NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 24: Static single assignment

Michael Engel

based on slides and lecture notes of Frank Pfenning (CMU)

Static Single Assignment (SSA)

- In this lecture we introduce Static Single Assignment (SSA) form
- SSA is a way to structuring the intermediate representation so that every variable is assigned exactly once
  - This is formally equivalent to continuation-passing style (CPS) IR
- Proposed by Rosen, Wegman and Zadeck in 1988 [1]
- Algorithm to compute SSA form efficiently by Cytron, Ferrante, Rosen, Wegman, and Zadeck at IBM in 1991 [2]

[1] Barry Rosen; Mark N. Wegman; F. Kenneth Zadeck (1988). "Global value numbers and redundant computations". Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages

[2] Cytron, Ron; Ferrante, Jeanne; Rosen, Barry K.; Wegman, Mark N. & Zadeck, F. Kenneth (1991). "Efficiently computing static single assignment form and the control dependence graph". ACM Transactions on Programming Languages and Systems. 13 (4): 451–490

Advantages of SSA

- Why do compiler writers use SSA?
  - SSA form makes use-def chains explicit in the IR, which in turn helps to simplify some optimizations

- Before getting into the details of SSA form, let's look at redundancy elimination as a motivating example
  - Redundancy elimination optimizations attempt to remove redundant computations

Caution…

- SSA form is seductive
  - The optimization benefits are real but not significant in simple compilers (like yours)
  - It looks easy but it isn’t

- My suggestion:
  - Think about it but probably not wise to attempt it

Redundancy elimination

- Common **redundancy elimination optimizations** are
  - value numbering
  - conditional constant propagation
  - common-subexpression elimination (CSE)
  - partial-redundancy elimination

What they do

```
read(i);
j = i + 1;
k = i;
l = k + 1;
```

```
i = 2;
j = i * 2;
k = i + 2;
```

```
    read(i);
    l = 2 * i + i;
    if (i>0) goto L1;
    i = i + 1;
    goto L2;
L1: k = 2 * i * l;
L2:
```

Value numbering determines that `j==l`

Constant propagation determines that `j==k`

Common subexpression elimination (CSE) determines that the second “`2*i`” is redundant

Value numbering

- Basic idea:
  - associate a symbolic value to each computation, in a way that any two computations with the same symbolic value always compute the same value

Congruence of expressions

- We define a notion of ***congruence*** of expressions:
  - $x \oplus y$ is congruent to $a \otimes b$ if $\oplus$ and $\otimes$ are the same operator, and:
    - x is congruent to a
    - and y is congruent to b
  - Typically, we will also take commutativity into account

Value numbering

- Suppose we have
  - `t1 = t2 + 1`
- Look up the key "`t2+1`" in a hash table
  - Use a hash function that assigns the same hash value (ie, the same value number) to expressions `e1` and `e2` if they are congruent
- If key "`t2+1`" is not in the table, then put it in with value "`t1`"
  - the next time we hit on "`t2+1`", can replace it in the IR with "`t1`"

Example

```
read(i);
j = i + 1;
k = i;
l = k + 1;
```

i = v1

Hash(v1 + 1) → j

j = v2

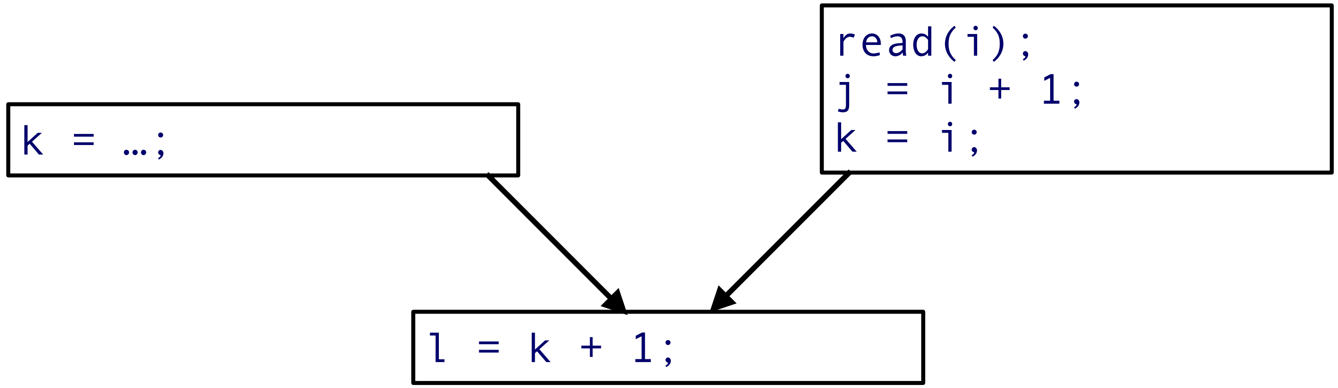
k = v1

Hash(v1 + 1) → j

Therefore `l = j`

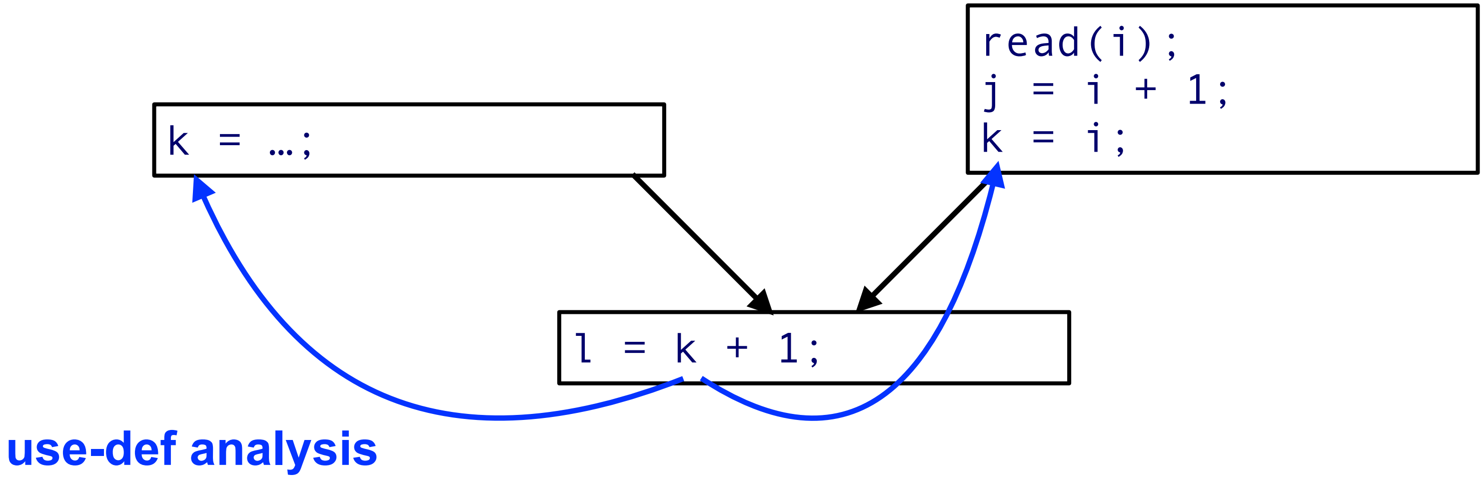
Global value numbering

- **Local** (i.e. within a basic block) value numbering is easy enough
- But what about **global** (i.e. within a procedure) value numbering?



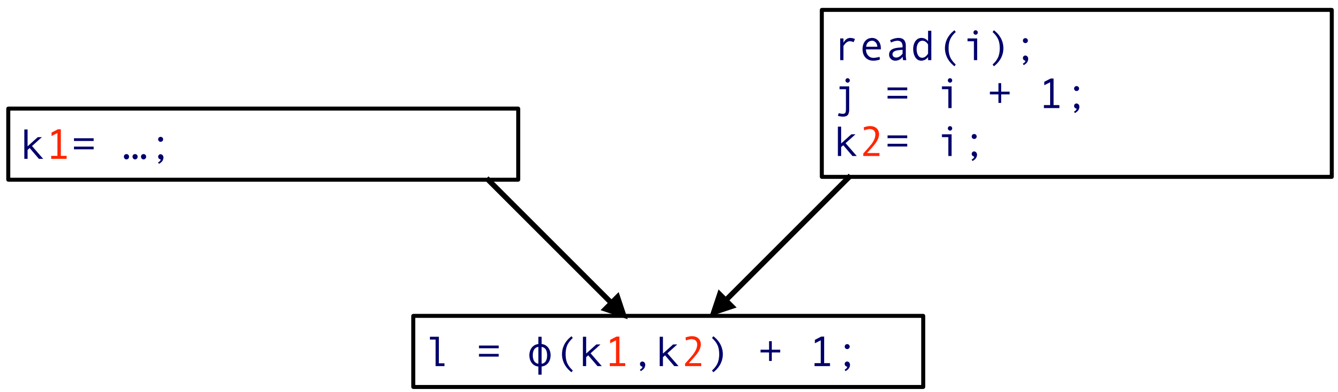
Importance of use-def info

- In the global case we must watch out for ***multiple assignments***
- We could do a data flow analysis to extend value numbering to the global case



Embedding use-def into the IR

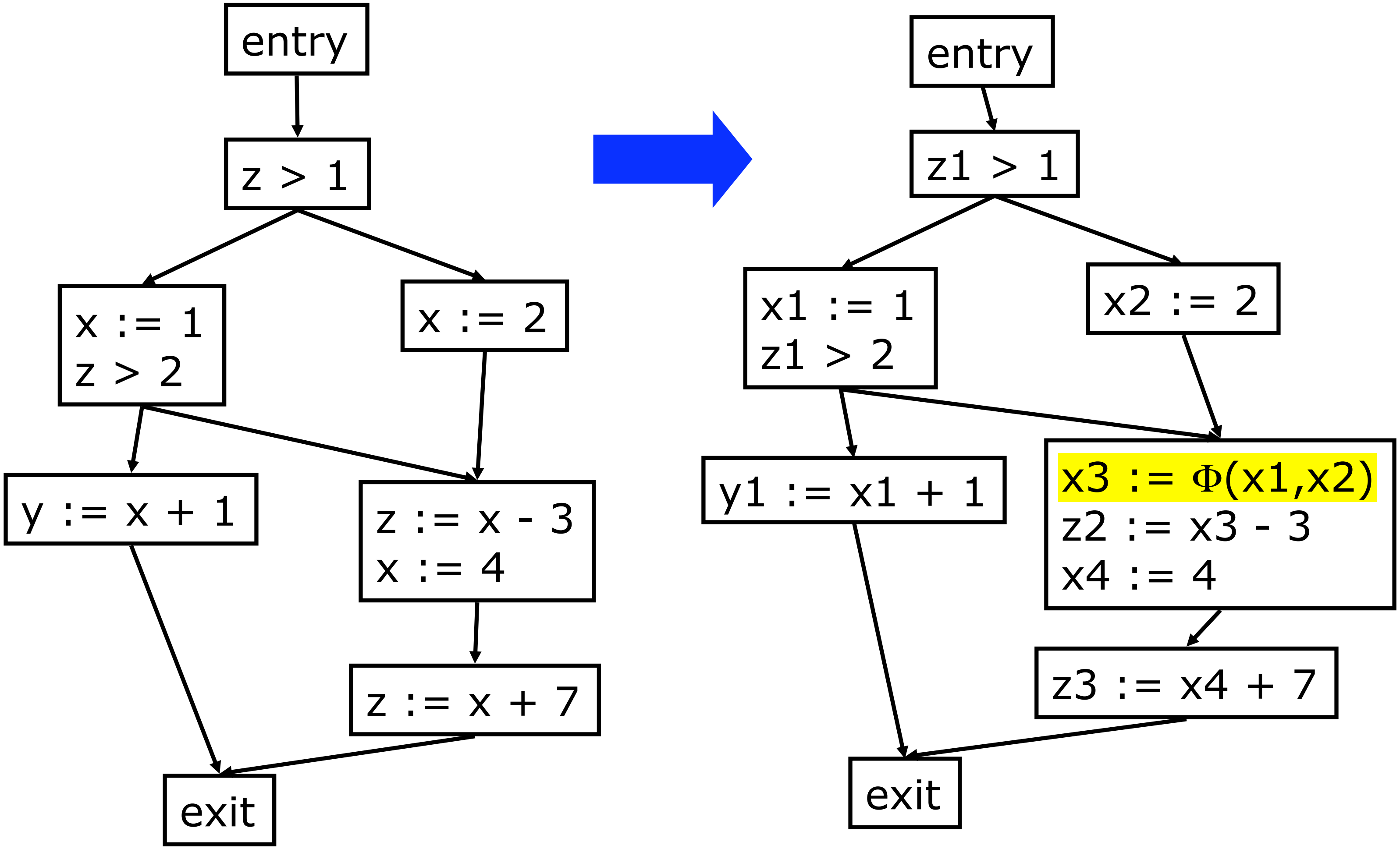
- Use-def information is central to several important optimizations
- The point of ***static single assignment*** form (SSA form) is to represent use-def information explicitly



SSA form

- Static single-assignment form arranges for every value computed by a program to have a unique assignment (aka “definition”)
- A procedure is in SSA form if every variable has (statically) exactly one definition
- SSA form simplifies several important optimizations, including various forms of redundancy elimination

Example



entry
z > 1
x := 1
z > 2
x := 2
y := x + 1
z := x - 3
x := 4
z := x + 7
exit
entry
z1 > 1
x1 := 1
z1 > 2
x2 := 2
y1 := x1 + 1
x3 := Φ(x1,x2)
z2 := x3 - 3
x4 := 4
z3 := x4 + 7
exit

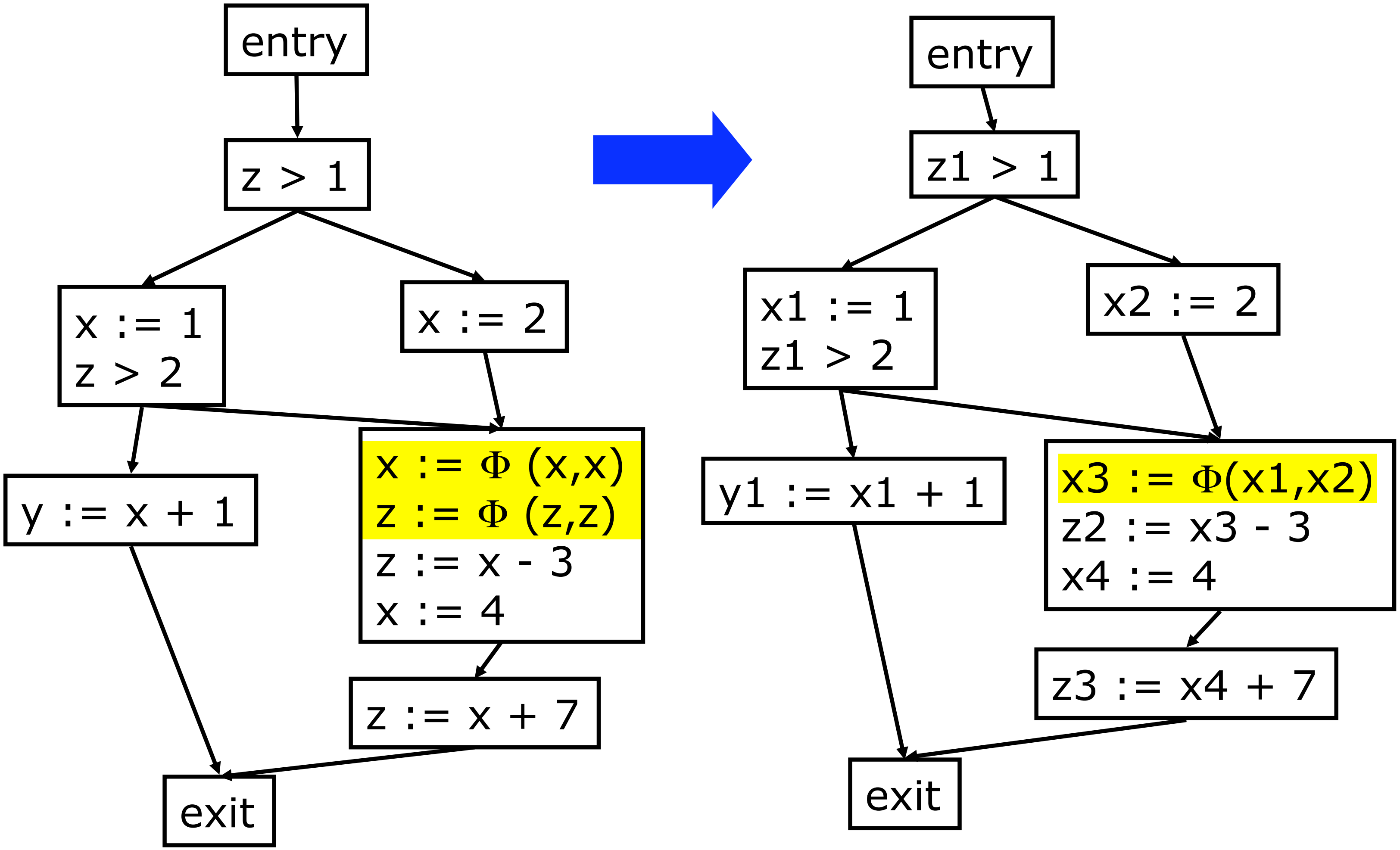
Value numbering in SSA

- In SSA form, if x and a are variables, they are congruent only if they are both live and they are the same variable
- …or if they are provably the same value (by constant or copy propagation)

Creating SSA form

- To translate into SSA form:
  - Insert trivial Φ functions at join points for each live variable
    - `Φ(t,t,...,t)`, where the number of `t`'s is the number of incoming flow edges
  - Globally analyze and rename definitions and uses of variables to establish SSA property
- After we are done with our optimizations, we can throw away all of the statements involving Φ functions
  (i.e. "unSSA")

Example



entry
z > 1
x := 1
z > 2
x := 2
x := Φ (x,x)
z := Φ (z,z)
z := x - 3
x := 4
y := x + 1
z := x + 7
exit
entry
z1 > 1
x1 := 1
z1 > 2
x2 := 2
x3 := Φ(x1,x2)
z2 := x3 - 3
x4 := 4
y1 := x1 + 1
z3 := x4 + 7
exit

SSA form for general graphs

- An SSA form with the minimum number of Φ functions can be created by using ***dominance frontiers***
- Definitions:
  - In a flowgraph, node `a` dominates node `b` (“`a dom b`”) if every possible execution path from entry to `b` includes `a`
  - If `a` and b are different nodes, we say that a strictly dominates `b` (“`a sdom b`”)
  - If `a` sdom `b`, and there is no `c` such that (`a` sdom `c`) and (`c` sdom `b`), we say that `a` is the immediate dominator of `b` (“`a idom b`”)

Dominance frontier

- For a node `a`, the dominance frontier of `a`, `DF[a]`, is the set of all nodes `b` such that `a` strictly dominates an immediate precedessor of `b` but not `b` itself

- More formally:

`DF[a] = {b | ∃ c ∈` Pred`(b)` such that
`a dom c` *but not* `a sdom b}`

Computing DF[a]

- A naïve approach to computing DF[a] for all nodes a would require quadratic time
- However, an approach that usually is linear time involves cutting into parts:
  - $DF_l[a] = \{ b \in Succ(a) \mid idom(b) \neq a \}$
  - $DF_u[a,c] = \{ b \in DF[c] \mid idom(c)=a \wedge idom(b) \neq a \}$

- Then:

$$DF[a] = DF_l[a] \cup \bigcup_{\substack{c \in G \\ (idom(c)=a)}} DF_u[a,c]$$

DF computation, cont'd

- What we want, in the end, is the set of nodes that need Φ functions, for each variable
- So we define DF[S], for a set of flow graph nodes S:

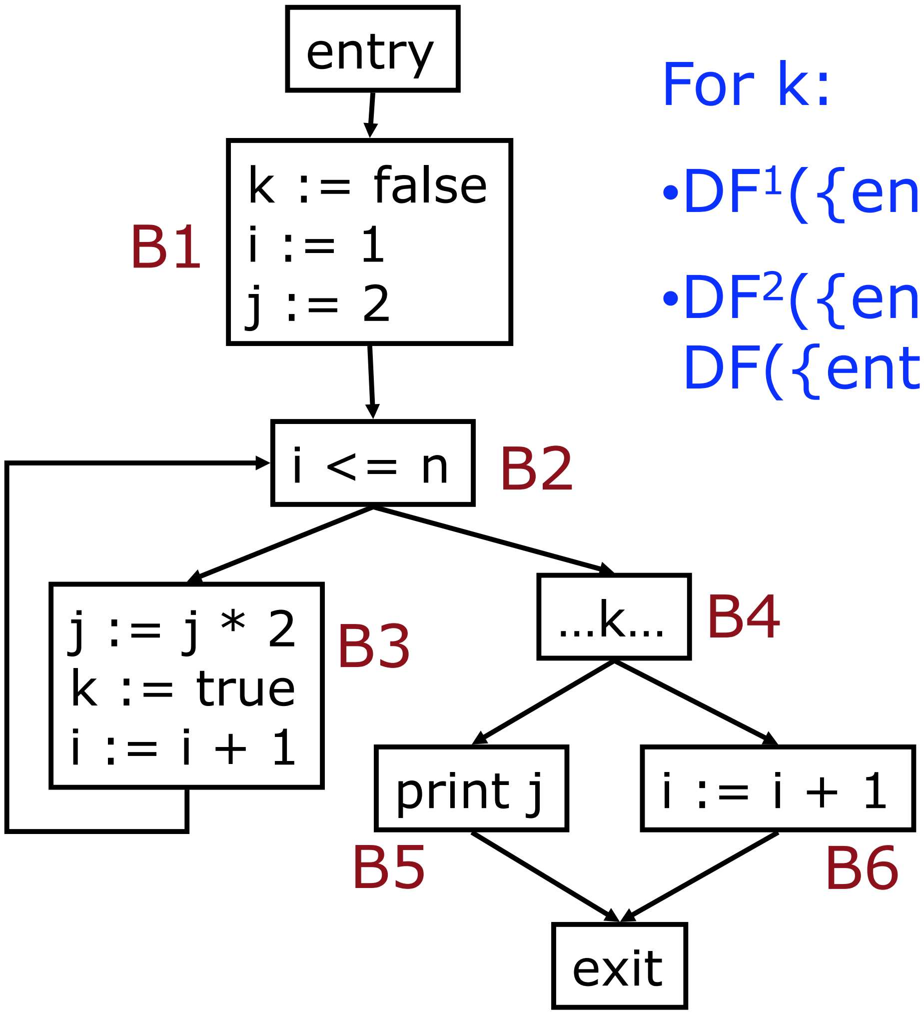
$$DF[S] = \bigcup_{a \in S} DF[a]$$

Iterated DF

- Then, the iterated dominance frontier is defined as follows:
  - $DF^{+}[S] = \lim_{i \to \infty} DF^{i}[S]$
  - where
    - $DF^{1}[S] = DF[S]$
    - $DF^{i+1}[S] = DF[S \cup DF^{i}[S]]$

- If S is the set of nodes that assign to variable t, then $DF^{+}[S \cup \{entry\}]$ is the set of nodes that need Φ functions for t

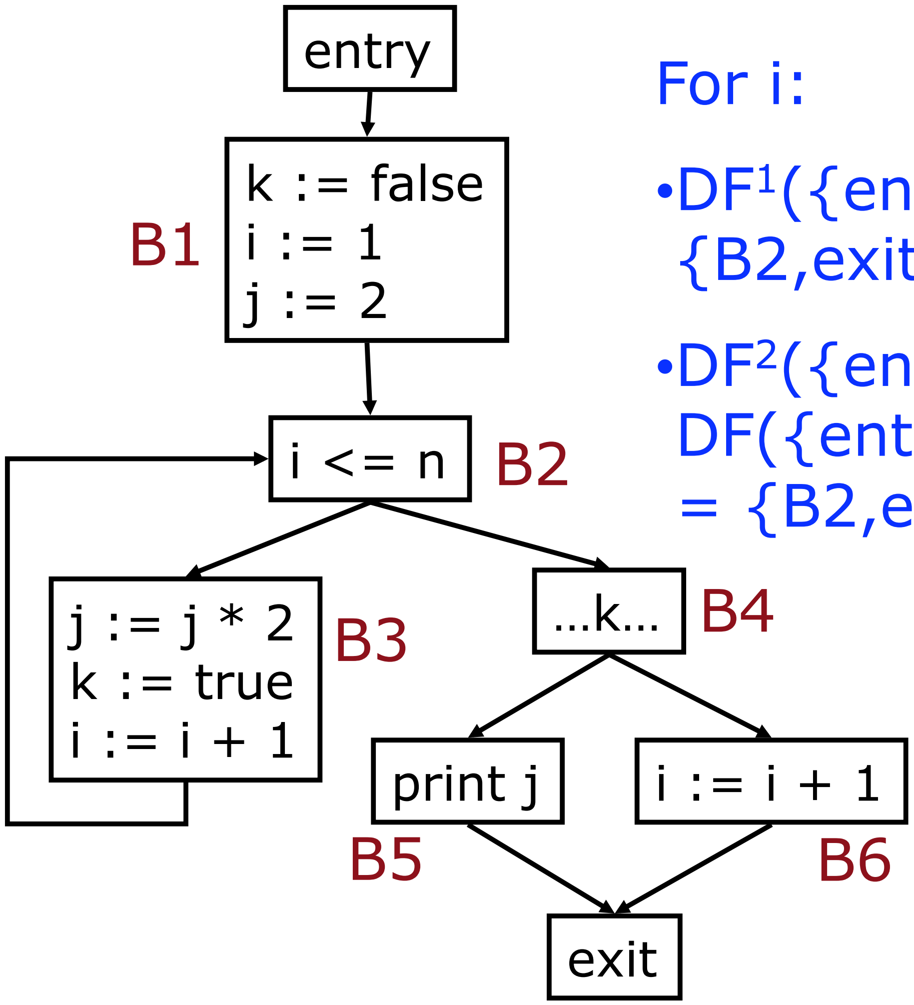
Example



For k:

- DF^1({entry,B1,B3}) = {B2}
- DF^2({entry,B1,B3}) = DF({entry,B1,B2,B3}) = {B2}

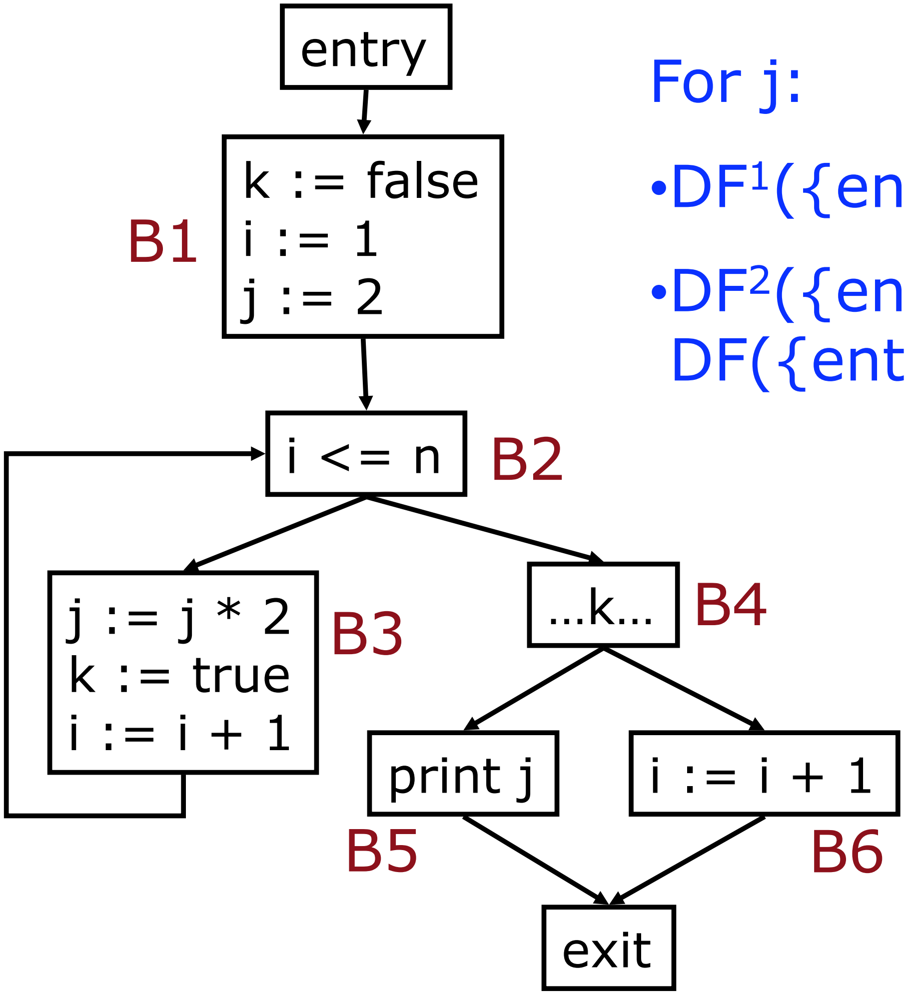
Example



For i:

- $DF^1(\{entry, B1, B3, B6\}) = \{B2, exit\}$
- $DF^2(\{entry, B1, B3, B6\}) = DF(\{entry, B1, B2, B3, B6, exit\}) = \{B2, exit\}$

Example



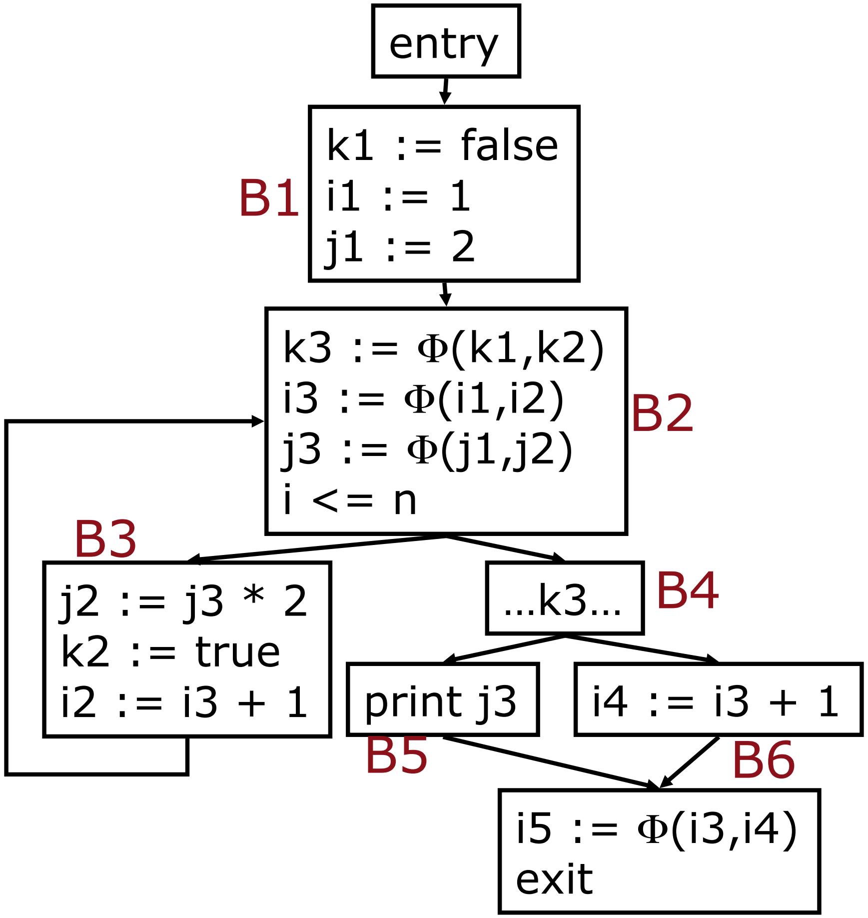
For j:

- $DF^1(\{entry, B1, B3\}) = \{B2\}$
- $DF^2(\{entry, B1, B3\}) = DF(\{entry, B1, B2, B3\}) = \{B2\}$

Example

So, Φ nodes for i, j, and k are needed in B2, and i also needs one in exit

- exit Φ nodes are usually pruned



Other ways to get SSA

- Although computing iterated dominance frontiers will result in the minimal SSA form, there are easier ways that work well for simple languages
- Without knowing the details of your project, I would guess that your translator always knows when it is creating a join point and can keep track of the immediate dominator
- If so, it can also create the necessary Φ nodes during translation

Summary

- SSA form has had a huge impact on compiler design
- Most modern production compilers use SSA form (including, for example, gcc, suif, LLVM, hotspot, ...)
- Compiler frameworks (i.e. toolkits for creating compilers) all use SSA form
- The advantages for simple compilers such as our VSL compiler are low, so using SSA in our project is probably too much overhead…