SSA form

Michel Schinz

Advanced Compiler Construction / 2006-06-16

SSA form

SSA form

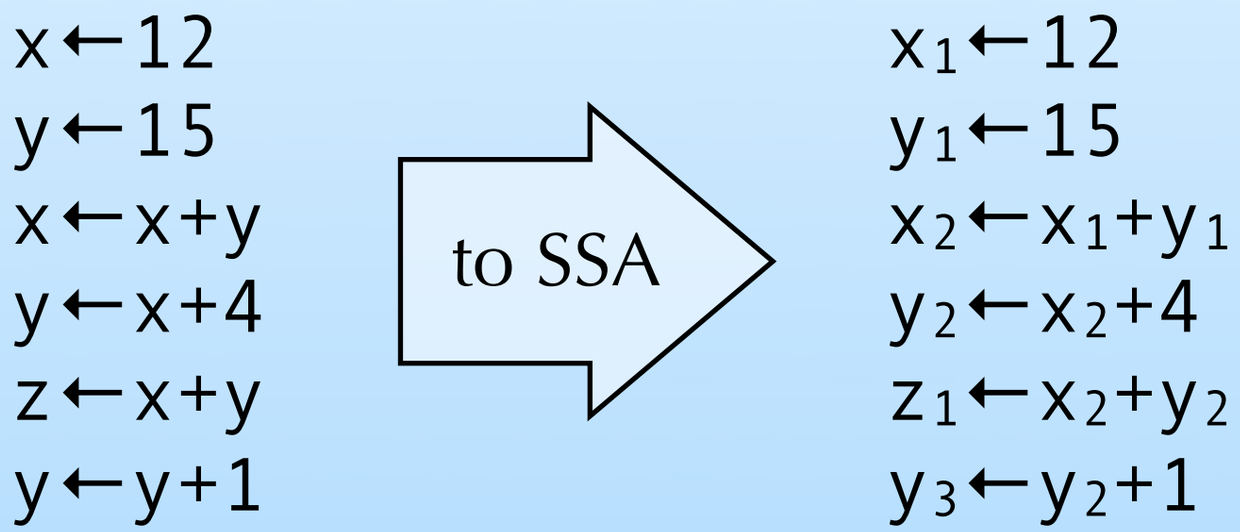
Static single-assignment (or **SSA**) form is an intermediate representation in which each variable has only one definition in the program.

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 interesting because it simplifies several optimisations and analysis, as we will see.

SSA form for 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:

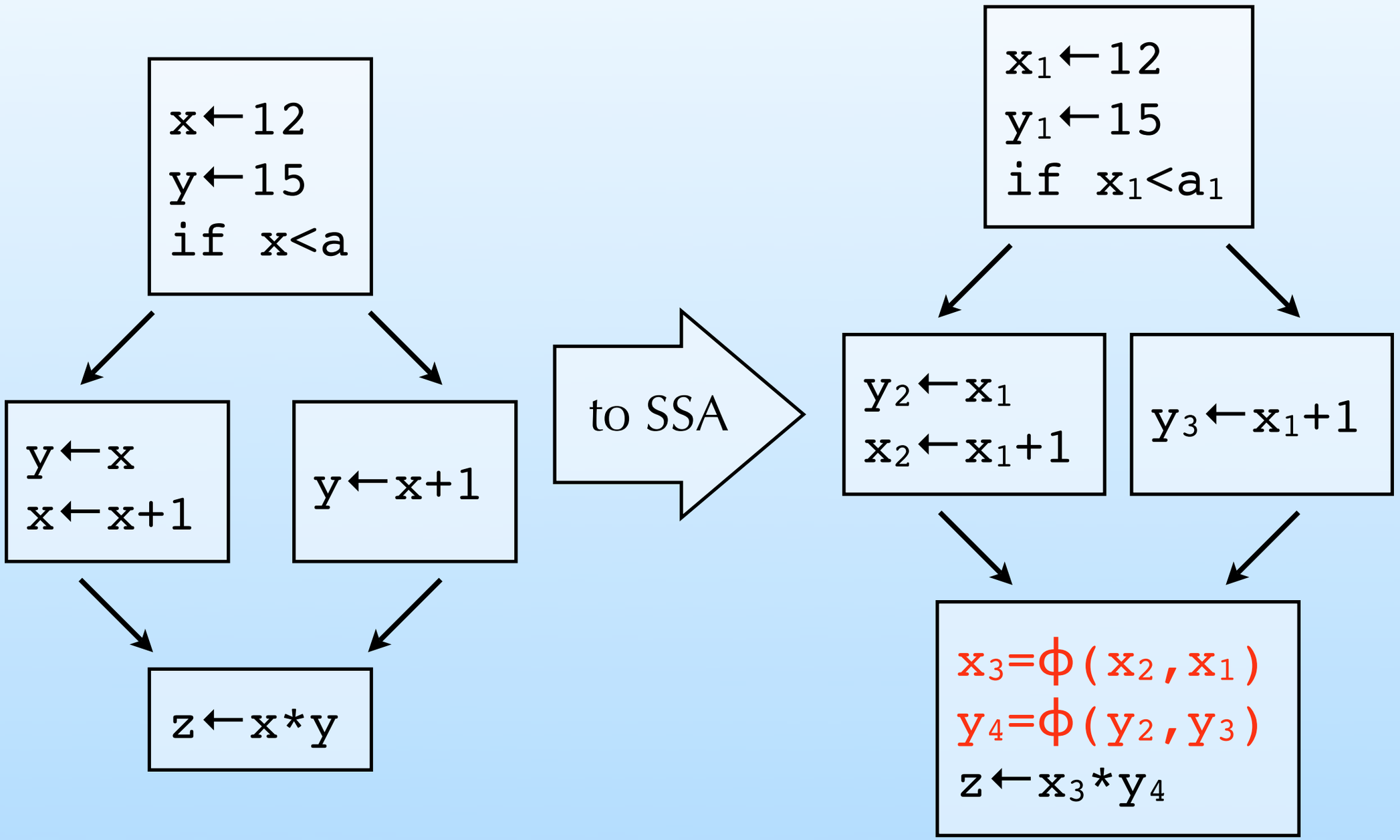


ϕ-functions

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 control flow.

ϕ-functions example



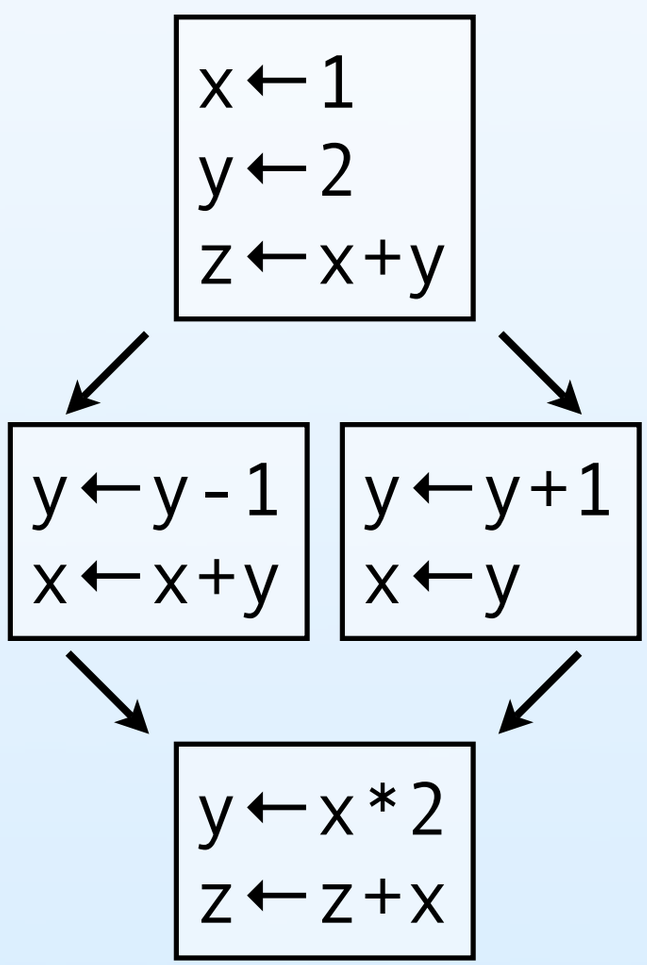
Building SSA form (naïve technique)

Naïve technique to build SSA form:

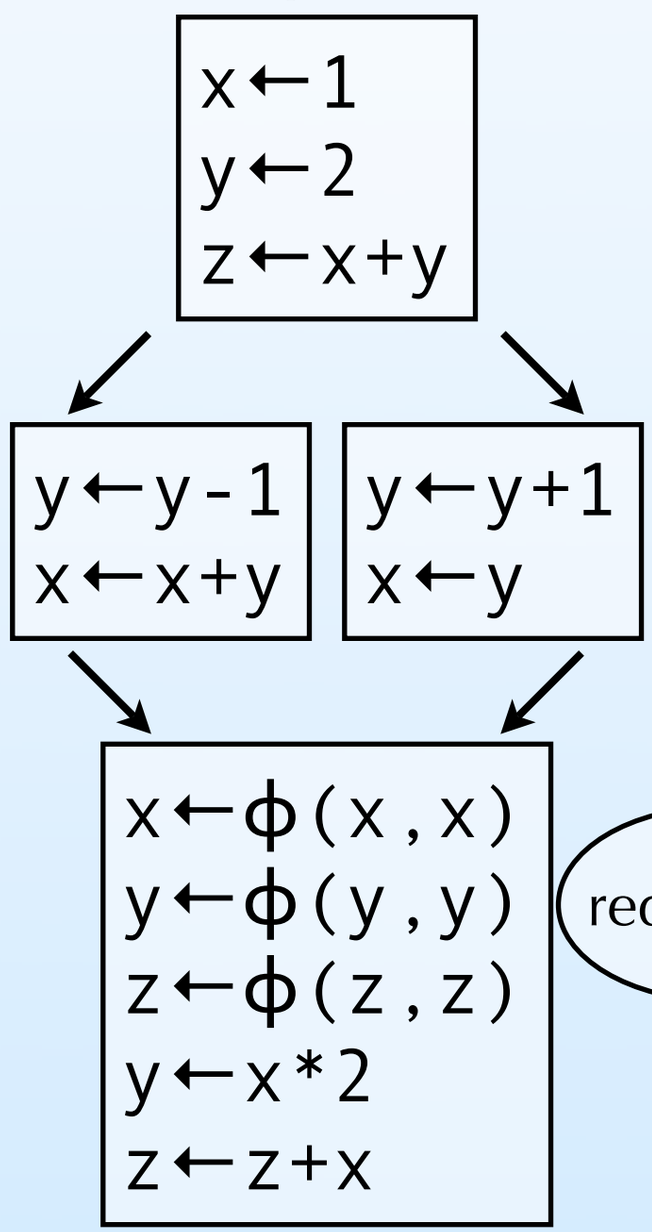
- for each variable *x* of the CFG, at each join point *n*, insert a ϕ-function of the form $x=\phi(x,\ldots,x)$ with as many parameters as *n* has predecessors,
- compute reaching definitions, and use that information to rename any use of a variable according to the – now unique – definition reaching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

Building SSA form (naïve technique)

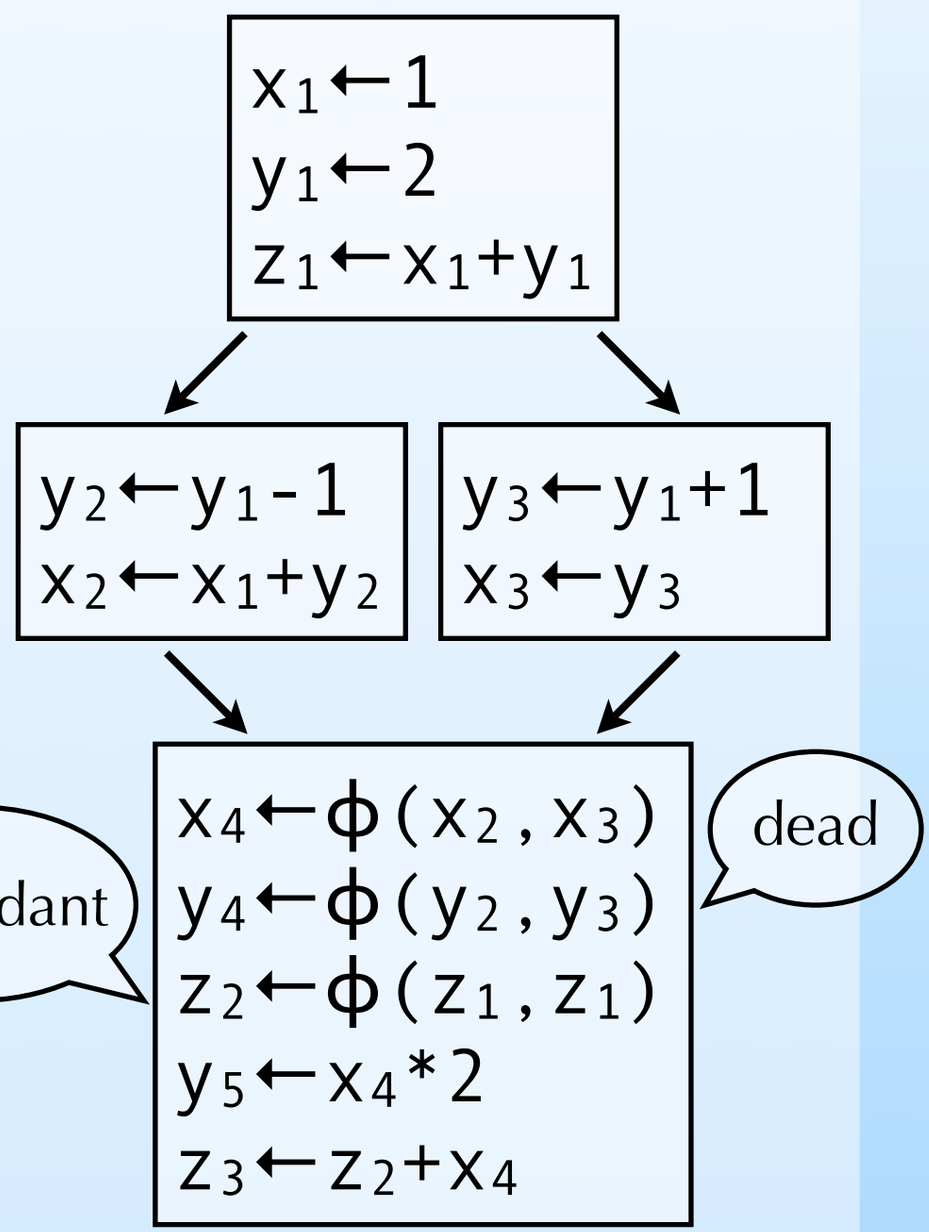
CFG



After phase 1



After phase 2



Better SSA construction techniques

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

However, it introduces too many ϕ-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

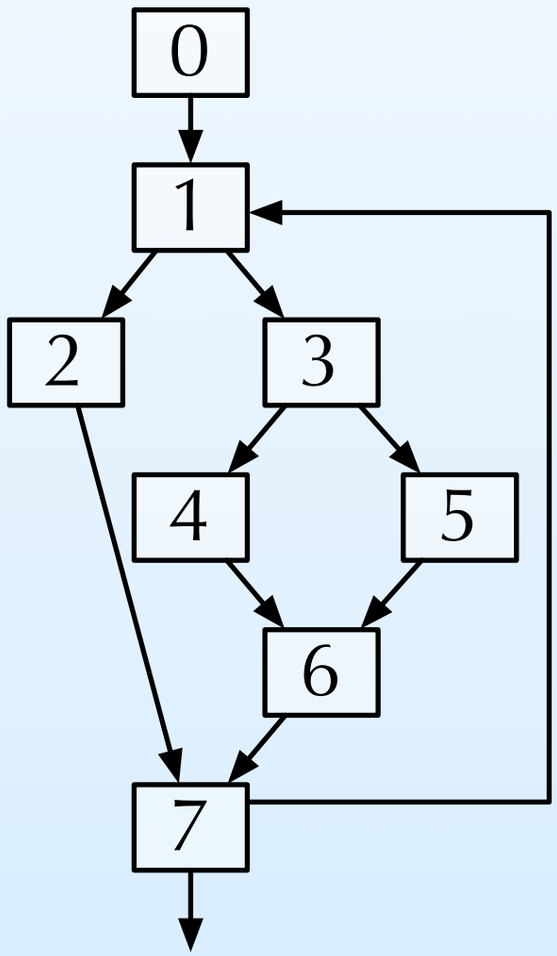
In a control-flow graph, a node n_1 **dominates** a node n_2 if all paths from the start node to n_2 pass through n_1.

By definition, the domination relation is reflexive, that is a node n always dominates itself. We then say that node n_1 **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 }

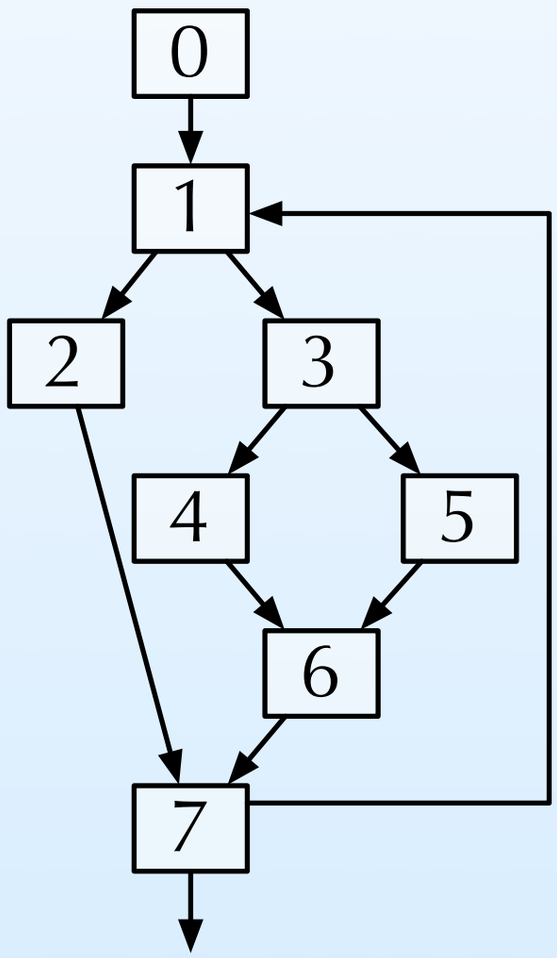
Dominator tree

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

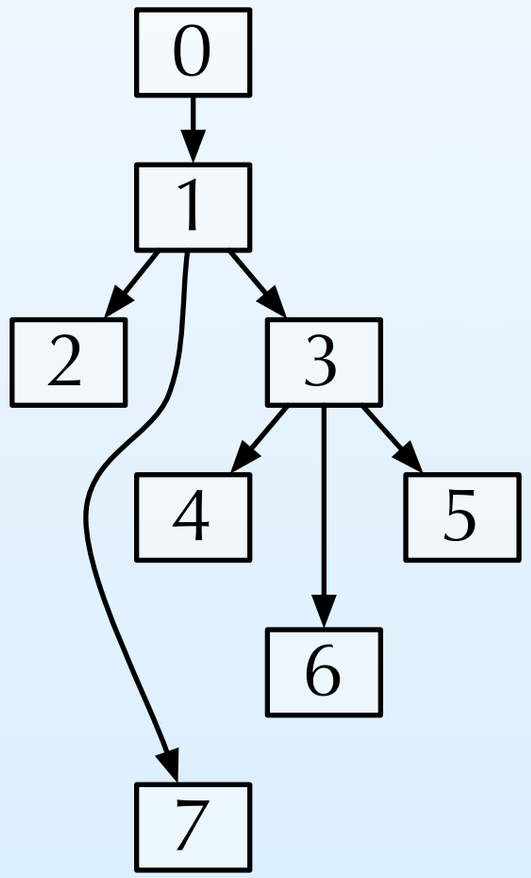
The nodes of the tree are the nodes of the CFG, and a node n_1 is a parent of a node n_2 if and only if n_1 is the immediate dominator of n_2.

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 which 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 $p_1, \ldots, p_k$ are the predecessors of n.

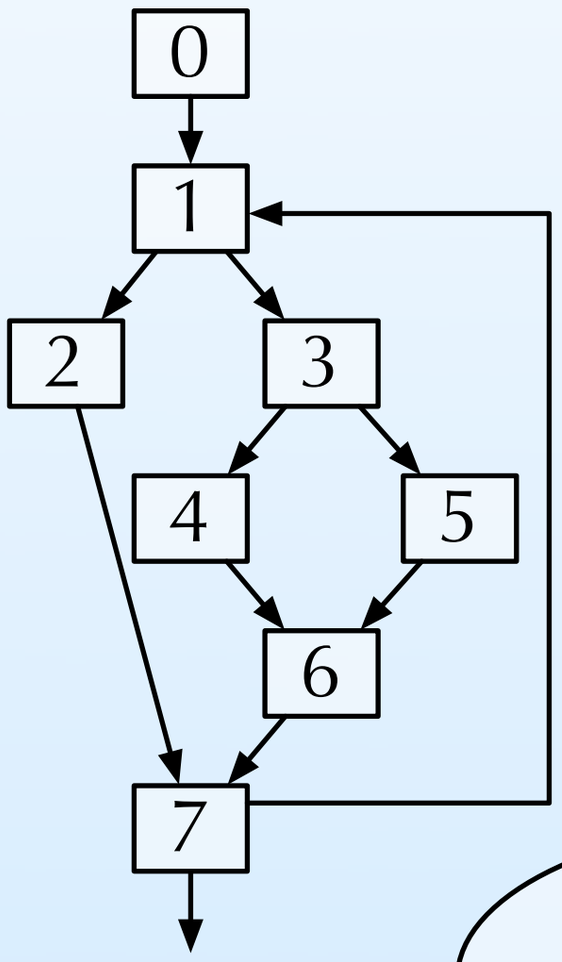
Dominance frontier

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.

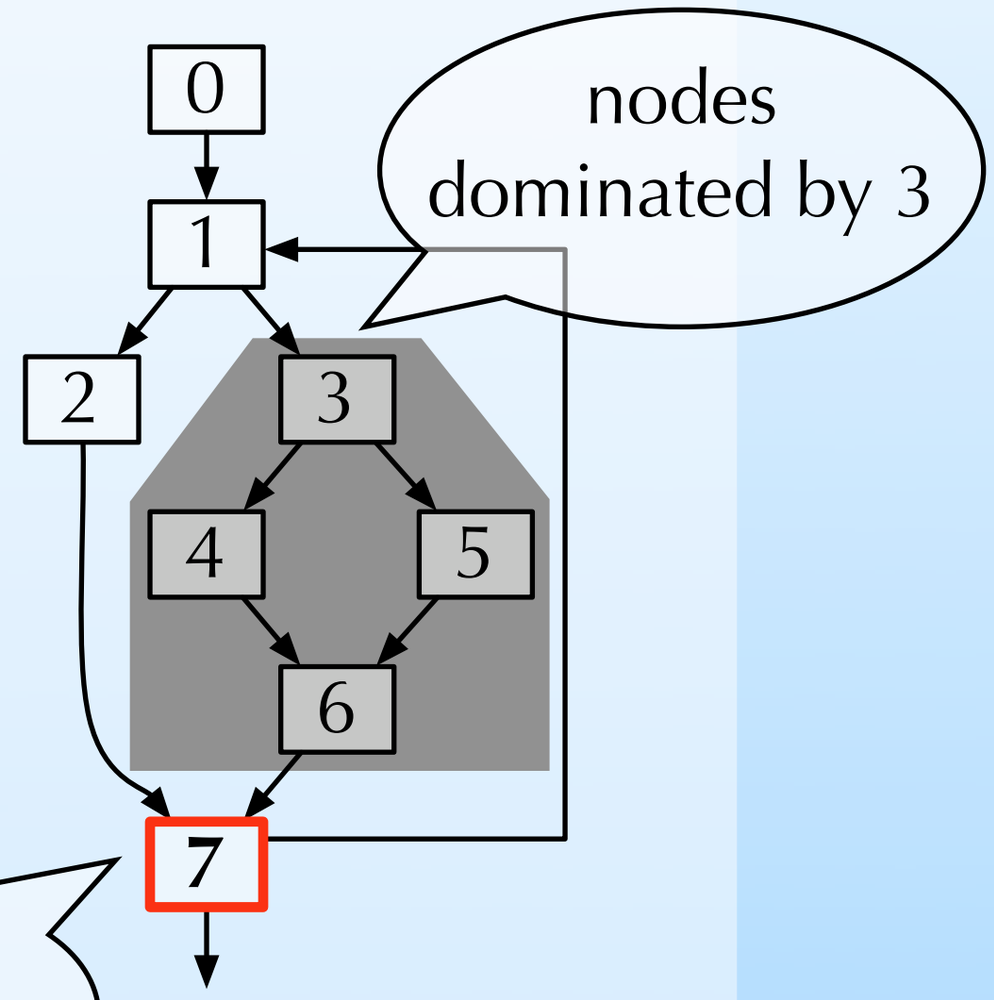
Informally, the dominance frontier of *n* contains the first nodes which are reachable from *n* but which are not strictly dominated by *n*.

Dominance frontier example

CFG



Dominance frontier



dominance
frontier of 3={7}

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.

Going further than minimal SSA

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 – still not optimal – and pruned SSA form.

Semi-pruned SSA form

Observation: a variable which 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** – names live across more than one node – and producing ϕ-functions for these names only.

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

Algorithm to build 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.

Semi-pruned SSA: inserting ϕ-functions

Before inserting ϕ-functions, the set G of global names must be computed. Once this is done, insertion of ϕ-functions proceeds 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 }
```

Semi-pruned SSA: renaming variables

Renaming is done by a pre-order traversal of the dominator tree.

For each visited node *n*, the following is done:

- definitions and uses of variables occurring in *n* are renamed,
- the parameter corresponding to *n* in all ϕ-functions of all successors of *n* in the CFG is renamed.

Example

(see blackboard)

Generating code from SSA form

Generating code from SSA form

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

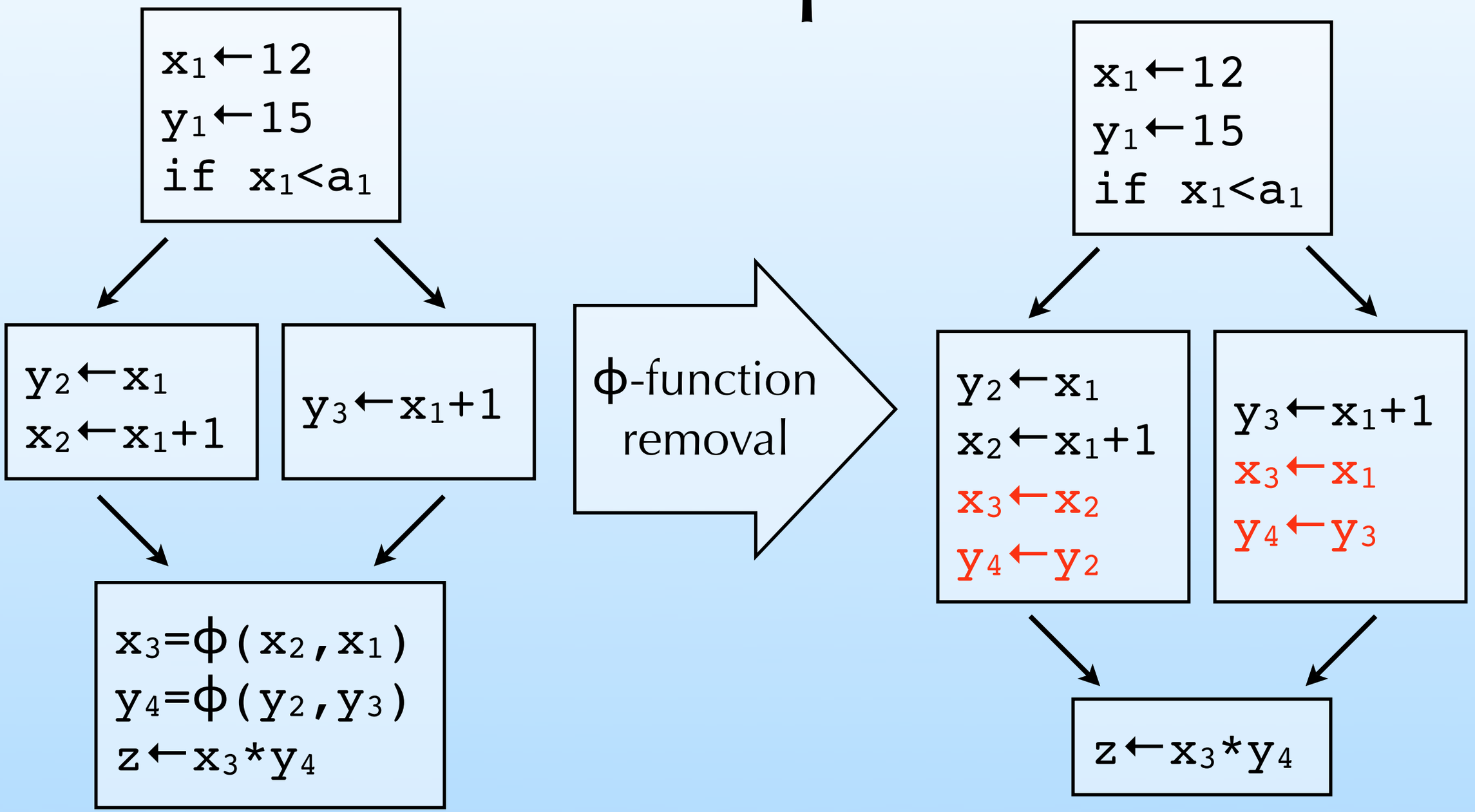
This implies in particular that ϕ-functions must be removed, as they cannot be implemented on standard machines.

Removing ϕ-functions

A ϕ-function of the form $x_i \leftarrow \phi(x_1,\ldots,x_n)$ can be 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 $x_i \leftarrow x_j$ (*i.e.* move instructions), but most of them will be removed later during register allocation, thanks to coalescing.

Removing ϕ-functions example



Critical edges

CFG edges which 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 redundant move instructions in n_1, corresponding to the ϕ-functions of n_2. Ideally, they should be executed only if control reaches n_2 later, but this is not certain when n_1 executes.

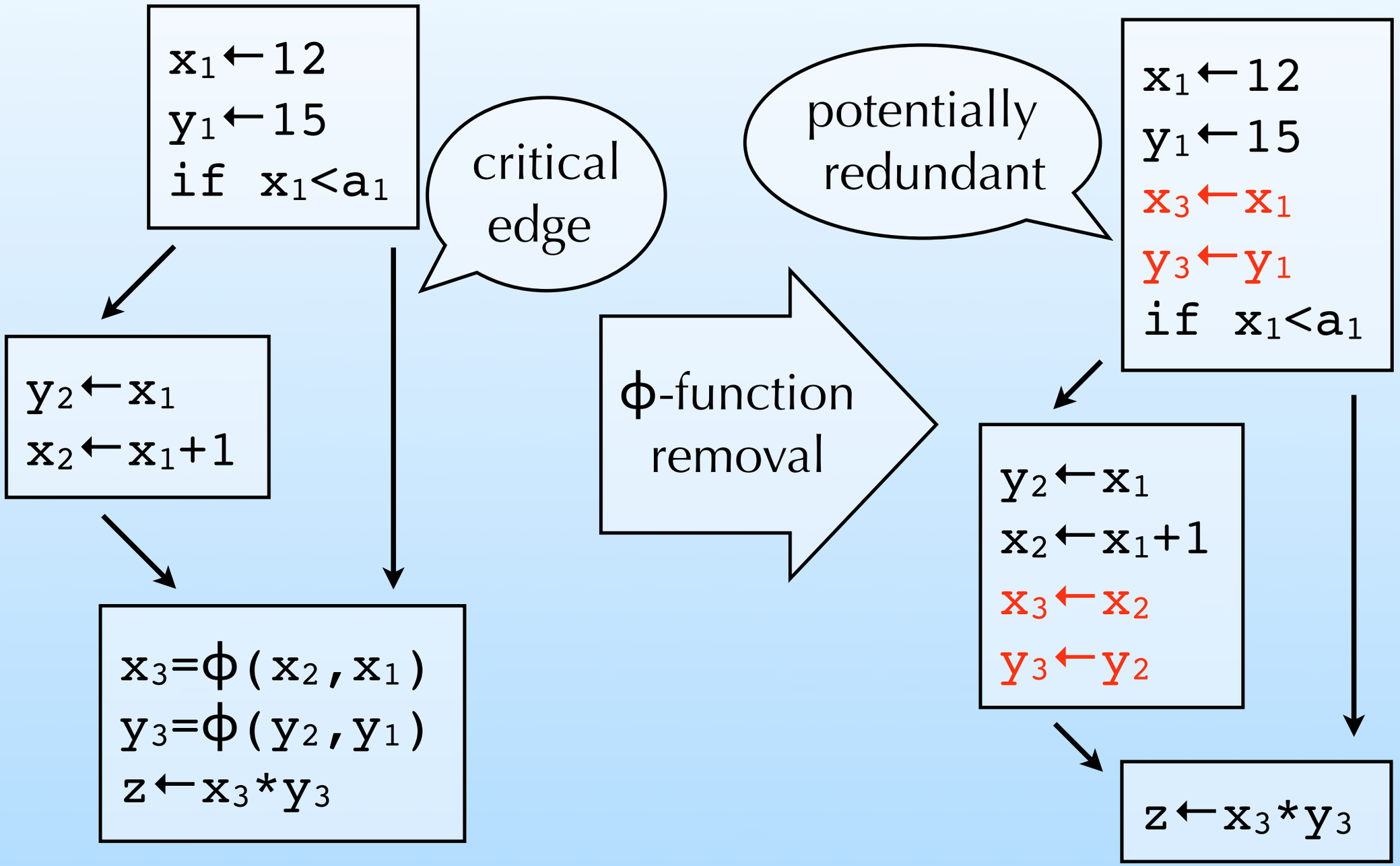
Edge splitting

Critical edges can easily be avoided completely using **edge splitting**.

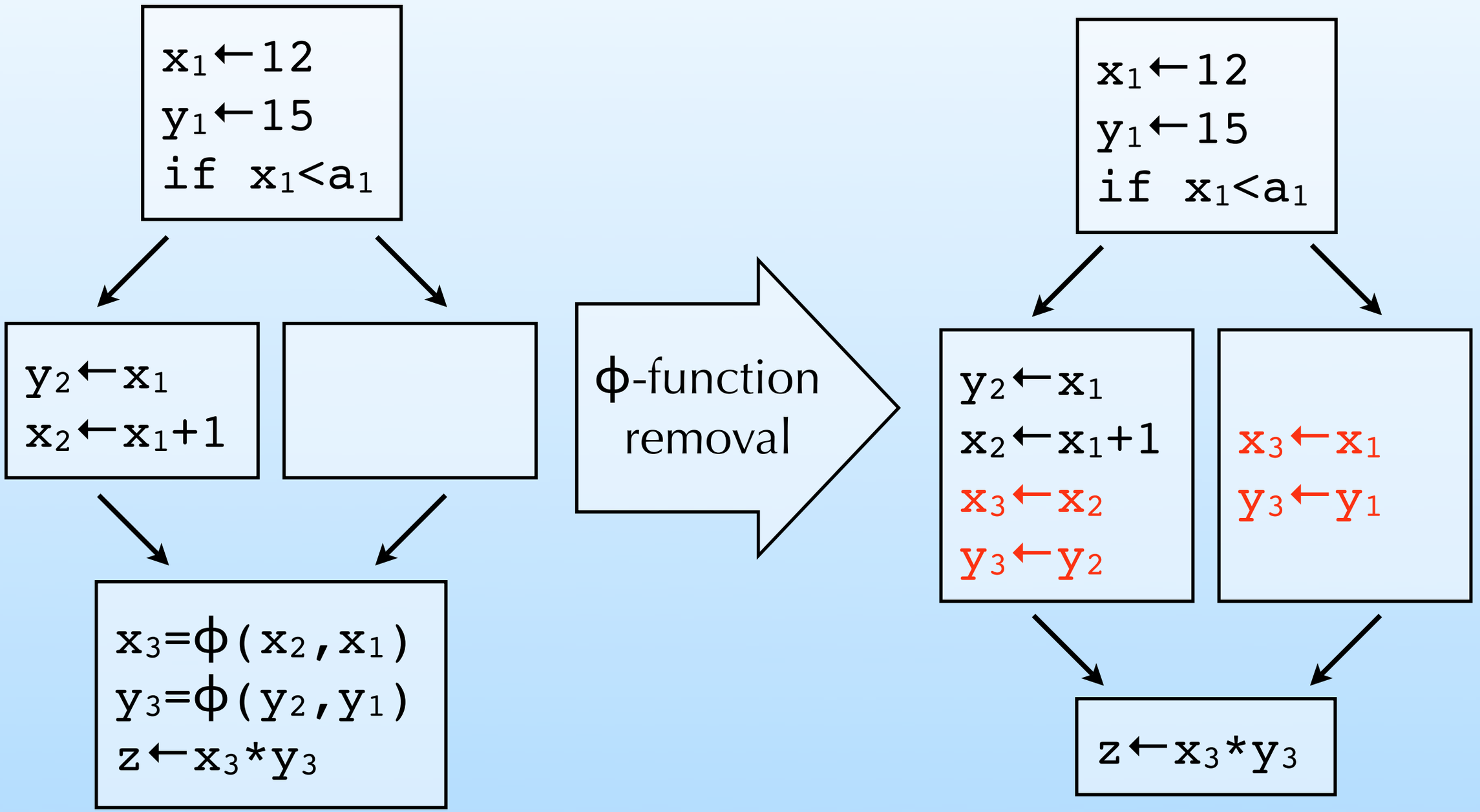
Edge splitting consists in replacing all critical edges leading from a node n_1 to a node n_2 by two edges: one from n_1 to a new empty node n_3, and one from n_3 to n_2.

Since the new empty block n_3 has only one predecessor and one successor, this effectively removes the critical edge.

Without edge splitting



With edge splitting



Using SSA form

Dead-code elimination

Basic dead-code elimination is trivial in SSA form: if a variable x_i is not used in some expression, then its definition – of the form $x_i \leftarrow y_j$ op z_k or $x_i \leftarrow \phi(x_j, \ldots, x_k)$ – can be deleted. Of course, this is only true if that definition does not have side-effects.

The deletion of a definition can remove the last use of some other variable, in which case its definition can be deleted too, and so on…

Simple constant propagation

SSA form also simplifies constant propagation: whenever a definition of the form $x_i \leftarrow c$ – where c is a constant – is encountered, then all uses of x_i can be replaced by c. Moreover, the definition itself can be deleted from the program, being now dead.

Also, a ϕ-function of the form $x_i \leftarrow \phi(c_1,\ldots,c_n)$ where $c_1=\ldots=c_n$ can be replaced by $x_i \leftarrow c_1$, which is then simplified as above.

Copy propagation and constant folding

Copy propagation can be handled in a similar fashion as constant propagation: definitions of the form $x_i \leftarrow y_j$, and single-argument ϕ-functions of the form $x_i \leftarrow \phi(y_j)$ can be deleted, and all uses of x_i replaced by uses of y_j.

The same is true of constant folding: a definition of the form $x_i \leftarrow c_1$ op c_2 – where c_1 and c_2 are constants – can be deleted and all uses of x_i replaced by the value of c_1 op c_2.

Liveness analysis

SSA form also simplifies liveness analysis, and hence the construction of the interference graph needed by register allocation.

To compute the region where a variable x_i is live in SSA form, it is sufficient to start from all uses of x_i and walk backwards in the CFG until the definition of x_i is encountered. The statements encountered during that walk are those during which x_i is live.

Summary

Static single-assignment (SSA) form is an intermediate representation where all names are defined exactly once. To enable this, ϕ-functions have to be inserted at join points in the CFG.

Transforming a program to SSA form is not completely trivial since unnecessary ϕ-functions should be avoided.

SSA encodes the data-flow of the program in its names, making several optimisations easier.