Intermediate representations

Michel Schinz
Advanced Compiler Construction – 2009-05-15

Intermediate representations

The term **intermediate representation** (**IR**) or **intermediate language** designates the data-structure(s) used by the compiler to represent the program being compiled.

Choosing a good IR is crucial, as many analyses and transformations (*e.g.* optimizations) are substantially easier to perform on some IRs than on others.

Most non-trivial compilers actually use several IRs during the compilation process, and they tend to become more low-level as the code approaches its final form.

Impact of IR on optimizations

Example 1: constant prop.

To illustrate the impact of IR on optimizations, consider the following simple program fragment:

```
x←7
. . .
```

Is it legal to perform **constant propagation** and blindly replace all later occurrences of x by 7?

The answer depends on the IR:

- If the IR allows multiple assignments to the same variable, then additional (data-flow) analyses are required to answer the question, as x might be re-assigned later.
- However, if the IR does *not* allow multiple assignments to the same variable, then yes, all occurrences of x can be unconditionally replaced by 7!

Other simple optimizations

Apart from constant propagation, many simple optimizations are made hard by the presence of multiple assignments to a single variable:

- **common-subexpression elimination**, which consists in avoiding the repeated evaluation of expressions,
- (simple) **dead code elimination**, which consists in removing assignments to variables whose value is not used later,
- etc.

In all cases, analyses are required to distinguish the various “versions” of a variable that appear in the program.

Conclusion: a good IR should not allow multiple assignments to a variable!

Example 2: inlining

Inlining (or **in-line expansion**) consists in replacing a call to a function by a copy of the body of that function, with parameters replaced by the actual arguments. It is a very important compiler optimization, as it often opens the door to further optimizations.

Some aspects of the intermediate representation can have an important impact on the implementation of inlining. To illustrate this, let us examine some problems that can occur when performing inlining directly on the AST – a choice that might seem reasonable at first sight.

Naïve inlining: problem #1

```
(define print/ret (lambda (x) (print-int x) x))
(define twice (lambda (y) (+ y y)))
(define f (lambda (z) (twice (print/ret z))))
```

incorrect inlining
of `twice` in `f`

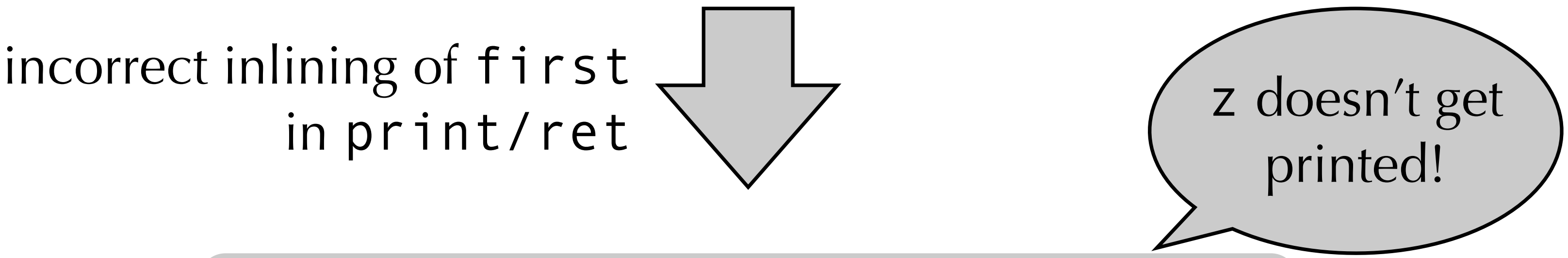
```
(define f (lambda (z)
            (+ (print-and-ret z)
               (print-and-ret z))))
```

`z` gets printed twice!

Possible solution: bind actual parameters to variables (using a `let`) to ensure that they are evaluated *at most* once.

Naïve inlining: problem #2

```
(define first (lambda (x y) x))
(define print/ret
  (lambda (z) (first z (print-int z))))
```



incorrect inlining of `first` in `print/ret`

`z` doesn't get printed!

```
(define print/ret (lambda (z) z))
```

Possible solution: bind actual parameters to variables (using a `let`) to ensure that they are evaluated *at least* once.

Easy inlining

The two pitfalls presented earlier can be avoided by bindings actual arguments to variables (using a `let`) before using them in the body of the inlined function.

However, a properly-designed IR can also avoid the problems altogether by ensuring that actual parameters are *always* atoms, *i.e.* variables or constants.

Conclusion: a good IR should only allow atomic arguments to functions.

IR #1
standard RTL/CFG

Register transfer language

A **register-transfer language** (**RTL**) is a kind of intermediate representation in which most operations compute a function of one or two virtual registers (*i.e.* variables) and store the result in another virtual register.

For example, the instruction adding variables `y` and `z`, storing the result in `x` could be written `x ← y + z`. Such instructions are sometimes called **quadruples**, because they typically have four components: the three variables (`x`, `y` and `z` here) and the operation (+ here).

RTLs are very close to assembly languages, the main difference being that the number of virtual registers is usually not bounded.

Control-flow graph

A **control-flow graph** (**CFG**) is a directed graph whose nodes are the individual instructions of a function, and whose edges represent control-flow.

More precisely, there is an edge in the CFG from a node n_1 to a node n_2 if and only if the instruction of n_2 can be executed immediately after the instruction of n_1.

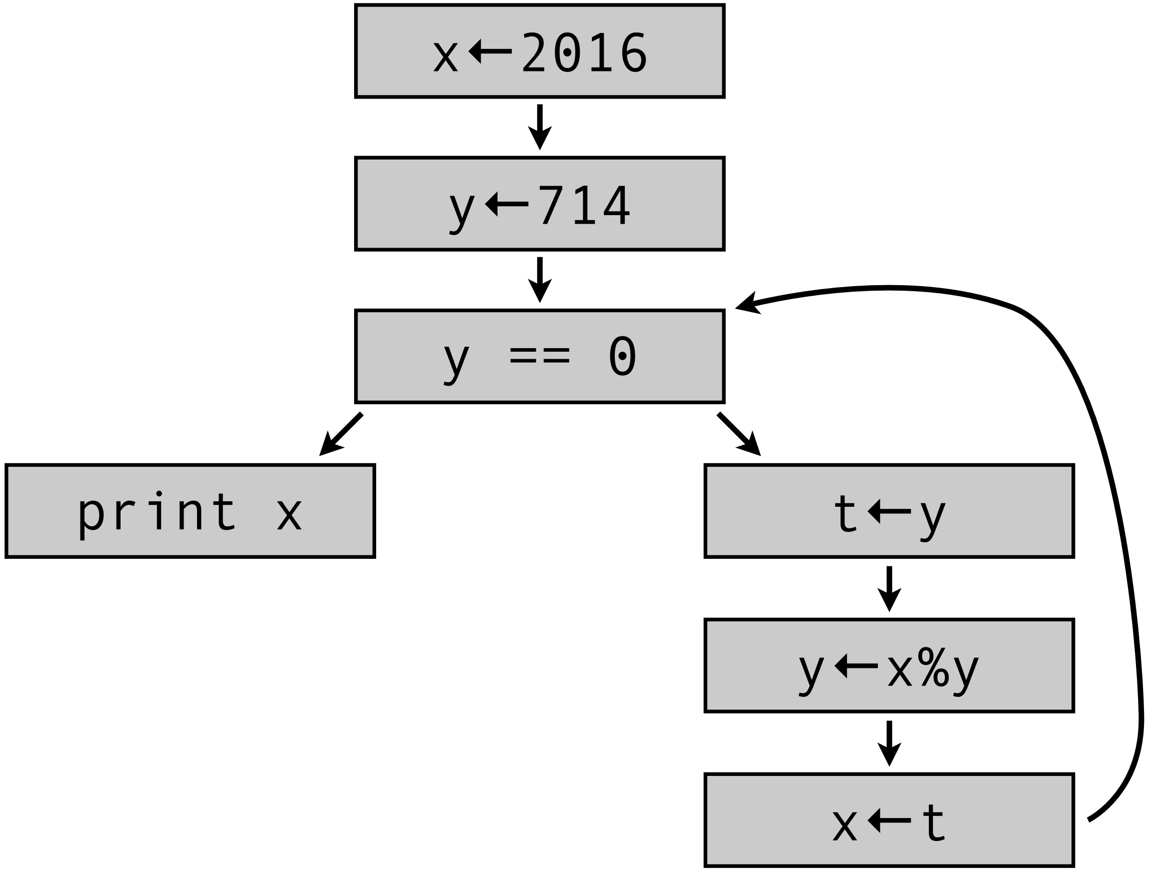
RTL/CFG

RTL/CFG is the name given to intermediate representations where each function of the program is represented as a control-flow graph whose node contain RTL instructions.

This kind of representation is very common in the late stages of compilers, especially those for imperative languages.

RTL/CFG example

Computation of the greatest common divisor of 2016 and 714 in a typical RTL/CFG representation.



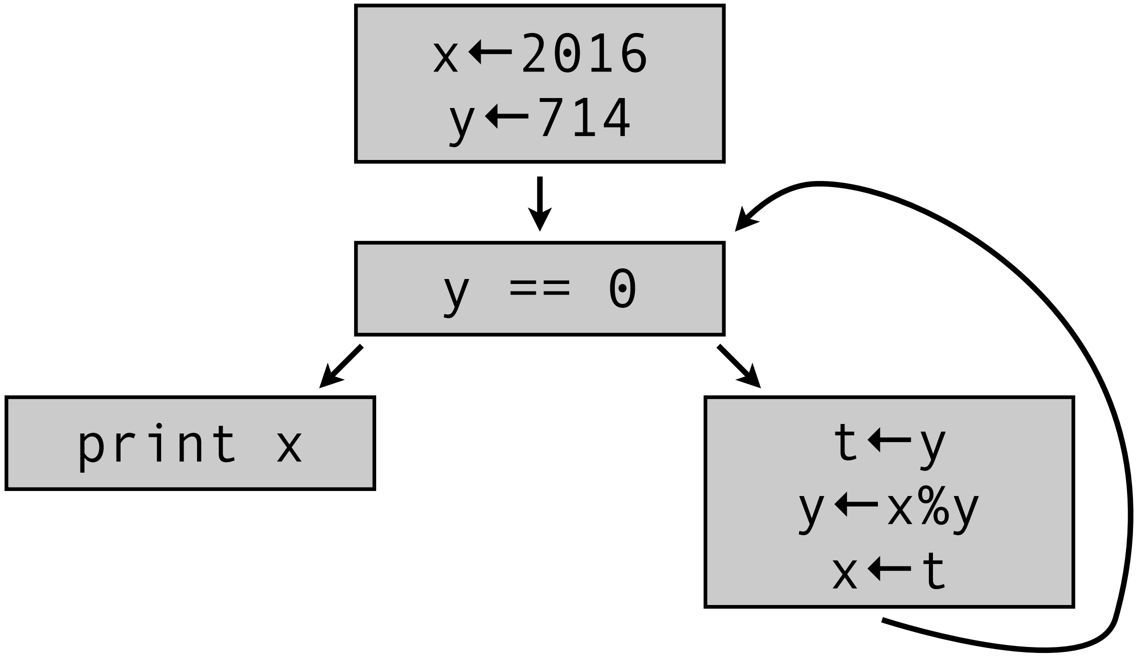
Basic blocks

A **basic block** is a maximal sequence of instruction for which control can only enter through the first instruction of the block and leave through the last.

Basic blocks are sometimes used as the nodes of the CFG, instead of individual instructions. This has the advantage of reducing the number of nodes in the CFG, but also complicates data-flow analyses. It is therefore far from being clear that basic blocks are still useful today.

RTL/CFG example

Same examples as before, but with basic blocks instead of individual instructions.



RTL/CFG pros and cons

Positive aspects of RTL/CFG:

- All intermediate values (*i.e.* subexpressions) are named, which helps when performing some optimizations like common-subexpression elimination.

Negative aspects of RTL/CFG:

- Even very simple optimizations (*e.g.* constant propagation, common-subexpression elimination) require data-flow analyses. This is because a single variable can be assigned multiple times.

IR #2
RTL/CFG in SSA form

SSA form

An RTL/CFG program is said to be in **static single-assignment** (**SSA**) form if each variable has only one definition in the program.

That single definition can be executed many times when the program is run – if it is inside a loop – hence the qualifier static.

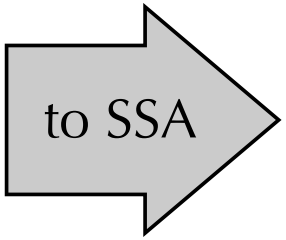
SSA form is popular because it simplifies several optimizations and analysis, as we will see.

Most (imperative) programs are not naturally in SSA form, and must therefore be transformed so that they are.

Straight-line code

Transforming a piece of straight-line code – *i.e.* without branches – to SSA is trivial: each definition of a given name gives rise to a new version of that name, identified by a subscript:

```
x←12
y←15
x←x+y
y←x+4
z←x+y
y←y+1
```



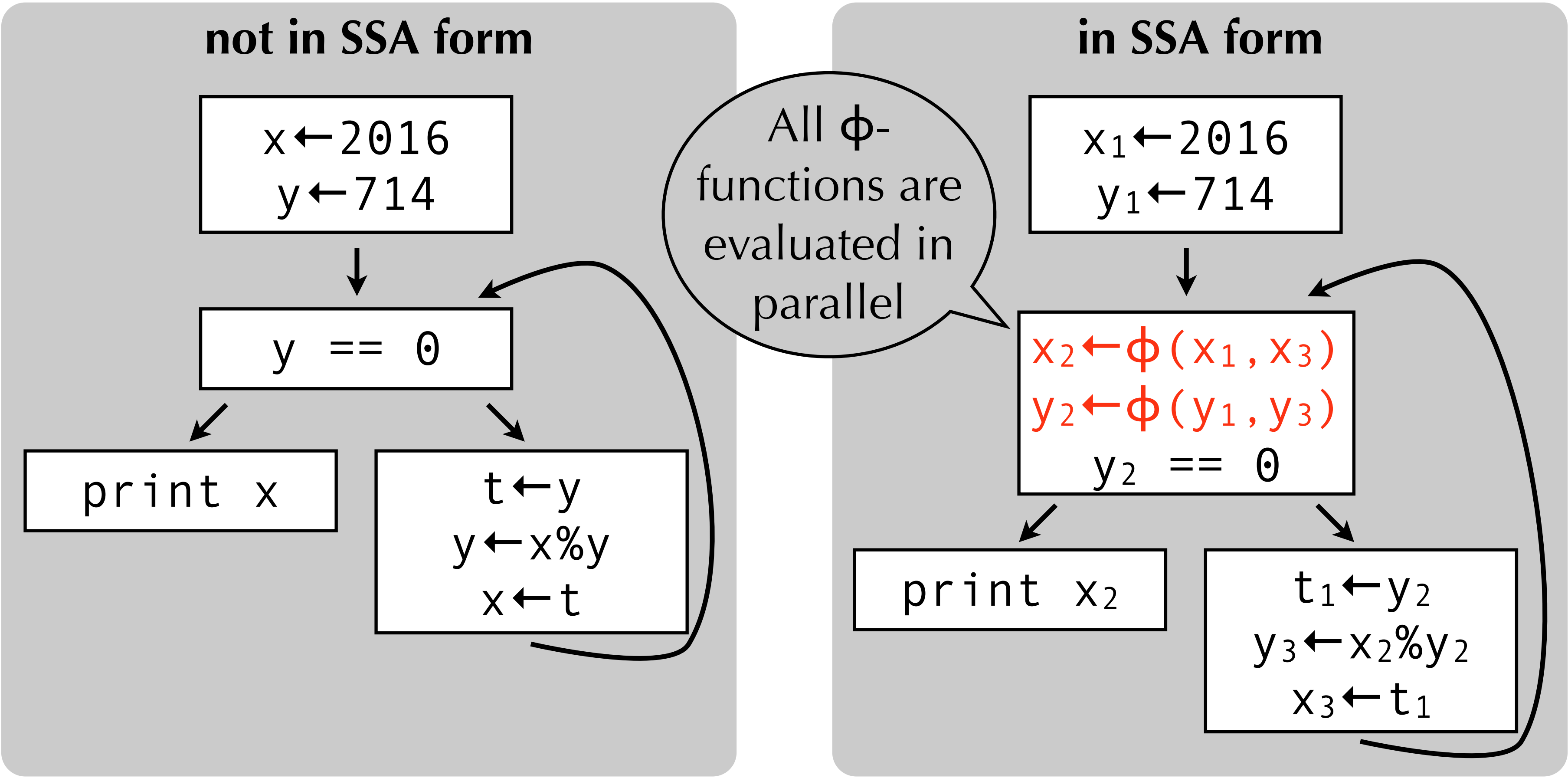
$$\begin{aligned}
&x_1 \leftarrow 12\\
&y_1 \leftarrow 15\\
&x_2 \leftarrow x_1 + y_1\\
&y_2 \leftarrow x_2 + 4\\
&z_1 \leftarrow x_2 + y_2\\
&y_3 \leftarrow y_2 + 1
\end{aligned}$$

ϕ-functions

Join-points in the CFG – nodes with more than one predecessors – are more problematic, as each predecessor can bring its own version of a given name.

To reconcile those different versions, a fictional **ϕ-function** is introduced at the join point. That function takes as argument all the versions of the variable to reconcile, and automatically selects the right one depending on the flow of control.

ϕ-functions example



not in SSA form

$x \leftarrow 2016$
$y \leftarrow 714$

$y == 0$

print x

$t \leftarrow y$
$y \leftarrow x\%y$
$x \leftarrow t$

in SSA form

$x_1 \leftarrow 2016$
$y_1 \leftarrow 714$

$x_2 \leftarrow \phi(x_1, x_3)$
$y_2 \leftarrow \phi(y_1, y_3)$
$y_2 == 0$

print x_2

$t_1 \leftarrow y_2$
$y_3 \leftarrow x_2\%y_2$
$x_3 \leftarrow t_1$

All ϕ-functions are evaluated in parallel

Evaluation of ϕ-functions

It is crucial to understand that all ϕ-functions of a block are evaluated *in parallel*, and not in sequence as the representation might suggest!

To make this clear, some authors write ϕ-functions in matrix form, with one row per predecessor:

$$(x_2, y_2) \leftarrow \phi\begin{pmatrix} x_1 & y_1 \\ x_3 & y_3 \end{pmatrix} \quad \text{instead of} \quad \begin{array}{l} x_2 \leftarrow \phi(x_1, x_3) \\ y_2 \leftarrow \phi(y_1, y_3) \end{array}$$

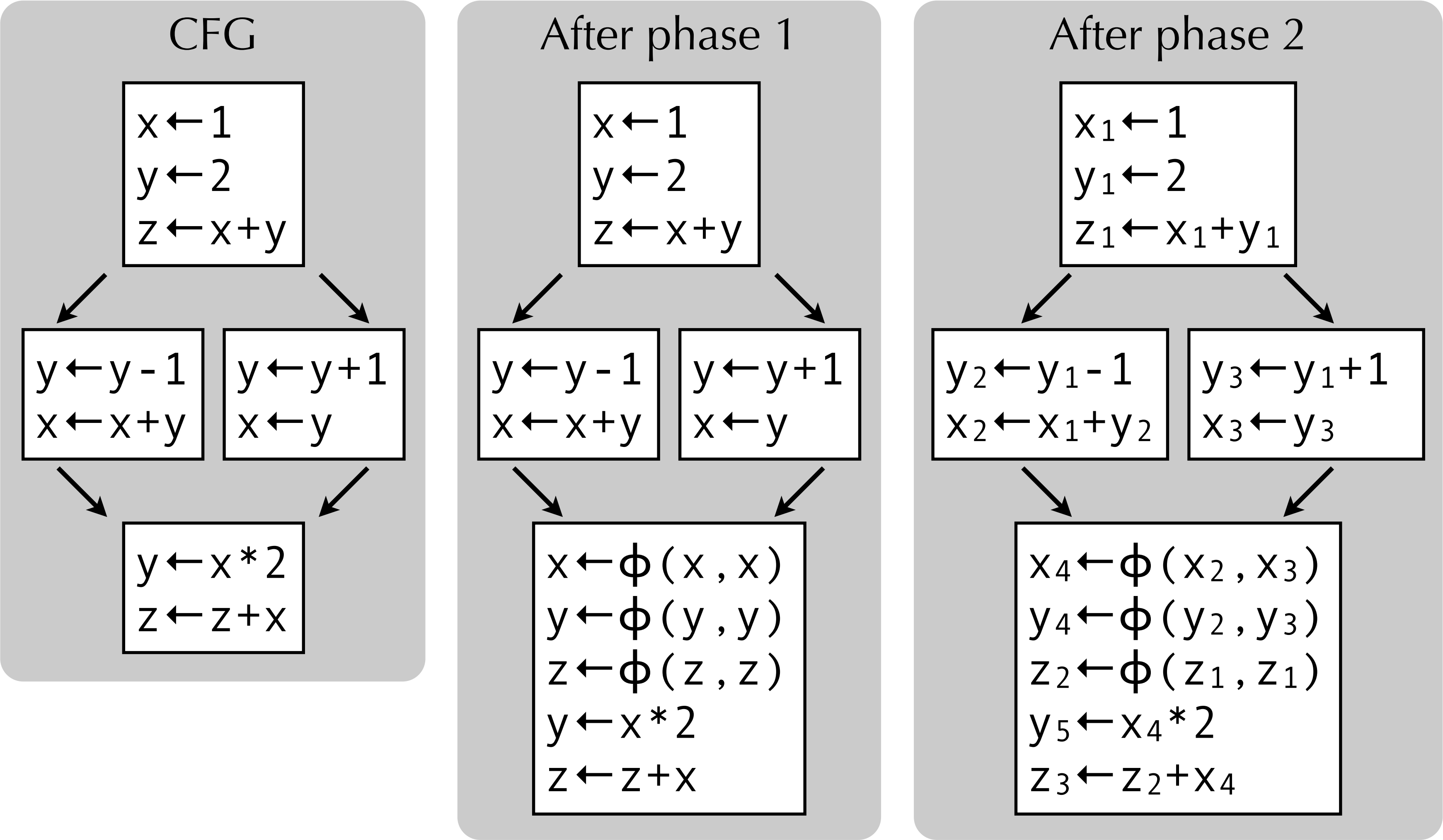
In the following slides, we will usually stick to the common, linear representation, but keep the parallel nature of ϕ-functions in mind.

(Naïve) building of SSA form

Naïve technique to build SSA form:

- for each variable *x* of the CFG, at each join point *n*, insert a ϕ-function of the form $x = \phi(x, \ldots, x)$ with as many parameters as *n* has predecessors,
- compute reaching definitions, and use that information to rename any use of a variable according to the – now unique – definition reaching it.

(Naïve) building of SSA form



CFG

$x \leftarrow 1$
$y \leftarrow 2$
$z \leftarrow x+y$

$y \leftarrow y-1$
$x \leftarrow x+y$

$y \leftarrow y+1$
$x \leftarrow y$

$y \leftarrow x*2$
$z \leftarrow z+x$

After phase 1

$x \leftarrow 1$
$y \leftarrow 2$
$z \leftarrow x+y$

$y \leftarrow y-1$
$x \leftarrow x+y$

$y \leftarrow y+1$
$x \leftarrow y$

$x \leftarrow \phi(x, x)$
$y \leftarrow \phi(y, y)$
$z \leftarrow \phi(z, z)$
$y \leftarrow x*2$
$z \leftarrow z+x$

After phase 2

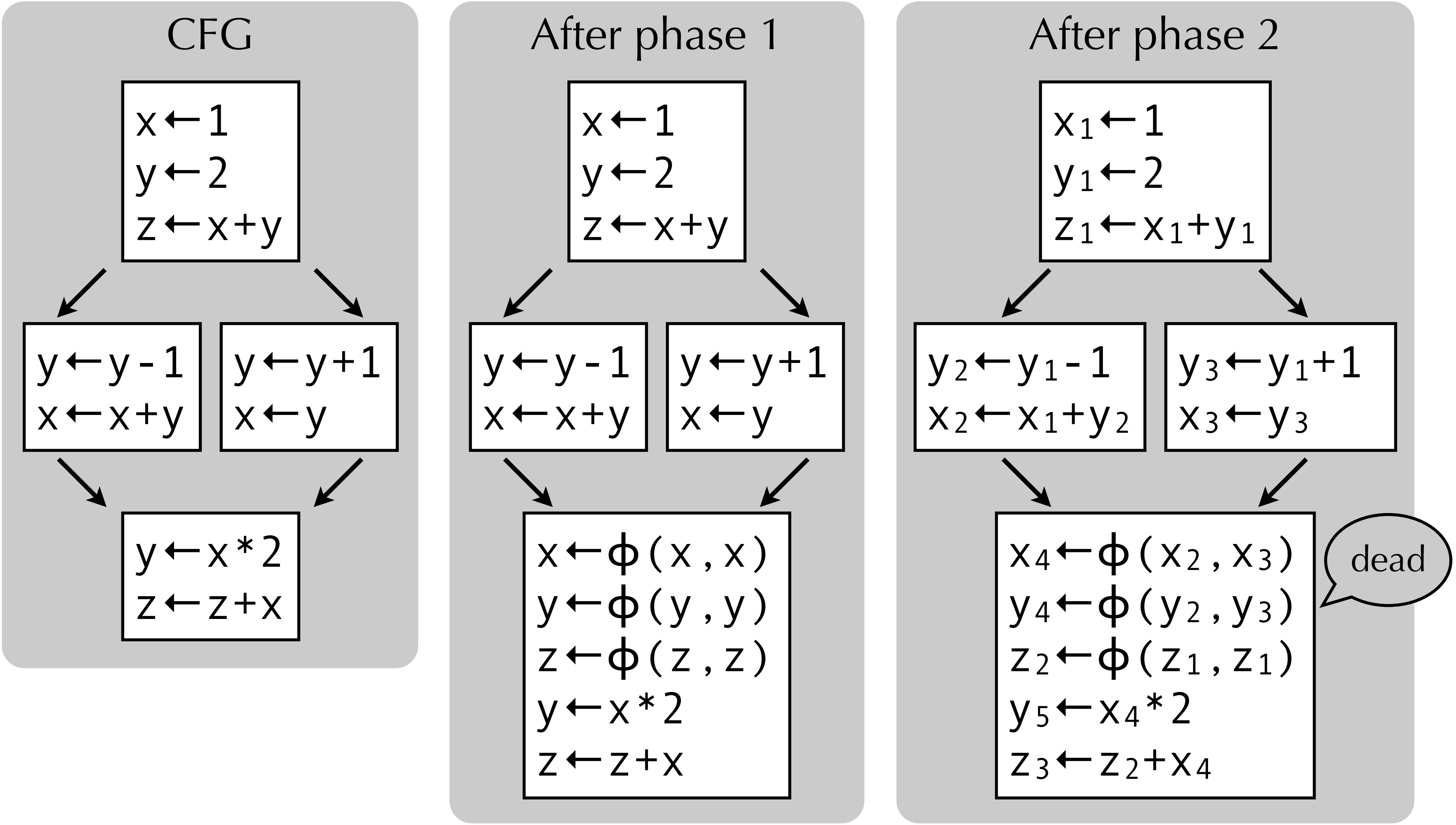
$x_1 \leftarrow 1$
$y_1 \leftarrow 2$
$z_1 \leftarrow x_1+y_1$

$y_2 \leftarrow y_1-1$
$x_2 \leftarrow x_1+y_2$

$y_3 \leftarrow y_1+1$
$x_3 \leftarrow y_3$

$x_4 \leftarrow \phi(x_2, x_3)$
$y_4 \leftarrow \phi(y_2, y_3)$
$z_2 \leftarrow \phi(z_1, z_1)$
$y_5 \leftarrow x_4*2$
$z_3 \leftarrow z_2+x_4$

(Naïve) building of SSA form



CFG

x←1
y←2
z←x+y

y←y-1
x←x+y

y←y+1
x←y

y←x*2
z←z+x

After phase 1

x←1
y←2
z←x+y

y←y-1
x←x+y

y←y+1
x←y

x←ϕ(x,x)
y←ϕ(y,y)
z←ϕ(z,z)
y←x*2
z←z+x

After phase 2

x_1←1
y_1←2
z_1←x_1+y_1

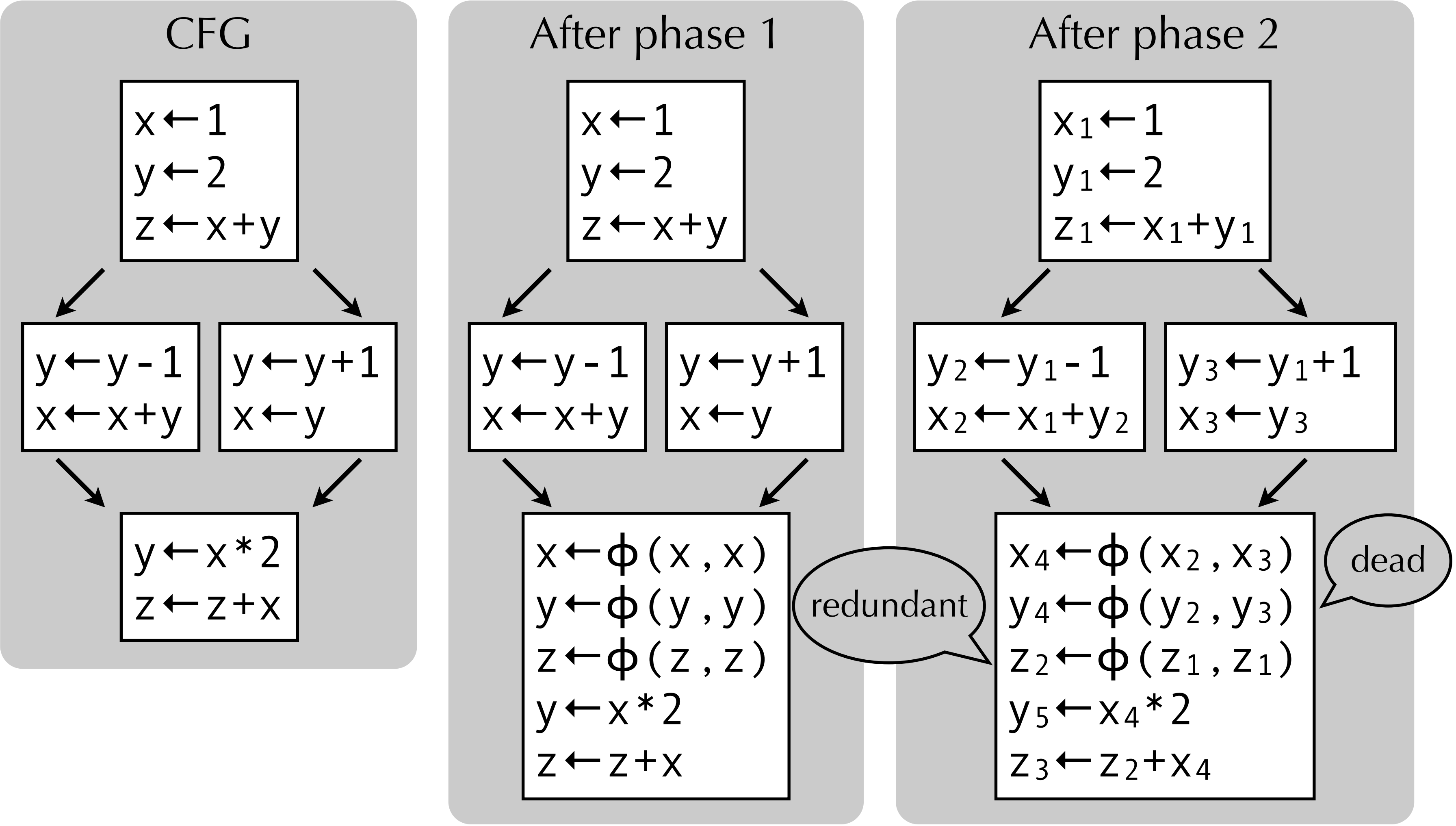
y_2←y_1-1
x_2←x_1+y_2

y_3←y_1+1
x_3←y_3

x_4←ϕ(x_2,x_3)
y_4←ϕ(y_2,y_3)
z_2←ϕ(z_1,z_1)
y_5←x_4*2
z_3←z_2+x_4

dead

(Naïve) building of SSA form



CFG

$x \leftarrow 1$
$y \leftarrow 2$
$z \leftarrow x+y$

$y \leftarrow y-1$
$x \leftarrow x+y$

$y \leftarrow y+1$
$x \leftarrow y$

$y \leftarrow x*2$
$z \leftarrow z+x$

After phase 1

$x \leftarrow 1$
$y \leftarrow 2$
$z \leftarrow x+y$

$y \leftarrow y-1$
$x \leftarrow x+y$

$y \leftarrow y+1$
$x \leftarrow y$

$x \leftarrow \phi(x, x)$
$y \leftarrow \phi(y, y)$
$z \leftarrow \phi(z, z)$
$y \leftarrow x*2$
$z \leftarrow z+x$

redundant

After phase 2

$x_1 \leftarrow 1$
$y_1 \leftarrow 2$
$z_1 \leftarrow x_1+y_1$

$y_2 \leftarrow y_1-1$
$x_2 \leftarrow x_1+y_2$

$y_3 \leftarrow y_1+1$
$x_3 \leftarrow y_3$

$x_4 \leftarrow \phi(x_2, x_3)$
$y_4 \leftarrow \phi(y_2, y_3)$
$z_2 \leftarrow \phi(z_1, z_1)$
$y_5 \leftarrow x_4*2$
$z_3 \leftarrow z_2+x_4$

dead

Better building techniques

The naïve technique just presented works, in the sense that the resulting program is in SSA form and is equivalent to the original one.

However, it introduces too many ɸ-functions – some dead, some redundant – to be useful in practice. It builds the **maximal** SSA form.

We will examine better techniques later, but to understand them we must first introduce the notion of dominance in a CFG.

Dominance

Dominance

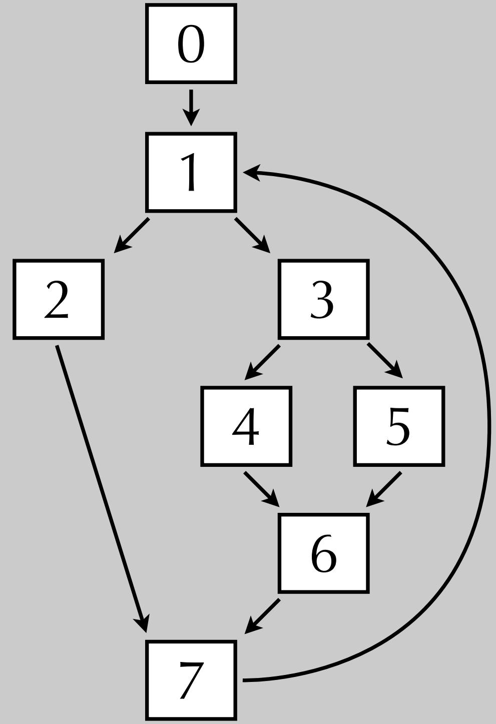
In a control-flow graph, a node n_1 **dominates** a node n_2 if all paths from the start node to n_2 pass through n_1.

By definition, the domination relation is reflexive, that is a node n always dominates itself. We then say that node *n1* **strictly dominates** n_2 if n_1 dominates n_2 and $n_1 \neq n_2$.

The **immediate dominator** of a node n is the strict dominator of n closest to n.

Dominance example

CFG



Dominance

Node	Dominators
0	{ 0 }
1	{ **0**, 1 }
2	{ 0, **1**, 2 }
3	{ 0, **1**, 3 }
4	{ 0, 1, **3**, 4 }
5	{ 0, 1, **3**, 5 }
6	{ 0, 1, **3**, 6 }
7	{ 0, **1**, 7 }

(immediate dominator in bold)

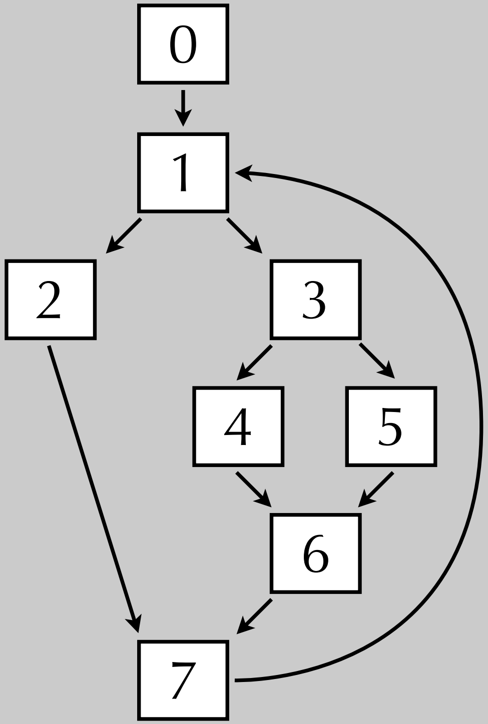
Dominator tree

The **dominator tree** is a tree representing the dominance relation.

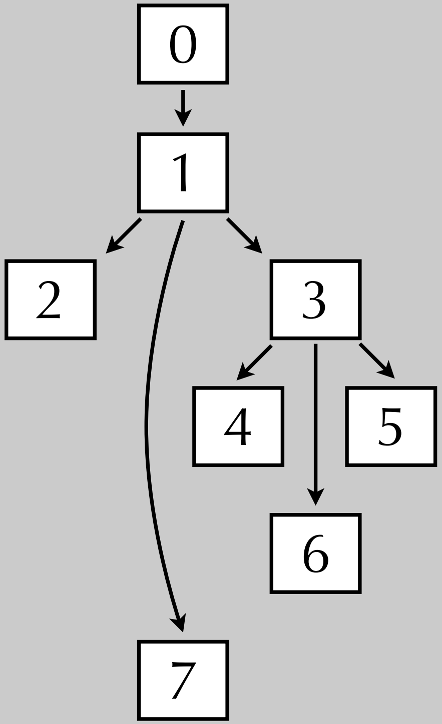
The nodes of the tree are the nodes of the CFG, and a node n_1 is a parent of a node n_2 if and only if n_1 is the immediate dominator of n_2.

Dominator tree example

CFG



Dominator tree



Computing dominance

Dominance can be computed using data-flow analysis.

To each node n of the CFG we attach a variable v_n giving the set of nodes that dominate n. The value of v_n is given by the following equation:

$$v_n = \{ n \} \cup (v_{p1} \cap v_{p2} \cap \ldots \cap v_{pk})$$

where p_1, …, p_k are the predecessors of n.

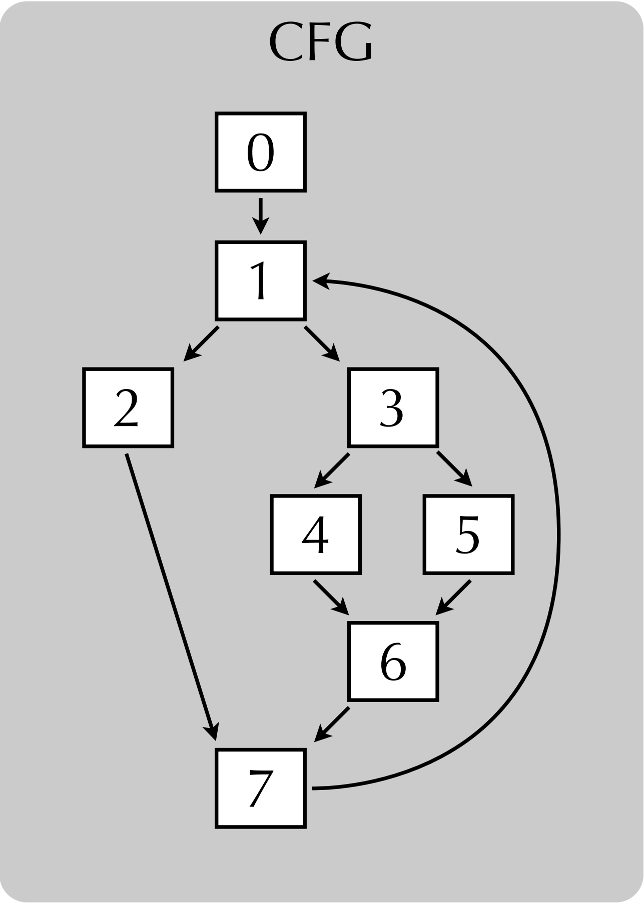
Dominance frontier

The **dominance frontier** of a node *n* – written DF(*n*) – is the set of all nodes *m* such that *n* dominates a predecessor of *m*, but does not strictly dominates *m* itself.

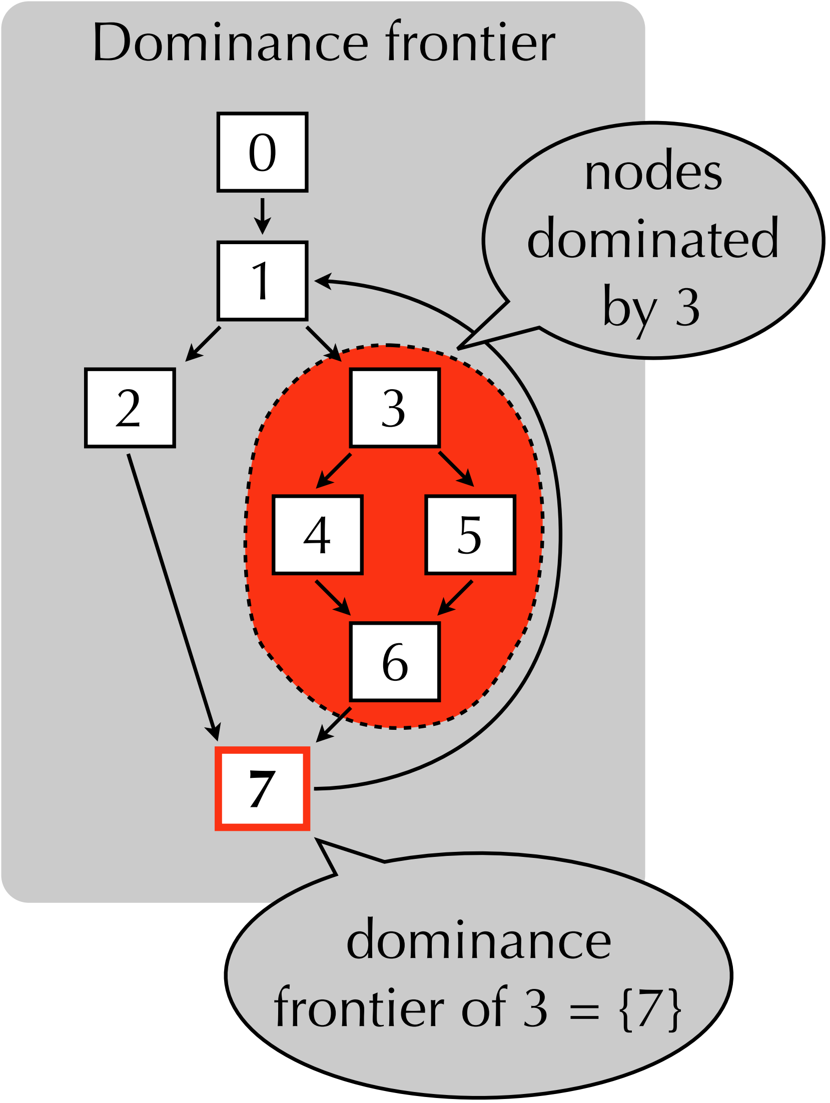
Informally, the dominance frontier of *n* contains the first nodes that are reachable from *n* but are not strictly dominated by *n*.

Dominance frontier example

CFG



Dominance frontier



nodes dominated by 3

dominance frontier of 3 = {7}

Dominance property of SSA

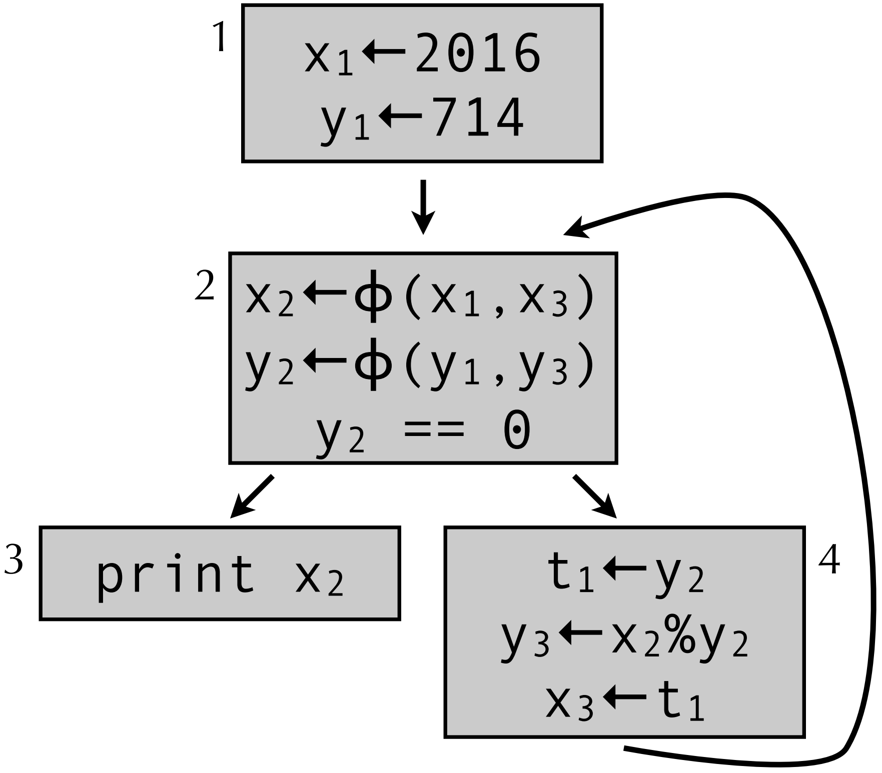
A program is said to be in **strict SSA form** if it satisfies the following **dominance property**:

All uses of a variable are dominated by its (single) definition.

Transformations (*e.g.* optimizations) on programs in SSA form often assume that the input program is in strict form, and must preserve this property.

Dominance and ϕ-functions

In our example, uses of x_3 and y_3 in the ϕ-functions of block 2 apparently violate the dominance property. This is an illusion, however, as they will be used only when coming from block 4.



1

$x_1 \leftarrow 2016$

$y_1 \leftarrow 714$

2

$x_2 \leftarrow \phi(x_1, x_3)$

$y_2 \leftarrow \phi(y_1, y_3)$

$y_2 == 0$

3

print x_2

4

$t_1 \leftarrow y_2$

$y_3 \leftarrow x_2 \% y_2$

$x_3 \leftarrow t_1$

Building SSA form

Minimal SSA form

The naïve technique to build SSA form presented earlier inserts ϕ-functions for every variable at the beginning of every join point.

Using dominance information, it is possible to do better, and compute **minimal** SSA form: for each definition of a variable *x* in a node *n*, insert a ϕ-function for *x* in all nodes of DF(*n*).

Notice that the inserted ϕ-functions are definitions, and can therefore force the insertion of more ϕ-functions.

Improving on minimal SSA

Reminder: the naïve technique to build SSA form presented at the beginning computes maximal SSA form.

The better technique just presented computes minimal SSA form.

Unfortunately, minimal SSA form is not necessarily optimal, and can contain dead ϕ-functions. To solve that problem, improved techniques have been developed to build semi-pruned – which is still not optimal – and pruned SSA form.

Semi-pruned SSA form

Observation: a variable that is only live in a single node can never have a live ϕ-function.

Therefore, the minimal technique can be further refined by first computing the set of **global names** – defined as the names that are live across more than one node – and producing ϕ-functions for these names only.

This is called **semi-pruned SSA form**.

Building semi-pruned SSA form

Like the naïve technique to build maximal SSA form, the algorithm to build semi-pruned SSA form is composed of two phases:

1. ϕ-functions are inserted for global names, according to dominance information,
2. variables are renamed.

Phase 1: inserting ϕ-functions

Before inserting ϕ-functions, the set *G* of global names must be computed. Once this is done, insertion of ϕ-functions is done as follows:

```
for each name $x$ in $G$
  work list = all nodes in which $x$ is defined
  for each node $n$ in work list
    for each node $m$ in DF($n$)
      insert a ϕ-function for $x$ in $m$
      work list = work list ∪ { $m$ }
```

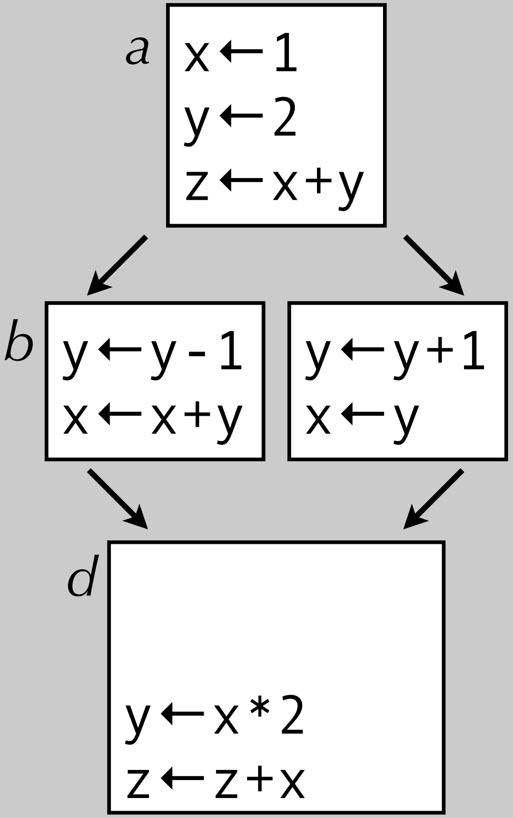
Phase 2: renaming variables

Renaming is done by a pre-order traversal of the dominator tree, as follows:

```
for each node n in the dominator tree
  rename definitions and uses of variables in n
  rename ϕ-functions parameters corresponding to n in all
    successors of n in the CFG.
```

Example: phase 1

CFG



DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

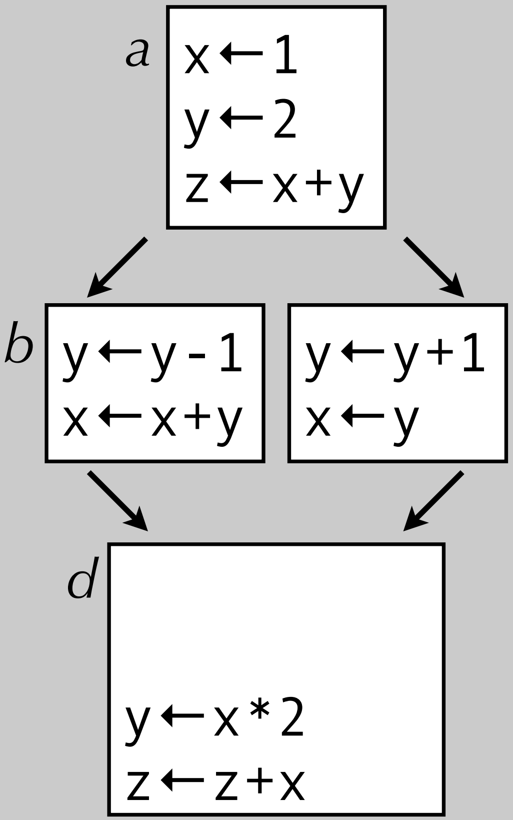
Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

Example: phase 1

CFG



DF(*a*) = DF(*d*) = {}
DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

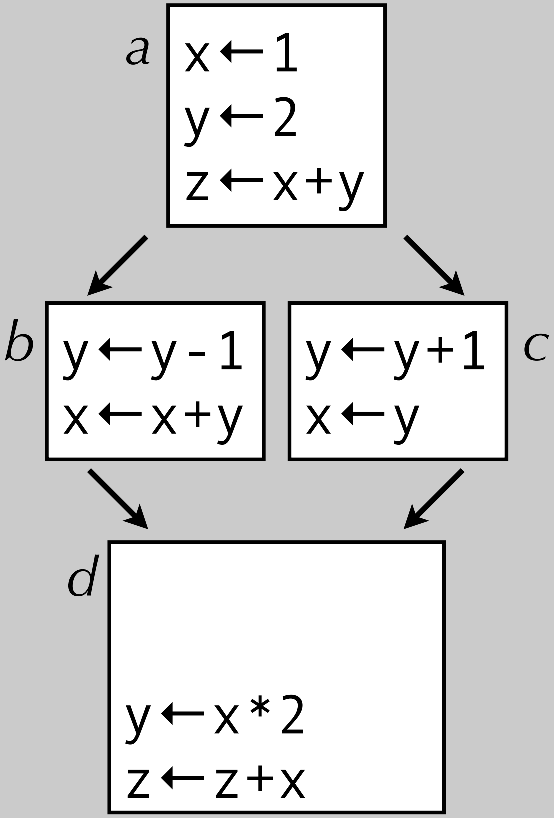
```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

Example: phase 1

CFG



a

x←1
y←2
z←x+y

b

y←y-1
x←x+y

c

y←y+1
x←y

d

y←x*2
z←z+x

DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

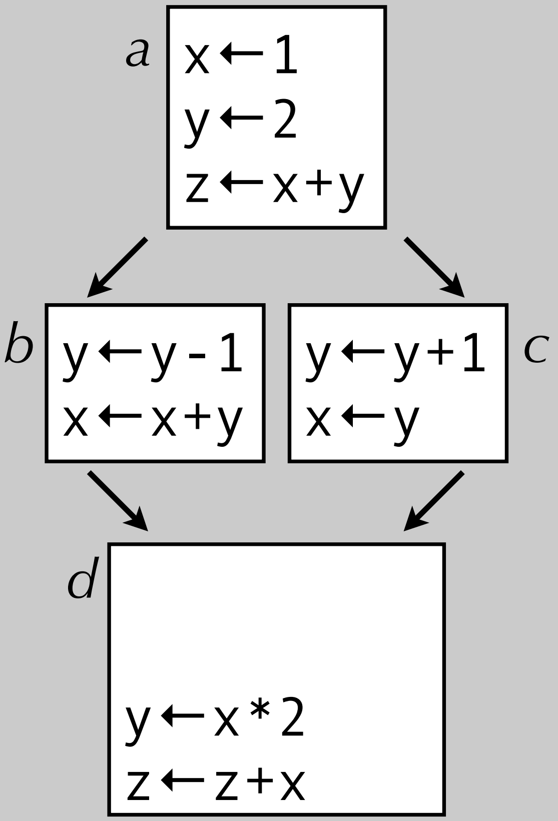
Result

name x

wrk lst	**ϕ-fun.**

Example: phase 1

CFG



a

x←1
y←2
z←x+y

b

y←y-1
x←x+y

c

y←y+1
x←y

d

y←x*2
z←z+x

DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

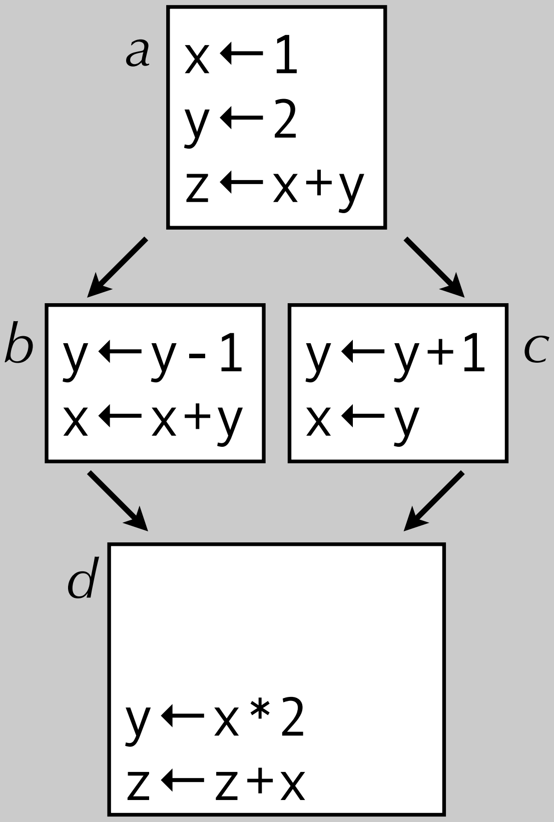
Result

name x

wrk lst	**ϕ-fun.**
[**a**,*b*,*c*]	

Example: phase 1

CFG



a: x←1, y←2, z←x+y

b: y←y-1, x←x+y

c: y←y+1, x←y

d: y←x*2, z←z+x

DF(*a*) = DF(*d*) = {}
DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

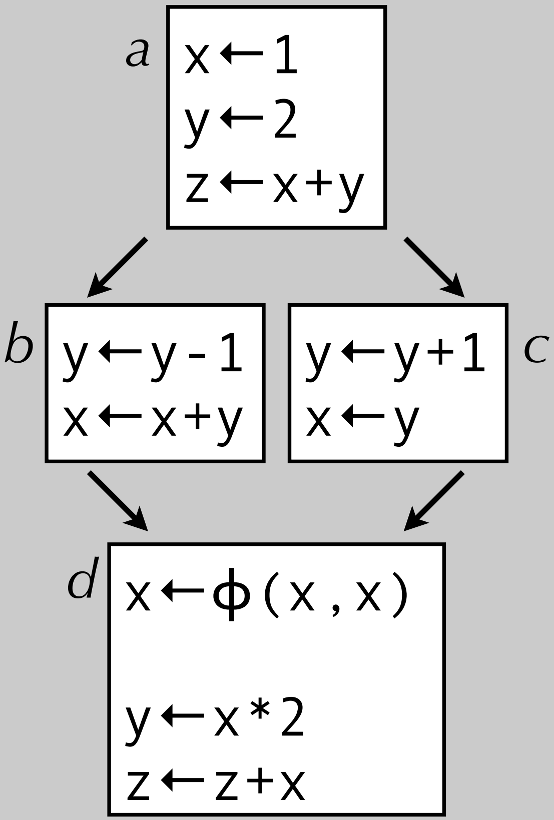
Result

name x

wrk lst	ϕ-fun.
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*

Example: phase 1

CFG



DF(*a*) = DF(*d*) = {}
DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

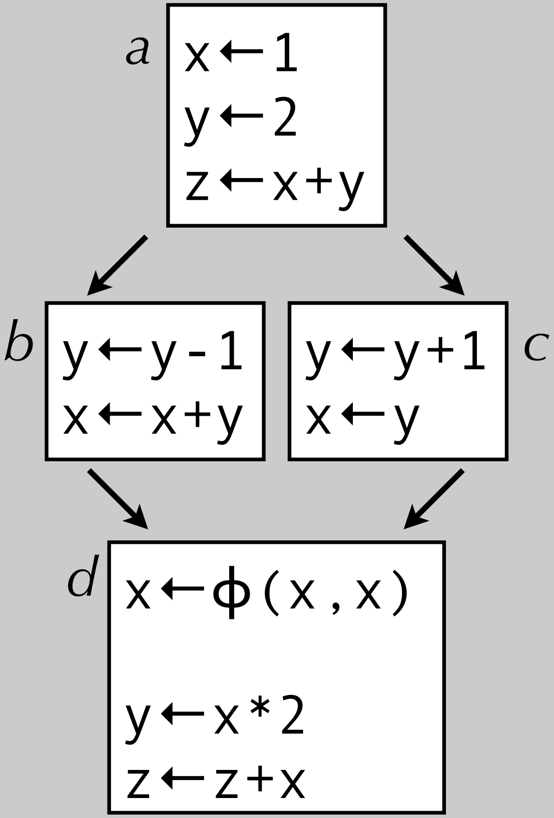
Result

name x

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*

Example: phase 1

CFG



a

x←1
y←2
z←x+y

b

y←y-1
x←x+y

c

y←y+1
x←y

d

x←ϕ(x,x)
y←x*2
z←z+x

DF(*a*) = DF(*d*) = {}
DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name $x$ in {x,y,z}
  work list = all nodes in which $x$ is defined
  for each node $n$ in work list
    for each node $m$ in DF($n$)
      insert a ϕ-function for $x$ in $m$
      work list = work list ∪ { $m$ }
```

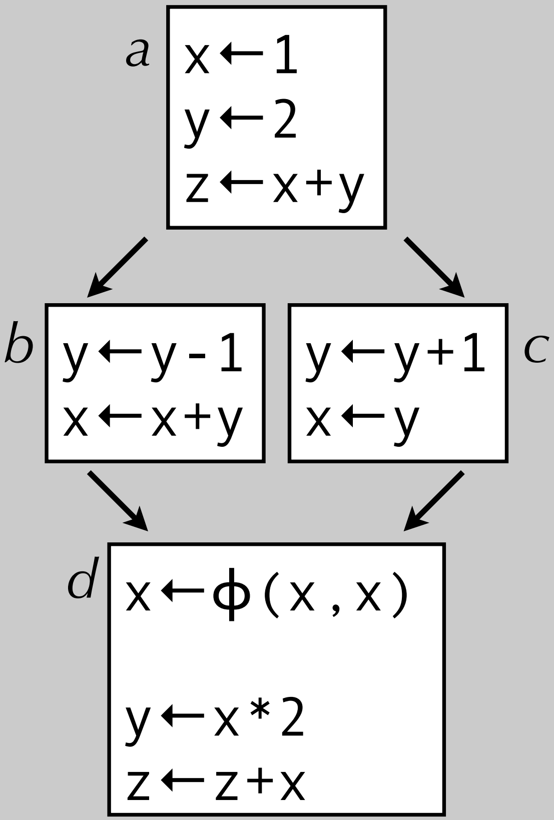
Result

name x

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*

Example: phase 1

CFG



```
a:
x←1
y←2
z←x+y

b:
y←y-1
x←x+y

c:
y←y+1
x←y

d:
x←ϕ(x,x)

y←x*2
z←z+x
```

DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

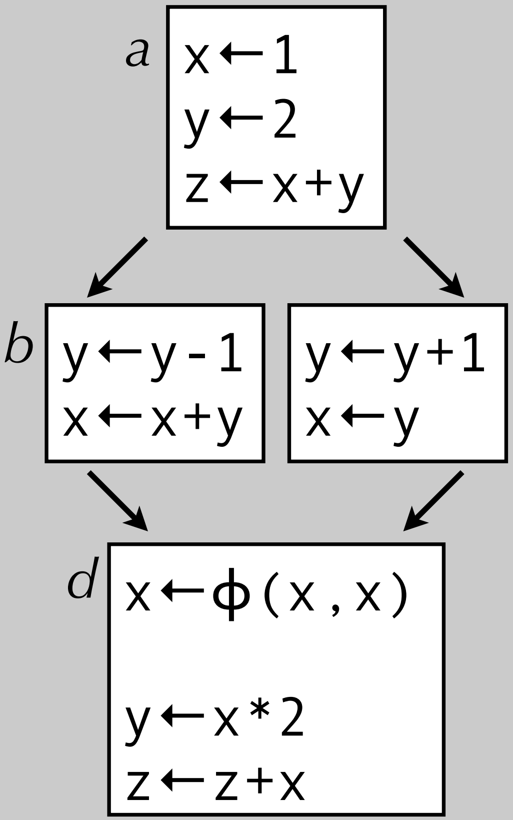
Result

name x

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	

Example: phase 1

CFG



a

x←1
y←2
z←x+y

b

y←y-1
x←x+y

c

y←y+1
x←y

d

x←ϕ(x,x)

y←x*2
z←z+x

DF(*a*) = DF(*d*) = {}
DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name $x$ in {x,y,z}
  work list = all nodes in which $x$ is defined
  for each node $n$ in work list
    for each node $m$ in DF($n$)
      insert a ϕ-function for $x$ in $m$
      work list = work list ∪ { $m$ }
```

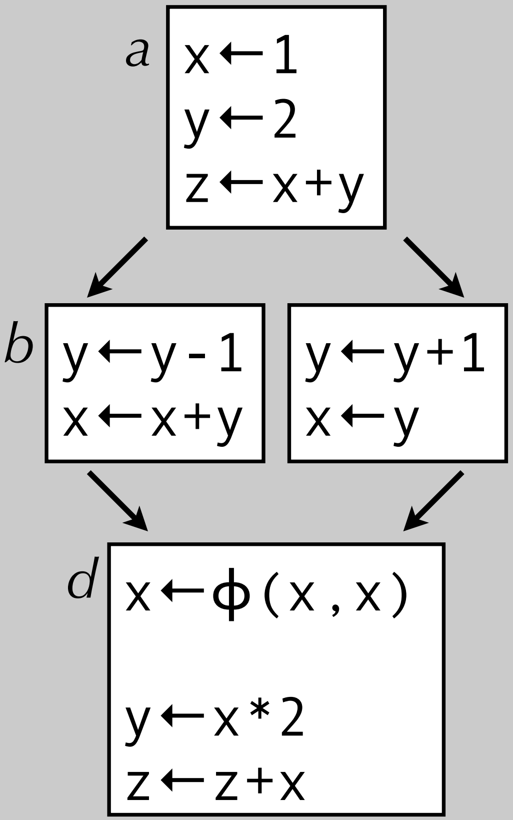
Result

name x

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

Example: phase 1

CFG



DF(*a*) = DF(*d*) = {}
DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
 work list = all nodes in which x is defined
 for each node n in work list
  for each node m in DF(n)
   insert a ϕ-function for x in m
   work list = work list ∪ { m }
```

Result

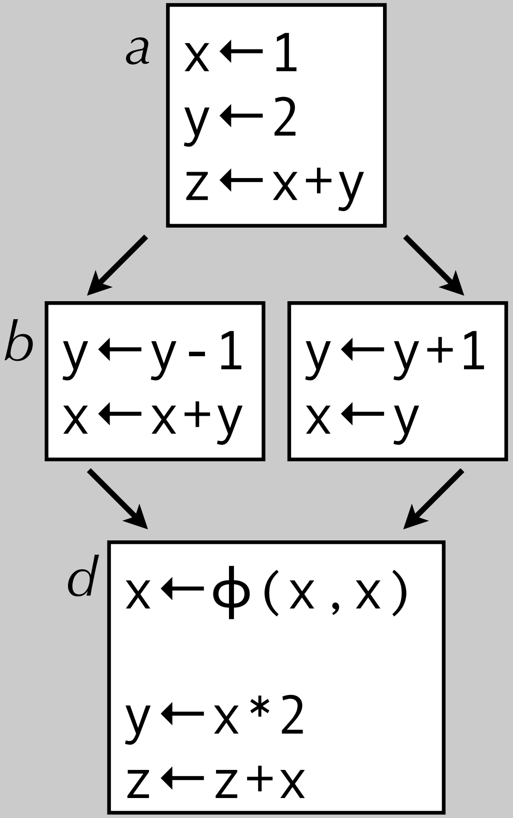
name x

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

Example: phase 1

CFG



a

```
x←1
y←2
z←x+y
```

b

```
y←y-1
x←x+y
```

c

```
y←y+1
x←y
```

d

```
x←ϕ(x,x)

y←x*2
z←z+x
```

DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

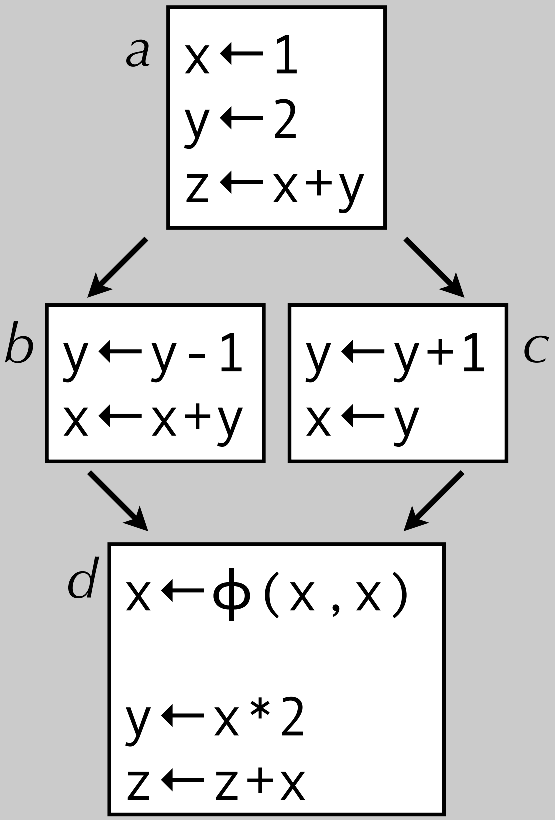
wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

wrk lst	**ϕ-fun.**

Example: phase 1

CFG



a
x←1
y←2
z←x+y

b
y←y-1
x←x+y

c
y←y+1
x←y

d
x←ϕ(x,x)

y←x*2
z←z+x

DF(*a*) = DF(*d*) = {}
DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

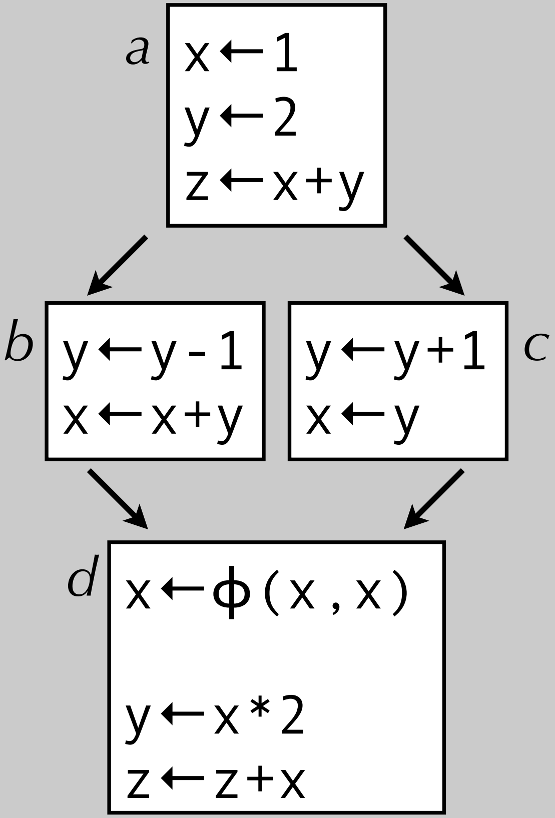
wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*,*d*]	

Example: phase 1

CFG



a
x←1
y←2
z←x+y

b
y←y-1
x←x+y

c
y←y+1
x←y

d
x←ϕ(x,x)

y←x*2
z←z+x

DF(*a*) = DF(*d*) = {}
DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

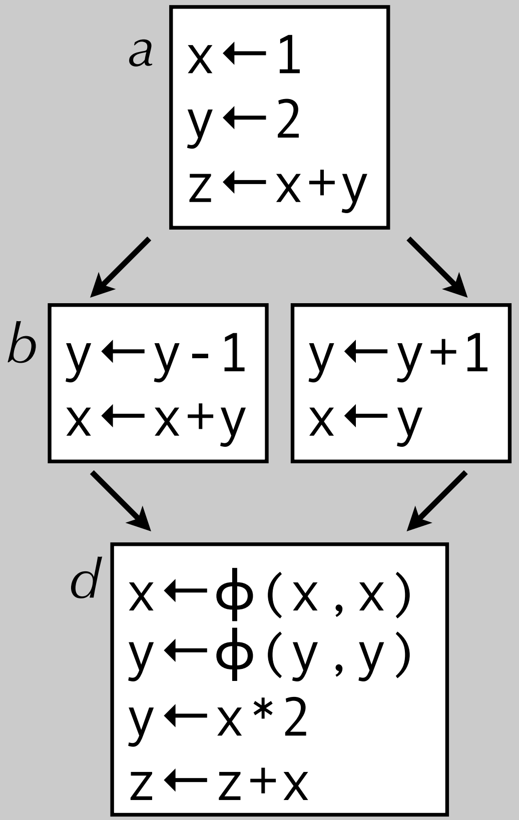
wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*,*d*]	
[***b***,*c*,*d*]	for y in *d*

Example: phase 1

CFG



DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

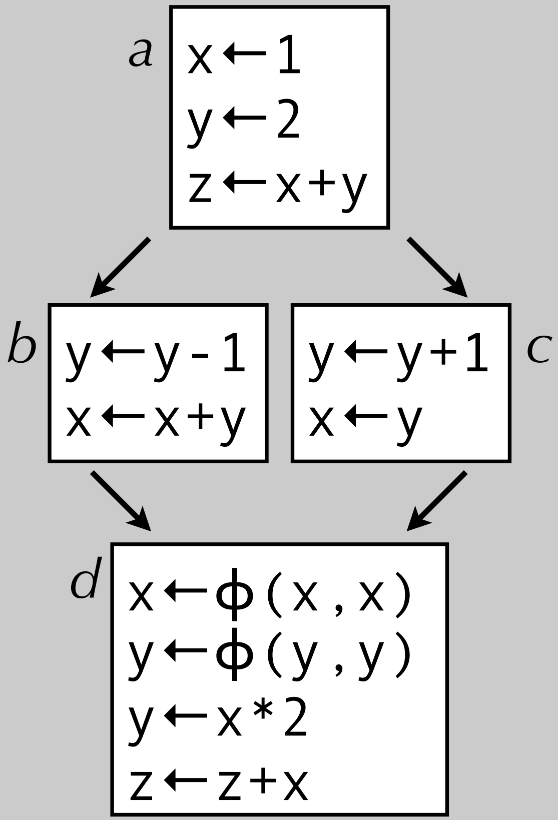
wrk lst	**ϕ-fun.**
[***a***,*b*,c]	
[***b***,c]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

wrk lst	**ϕ-fun.**
[***a***,*b*,c,*d*]	
[***b***,c,*d*]	for y in *d*

Example: phase 1

CFG



a

```
x←1
y←2
z←x+y
```

b

```
y←y-1
x←x+y
```

c

```
y←y+1
x←y
```

d

```
x←ϕ(x,x)
y←ϕ(y,y)
y←x*2
z←z+x
```

DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

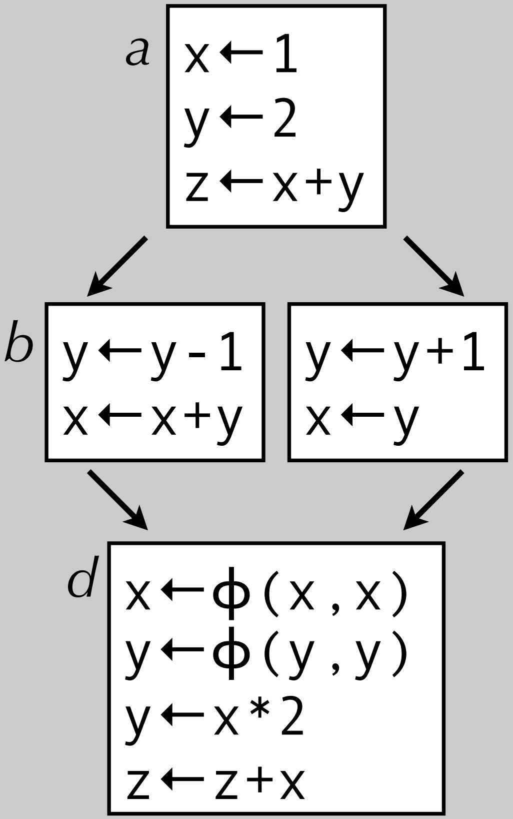
wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*,*d*]	
[***b***,*c*,*d*]	for y in *d*
[***c***,*d*]	for y in *d*

Example: phase 1

CFG



DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

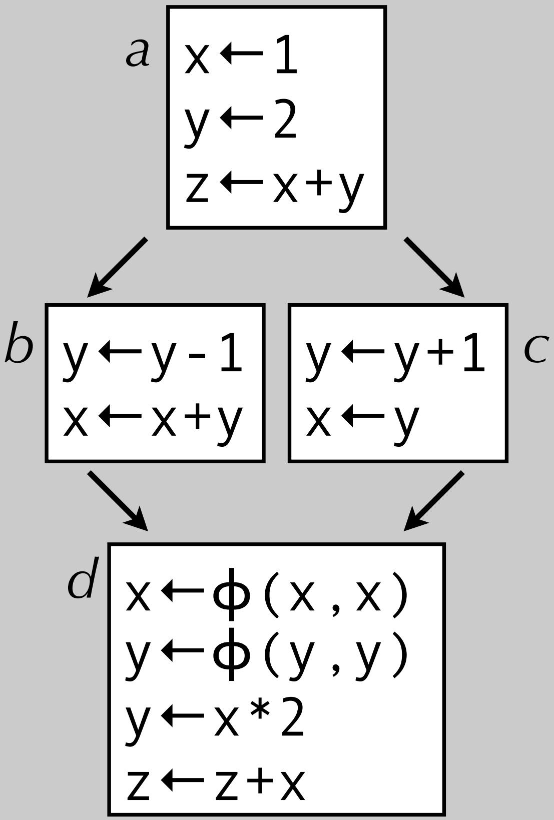
wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*,*d*]	
[***b***,*c*,*d*]	for y in *d*
[***c***,*d*]	for y in *d*
[***d***]	

Example: phase 1

CFG



DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

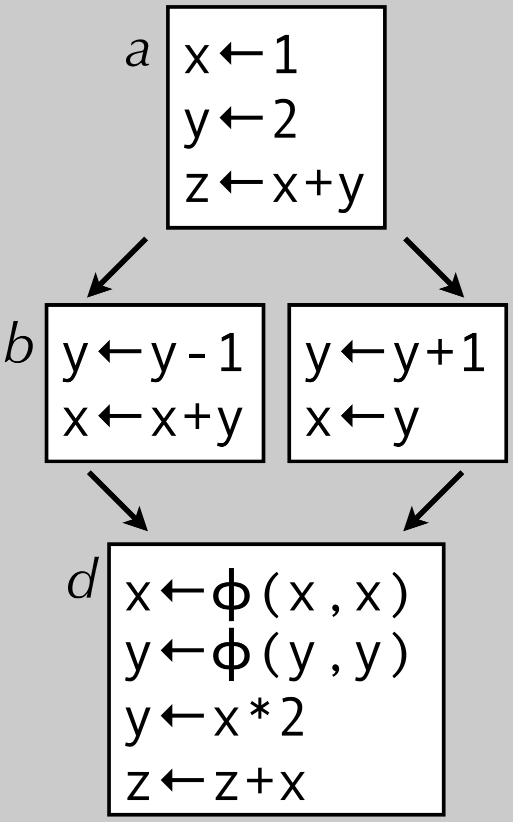
wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*,*d*]	
[***b***,*c*,*d*]	for y in *d*
[***c***,*d*]	for y in *d*
[***d***]	
[]	

Example: phase 1

CFG



DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

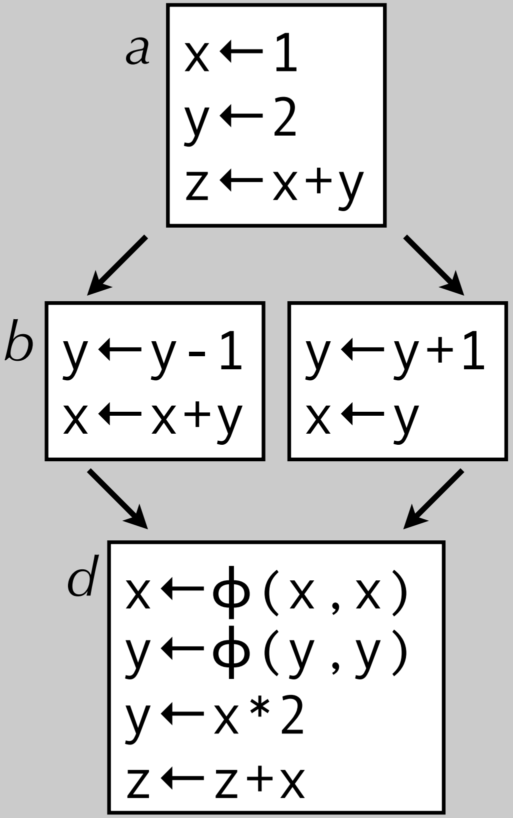
name y

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*,*d*]	
[***b***,*c*,*d*]	for y in *d*
[***c***,*d*]	for y in *d*
[***d***]	
[]	

name z

Example: phase 1

CFG



a

x←1
y←2
z←x+y

b

y←y-1
x←x+y

c

y←y+1
x←y

d

x←ϕ(x,x)
y←ϕ(y,y)
y←x*2
z←z+x

DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name $x$ in {x,y,z}
  work list = all nodes in which $x$ is defined
  for each node $n$ in work list
    for each node $m$ in DF($n$)
      insert a ϕ-function for $x$ in $m$
      work list = work list ∪ { $m$ }
```

Result

name x

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

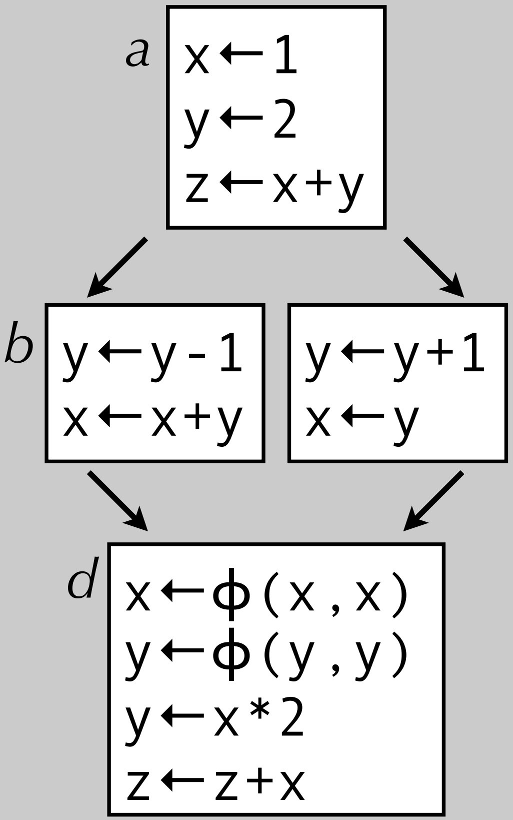
wrk lst	**ϕ-fun.**
[***a***,*b*,*c*,*d*]	
[***b***,*c*,*d*]	for y in *d*
[***c***,*d*]	for y in *d*
[***d***]	
[]	

name z

wrk lst	**ϕ-fun.**

Example: phase 1

CFG



DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

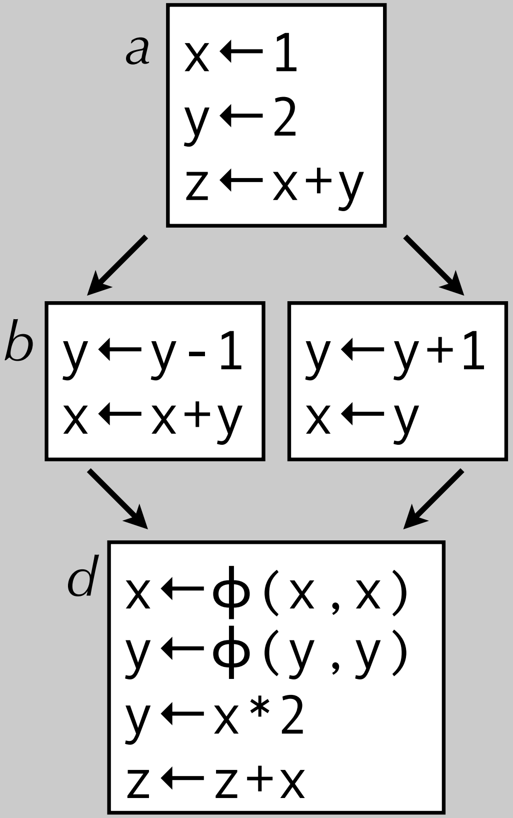
wrk lst	**ϕ-fun.**
[***a***,*b*,*c*,*d*]	
[***b***,*c*,*d*]	for y in *d*
[***c***,*d*]	for y in *d*
[***d***]	
[]	

name z

wrk lst	**ϕ-fun.**
[***a***,*d*]	

Example: phase 1

CFG



DF(*a*) = DF(*d*) = {}
DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

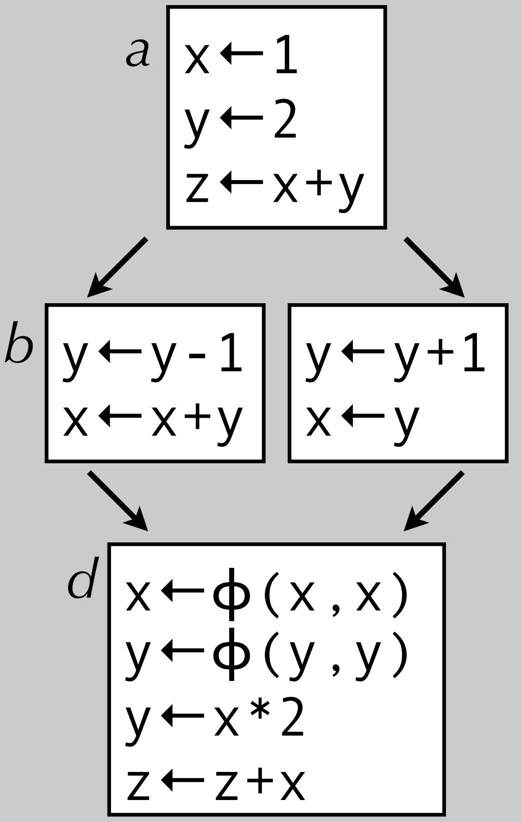
wrk lst	**ϕ-fun.**
[***a***,*b*,*c*,*d*]	
[***b***,*c*,*d*]	for y in *d*
[***c***,*d*]	for y in *d*
[***d***]	
[]	

name z

wrk lst	**ϕ-fun.**
[***a***,*d*]	
[***d***]	

Example: phase 1

CFG



DF(*a*) = DF(*d*) = {}

DF(*b*) = DF(*c*) = {*d*}

Algorithm (phase 1)

```
for each name x in {x,y,z}
  work list = all nodes in which x is defined
  for each node n in work list
    for each node m in DF(n)
      insert a ϕ-function for x in m
      work list = work list ∪ { m }
```

Result

name x

wrk lst	**ϕ-fun.**
[***a***,*b*,*c*]	
[***b***,*c*]	for x in *d*
[***c***,*d*]	for x in *d*
[***d***]	
[]	

name y

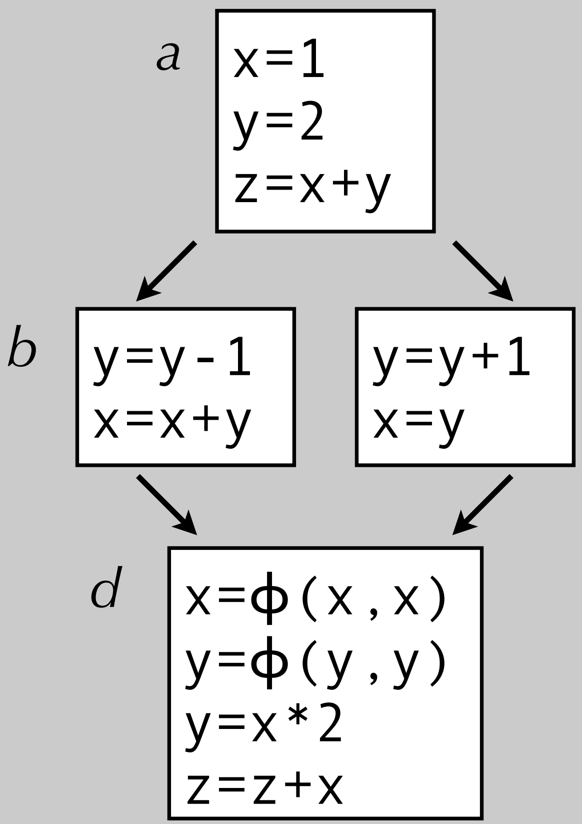
wrk lst	**ϕ-fun.**
[***a***,*b*,*c*,*d*]	
[***b***,*c*,*d*]	for y in *d*
[***c***,*d*]	for y in *d*
[***d***]	
[]	

name z

wrk lst	**ϕ-fun.**
[***a***,*d*]	
[***d***]	
[]	

Example: phase 2

CFG



a

```
x=1
y=2
z=x+y
```

b

```
y=y-1
x=x+y
```

c

```
y=y+1
x=y
```

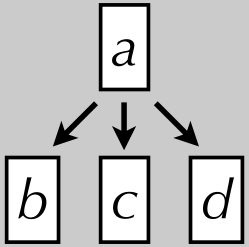
d

```
x=ϕ(x,x)
y=ϕ(y,y)
y=x*2
z=z+x
```

Algorithm (phase 2)

```
for each node n in the dominator tree (pre-order)
  rename definitions and uses of variables in n
  rename ϕ-functions parameters corresponding
    to n in all successors of n in the CFG.
```

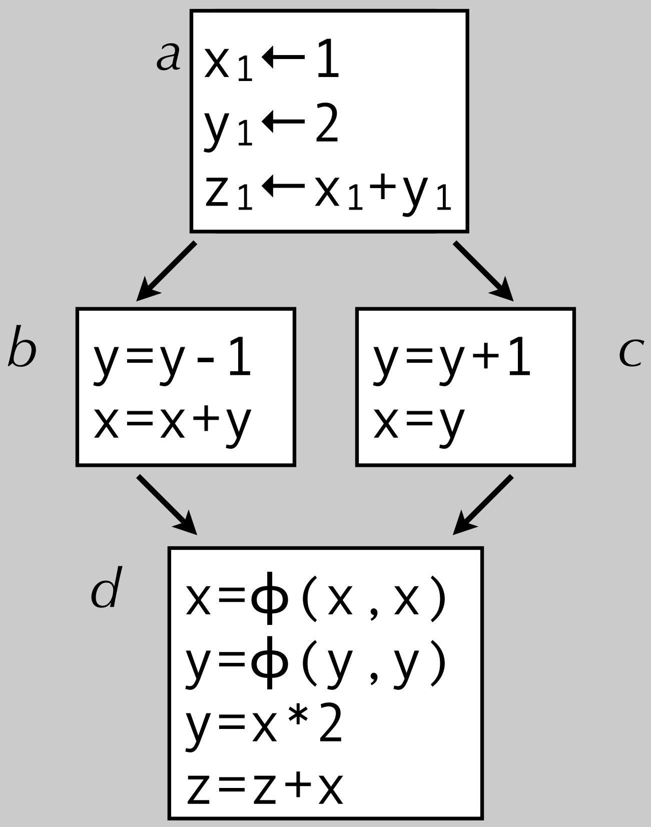
Dominator tree



chosen pre-order:
a,*b*,*d*,*c*

Example: phase 2

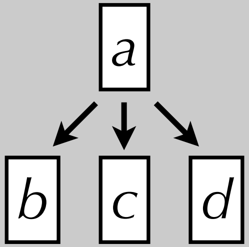
CFG



Algorithm (phase 2)

```
for each node n in the dominator tree (pre-order)
  rename definitions and uses of variables in n
  rename ϕ-functions parameters corresponding
    to n in all successors of n in the CFG.
```

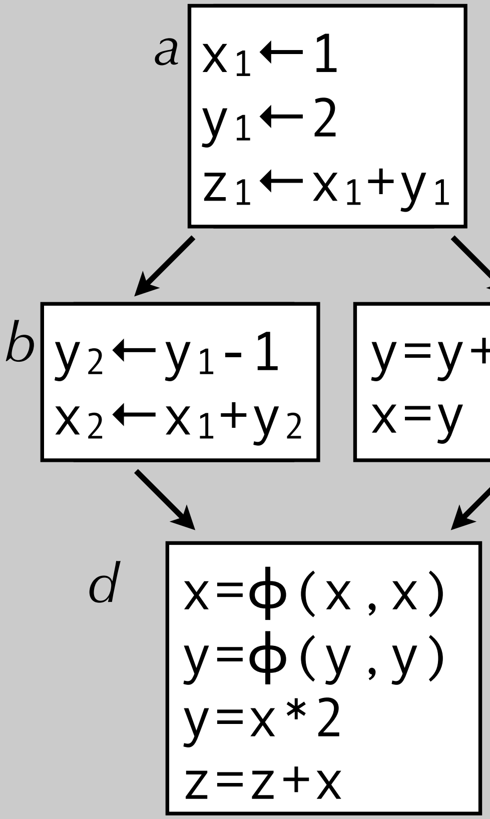
Dominator tree



chosen pre-order:
a,b,d,c

Example: phase 2

CFG



$x_1 \leftarrow 1$
$y_1 \leftarrow 2$
$z_1 \leftarrow x_1 + y_1$

$y_2 \leftarrow y_1 - 1$
$x_2 \leftarrow x_1 + y_2$

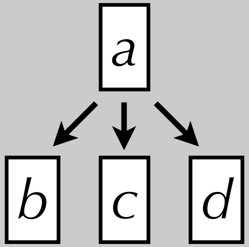
y=y+1
x=y

x=ϕ(x,x)
y=ϕ(y,y)
y=x*2
z=z+x

Algorithm (phase 2)

```
for each node $n$ in the dominator tree (pre-order)
  rename definitions and uses of variables in $n$
  rename ϕ-functions parameters corresponding
    to $n$ in all successors of $n$ in the CFG.
```

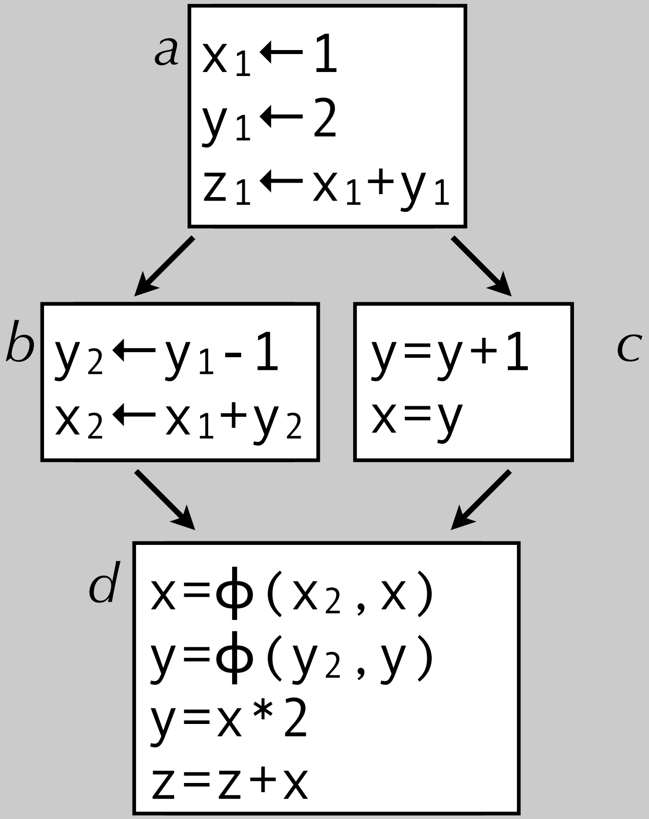
Dominator tree



chosen pre-order:
a,b,d,c

Example: phase 2

CFG



$x_1 \leftarrow 1$
$y_1 \leftarrow 2$
$z_1 \leftarrow x_1 + y_1$

$y_2 \leftarrow y_1 - 1$
$x_2 \leftarrow x_1 + y_2$

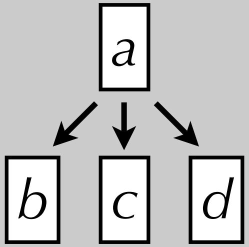
$y=y+1$
$x=y$

$x=\phi(x_2,x)$
$y=\phi(y_2,y)$
$y=x*2$
$z=z+x$

Algorithm (phase 2)

```
for each node $n$ in the dominator tree (pre-order)
  rename definitions and uses of variables in $n$
  rename $\phi$-functions parameters corresponding
    to $n$ in all successors of $n$ in the CFG.
```

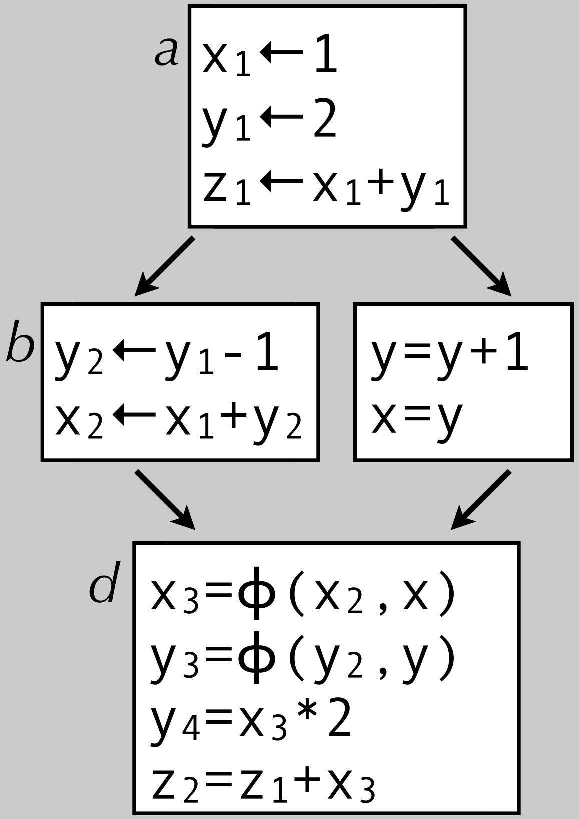
Dominator tree



chosen pre-order:
a,b,d,c

Example: phase 2

CFG



a

$x_1 \leftarrow 1$

$y_1 \leftarrow 2$

$z_1 \leftarrow x_1 + y_1$

b

$y_2 \leftarrow y_1 - 1$

$x_2 \leftarrow x_1 + y_2$

c

y=y+1

x=y

d

$x_3 = \phi(x_2, x)$

$y_3 = \phi(y_2, y)$

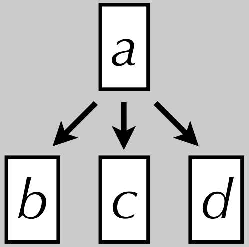
$y_4 = x_3 * 2$

$z_2 = z_1 + x_3$

Algorithm (phase 2)

```
for each node n in the dominator tree (pre-order)
  rename definitions and uses of variables in n
  rename ϕ-functions parameters corresponding
    to n in all successors of n in the CFG.
```

Dominator tree

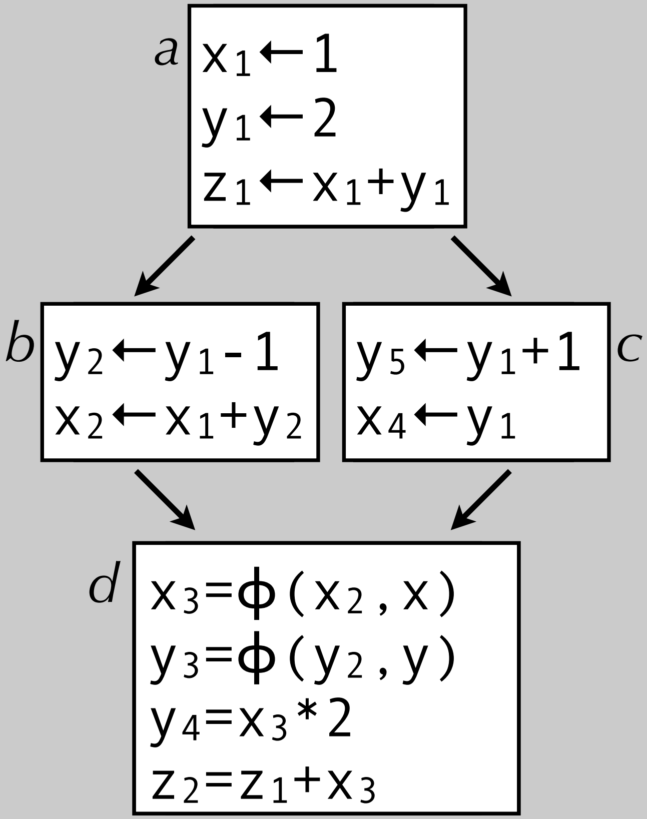


chosen pre-order:

a,b,d,c

Example: phase 2

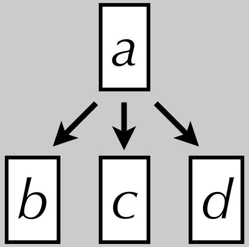
CFG



Algorithm (phase 2)

```
for each node n in the dominator tree (pre-order)
  rename definitions and uses of variables in n
  rename φ-functions parameters corresponding
    to n in all successors of n in the CFG.
```

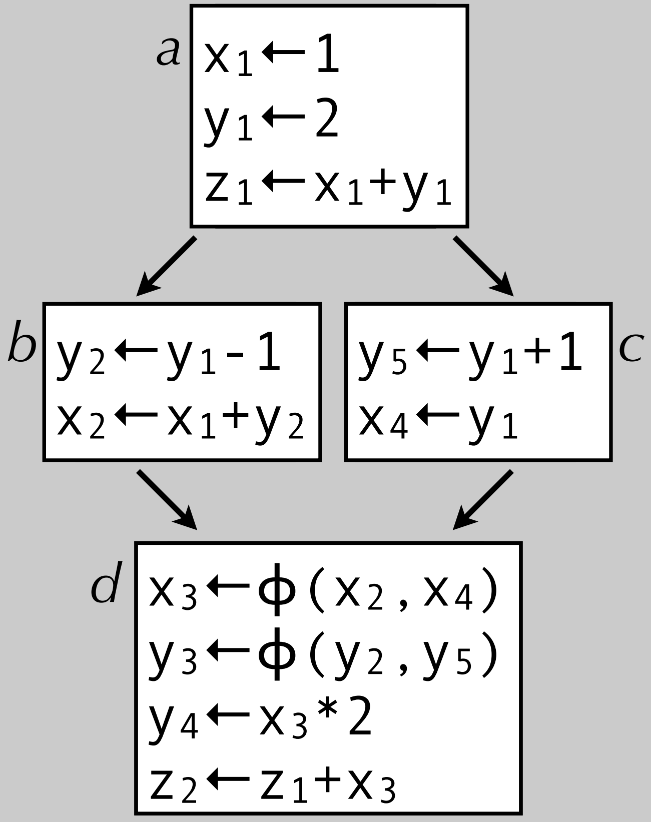
Dominator tree



chosen pre-order:
a,b,d,c

Example: phase 2

CFG

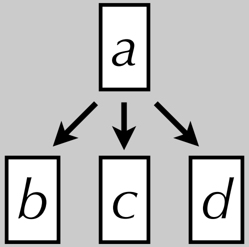


```
a:
x₁←1
y₁←2
z₁←x₁+y₁

b:
y₂←y₁-1
x₂←x₁+y₂

c:
y₅←y₁+1
x₄←y₁

d:
x₃←ϕ(x₂,x₄)
y₃←ϕ(y₂,y₅)
y₄←x₃*2
z₂←z₁+x₃
```

Algorithm (phase 2)

```
for each node n in the dominator tree (pre-order)
  rename definitions and uses of variables in n
  rename ϕ-functions parameters corresponding
    to n in all successors of n in the CFG.
```

Dominator tree



chosen pre-order:
a,b,d,c

Getting out of SSA form

Getting out of SSA form

After the program has been turned into SSA form and the various optimizations performed on that representation, it must be transformed into executable form.

This implies in particular that ϕ-functions must be removed, as they cannot be implemented on standard machines.

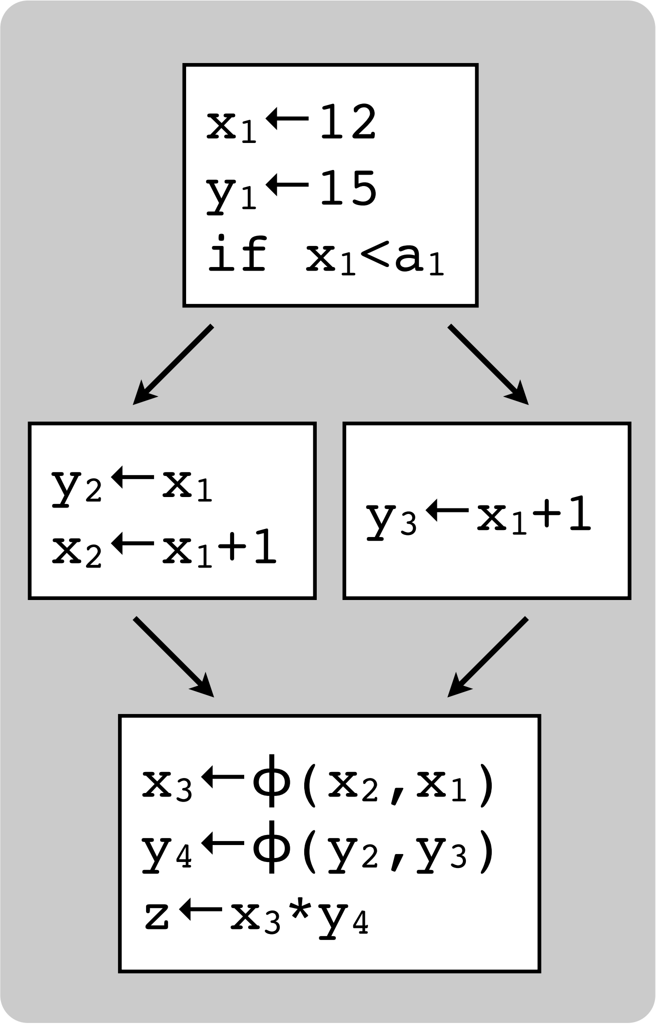
Removing ϕ-functions

First idea: a ϕ-function of the form $x_i = \phi(y_1, \ldots, z_n)$ is removed by inserting appropriate assignments to x_i in all predecessors of the node containing that function.

This will introduce many assignments of the form $x_i \leftarrow y_j$ – *i.e.* `MOVE` instructions – but most of them will be removed later during register allocation, thanks to coalescing.

Unfortunately, as we will see, this naïve technique has two problems, and cannot therefore be used as-is.

Removing ϕ-functions



$x_1 \leftarrow 12$
$y_1 \leftarrow 15$
$\mathtt{if}\ x_1 < a_1$

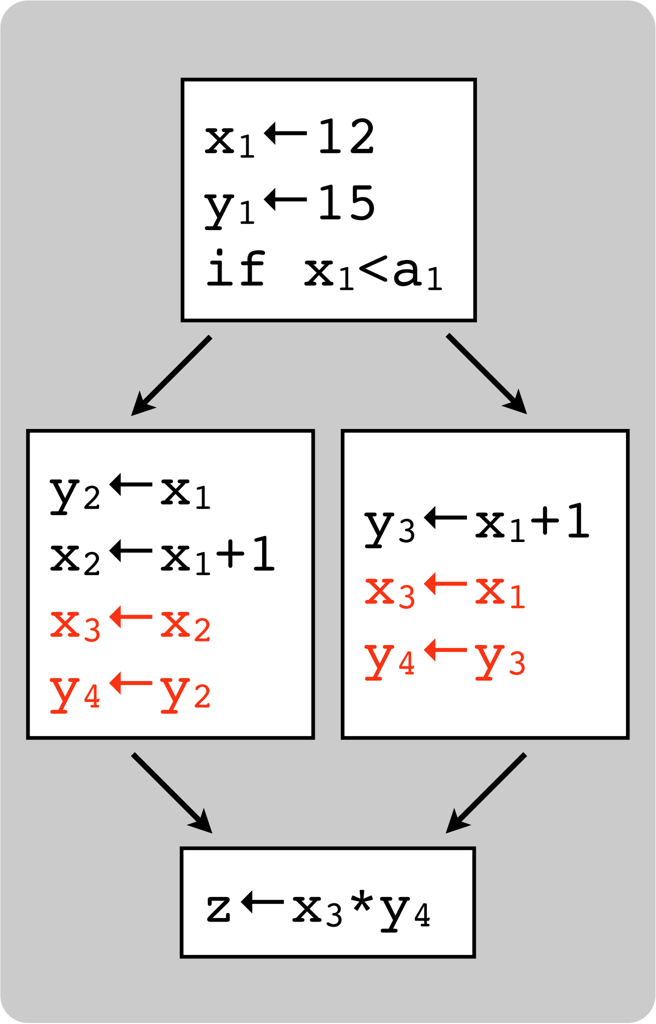
$y_2 \leftarrow x_1$
$x_2 \leftarrow x_1 + 1$

$y_3 \leftarrow x_1 + 1$

$x_3 \leftarrow \phi(x_2, x_1)$
$y_4 \leftarrow \phi(y_2, y_3)$
$z \leftarrow x_3 * y_4$

ϕ-function removal

(In this case, the naïve technique works)



$x_1 \leftarrow 12$
$y_1 \leftarrow 15$
$\mathtt{if}\ x_1 < a_1$

$y_2 \leftarrow x_1$
$x_2 \leftarrow x_1 + 1$
$x_3 \leftarrow x_2$
$y_4 \leftarrow y_2$

$y_3 \leftarrow x_1 + 1$
$x_3 \leftarrow x_1$
$y_4 \leftarrow y_3$

$z \leftarrow x_3 * y_4$

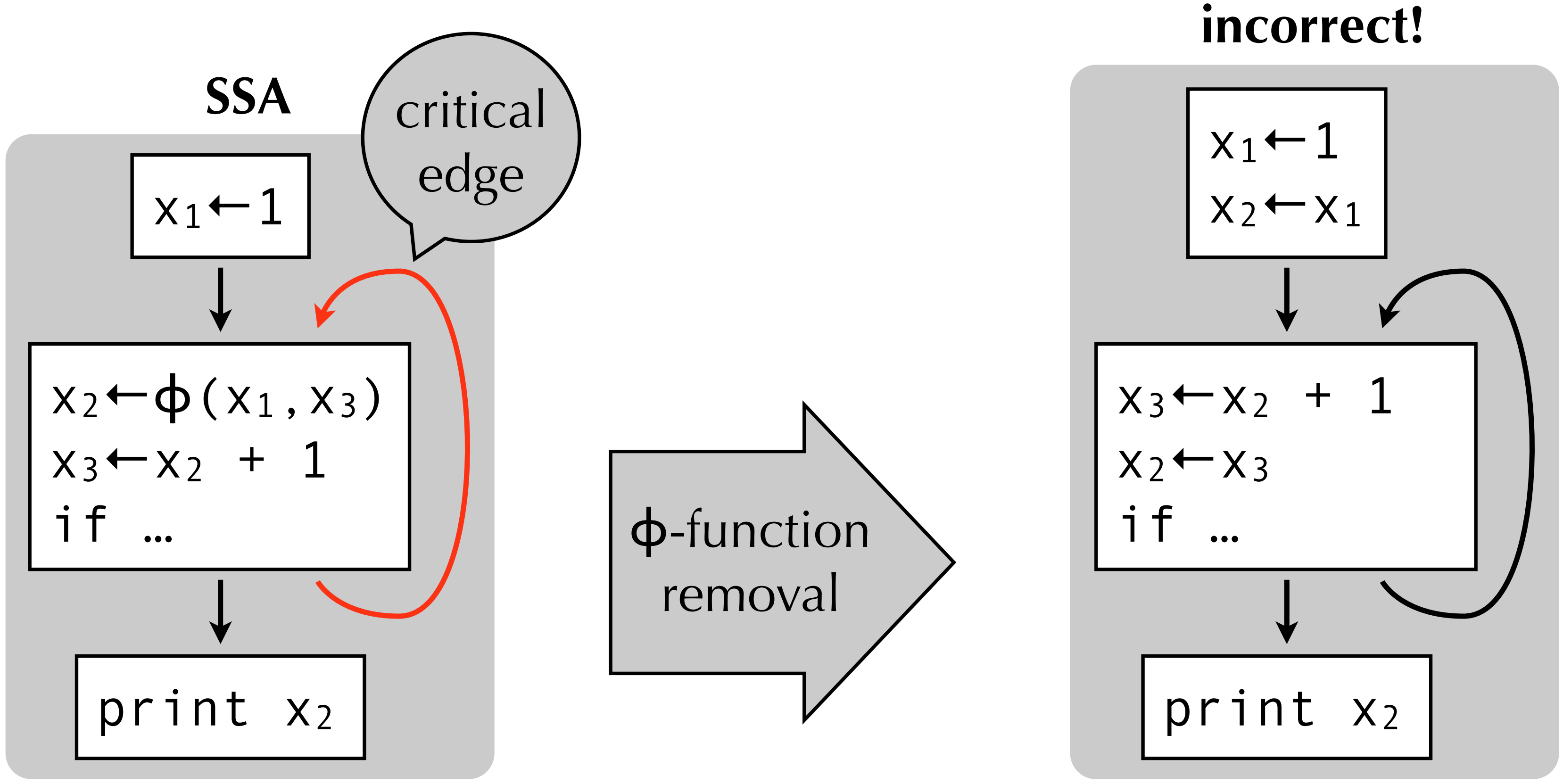
Problem #1: critical edges

CFG edges that go from a node with multiple successors to a node with multiple predecessors are called **critical edges**.

While removing ϕ-functions, the presence of a critical edge from n_1 to n_2 leads to the insertion of useless and sometimes incorrect move instructions in n_1, corresponding to the ϕ-functions of n_2. These should be executed only if control reaches n_2 later, but this is not certain when n_1 executes.

This problem can be solved by **splitting** critical edges, *i.e.* inserting a new node in the middle of them.

Without edge splitting



$x_1 \leftarrow 1$

$x_2 \leftarrow \phi(x_1, x_3)$
$x_3 \leftarrow x_2 + 1$
`if …`

`print` x_2

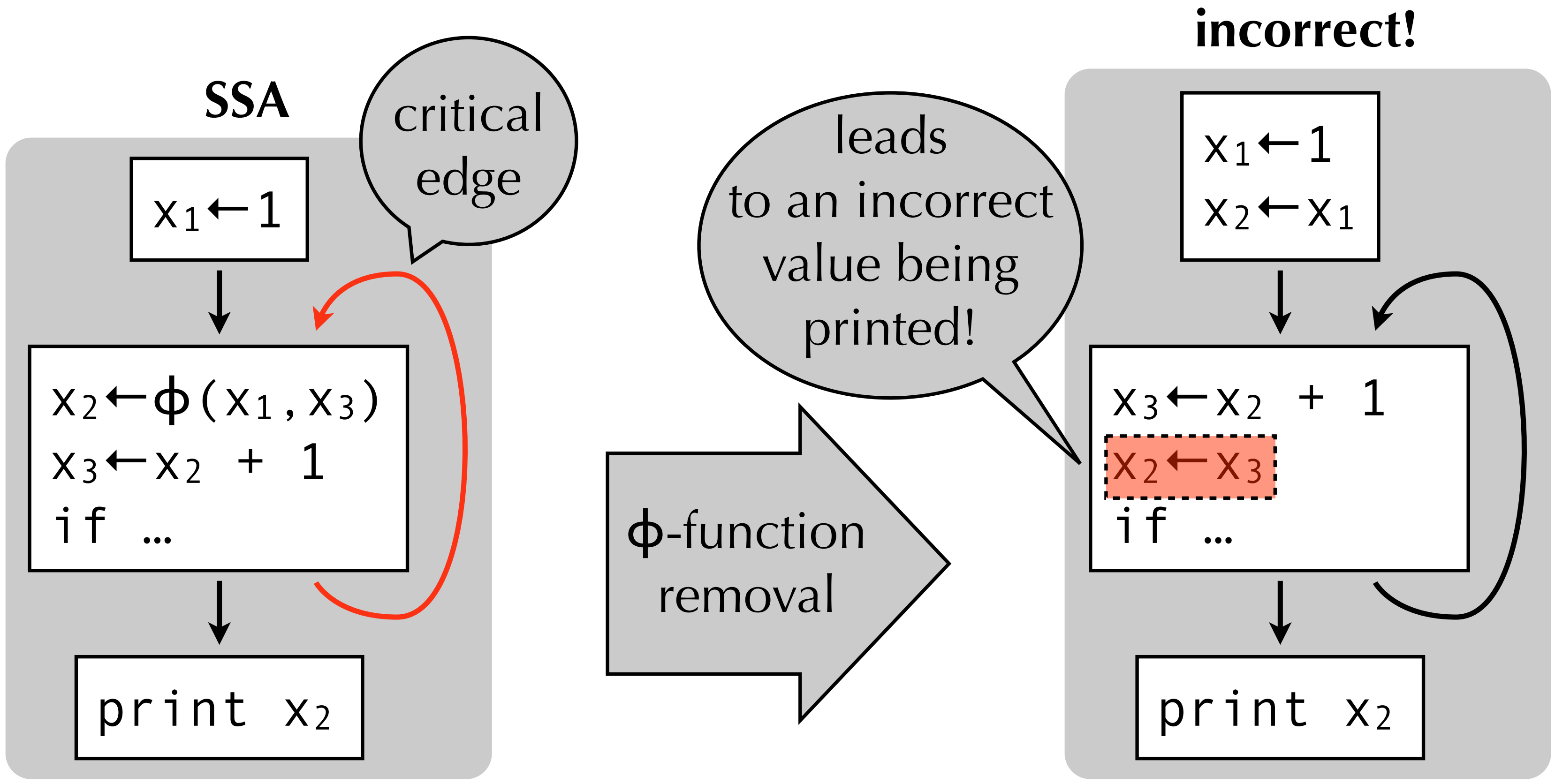
$x_1 \leftarrow 1$
$x_2 \leftarrow x_1$

$x_3 \leftarrow x_2 + 1$
$x_2 \leftarrow x_3$
`if …`

`print` x_2

(This problem is known as *the lost copy problem*.)

Without edge splitting



SSA

$x_1 \leftarrow 1$

$x_2 \leftarrow \phi(x_1, x_3)$
$x_3 \leftarrow x_2 + 1$
$\texttt{if} \ldots$

$\texttt{print } x_2$

critical edge

ϕ-function removal

leads to an incorrect value being printed!

incorrect!

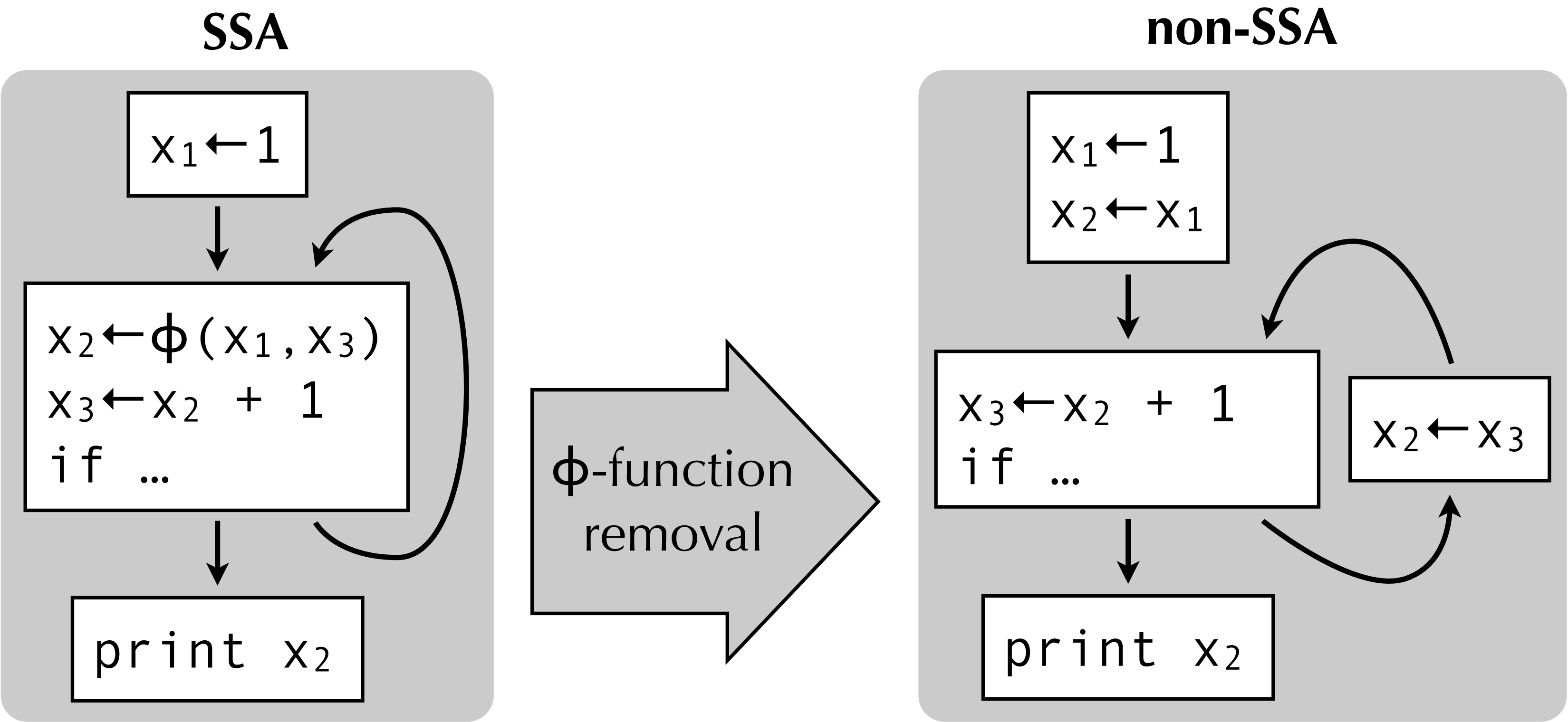
$x_1 \leftarrow 1$
$x_2 \leftarrow x_1$

$x_3 \leftarrow x_2 + 1$
$x_2 \leftarrow x_3$
$\texttt{if} \ldots$

$\texttt{print } x_2$

(This problem is known as *the lost copy problem*.)

With edge splitting



SSA

$x_1 \leftarrow 1$

$x_2 \leftarrow \phi(x_1, x_3)$
$x_3 \leftarrow x_2 + 1$
if …

print x_2

ϕ-function removal

non-SSA

$x_1 \leftarrow 1$
$x_2 \leftarrow x_1$

$x_3 \leftarrow x_2 + 1$
if …

$x_2 \leftarrow x_3$

print x_2

Problem #2: parallel move

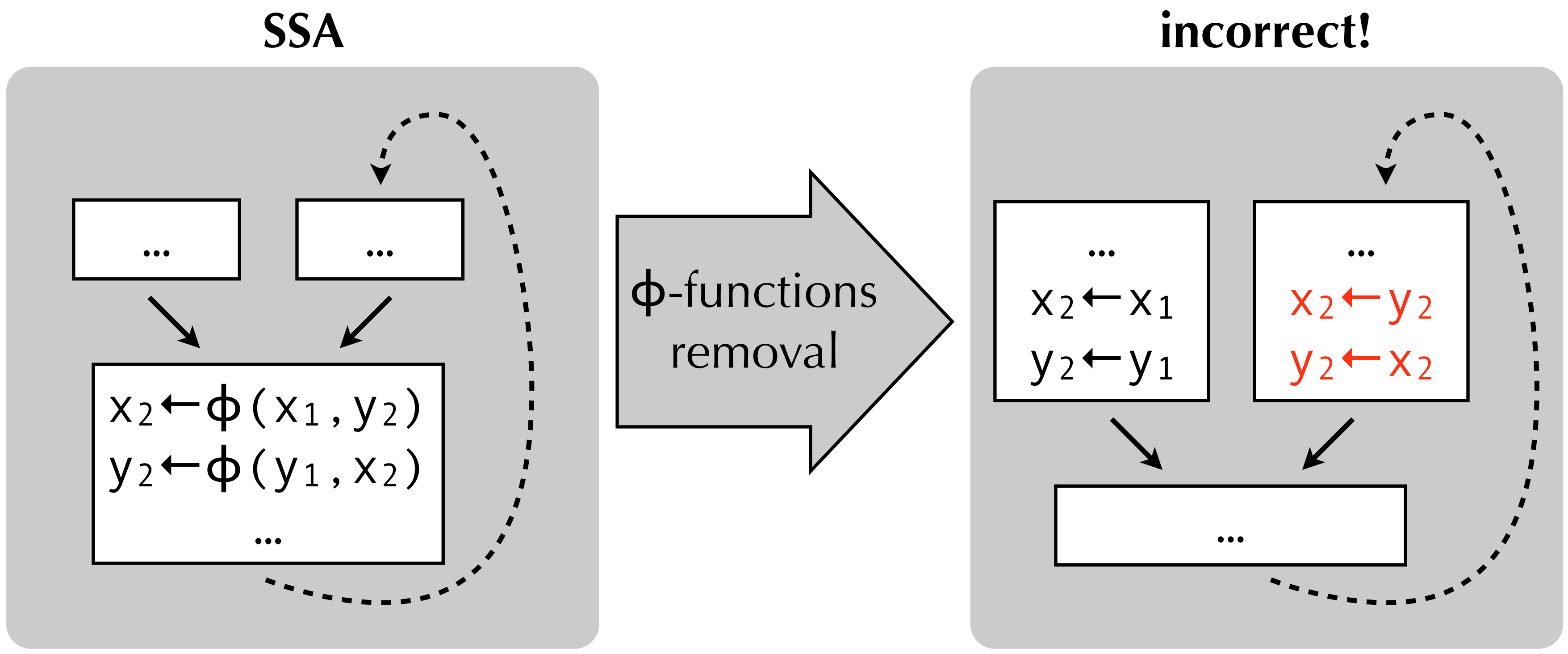
The semantics of SSA impose that all ϕ-functions of a block are evaluated in parallel.

For that reason, ϕ-functions should rigorously not be replaced by a sequence of assignments in the predecessor, but rather by a single *parallel* assignment, *e.g.* $(x_2, y_2) \leftarrow (x_1, y_1)$

If the target language does not offer parallel assignment, care should be taken to make sure that the sequence of assignments is equivalent to a parallel assignment. In the case of cyclic dependencies, this requires the use of an additional temporary variable. Example:

$$(x_2, y_2) \leftarrow (y_2, x_2) \equiv \begin{array}{l} t \leftarrow x_2 \\ x_2 \leftarrow y_2 \\ y_2 \leftarrow t \end{array}$$

Parallel move problem



SSA

...

...

$x_2 \leftarrow \phi(x_1, y_2)$
$y_2 \leftarrow \phi(y_1, x_2)$
...

ϕ-functions removal

incorrect!

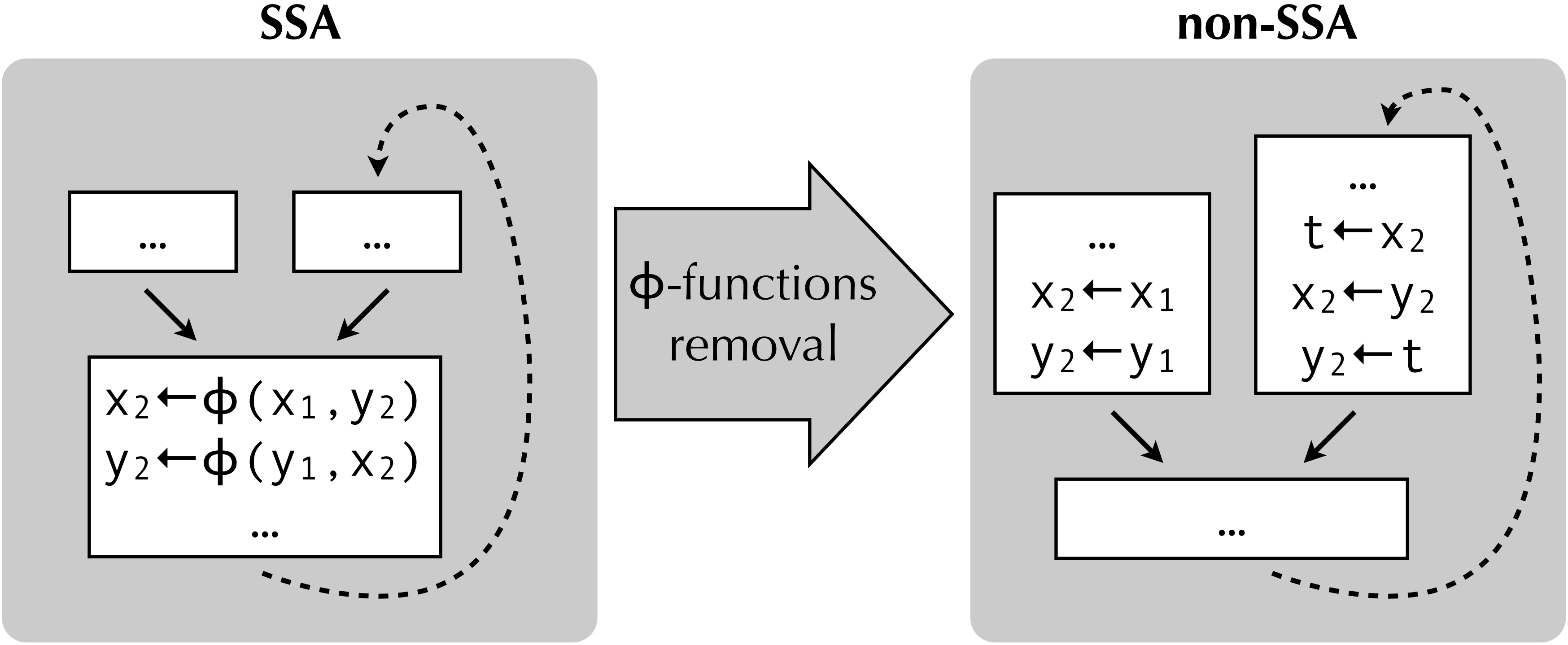
...
$x_2 \leftarrow x_1$
$y_2 \leftarrow y_1$

...
$x_2 \leftarrow y_2$
$y_2 \leftarrow x_2$

...

(This problem is know as *the swap problem*.)

Parallel move problem



SSA and functional programming

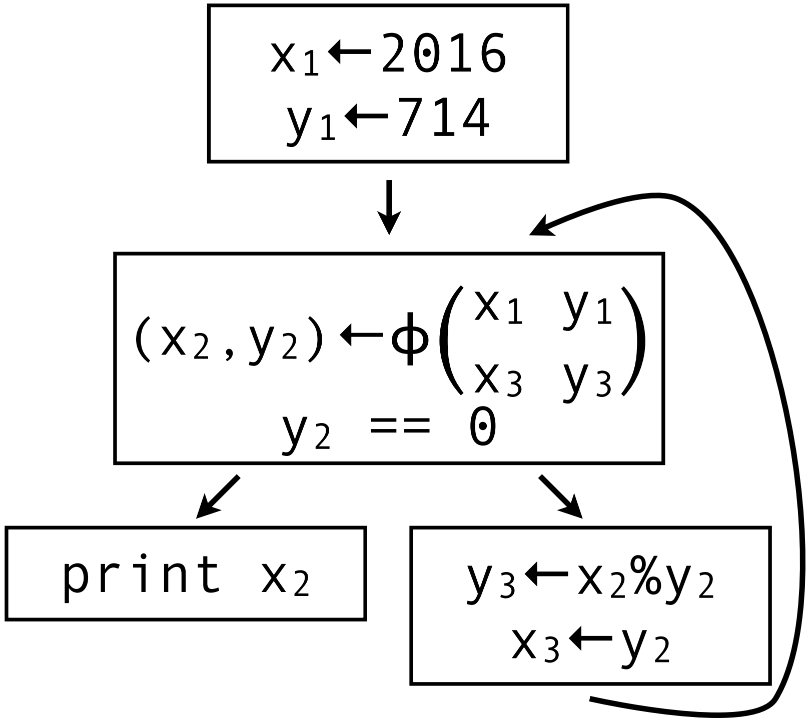
SSA *vs*. functional programming

SSA and (pure) functional programming languages share the characteristic that variables can be “assigned” once only. This is what makes programs in SSA form easier to analyze for the compiler, and a part of what makes functional programs easier to reason about for programmers.

The relation between the two is much deeper, however: SSA is basically functional programming with a different syntax!

SSA *vs.* functional programming

RTL/CFG in SSA form



$x_1 \leftarrow 2016$
$y_1 \leftarrow 714$

$(x_2, y_2) \leftarrow \phi\begin{pmatrix} x_1 & y_1 \\ x_3 & y_3 \end{pmatrix}$
$y_2 == 0$

print x_2

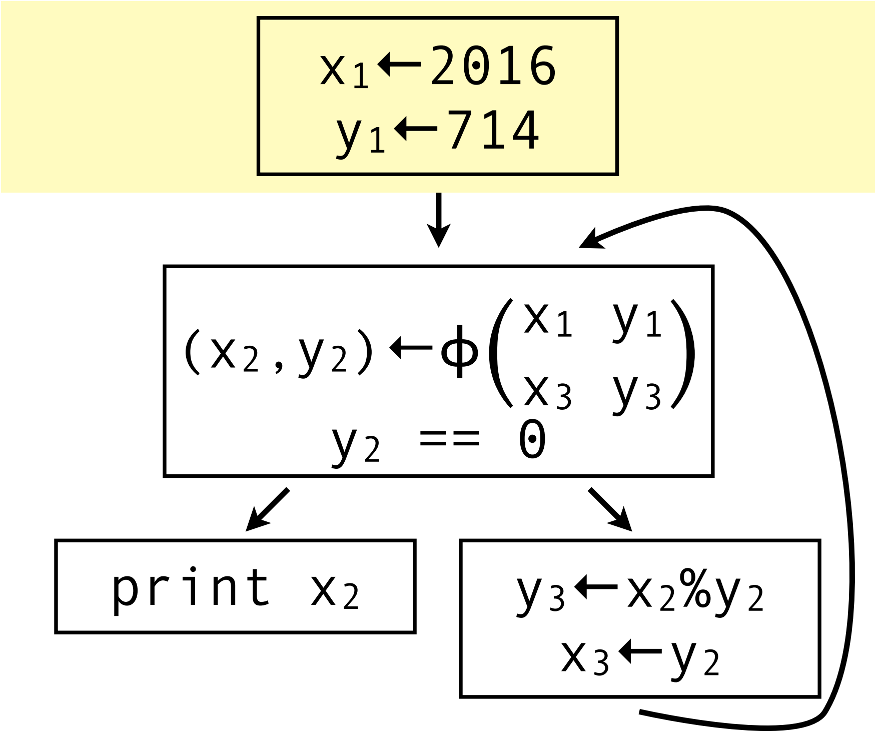
$y_3 \leftarrow x_2 \% y_2$
$x_3 \leftarrow y_2$

functional program

```
**val** $x_1$ = 2016
**val** $y_1$ = 714
loop($x_1$, $y_1$)
**def** loop($x_2$,$y_2$) =
    **if** ($y_2$ == 0)
        print($x_2$)
    **else** {
        **val** $y_3$ = $x_2$ % $y_2$
        **val** $x_3$ = $y_2$
        loop($x_3$, $y_3$)
    }
```

SSA *vs.* functional programming

RTL/CFG in SSA form

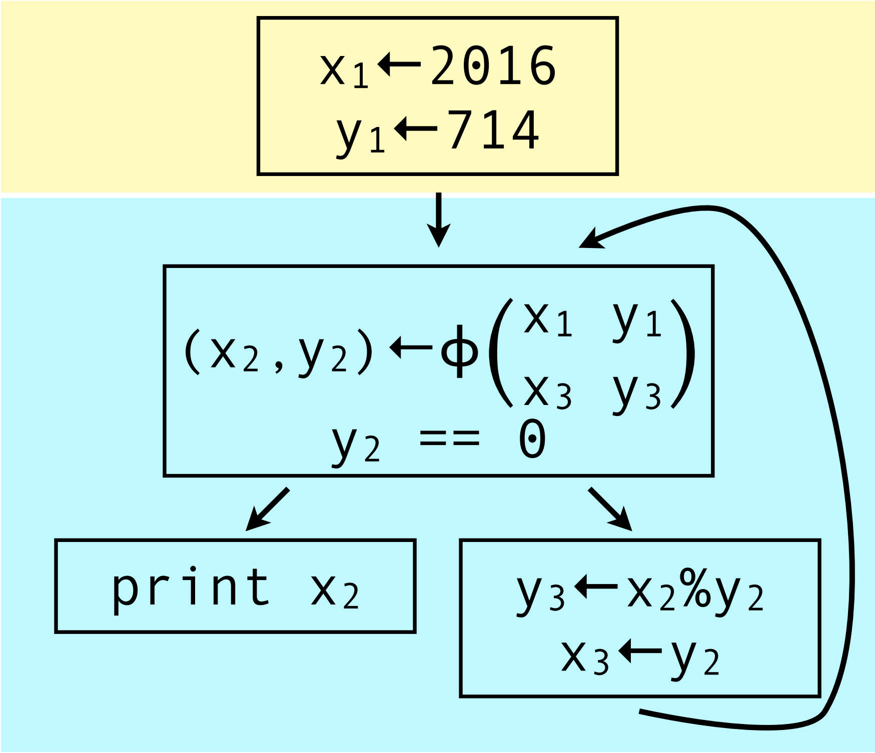

$x_1 \leftarrow 2016$
$y_1 \leftarrow 714$
$(x_2, y_2) \leftarrow \phi\begin{pmatrix} x_1 & y_1 \\ x_3 & y_3 \end{pmatrix}$
y_2 == 0
print x_2
$y_3 \leftarrow x_2 \% y_2$
$x_3 \leftarrow y_2$

functional program

```
val x1 = 2016
val y1 = 714
loop(x1, y1)
def loop(x2,y2) =
   if (y2 == 0)
      print(x2)
   else {
      val y3 = x2 % y2
      val x3 = y2
      loop(x3, y3)
   }
```

SSA *vs.* functional programming

RTL/CFG in SSA form



$x_1 \leftarrow 2016$
$y_1 \leftarrow 714$

$(x_2, y_2) \leftarrow \phi\begin{pmatrix} x_1 & y_1 \\ x_3 & y_3 \end{pmatrix}$
$y_2 == 0$

print x_2

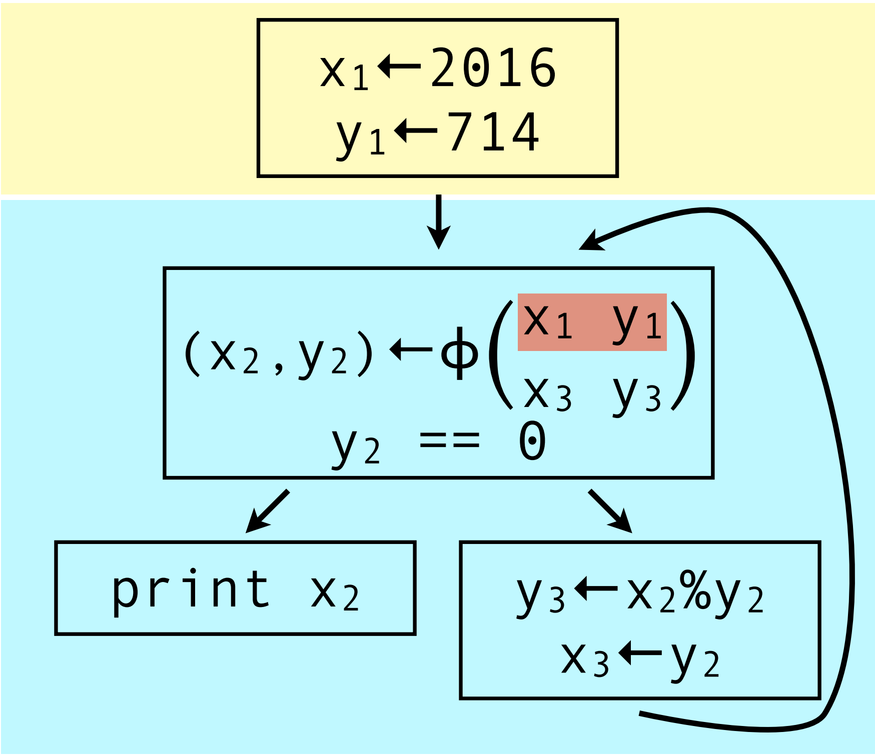
$y_3 \leftarrow x_2 \% y_2$
$x_3 \leftarrow y_2$

functional program

```
val $x_1$ = 2016
val $y_1$ = 714
loop($x_1$, $y_1$)
def loop($x_2$,$y_2$) =
    if ($y_2$ == 0)
        print($x_2$)
    else {
        val $y_3$ = $x_2$ % $y_2$
        val $x_3$ = $y_2$
        loop($x_3$, $y_3$)
    }
```

SSA *vs.* functional programming

RTL/CFG in SSA form



$x_1 \leftarrow 2016$
$y_1 \leftarrow 714$

$(x_2, y_2) \leftarrow \phi\begin{pmatrix} x_1 & y_1 \\ x_3 & y_3 \end{pmatrix}$
$y_2 == 0$

print x_2

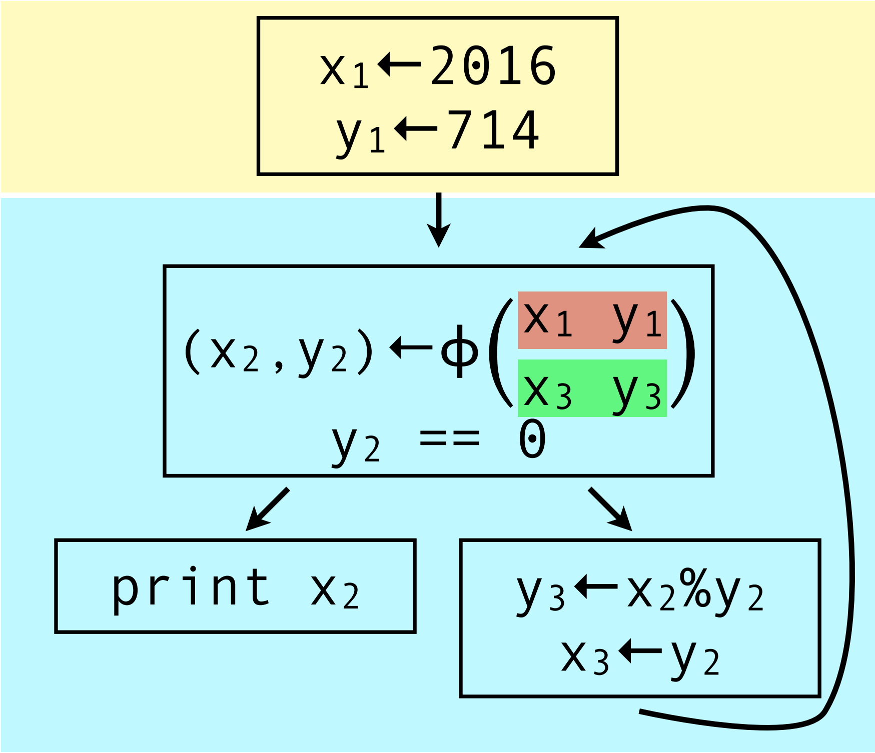
$y_3 \leftarrow x_2 \% y_2$
$x_3 \leftarrow y_2$

functional program

```
val x1 = 2016
val y1 = 714
loop(x1, y1)
def loop(x2,y2) =
   if (y2 == 0)
      print(x2)
   else {
      val y3 = x2 % y2
      val x3 = y2
      loop(x3, y3)
   }
```

SSA *vs.* functional programming

RTL/CFG in SSA form


$x_1 \leftarrow 2016$
$y_1 \leftarrow 714$
$(x_2, y_2) \leftarrow \phi\begin{pmatrix} x_1 & y_1 \\ x_3 & y_3 \end{pmatrix}$
y_2 == 0
print x_2
$y_3 \leftarrow x_2 \% y_2$
$x_3 \leftarrow y_2$

functional program

```
val $x_1$ = 2016
val $y_1$ = 714
loop($x_1$, $y_1$)
def loop($x_2$,$y_2$) =
  if ($y_2$ == 0)
    print($x_2$)
  else {
    val $y_3$ = $x_2$ % $y_2$
    val $x_3$ = $y_2$
    loop($x_3$, $y_3$)
  }
```

SSA *vs.* functional programming

SSA		Functional programming
code blocks starting with ϕ-functions	≈	functions with parameters
("calls" to) ϕ-functions	≈	function parameters
jumps	≈	tail calls to functions
dominance property	≈	variable scope

SSA pros and cons

Positive aspects of SSA form:

- Several optimizations and analysis are simpler when the RTL/CFG program is in SSA form, thanks to the single-assignment property.

Negative aspects of SSA form:

- ϕ-functions are an additional concept that must be handled by all code that manipulates the IR.

As we have seen, basic blocks with ϕ-functions are equivalent to functions with arguments. This suggests that a functional language with nested functions might be as powerful than RTL/CFG in SSA form, but simpler and cleaner.

IR #3
Functional IR

Functional IRs

A **functional IR** is an intermediate representation that is close to a (very) simple functional programing language.

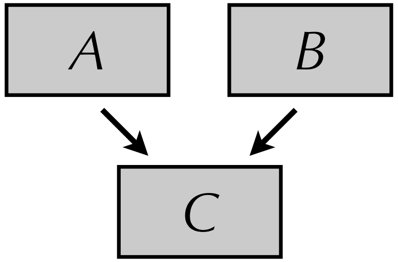
Typical functional IRs have the same interesting characteristics as RTL/CFG in SSA form, namely:

- all operations (*e.g.* arithmetic operations) are performed on atomic values (variables or constants), and the result of these operations is *always* named,
- variables can be “assigned” only once.

But they also bring several advantages compared to RTL/CFG in SSA form, as we will see later.

Code sharing

In RTL/CFG, a block can have multiple predecessors, like the block *C* below:



In a functional IR, a (local) function can be used to represent the block *C*. Jumps to *C* are simply represented as tail calls to the *C* function:

```
let C() = ...   // code for C
in ... C()      // in the code for A
   ... C()      // in the code for B
```

Notice that the function *C* is basically a continuation!

Functional IRs in CPS

The fact that continuations can be used to represent code blocks with multiple predecessors suggests that a functional IR might benefit from being in CPS.

Apart from shared code blocks, continuations also make it possible to express other language features:

- function returns, which are nothing but a call to the return continuation,
- exceptions, which can be implemented by passing a second continuation to every function, representing the current exception handler.

A functional IR in CPS

The syntax of a simple functional IR in CPS could be:

T ::=

letval x = V **in** T

let $x_1 = x_2 \oplus x_3$ **in** T where $\oplus$ is one of { +, -, *, … }

letcont $k\ (x_1, \ldots, x_n) = T_1$ **in** T_2

$f\ (k, x_1, \ldots, x_n)$

$k\ (x_1, \ldots, x_n)$

if $x_1 \circeq x_2$ **then** k_1 **else** k_2 where $\circeq$ is one of { =, ≠, <, … }

V ::=

integer

$\lambda(x_1, \ldots, x_n)\ T$

Code example

In our functional IR, the code to compute the gcd of 2016 and 714 would look as follows:

```
letcont loop(x, y) =
    letcont $k_1$() = print(x) in
    letcont $k_2$() =
        let t = x % y in
        loop(y, t)
    in
    letval z = 0 in
    if y = z then $k_1$ else $k_2$
in
letval x = 2016 in
letval y = 714 in
loop(x, y)
```

Scope *vs*. dominance property

Like in a standard functional language, all variables in a functional IR have a **scope** outside which they cannot be referenced.

This notion of scope plays the same role as the dominance property in SSA (reminder: the dominance property specifies that all uses of a variable *v* must be dominated by the definition of *v*).

Notice, however, that checking that all uses of a variable are in the scope of its definition is much easier with a functional IR, as the dominance relation does not have to be computed: scope is purely syntactical.

In our IR, the scope of all variables is the term following the keyword **in**.

Functional IR pros and cons

Positive aspects of functional IRs:

- Well-designed functional IRs have all the advantages of RTL/CFG programs in SSA form, but are simpler because they do not have ϕ-functions.

Negative aspects of functional IRs:

- Most (current) literature on compiler optimization uses RTL/CFG in SSA form, which means that its algorithms must be adapted before being applicable to a functional IR.

Summary

Choosing the right intermediate representation is one of the most crucial design choice for a compiler author.

RTL/CFG is a classical intermediate representation that is close to the instruction set of a typical von Neumann computer. It is widely used, but its imperative nature makes it difficult to analyze and reason about.

RTL/CFG can be improved by using SSA form, which is basically a functional version of RTL/CFG.

Functional intermediate languages have all the advantages of SSA form. However, by modeling code blocks as functions with arguments, they can do without ϕ-functions. This makes them probably the best kind of intermediate languages, even when compiling imperative languages.