Static Single Assignment Form

Overview Many dataflow analyses need to find the **use-sites** of each *defined* variable or the **def-sites** of of each variable *used* in an expression. The **def-use** chain is a data structure that makes this efficient: for each statement in the flowgraph, the compiler can keep a list of pointers to all the use-sites of the variables defined there, and a list of pointers to all def-sites of the variables used there. But when a variable has N definitions and M uses, we need $N \cdot M$ pointers to connect them.

SSA avoids this problem by "getting the right number of names." Even if we see only one variable, x, based on its definition, the variable has many versions (x defined in statement S_1, x defined in statement S_2, etc.), each deserve its own name.

In SSA, **each variable in the program has only one definition** – it is assigned only once. The assignment might be, in a loop, which is executed many times; so single assignment is a *static property* of the program text, not a dynamic property of program execution.

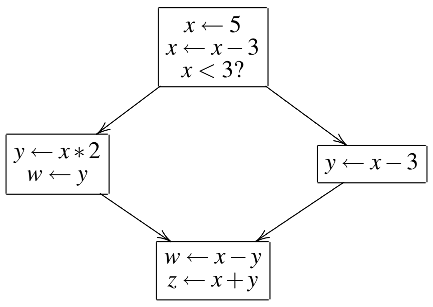
To achieve single-assignment, we make up a new variable name for each assignment to the variable. SSA simplifies and improves the results of various compiler optimizations, by simplifying the properties of variables. For example, given:

```
original:  y = 1;        SSA: y1 = 1
           y = 2;  ===>       y2 = 2
           x = y;             x1 = y2
```

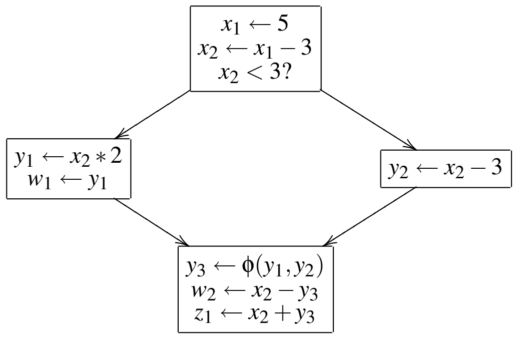
Reaching definition analysis which determines that the first `y = 1` is useless can be easily performed in SSA form. The following compiler optimizations can benefit from the use of SSA:

- constant propagation
- sparse conditional constant propagation
- dead code elimination
- global value numbering
- partial redundancy elimination
- strength reduction
- register allocation

Converting flowgraph to SSA graph Converting flowgraph into an SSA graph involves *replacing the target of each* ***definition*** *with a new variable, and the* ***use*** *of a variable with the "version" of the variable* ***reaching*** *that point*. Given:



its SSA form is:



The ϕ-function Creating SSA forms when there is no jumps is easy. However, when two control-flow edges join together, **carry different values of some variable** x, we must somehow merge the two values. This is done using the ϕ-function.

The ϕ-function at the last block, in the figure above, is inserted to reconcile the conflict between two definitions, y_1 and y_2, along different paths. A ϕ-function tasks as arguments the SSA-names for the value on *each control-flow edge entering the block*. It defines a new name for subsequent uses.

Computing minimal SSA using dominance frontiers *Dominance frontiers capture the precise places at which we need ϕ-functions:* if the node A defines a certain variable, then that definition and that definition alone (or redefinitions) will reach every node A dominates.

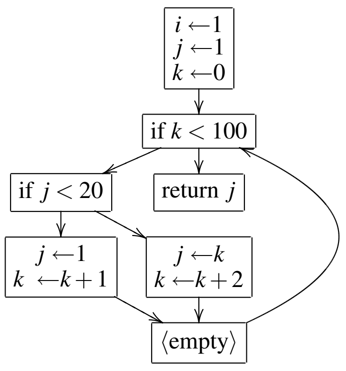
Global value numbering Global value numbering (GVN) eliminates redundancy by constructing a **value graph** of the program, and then determines which values are computed by equivalent expressions. GVN is able to identify some redundancy that **common subexpression elimination** cannot, and vice versa.

Sparse conditional constant propagation This optimization is effectively equivalent to iteratively performing **constant propagation**, constant folding, and dead code elimination until there is no change, but is much more efficient. This optimization symbolically executes the program, simultaneously propagating constant values and eliminating portions of the control flow graph that this makes unreachable.

Appel: SSA is functional programming Consider the following program:

```
i = 1;
j = 1;
k = 0;
while (k < 100) {
  if (j < 20) {
    j = i;
    k = k + 1;
  }
  else {
    j = k;
    k = k + 2;
    }
  }
```

Its flowgraph is given below:


$i \leftarrow 1$
$j \leftarrow 1$
$k \leftarrow 0$
if $k < 100$
if $j < 20$
return j
$j \leftarrow 1$
$k \leftarrow k+1$
$j \leftarrow k$
$k \leftarrow k+2$
⟨empty⟩