\[ x = a + b \]  \#1

\[ d = x + 1 \]  \#2

\[ b = a + c \]  \#3

\[ \text{if} (\ldots) d < 7 \]  \#4

\[ y = b + x \]  \#5

For the IR:

- Like to record that "x" produced by stmt 1 is used by stmt 2
- Must make sure that stmt 1 reads old value of "b" (not the b produced in stmt 3)

Sufficient solution: stmt 3 always follows stmt 2.

Important: old value is used (in stmt 1)
- new value is used in stmt 3 -- (e.g., stmt 5)
Idea: remove unnecessary dependences by renaming of variables

introduce new variable names

\[ x = a + b \]

\[ y = a + \bar{c} \]

use subscripts to indicate that a different version of a variable is used

\( b_0, b_1, b_2, \ldots \)

Could use any name... but programs are easier to read if we use subscripts.

New version is created whenever there is an assignment.
In our example:

\[ x_1 = a_0 + b_0 \]
\[ d_1 = x_1 + 1 \]
\[ b_1 = a_0 + c_0 \]

if (...) \( x < 3 \)

\[ y_1 = b_1 + x_1 \]
\[ y_2 = \ldots \]

// next assignment to y

won't work for fields of an object.
This representation is known as "SSA".

Static Single Assignment (SSA) form represents each variable in the program text written by exactly one statement or operation.

- Data dependences are explicit.
- No constraints due to choice of variable names.
2.0 SSA representation

SSA is used in many production compilers

- Intro: basic block operations
- IF statements
- Control flow constructs
- Control Flow Graph (CFG)
- Simple way to implement the SSA form for programs written in a simple language (Pascal, Java: ✓
  C, assembly: ✓)

3.0 Introducing SSA into arbitrary program
2.1 SSA for basic blocks

Assume: program broken into basic blocks

Goal: transform one basic block into SSA format

For each (method-local) variable, we need a counter. For \( v \) we use \( C_v \)

Easy: given a statement

\[
\text{dest} = \text{source}_A + \text{source}_B
\]

\[x = a + b\]

look up \( C_x \)

new-\( x \)-version = \( C_x + \)

look up \( C_a \), \( C_b \)
2.2 IF statements

How do we handle a conditional statement

\[ a = 1; \]
\[ \text{if}(b_0 \neq 0) \]
\[ a = 0; \]
\[ x = a \]

we need counters

\[ Ca = 0 + 2 \]
\[ Cb = 0 \]
\[ Cx = -1 \]

\[ a_2 = 1; \]
\[ \text{if}(b_0 \neq 0) \]
\[ a_2 = 0; \]
\[ x_1 = a_2 \]

can't use \( a_1 \)

can't use \( a_2 \)
Idea: we introduce a magic function that picks the correct version

- $a_1 : \text{if } b == 0$
- $a_2 : \text{if } b \neq 0$

version picked depends on the path taken to the point where a decision must be made.

$\phi$ - function

$\phi \ (\text{arg}_1, \text{arg}_2)$

selects the right version based on path taken
\( a_1 = 1; \)
if \( (b_0 \neq 0) \)
\( t \)
\( a_2 = 0; \)
\( 3 \)

\( a_3 = \phi(a_1, a_2) \)
\( x_1 = a_3 \)

<table>
<thead>
<tr>
<th>path 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>path 2</td>
</tr>
</tbody>
</table>

\( a_1, a_2 \): are set along the different paths through the program.

\( \phi(\text{args}_1, \text{args}_2) \)

Version for path 1,
Version for path 2.

If the compiler manages to place \( a_1 \) and \( a_2 \) into some storage location (e.g., registers) then \( \phi \)-fact is trivial to implement.
2.3 $\phi$-Function (or $\phi$-node)

$\phi$-function selects the correct version based on path taken
if a basic block $B'$ has $m$ predecessors (in the control flow graph)

then the $\phi$-Function must allow $m$ arguments

$$V_{new} = \phi(V_1, V_2, \ldots, V_m)$$
(Aside: no limit on the number of arguments for Java, C, (but 2 are enough for Java!))

- \( \phi \)-Functions appear at the beginning of a block
- Result of \( \phi \)-Function is always assigned to a new variable
  (new version of variable that produced the versions that appear as arguments)
- Reading of the arguments is special
Example to illustrate

```c
if (oo) {
    x = 0;
    else x = 1;
}

y = x

x_1 (x_2) is read at the end of the "then" ("else") block
What is good about SSA?

- Some optimizations are easy
- Efficient (for the compiler)

**Issue 1: Support for optimizations**

**Example:** Common Subexpression Elimination (CSE)

Attempts to identify expressions that (at a given point) have already been evaluated

\[ x = a + b \]

\[ \text{if } (\cdot) \]

\[ x = a + b \]

\[ a + b \text{ is already evaluated} \]

"Common subexpr"
Not always easy to detect...

\[ \text{if} (\ldots) \{ \]

\[ a = \ldots \]

\[ f \]

\[ = a + b \]

\[ \sqrt{ \text{with SSA - easy to detect} } \]

\[ = a_i + b_i \]

\[ \text{if} (\ldots) \{ \]

\[ x_k = \ldots \]

\[ 3 \]

\[ = a_i + b_j \]

\[ \checkmark \]

\[ a + b \] is not a common subexpression \Rightarrow a + b \text{ must be re-evaluated}
Issue 2: Compiler efficiency

global data-flow analysis work a common representation, direct solution is over

intra-procedural (inside a method)

in a basic compiler:

def gen(B):
    for each bb in B:
        kill(B)

data-flow equation (N basic blocks)
2N equations

\[
\text{IN}[B] = \bigcup \text{OUT}[P] \\
\text{OUT}[B] = \text{gen}(B) \cup \text{IN}[B] - \text{kill}(B)
\]
Solution: computed by iteration

Sets are represented as bit vectors (lists, hash tables) ...

Length depends on number of definitions, number of variables, number of expressions:

- potentially unbounded
- wasted space, wasted cycles, missed opportunities
SSA does not have this problem.

For a given version:

\[ a_i = \ldots \]

\[ = a_i \]

\[ = a_i + a_i \]

The number of links that the compiler must maintain depends only on the number of uses in the program (no extra work if there is no use outside a basic block).