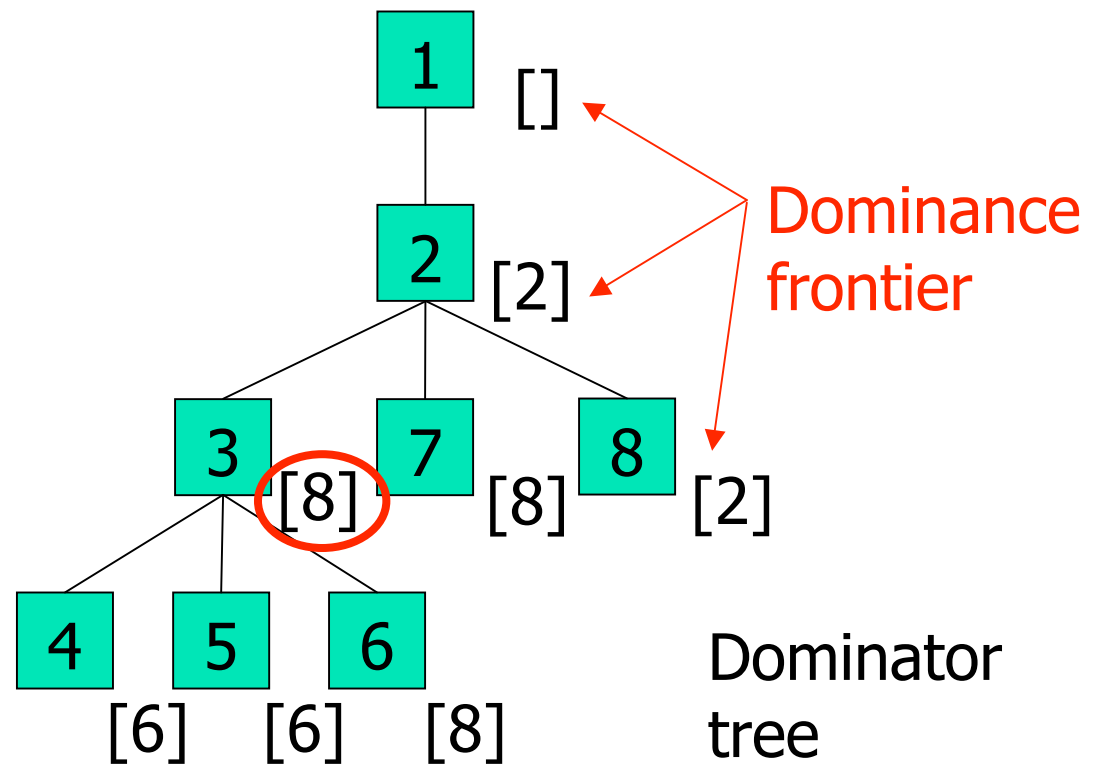
Example:

8 is the dominance frontier of 3

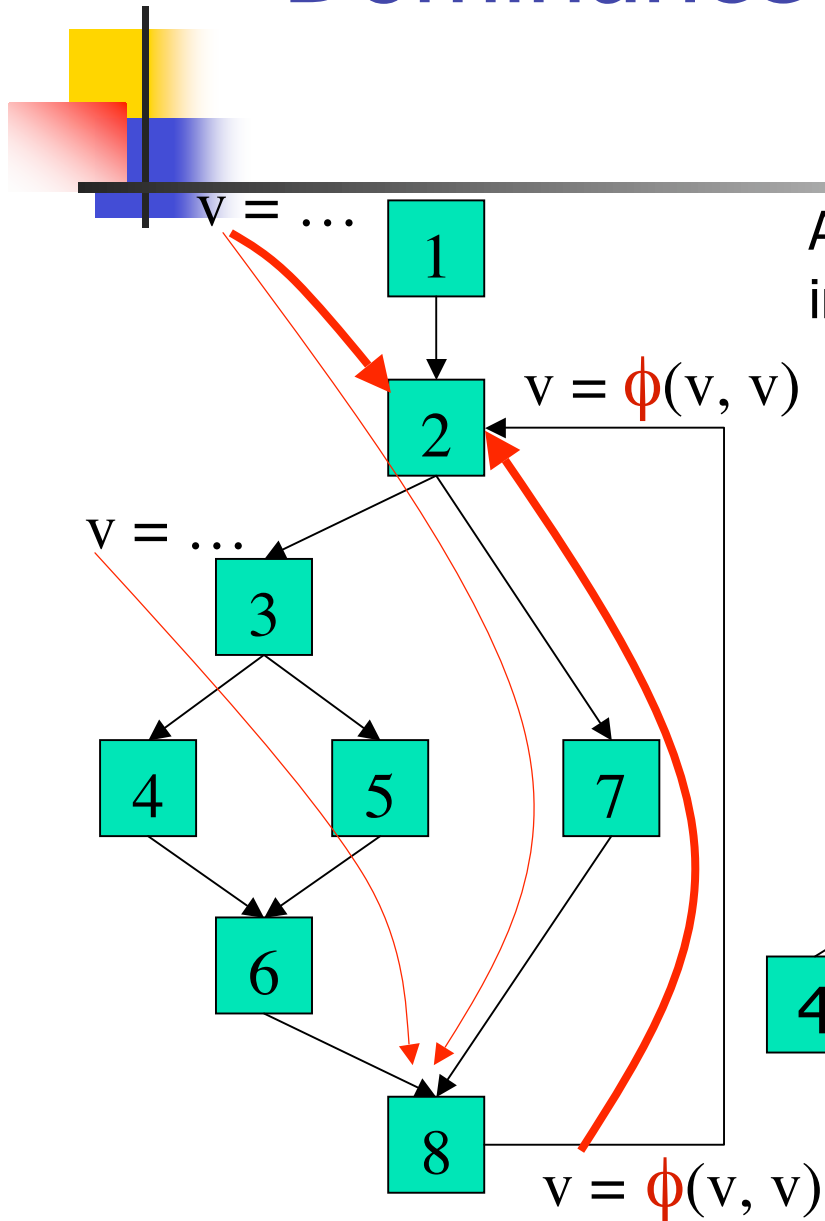
# Dominance frontier



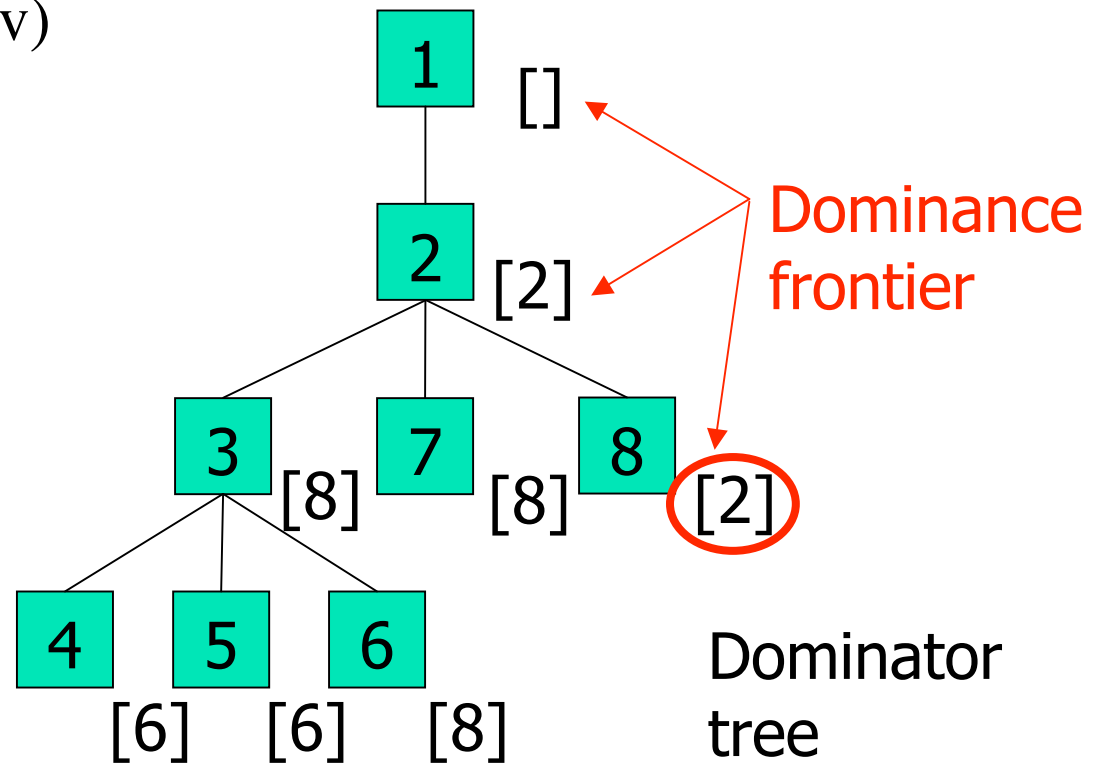
Insert  $\phi$  at the dominance frontier of 3.



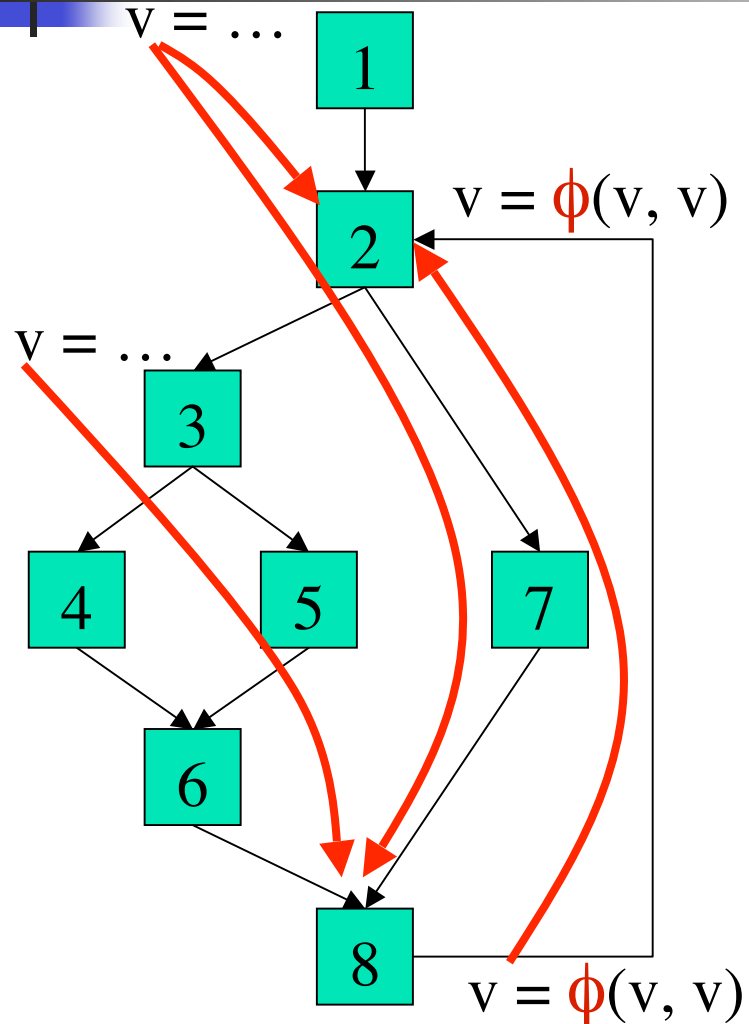
# Dominance frontier



As  $\phi$  is inserted in 8, another  $\phi$  should be inserted at the dominance frontier of 8.



# SSA translation: inserting $\phi$ -functions



The result of inserting  $\phi$ -functions

# Two major methods of SSA translation

## (i) Method by Cytron et al.

[Cytron et. al 91]

1. Compute the dominance frontier (DF) of all nodes in CFG
  - (before insertion phase of  $\phi$ -functions)
  - Compute DF using CFG and dominator tree
2. Insert  $\phi$ -functions
  - Insert  $\phi$ -functions at every dominance frontier of nodes where variables are defined (repeat if  $\phi$  is inserted).
3. Rename variables
  - Renames  $v$  to  $v_1, v_2, \dots$

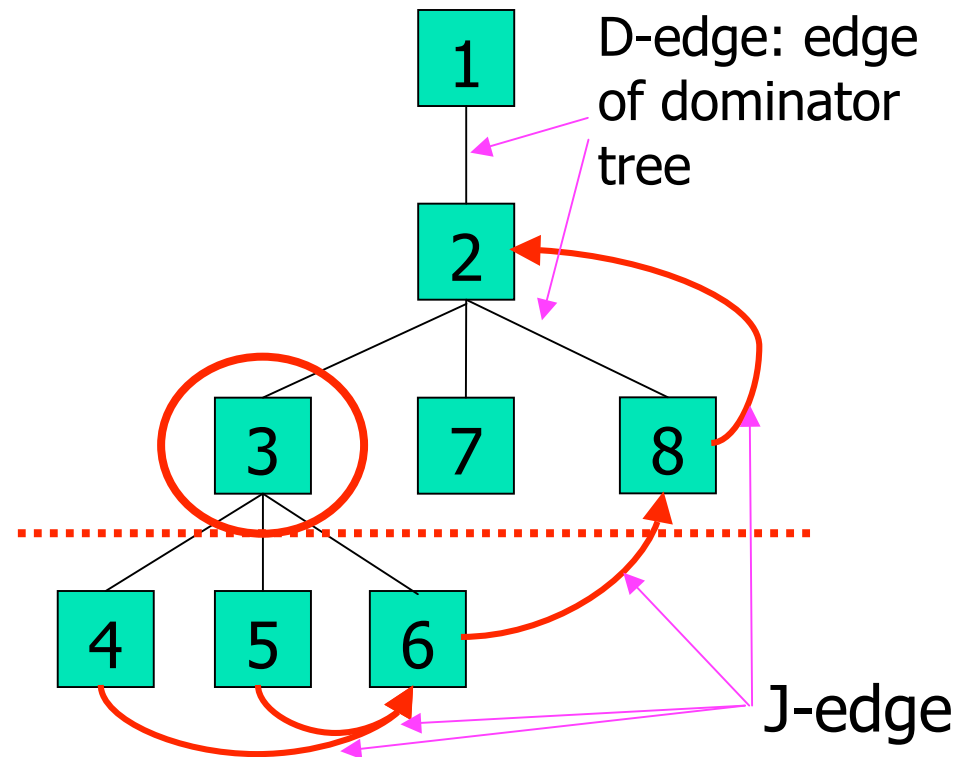
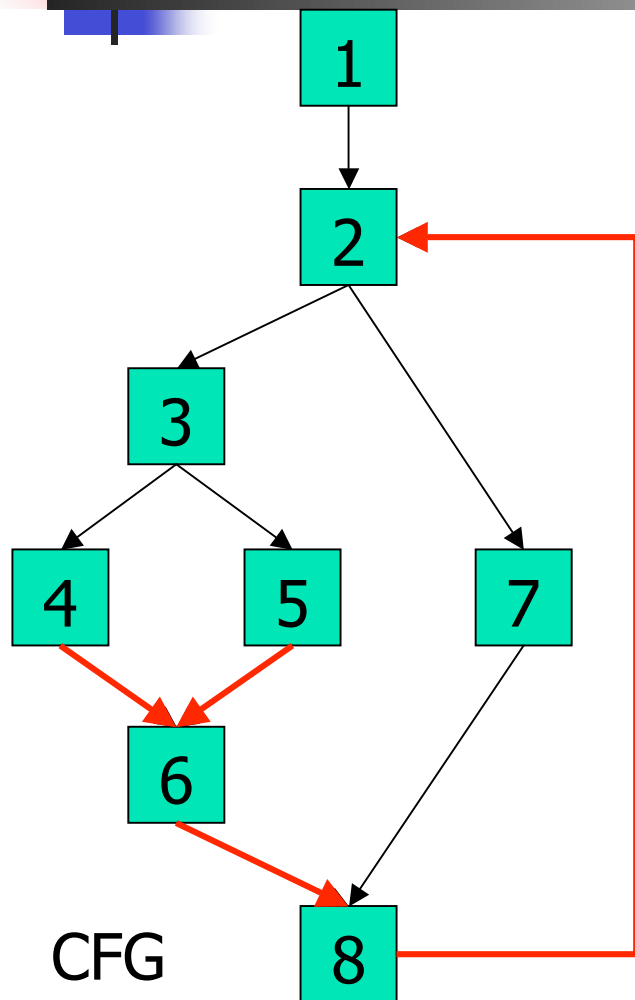


## (ii) Method by Sreedhar et al. [Sreedhar and Gao 95]

---

- An improvement of Cytron et al.'s method
- Use a data structure **DJ-graph**
- Computation of dominance frontier is made **during** the  $\phi$ -function insertion phase

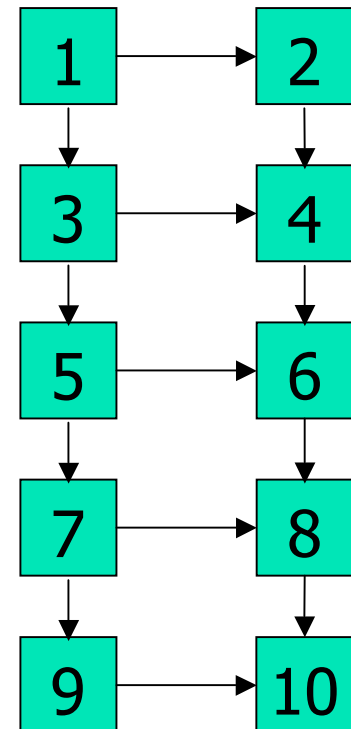
## (ii) Method by Sreedhar et al. [Sreedhar and Gao 95]



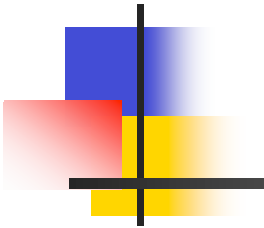
DJ-graph: Dominator tree +  $\alpha$

# Computational complexity of $\phi$ -function insertion

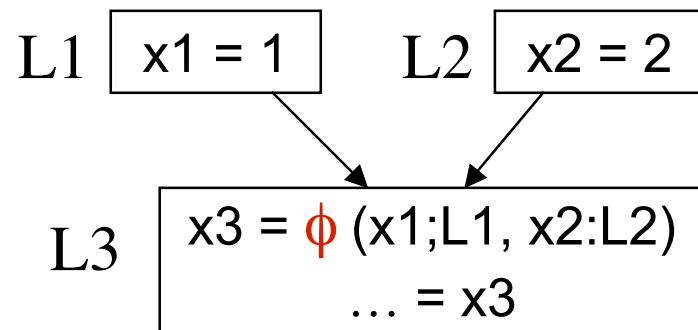
- Usual programs
  - The number of dominance frontier for each node is less than or equal to 2
  - Cytron: linear  $O(n)$  (wrt. the size of CFG)
  - Sreedhar: linear  $O(n)$
- Programs where the CFG is the ladder graph or nested loop
  - Cytron: quadratic  $O(n^2)$ 
    - The number of dominance frontier is large
  - Sreedhar: linear  $O(n)$
- Coins implements Cytron's method



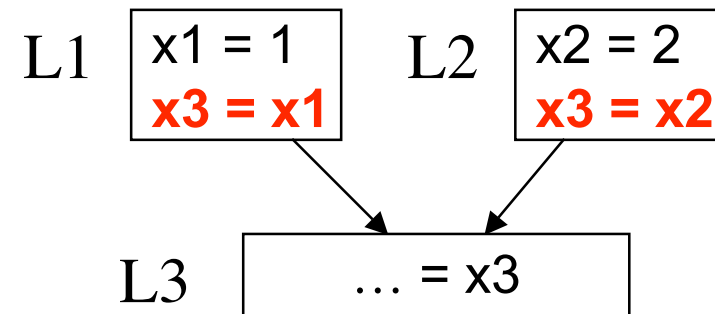
### 3 Back translation from SSA form (SSA back translation)



# Back translation from SSA form (SSA back translation)

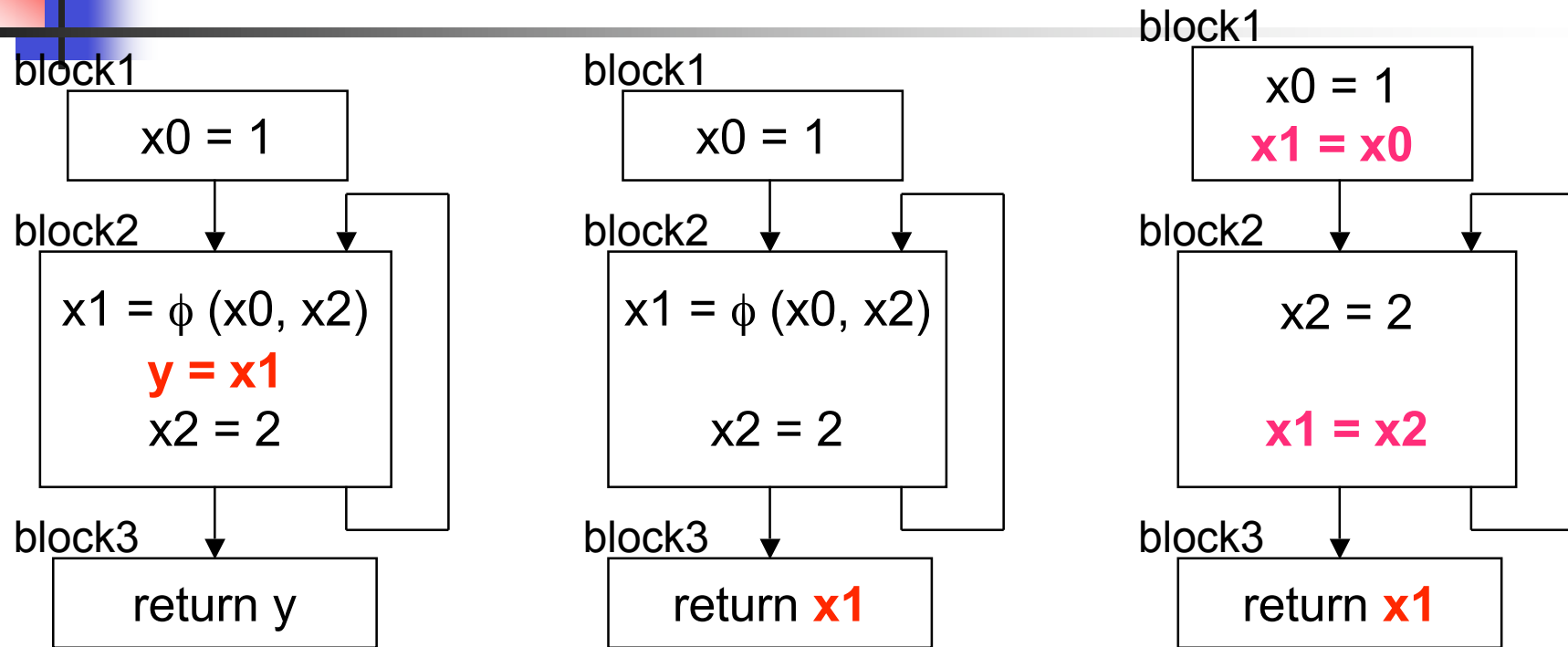


(a) SSA form



(b) Normal form

# Problems of naïve SSA back translation (Lost copy problem)



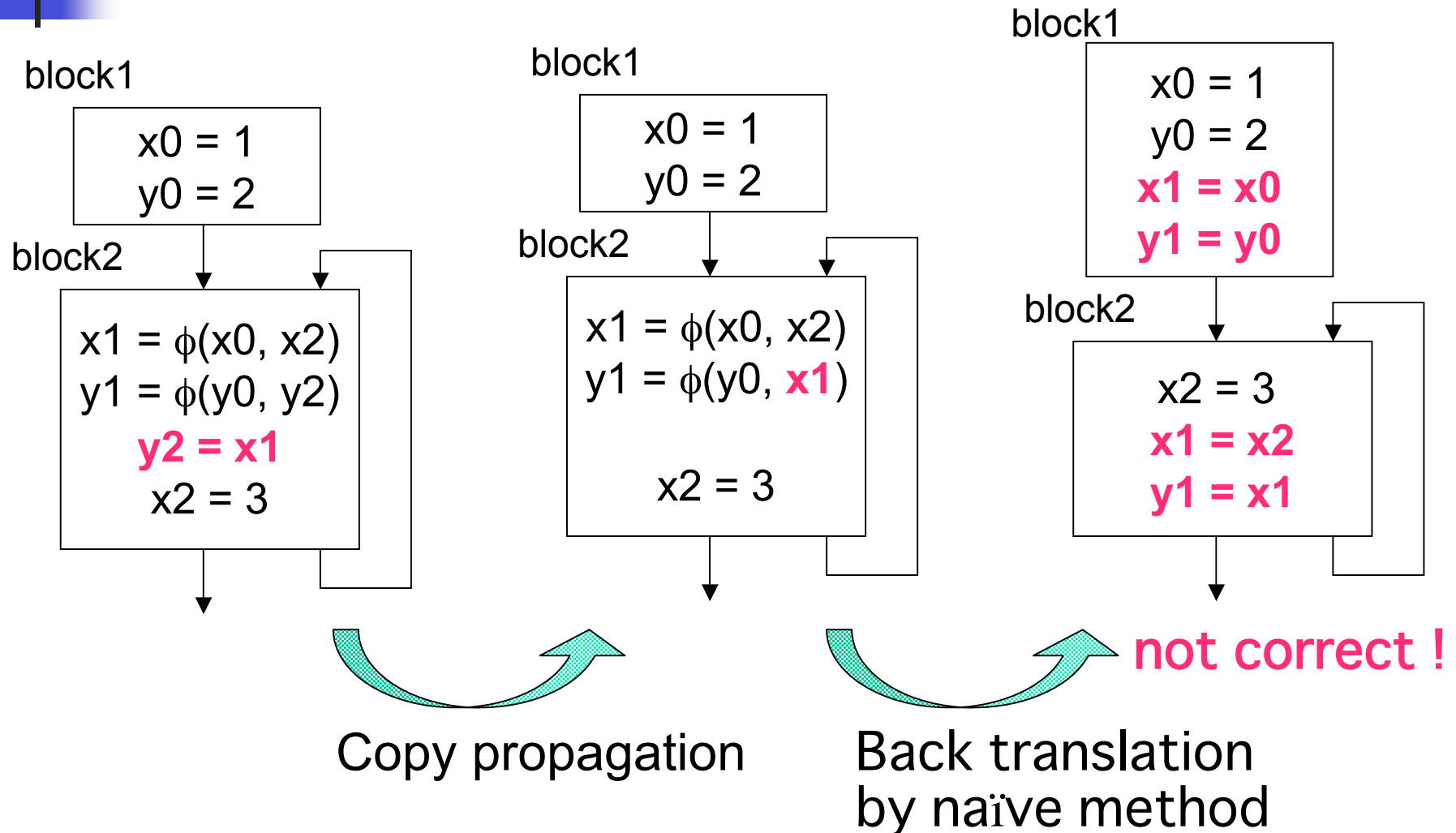
Copy propagation

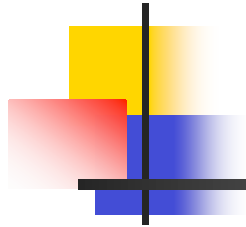
Back translation  
by naïve method

not correct !

# Problems of naïve SSA back translation

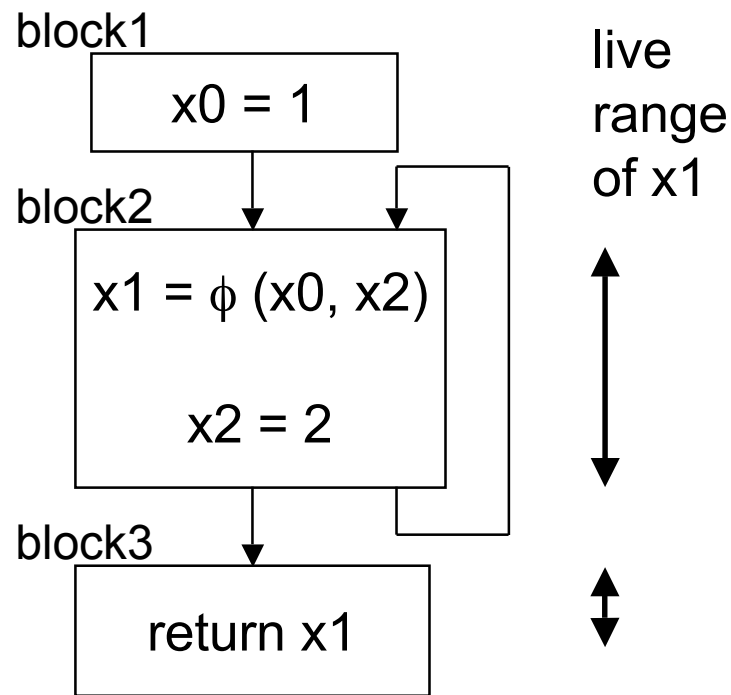
(Simple ordering problem:  
simultaneous assignments to  $\phi$ -functions)



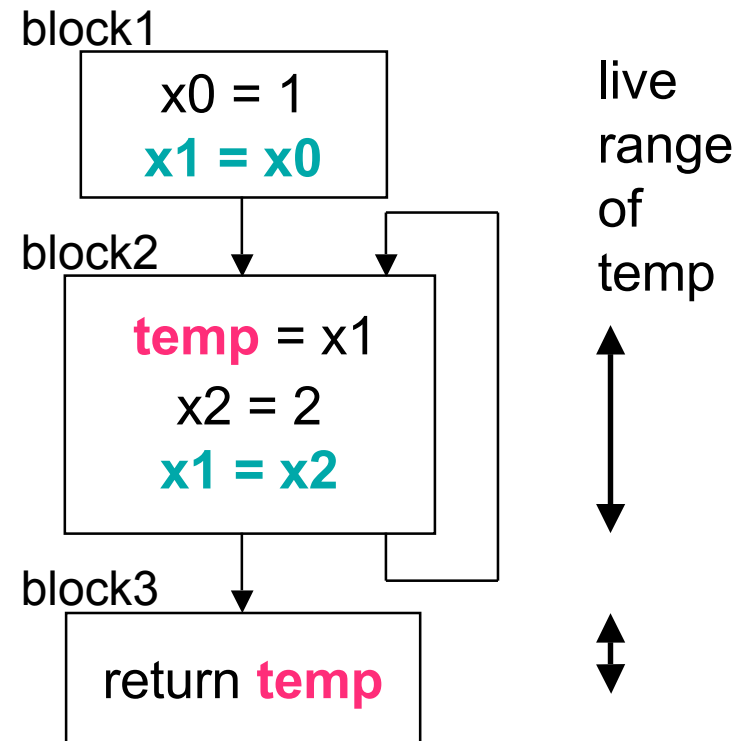


- To remedy these problems...
- SSA back translation algorithms by
- (i) Briggs et al. [1998]
    - Insert copy statements
  - (ii) Sreedhar et al. [1999]
    - Eliminate interference

# (i) SSA back translation algorithm by Briggs

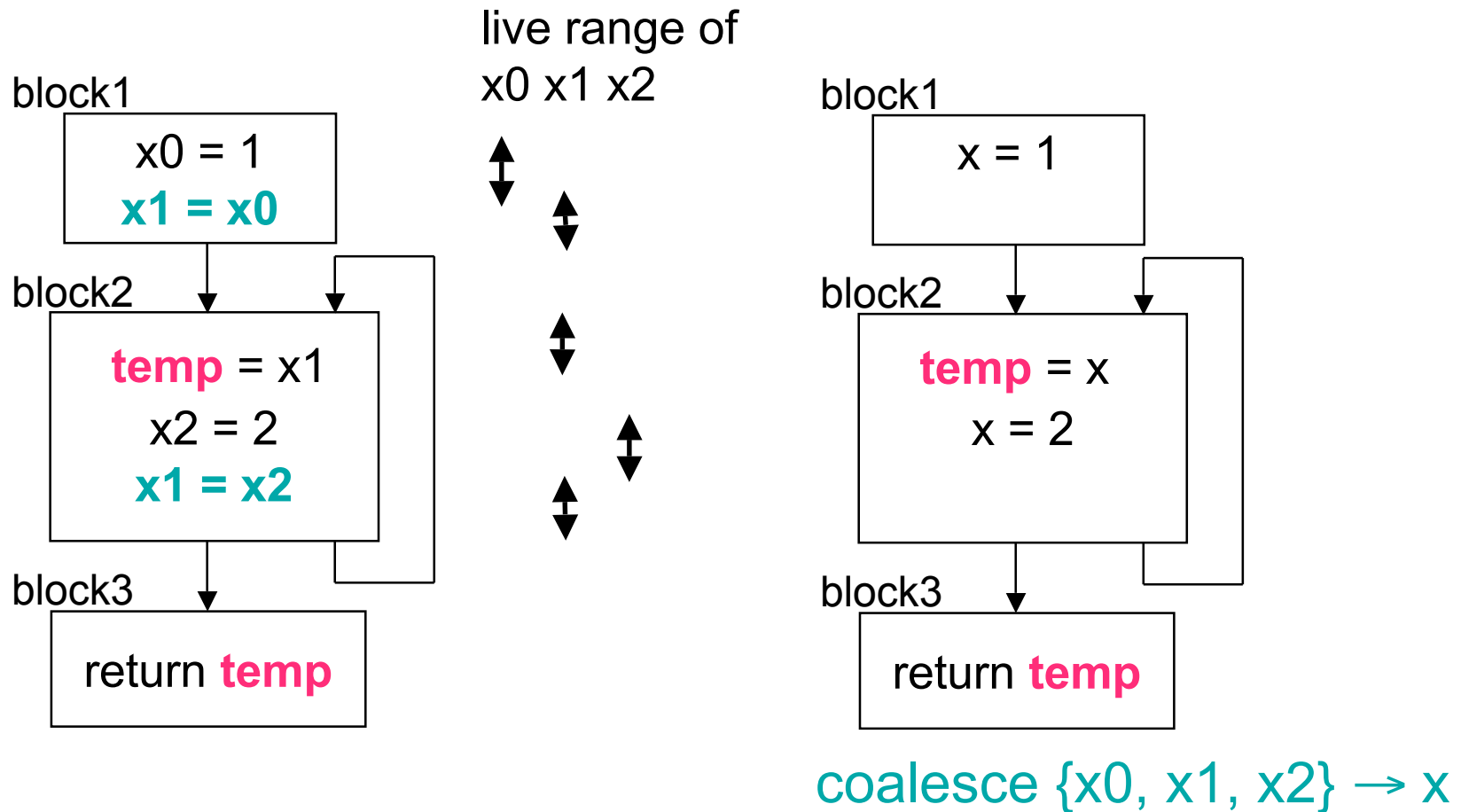


(a) SSA form



(b) normal form after back translation

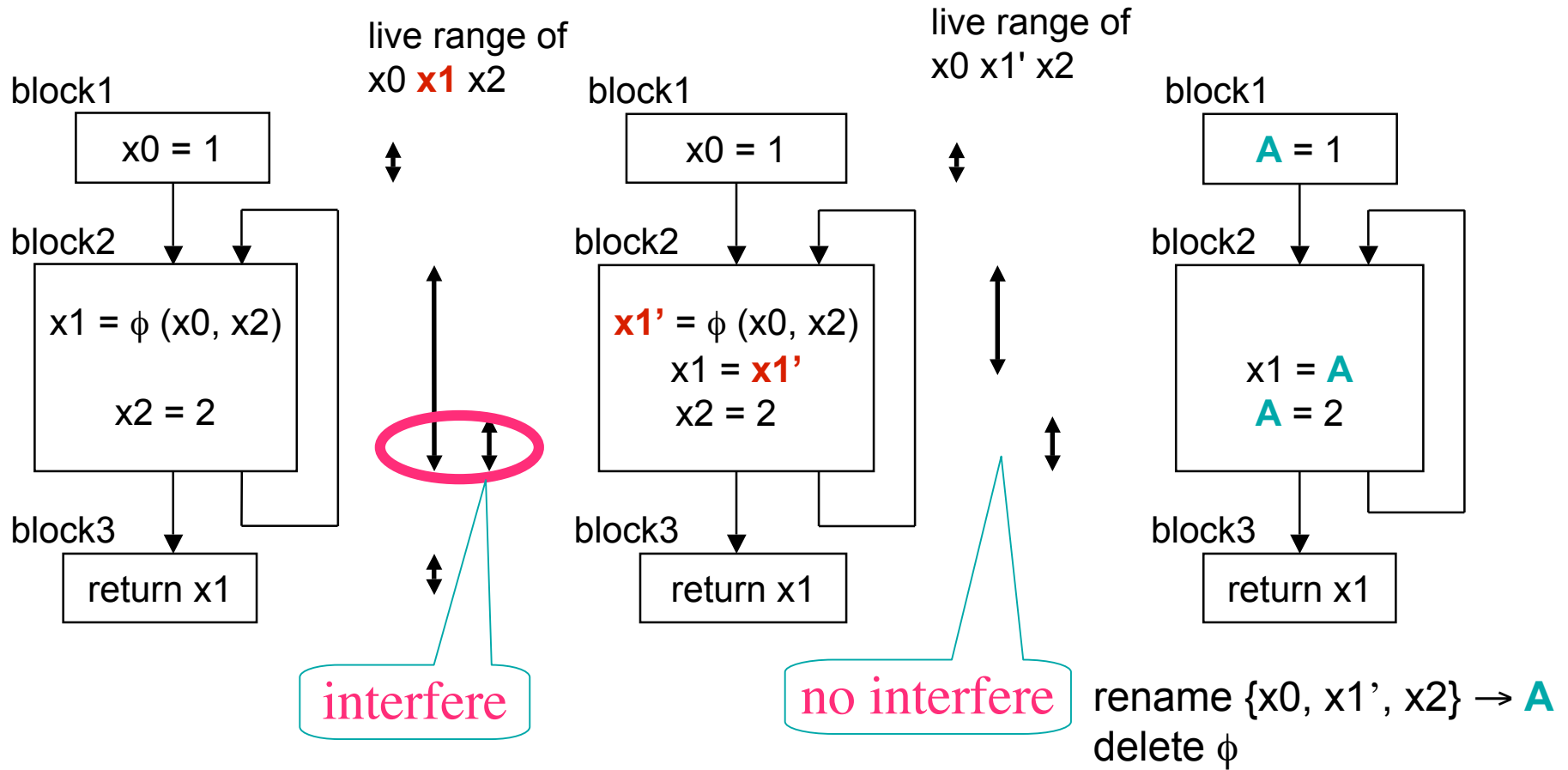
# (i) SSA back translation algorithm by Briggs (cont)



(b) normal form after back translation (same as before)

(c) after coalescing

# (ii) SSA back translation algorithm by Sreedhar



(a) SSA form

(b) eliminating interference

(c) normal form after back translation

# Empirical comparison of SSA back translation

No. of copies (no. of copies in loops)

	<b>SSA form</b>	<b>Briggs</b>	<b>Briggs + Coalescing</b>	<b>Sreedhar</b>
<b>Lost copy</b>	0	3	1 (1)	1 (1)
<b>Simple ordering</b>	0	5	2 (2)	2 (2)
<b>Swap</b>	0	7	5 (5)	3 (3)
<b>Swap-lost</b>	0	10	7 (7)	4 (4)
<b>do</b>	0	9	6 (4)	4 (2)
<b>fib</b>	0	4	0 (0)	0 (0)
<b>GCD</b>	0	9	5 (2)	5 (2)
<b>Selection Sort</b>	0	9	0 (0)	0 (0)
<b>Hige Swap</b>	0	8	3 (3)	4 (4)

(Coins implements Sreedhar's method)



# 4 SSA form optimization

---

# Conditional constant propagation [Wegman 91]

## Example source program

```
/* from Appel, A.: Modern compiler implementation in Java, 2nd ed.,  
   2002, Fig. 19.4 */  
int main() {  
    int i, j, k;  
    i = 1;  
    j = 1;  
    k = 0;  
    while (k < 100) {  
        if (j < 20) {  
            j = i;  
            k = k + 1;  
        } else {  
            j = k;  
            k = k + 2;  
        }  
    }  
    printf("%d\n", j);  
}
```

# Conditional constant propagation

Translate into SSA form

```
i = 1;
j = 1;
k = 0;
while (k < 100) {
  if (j < 20) {
    j = i;
    k = k + 1;
  } else {
    j = k;
    k = k + 2;
  }
}
printf("%d\n", j);
```

(a) source program



```
i1 = 1;
j1 = 1;
k1 = 0;
while (
  j2 =  $\phi$ (j1, j5);
  k2 =  $\phi$ (k1, k5);
  k2 < 100) {
  if (j2 < 20) {
    j3 = i1;
    k3 = k2 + 1;
  } else {
    j4 = k2;
    k4 = k2 + 2;
  }
  j5 =  $\phi$ (j3, j4);
  k5 =  $\phi$ (k3, k4);
}
printf("%d\n", j5);
```

(b) SSA form

# Conditional constant propagation

```
i1 = 1;
j1 = 1;
k1 = 0;
while (
  j2 =  $\phi$ (j1, j5);
  k2 =  $\phi$ (k1, k5);
  k2 < 100) {
  if (j2 < 20) {
    j3 = i1;
    k3 = k2 + 1;
  } else {
    j4 = k2;
    k4 = k2 + 2;
  }
  j5 =  $\phi$ (j3, j4);
  k5 =  $\phi$ (k3, k4);
}
printf("%d\n", j5);
```

(b) SSA form  
(same as before)

```
i1: 1
j1: 1
k1: 0

j2: 1
k2: not constant
k2 < 100:
j2 < 20: always true
j3: 1
k3: not constant
not reachable
j4:
k4:

j5: 1
k5: not constant
```

result of analysis

By analyzing the SSA form, the system knows that

- $j_2, j_3, j_5$  are always 1.
- the else part is not reachable

# Conditional constant propagation

```
i1 = 1;
j1 = 1;
k1 = 0;
while (
  j2 =  $\phi(j1, j5)$ ;
  k2 =  $\phi(k1, k5)$ ;
  k2 < 100) {
  if (j2 < 20) {
    j3 = i1;
    k3 = k2 + 1;
  } else {
    j4 = k2;
    k4 = k2 + 2;
  }
  j5 =  $\phi(j3, j4)$ ;
  k5 =  $\phi(k3, k4)$ ;
}
printf("%d\n", j5);
```

(b) SSA form  
(same as before)



```
while (
  k2 =  $\phi(0, k3)$ ;
  k2 < 100) {
  k3 = k2 + 1;
}
printf("%d\n", 1);
```

(c) SSA form after cond. const. prop.  
with dead code elimination

# Conditional constant propagation

```
while (  
  k2 =  $\phi(0, k3)$ ;  
  k2 < 100) {  
  k3 = k2 + 1;  
}  
printf("%d¥n", 1);
```



```
k = 0;  
while (k < 100) {  
  k = k + 1;  
}  
printf("%d¥n", 1);
```

(it is actually in intermediate representation, but shown here in structured C style)

(c) SSA form after cond. const. prop. with dead code elimination (same as before)

(d) after SSA back translation

# Conditional constant propagation (Summary)

```
i = 1;
j = 1;
k = 0;
while (k < 100) {
  if (j < 20) {
    j = i;
    k = k + 1;
  } else {
    j = k;
    k = k + 2;
  }
}
printf("%d\n",j);
```

(a) source program



```
k = 0;
while (k < 100) {
  k = k + 1;
}
printf("%d\n",1);
```

(in structured program style)

We see that constants are propagated, and the “else part” of the while statement is deleted.

(d) after SSA back translation

# Operator strength reduction of loop induction variables and linear function test replacement

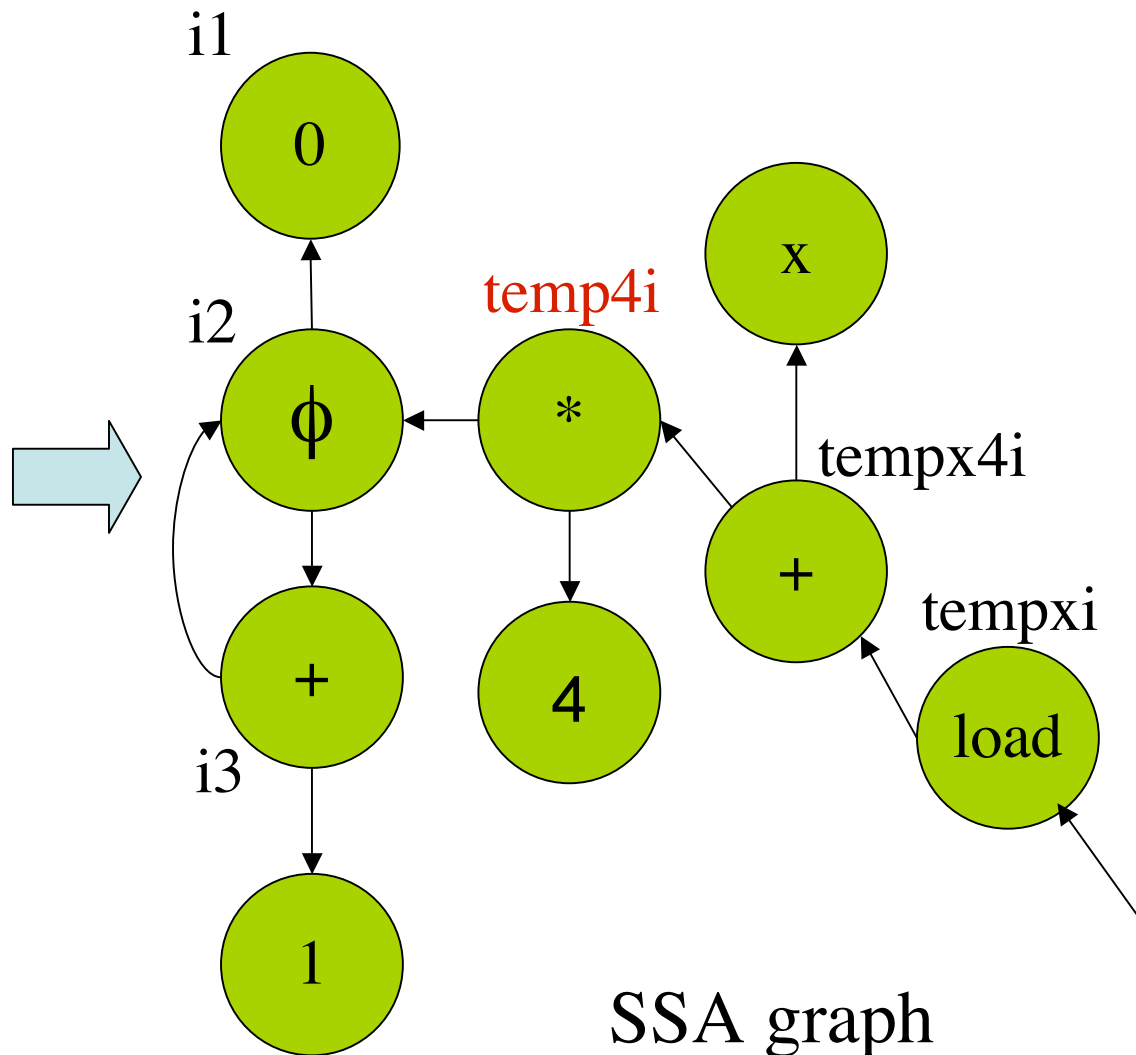
Example source program

```
/* addvectosr.c -- add vector for osr example */  
void addvect(int z[], int x[], int y[], int n) {  
    int i;  
    for (i = 0; i < n; i++) {  
        z[i] = x[i] + y[i];  
    }  
}
```

# Operator strength reduction of loop induction variables and linear function test replacement

```
i1 = 0
L1:
i2 = φ(i1, i3)
if (i2 >= n1) goto L6
temp4i = 4 * i2
tempx4i = x + temp4i
tempxi = * tempx4i
...
i3 = i2 + 1
goto L1
L6:
```

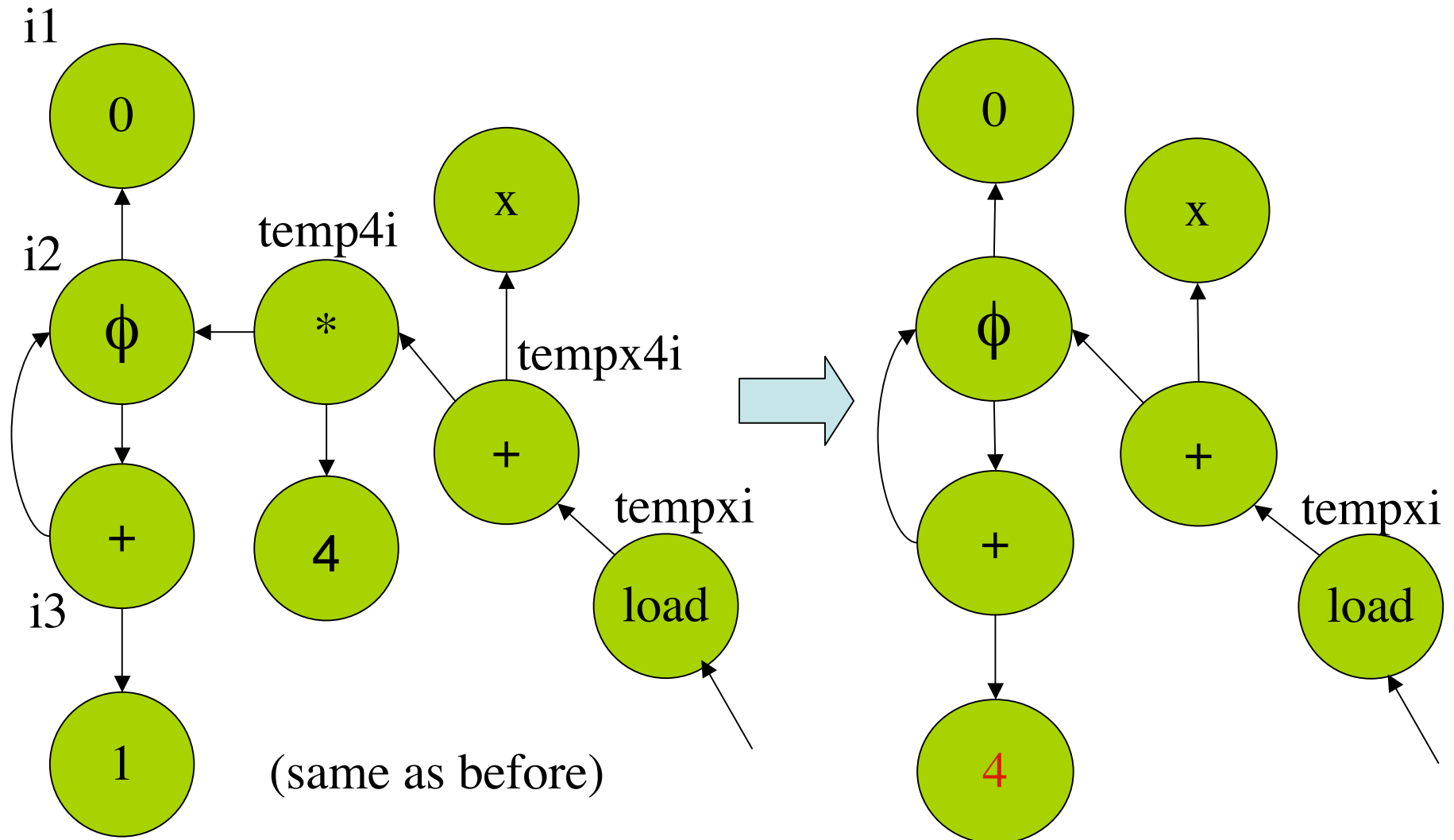
SSA form intermediate representation shown in C style



SSA graph

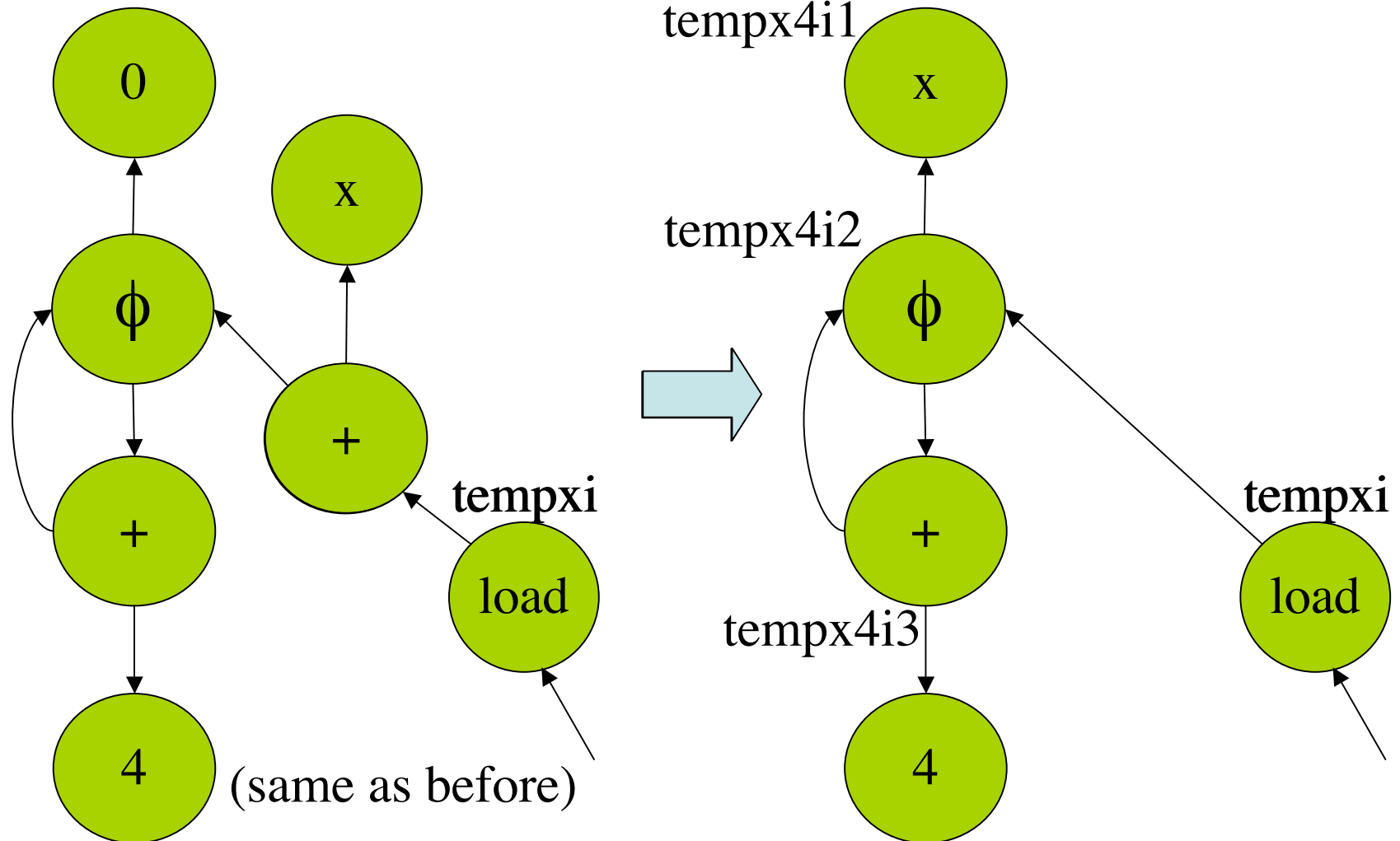
(‘load’ corresponds to prefix op ‘\*’ in C)

# Operator strength reduction of loop induction variables and linear function test replacement



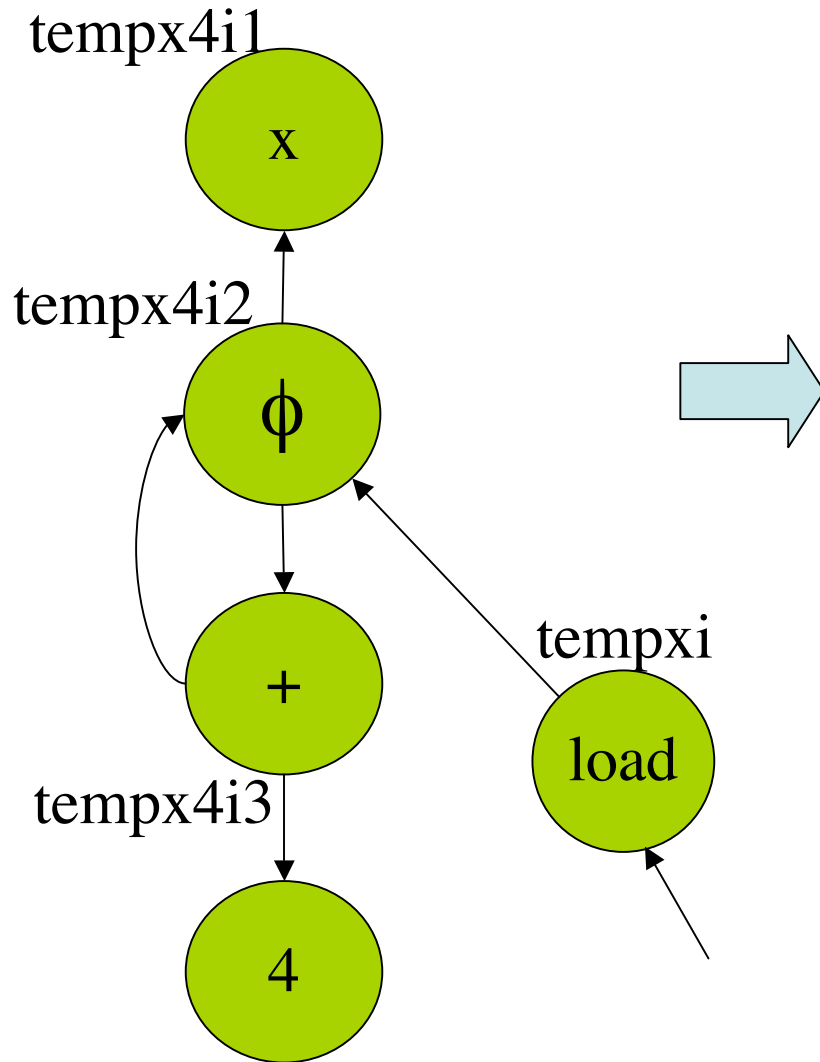
transformation of SSA graph - strength reduction

# Operator strength reduction of loop induction variables and linear function test replacement



further transformation of SSA graph - strength reduction

# Operator strength reduction of loop induction variables and linear function test replacement



SSA graph (same as before)

...

`temp_x4i1 = x`

L1:

`temp_x4i2 =  $\phi$ (temp_x4i1, temp_x4i3)`

...

`temp_xi = * temp_x4i2`

...

`temp_x4i3 = temp_x4i2 + 4`

...

reconstructed SSA form

# Operator strength reduction of loop induction variables and linear function test replacement

Example source program

```
void addvect (int z[], int x[],
int y[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        z[i] = x[i] + y[i];
    }
}
```

after strength reduction of ind var

```
...
temp4i = 0;
tempx4i = x;
tempy4i = y;
tempz4i = z;
i = 0;
if (i >= n) goto L6;
L4:
tempxi = * tempx4i;
tempyi = * tempy4i;
* tempz4i = tempxi + tempyi;
i = i + 1;
temp4i = temp4i + 4;
tempx4i = tempx4i+4;
tempy4i = tempy4i+4;
tempz4i = tempz4i+4;
if (temp4i < n << 2) goto L4;
L6:
```

In array accesses, 'multiply by 4' disappears. (Actually it is in SSA form intermediate representation, but shown here in C style without  $\phi$ .)

Test of induction variable 'i' is replaced by that of 'temp4i'

# Operator strength reduction of loop induction variables and linear function test replacement

after strength reduction of ind var

```
...
temp4i = 0;
tempx4i = x;
tempy4i = y;
tempz4i = z;
i = 0;
if (i >= n) goto L6;
L4:
tempxi = * tempx4i;
tempyi = * tempy4i;
* tempz4i = tempxi + tempyi;
i = i + 1;
temp4i = temp4i + 4;
tempx4i = tempx4i+4;
tempy4i = tempy4i+4;
tempz4i = tempz4i+4;
if (temp4i < n << 2) goto L4;
L6:
```

(same as before)

after all optimization

```
...
if (0 >= n) goto L6;
temp4n = n << 2;
temp4i = 0;
L4:
tempxi = * x;
tempyi = * y;
* z = tempxi + tempyi;
temp4i = temp4i + 4;
x = x + 4;
y = y + 4;
z = z + 4;
if (temp4i < temp4n) goto L4;
L6:
```

Test of induction variable 'i' is replaced by that of 'temp4i', and 'i' disappears.

# Operator strength reduction of loop induction variables and linear function test replacement

Note: Further optimization might be possible in some architecture like SPARC, as shown below. But it depends on instruction set.

```
...
if (0 >= n) goto L6;
temp4n = n << 2;
temp4i = 0;
L4:
tempxi = * x;
tempyi = * y;
* z = tempxi + tempyi;
temp4i = temp4i + 4;
x = x + 4;
y = y + 4;
z = z + 4;
if (temp4i < temp4n) goto L4;
L6:
```

(same as before)



```
...
if (0 >= n) goto L6;
temp4n = n << 2;
temp4i = 0;
L4:
tempxi = * (x + temp4i);
tempyi = * (y + temp4i);
* (z + temp4i) = tempxi + tempyi;
temp4i = temp4i + 4;

if (temp4i < temp4n) goto L4;
L6:
```

# Common subexpression elimination

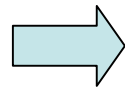


## Example source program

```
/* cse_test.c */  
...  
a = 1;  
b = 2;  
x = 3;  
y = a + b;  
c = x + y;  
if (a < 10)  
    z = a + b;  
else  
    z = a + b;  
d = x + z;  
printf("%d\n",d);
```

# Common subexpression elimination

```
a = 1;
b = 2;
x = 3;
y = a + b;
c = x + y;
if (a < 10)
    z = a + b;
else
    z = a + b;
d = x + z;
printf("%d\n", d);
```



```
a = 1;
b = 2;
x = 3;
y = a + b;
c = x + y;
if (a < 10)
    (z1 IS y) // a + b is a common subexpr
              // and z1 is equal to y.
else
    (z2 IS y) // a + b is a common subexpr
              // and z2 is equal to y.
z3 =  $\phi$ (z1,z2);
(z3 IS y) // therefore, z3 is equal to y
          // after if statement.
(d IS x+z3 IS x+y IS c) // d is equal to c
printf("%d\n", d);
```

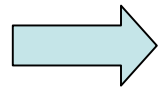
source program

after analysis in SSA form

# Common subexpression elimination

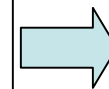
```
a = 1;
b = 2;
x = 3;
y = a + b;
c = x + y;
if (a < 10)
  (z1 IS y)
else
  (z2 IS y)
z3 =  $\phi$ (z1,z2);
(z3 IS y)
(d IS x+z3 ... IS c)
printf("%d\n", d);
```

after analysis  
(same as before)



```
a = 1;
b = 2;
x = 3;
y = a + b;
c = x + y;
if (a < 10) {
} else {
}
printf("%d\n", c);
```

after common  
subexpr elim



```
printf("%d\n", 6);
```

after all  
optimizations

# References



- [Appel-Tiger 02] Appel, A.: Modern Compiler Implementation in Java, second ed., Cambridge University Press, 2002.
- [Briggs et al. 98] Briggs, P., Cooper, K., Harvey, T. and Simpson, T.: Practical improvements to the construction and destruction of static single assignment form, *Softw. Pract. Exper.*, Vol. 28, No. 8, pp. 859-881, 1998.
- [Cooper et al. 01] Cooper, K., Simpson, T. and Vick, C.: Operator Strength Reduction, *Trans. Prog. Lang. Syst.*, Vol. 23, No. 5, pp. 603-625, 2001.
- [Cooper et al. 03] Cooper, K. and Torczon, L.: *Engineering a Compiler*, Morgan Kauffmann, 2003.
- [Cytron et al. 91] Cytron, R., Ferrante, J, Rosen, B. K., Wegman, M. N. and Zadeck, F. K.: Efficiently computing static single assignment form and the control dependence graph, *ACM Trans. Prog. Lang. Syst.* 13, 4, 451-490, 1991.
- [Nakata 99] Nakata, I.: *Organization and Optimization of Compilers*, Asakura-shoten, 1999. (in Japanese)

# References (cont)

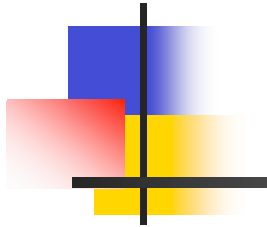


- [Sreedhar and Gao 95] Sreedhar, V. C. and Gao, G. R.: A Linear Time Algorithm for Placing  $\phi$ -nodes, ACM 22nd Symposium on Principles of Programming Languages, 62-73, 1995.
- [Sreedhar et al 99] Sreedhar, V. C., Ju, R.D.-C., Gillies, D.M. and Santhanam, V.: Translating out of static single assignment form, in Cortesi, A. and File, G. (Eds.) SAS'99, LNCS 1694, pp. 194-210, 1999.
- [Wegman and Zadeck 91] Wegman, M.N., and Zadeck, F.K.: Constant propagation with conditional branches, ACM Trans. Prog. Lang. Syst., 13, 2, 181-210, 1991.
- The COINS project web page. <http://www.coins-project.org/>
- SSA module of COINS, especially the external specification document. <http://www.coins-project.org/> and click SSA.
- Further references of SSA are available in the external specification document of SSA module in the Coins web page.

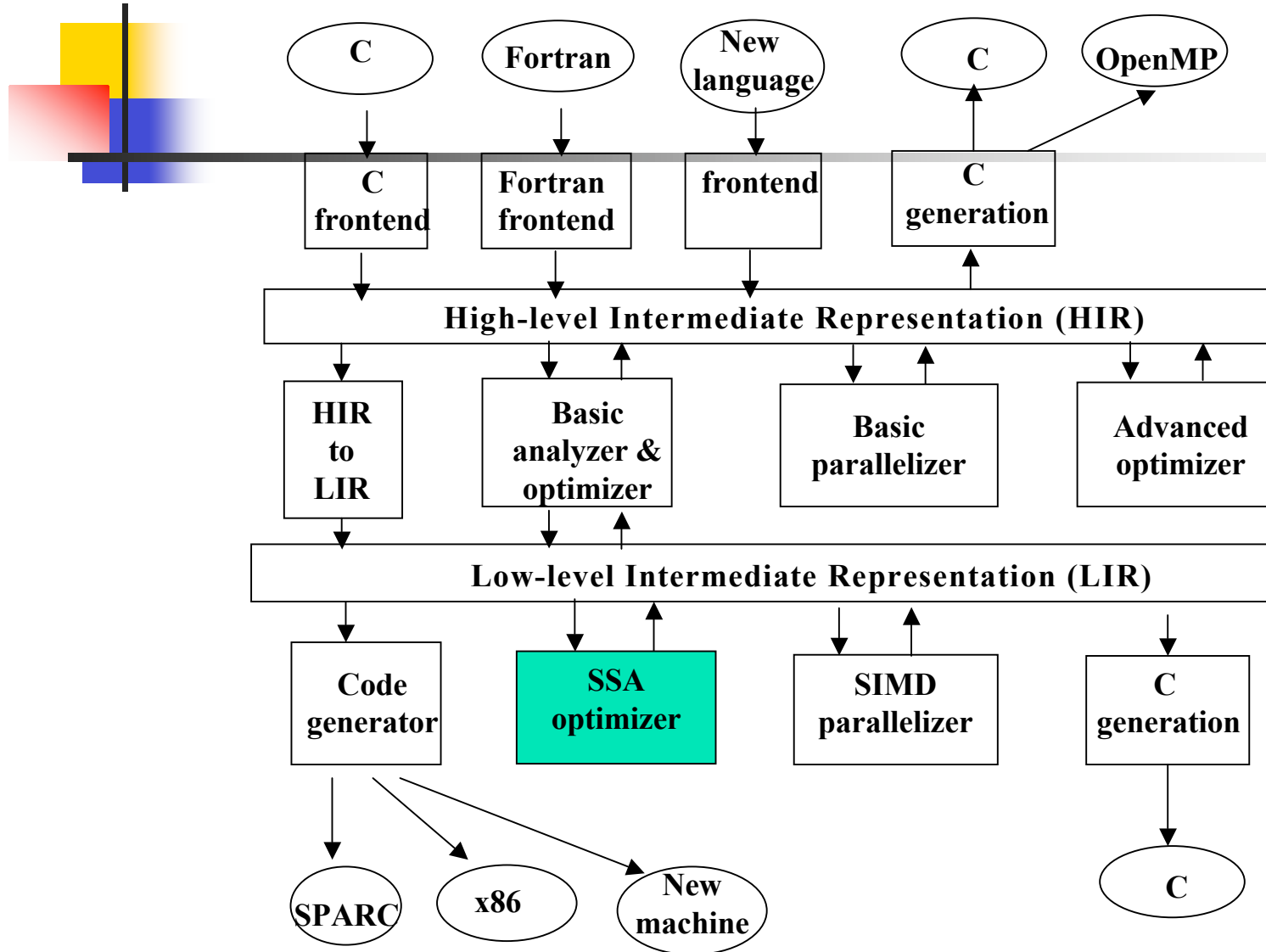
Appendix

SSA optimization module

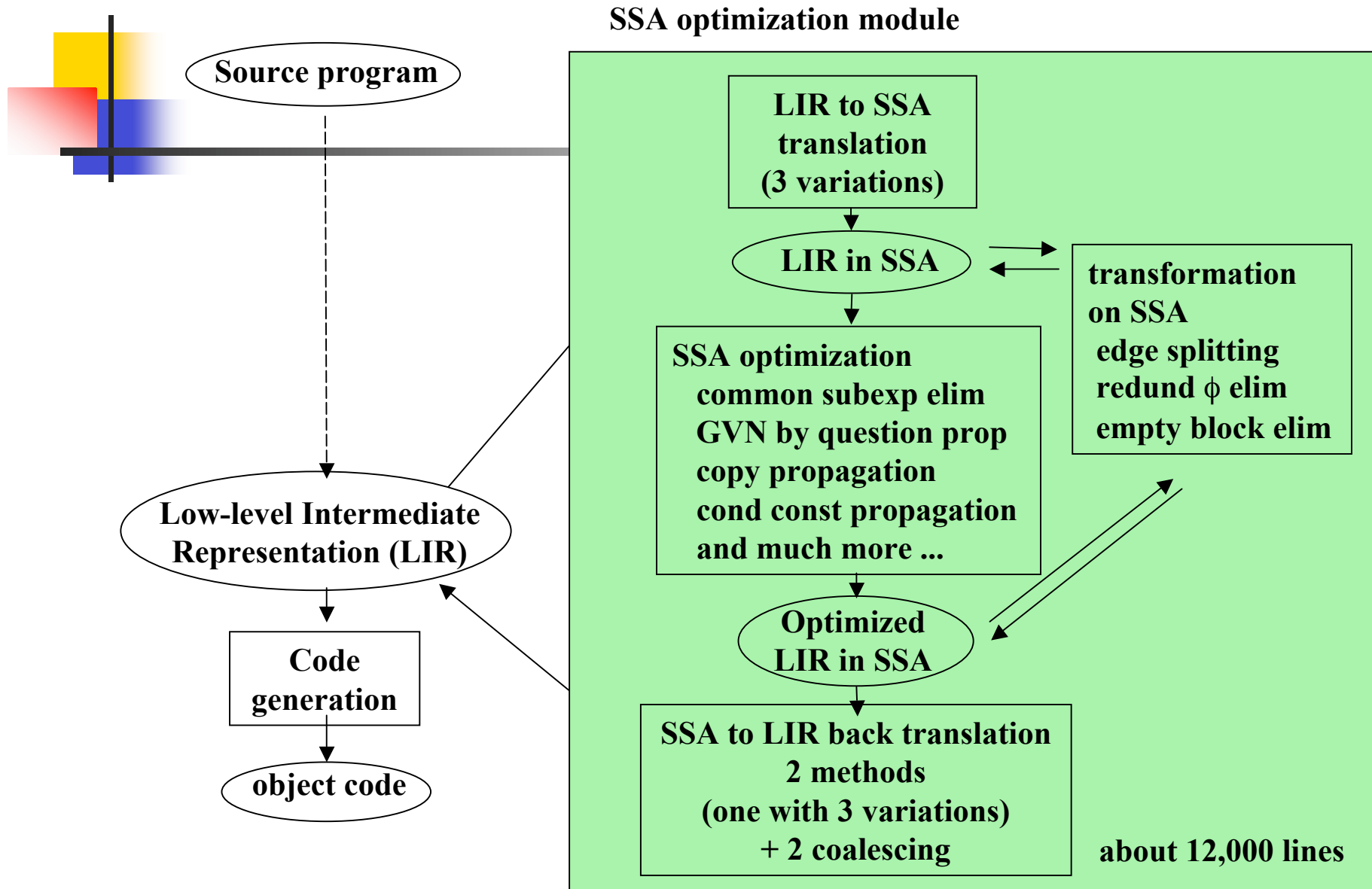
in COINS



# COINS compiler infrastructure



# SSA optimization module in COINS



# Outline of SSA module in COINS (1)



## Translation into and back from SSA form on Low-level Intermediate Representation (LIR)

---

- SSA translation
  - Use dominance frontier [Cytron et al. 91]
  - 3 variations: translation into Minimal , Semi-pruned and Pruned SSA forms
- SSA back translation
  - Sreedhar et al.'s method [Sreedhar et al. 99]
    - 3 variations: Method I, II, and III
  - Briggs et al.'s method [Briggs et al. 98] (under development)
- Coalescing
  - SSA-based coalescing during SSA back translation [Sreedhar et al. 99]
  - Chaitin's style coalescing after back translation
- Each variation and coalescing can be specified by options

# Outline of SSA module in COINS (2)



- Several optimization on SSA form:

dead code elimination, copy propagation, common subexpression elimination, global value numbering based on efficient query propagation, conditional constant propagation, loop invariant code motion, operator strength reduction for induction variable and linear function test replacement, empty block elimination, copy folding at SSA translation time ...

- Useful transformation as an infrastructure for SSA form optimization:

critical edge removal on control flow graph, loop transformation from 'while' loop to 'if-do-while' loop, redundant phi-function elimination, making SSA graph ...

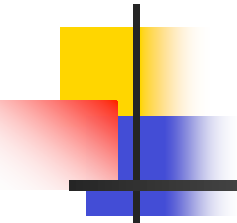
- Each variation, optimization and transformation can be made selectively by specifying options



# Demonstration

---

- Demo on a SPARC machine



(Foils for laboratory work written in Japanese follows)

---



# 本日の実習内容

---

- デモプログラムを実際に動かしてみる.  
次のいくつかを行ってみて, SSAモジュールの動かし方に慣れるとともに, 変換や最適化の前後のLIRやSSA形式, lir2cの結果, アセンブリコードなどを見る.
  - ◆ SSA変換とSSA逆変換
    - ◆ Readme-ssatrans参照
  - ◆ SSA最適化
    - ◆ Readme-ssaopt参照
- レポートは不要.



# レポート課題

次ページのプログラムについて、

- CoinsのSSA最適化のオプションをいろいろ変えて最適化を行い、どのような目的コードが出るかを確認する。（Sparcコードが難しい場合は、lir2cの結果でもよい）
- 最内ループの目的コードについて、さらなる最適化の余地がないかどうか調べる。Sunのccやgccによる目的コード、人手での最適化と比べてみるとよい。
- なお、最適化の結果が正しそうなことを、ccあるいはgccによる実行結果と比べておくとよい。
- プログラムは、`~sassa/coins/ssa-demo-ssa-shuchukougi0407/matmul-main.c` にある。

あるいは、SSA形式に関するものであれば、他の自由な課題を行ってもよい。

```
>less matmul-main.c
```

```
# include <stdio.h>
```

```
main(){
```

```
    double A[3][3], B[3][3], C[3][3], s;
```

```
    int i, j, k;
```

```
    for (i = 0; i < 3; i++ ) {
```

```
        for (j = 0; j < 3; j++ ) {
```

```
            A[i][j] = (double) i*2+j;
```

```
            B[i][j] = (double) i*2+j;
```

```
        }
```

```
    }
```

```
    for ( i=0; i<3; i++){
```

```
        for ( j=0; j<3; j++){
```

```
            s=0.0;
```

```
            for ( k=0; k<3; k++){
```

```
                s=s+A[i][k]*B[k][j];
```

```
            }
```

```
            C[i][j]=s;
```

```
        }
```

```
    }
```

```
    for (i = 0; i < 3; i++ ) {
```

```
        for (j = 0; j < 3; j++ ) {
```

```
            printf("%g %g %g\n", A[i][j], B[i][j], C[i][j]);
```

```
        }
```

```
    }
```

```
}
```



# 参考資料

---

- SPARC命令の表
- Coinsリリースにあるドキュメント：  
edu:...coins.../doc-en/README.SSA.en.txt,  
/doc-ja/README.driver.ja.txt, README.backend.ja.txtなど
- edu上の~sassa/coins/ssa-demo-shuchukougi0407/Readme-ssatrans, Readme-ssaopt
- <http://www.coins-project.org/>
- <http://www.is.titech.ac.jp/~sassa/coins-www-ssa/japanese/index.html>
- 上記Coins SSAウェブページからたどったSSA最適化仕様書
- Coinsバックエンドの資料
- ただし一部は最新情報とは限らないので注意