Chapter 10 Intermediate Representations

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: May 12, 2010

current version: June 16, 2022

Chapter outline: Intermediate Representations

1. Overview
2. Java virtual machine
3. Static single assignment form

References:

1. Michel Schinz, SSA form, act-2007-12-ssa_6.pdf
2. Static Single Assignment Form (SSA), Wikipedia, 20141201
3.

§10.1 Overview

The parser builds a *blank* AST. Semantics analysis adds semantic information to AST. Before generating the actual machine code, the compiler first generates a form of code known as *intermediate representation* (IR). IR is more concise and abstract and easier to generate than real machine code. Most compilers use one or more levels of IR before generating the real machine code.

We usually program in a high-level language, such as C or Java. High-level programs cannot be directly executed on hardware. Instead, only low-level instructions may be executed on hardware. A compiler bridges the high-level programs and the low-level instructions.

§10.1.1 Examples

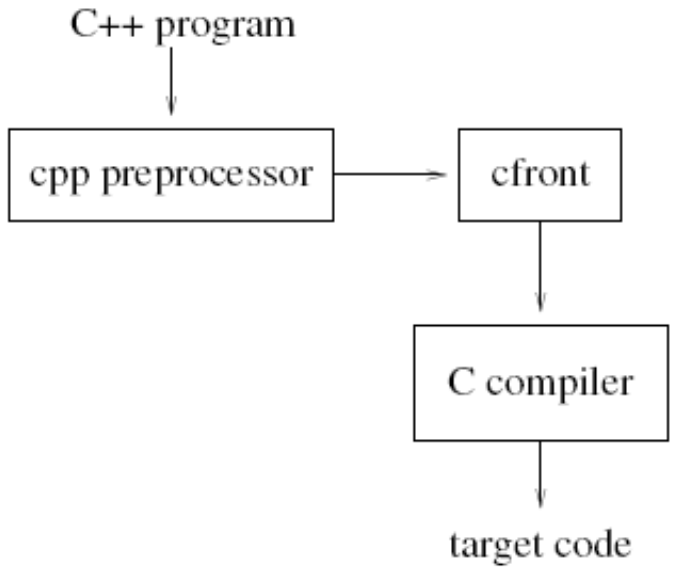


Figure 10.1: Use of `cfront` to translate C++ to C.

Early C++ uses *cpp* and *cfront* to translate C++ code to C code. The output from *cpp* and *cfront*, respectively, may be considered as an intermediate representation.

The *latex* formatting system, shown in Figure 10.2, also makes use of several intermediate languages: tex, dvi, and ps. These intermediate languages improve the portability of the *latex* system as new printers can be easily accommodated.

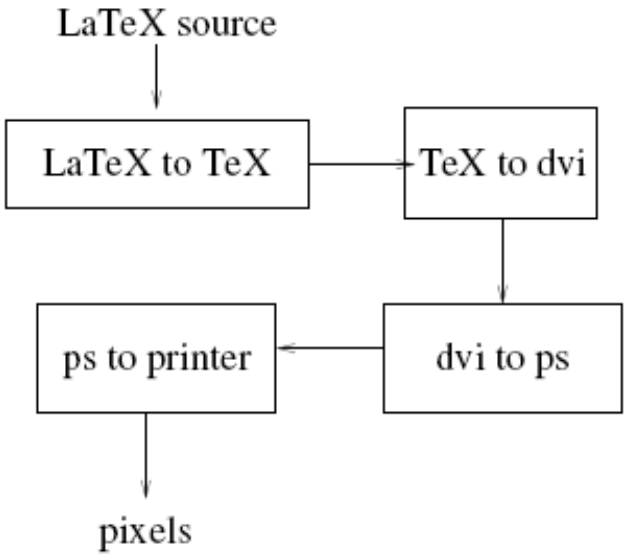


Figure 10.2: Translation from LaTeX into print.

The requirements of an intermediate language includes

1. An intermediate language must be precisely defined. That is, it must capture all the information in a source program.
2. There could be multiple intermediate languages. There are translators among the source language, intermediate languages, and the target language.
3. On each line in the intermediate representation, it should be possible to know the corresponding position in the source program. This makes, say, error messages easier to understand for a programmer.

Introducing several levels of intermediate representations in the compilation process actually slows down the process. However, appropriate IR's make it possible to break down the complicated compilation process into smaller steps and hence makes the task easier.

§10.1.2 The middle-end

The *front-end* is responsible for parsing the input, which is more concerned with the source language.

The *back-end* is responsible for generating the target code, which is more concerned with the target platform.

In a compiler, there are usually a set of components between the front-end and the back-end, which is called the *middle-end*.

For a multiple-source, multiple-target compiler system, such as GCC, putting the components common to all compilers makes compiler development more economical. See Figure 10.3.

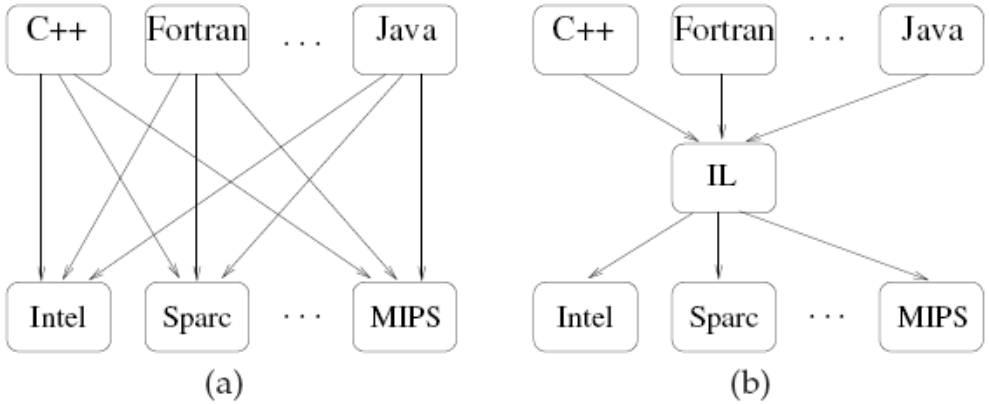


Figure 10.3: A middle-end and its ILs simplify construction of a compiler suite that must support multiple source languages and multiple target architectures.

Other advantages:

1. An IL allows many components to cooperate by sharing a common representation of programs. Information about a program, such as variable names, types, source lines, etc., is available. Many other tools, such as class browsers, debuggers, and profilers, can also be built around IR.
2. An IL simplifies development and testing.

3. A major portion of the IL can be re-used.
4. Different development teams/vendors can cooperate by agreeing on a common IL.
5. IL makes prototyping new algorithms (e.g., for CSE) and new compiler organizations (e.g., order of phases) easier in a research environment
6. The IL and its interpreter can serve as a reference definition of a programming language. For example, DIANA (descriptive intermediate attributed notation for Ada) is an IL that defines Ada.
7. Interpreters for an IL help testing and porting a compiler.
8. IL helps the development of retargetable code generators and hence enhances the portability of a compiler.

Common IL includes P-code (Pascal), bytecode (Java), bitcode (LLVM), DIANA (Ada), webassembly, etc. GCC uses two ILs. MS C uses CIL (common intermediate language). CLR (common language

runtime) is an interpreter for CIL. [a]

[a] IL and IR are synonyms.

§10.2 Java virtual machine

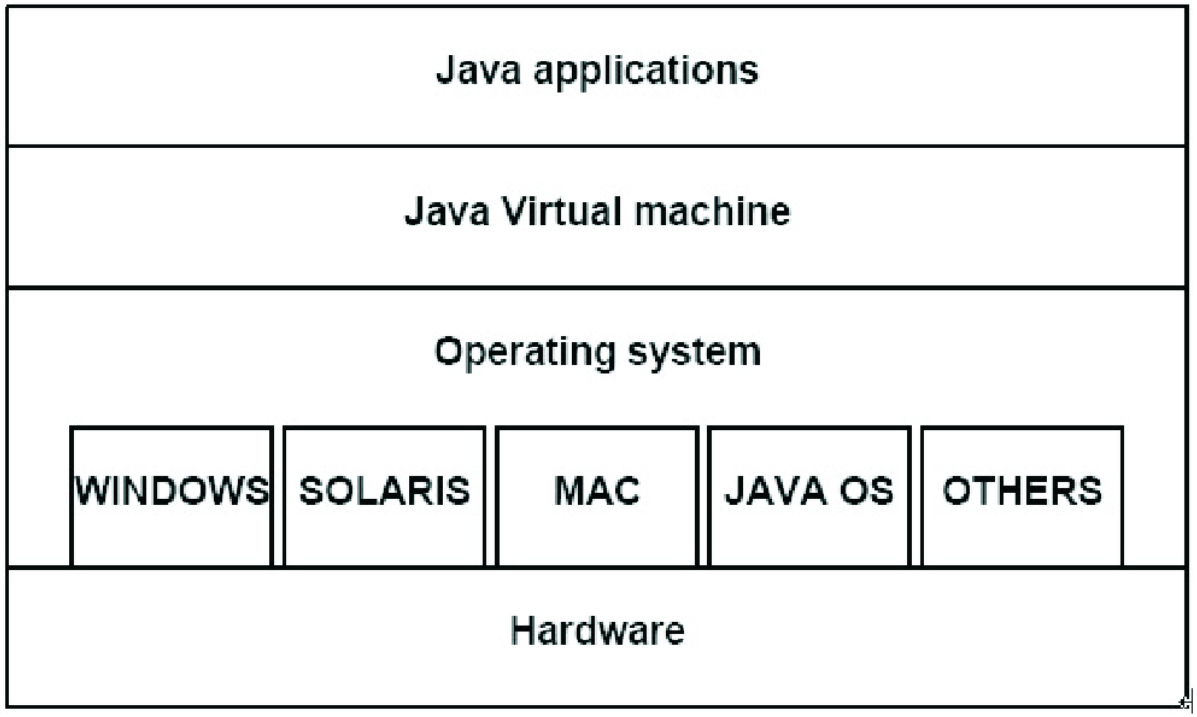


Figure 1: Java VM and applications

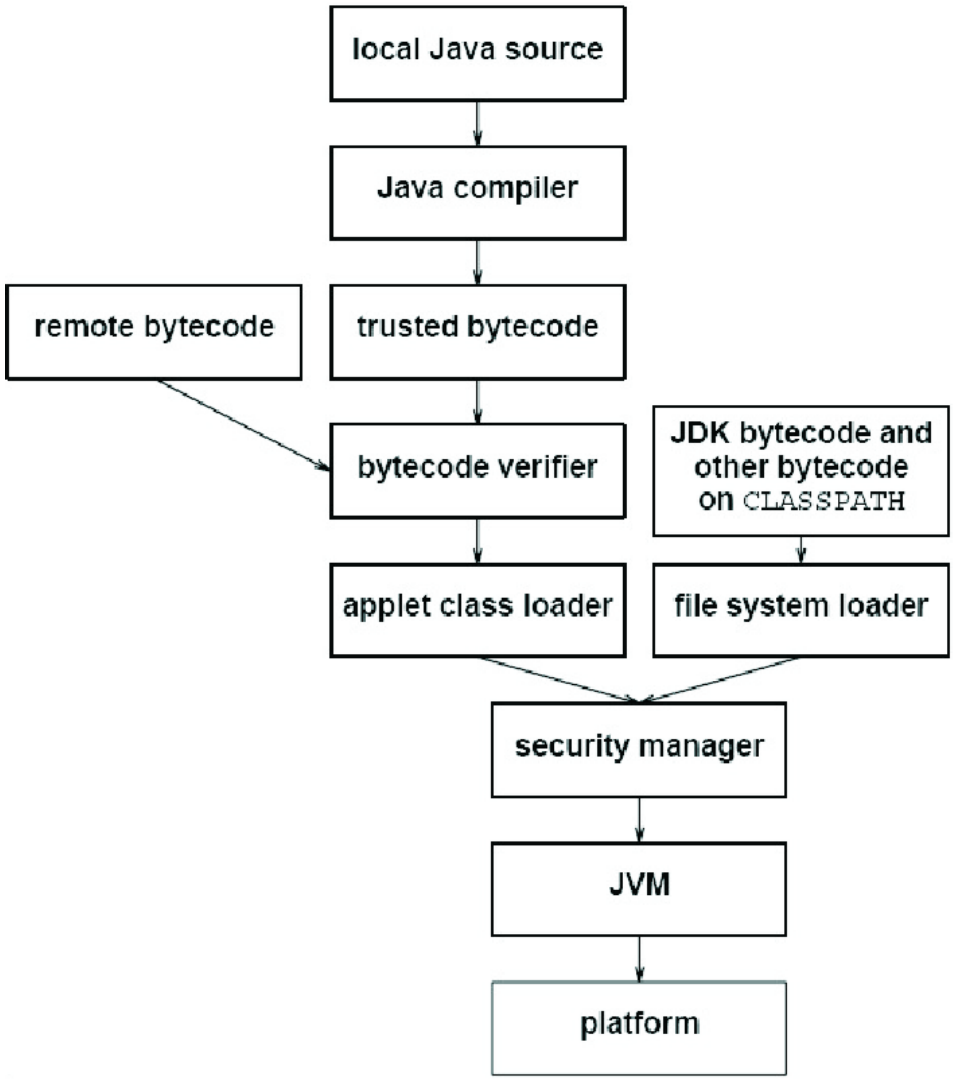


Figure 2: Java system

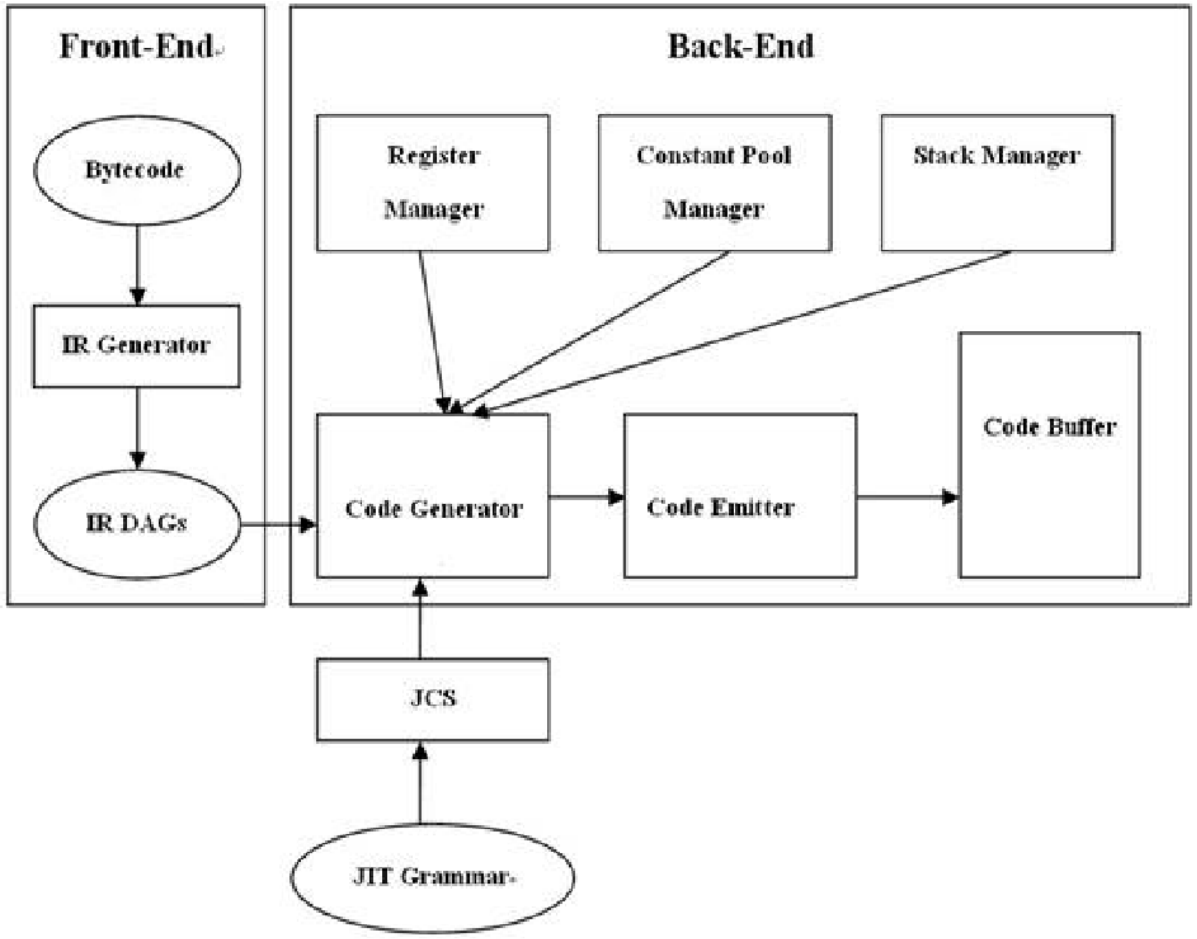


Figure 3: Structure of a JIT.

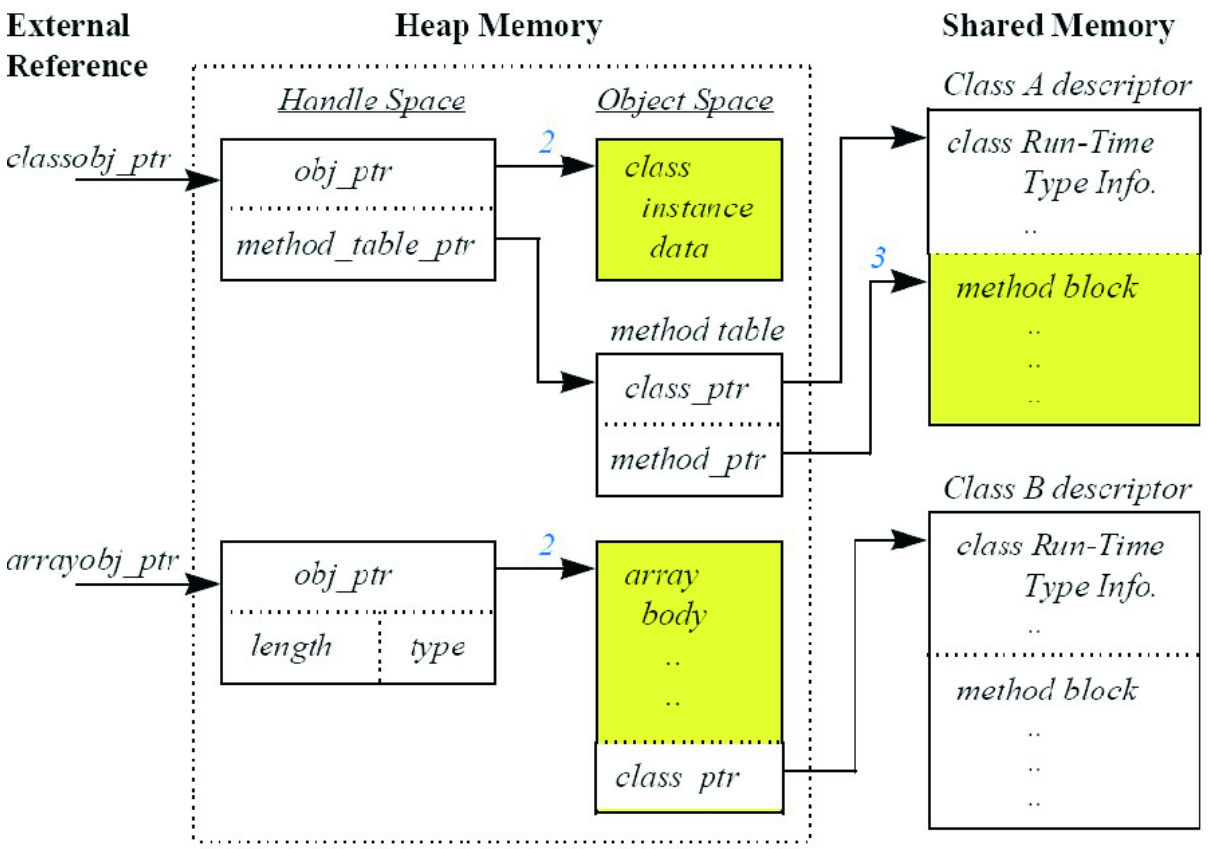


Figure 8. Run-time memory organization used by Java interpreter.

Figure 4: Java run-time memory organization.

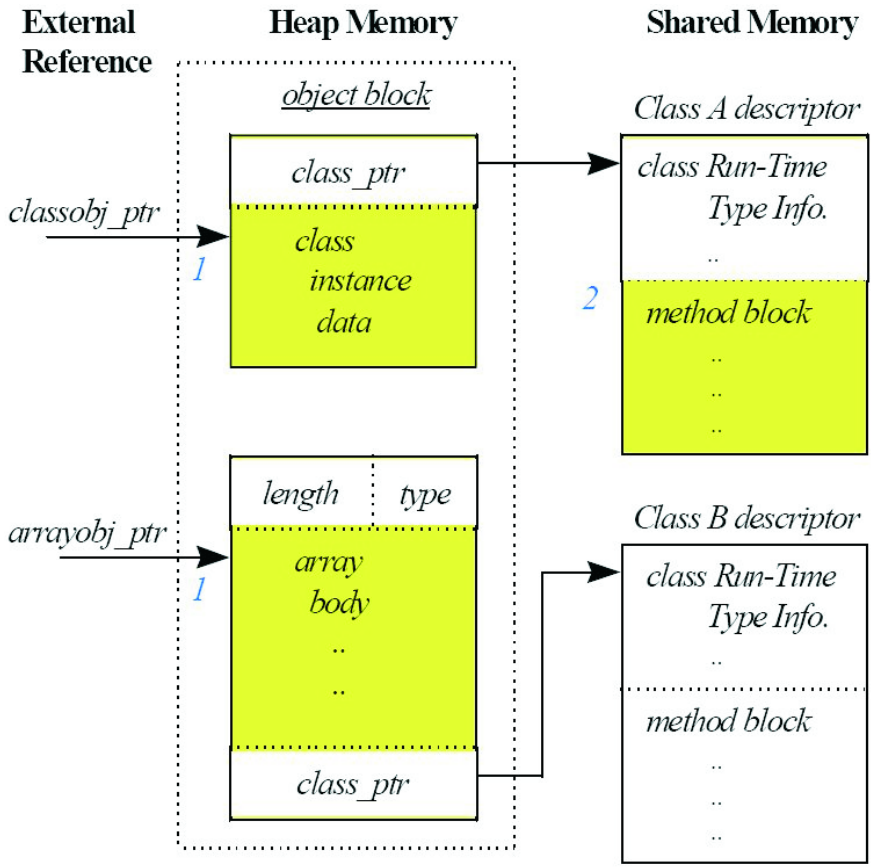


Figure 9. Run-time memory organization used by Java bytecode translator.

Figure 5: Revised Java run-time memory organization.

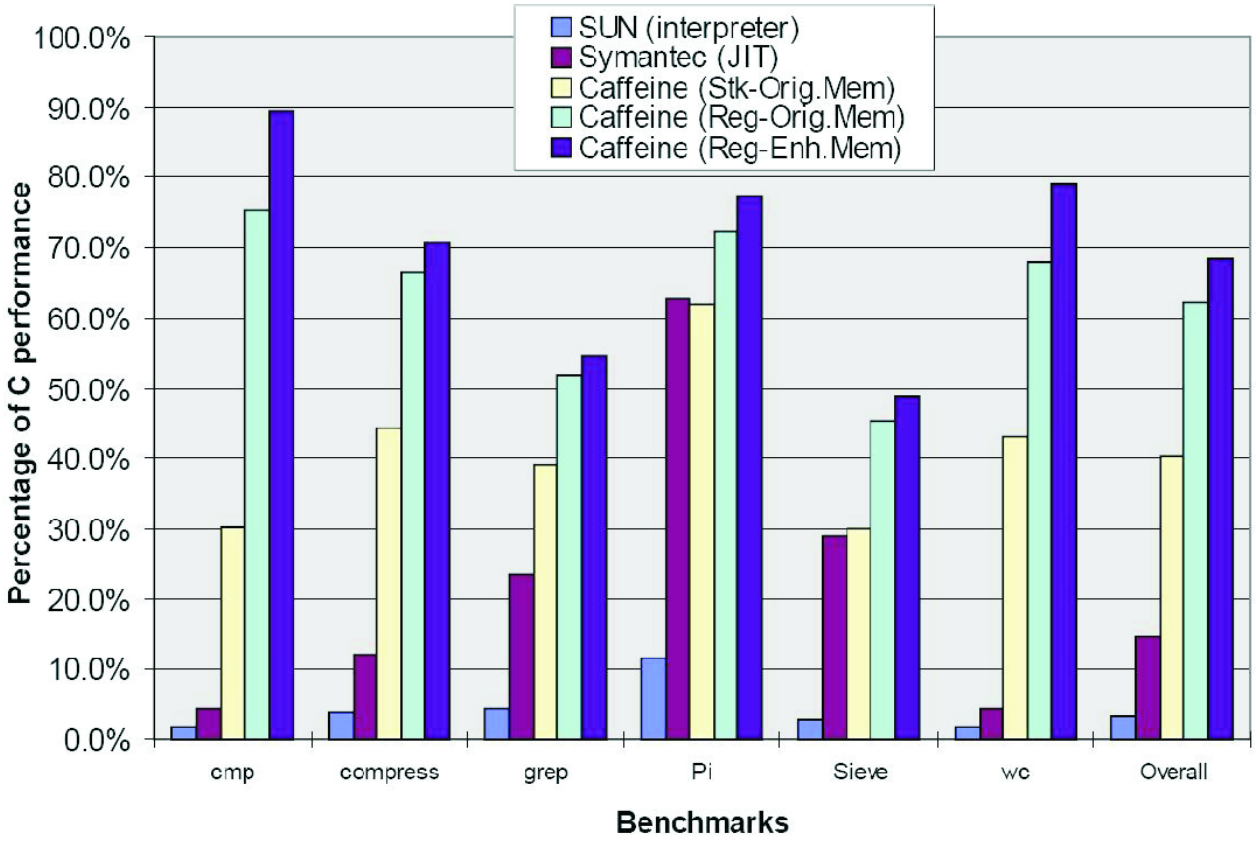


Figure 11. Experiment results on different approaches. All numbers are relative speed to the equivalent C code compiled by Microsoft Visual C/C++ compiler with optimization level two.

Figure 6: Real JIT performance.

Compactness A Java compiler compiles Java source code to *bytecode*, which is similar to conventional `.exe` files. Bytecode is put in a *classfile* (`.class`).

Bytecode is compact. It is aimed at a stack machine (Java virtual machine). All instructions, such as `iadd`, operate on operands on the stack top. Thus, an instruction usually does not have operand addresses (and is short).

There is no register in the bytecode. Note that stack manipulation is slower than registers. The stack machine allows many short instructions since instructions have implicit operands on the stack. This leads to smaller code size. On the other hand, since stack manipulation is slower, a stack machine is usually slower than a register machine.

Sometimes there are several instructions (of different lengths) that achieve the same effect. For example, `iconst_0` (1 byte) and `ldc_w 0` (3 bytes). Frequently used instructions have special short formats.

Safety JVM is designed to execute safely, not to compromise a user's computer. The instruction set and the virtual machine are designed with security in mind so that efficient security checks are possible. When a class file is loaded into JVM, a bytecode verifier will check the class file against various security requirements before execution.

In a purely zero-address form, to load a value of a register into the stack takes two steps:

1. First, compute the register number, and push it onto the stack
2. Next a `load` instruction pops off the register number and load the value of that register onto stack

Checking the validity of the register number must be done at run time.

In contrast, Java uses *immediate operand* (such as `iload 5`) as a register number. When loading the bytecode, the verifier can check if register 5 is accessible by the current method. No run-time check is needed.

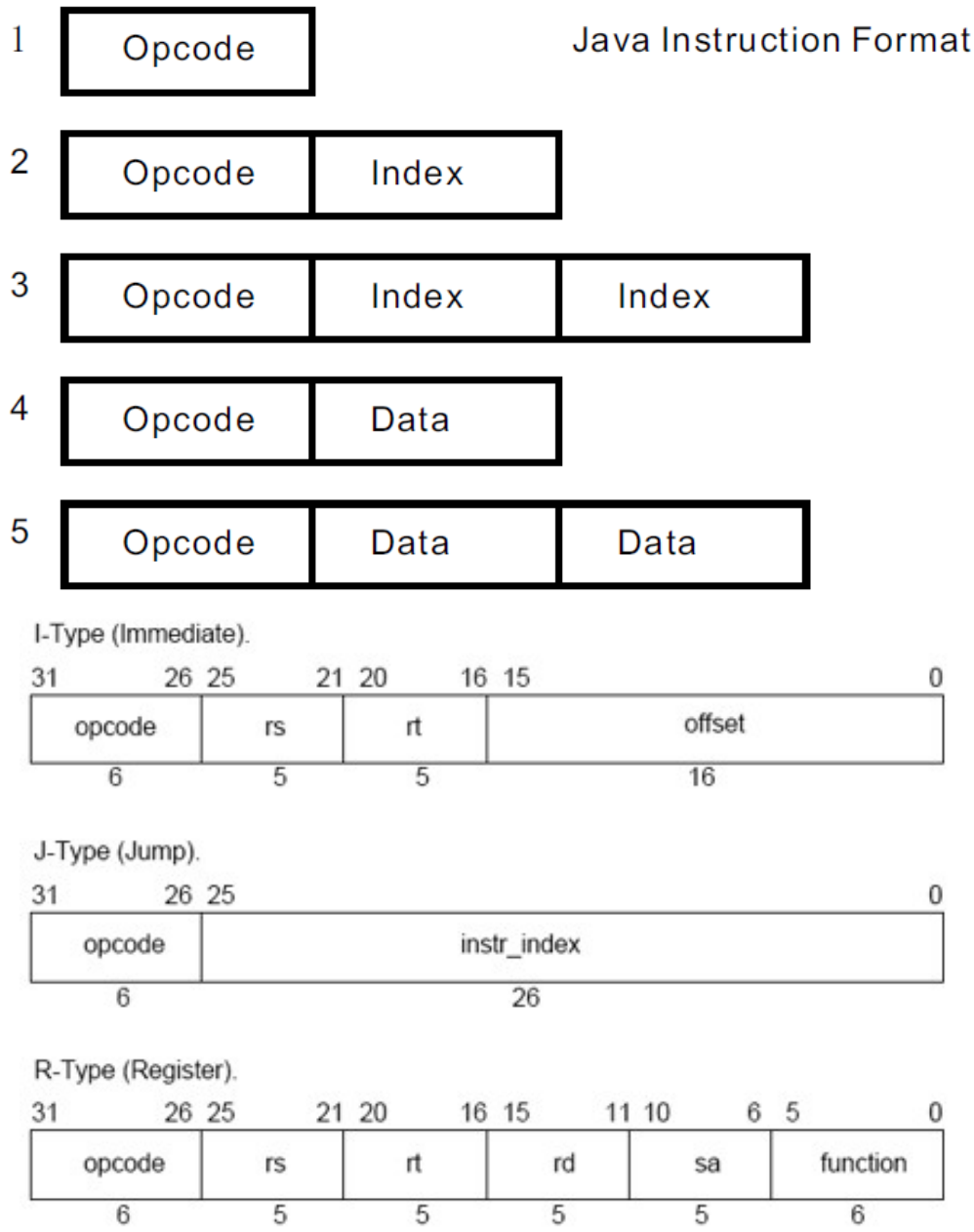


Figure 7: Comparison of Java and MIPS instruction formats.

§10.2. Contents of a class file

A class file is partitioned into several sections.

Java provides primitive and reference types. Figure 10.4 shows the abbreviations for types. Below is an example reference type:

```
Ljava/lang/String;
```

Type	JVM designation
`boolean`	`Z`
`byte`	`B`
`double`	`D`
`float`	`F`
`int`	`I`
`long`	`J`
`short`	`S`
`void`	`V`
Reference type *t*	L*t*;
Array of type *a*	[*a*

Figure 10.4: Java types and their designation in the JVM. All of the integer-valued types are signed. For reference types, *t* is a fully qualified class name. For array types, *a* can be a primitive, reference, or array type.

A class file contains a *constant pool*, which contains all the constants (integer, real, strings, etc.) used in that class. A constant is referenced through its ordinal position (1, 2, etc.), not by its byte-offset, inside the constant pool.

Some instructions, such as `ldc`, uses 1 byte for constant-pool references. Some instructions, such as `ldc_w`, uses 2 bytes for constant-pool references. (Hence, `ldc_w` can reference more entries.)

§10.2.3 JVM instructions

JVM supports the following kinds of instructions:

1. Arithmetic: `iadd` pops off two values from the stack, adds them as integers (32-bit 2's complement), and pushes the result back to the stack. There are other additions: `fadd`, `ladd`, `dadd`. There are similar instructions for substraction, multiplication, division, and remainder.

2. Register traffic: JVM has no real registers. Instead, it has an unlimited number of *virtual registers*. The virtual registers are used for a method's local variables. For static methods, register 0 is the 1st argument. For instance methods, register 0 holds `this`. When a method is invoked, argument values are automatically popped from the caller's stack and deposited into the low-numbered registers. JVM registers are untyped. `long` and `double` occupy a pair of even-odd registers. An example, `iload 2` takes two bytes. A similar instruction `iload_2` takes one byte.

   Example: `istore 10`.

Example: `fload n`.

JVM uses an integer interpretation for boolean: 0 as false and 1 as true.

`char`, `byte`, and `short` are treated as `int`.

An object reference takes four bytes. It is loaded and stored with `aload 4` and `astore 5` instructions. 0 is treated as `null`.

The bytecode verifier will check for type errors, such as "`astore 4`; `iload 4`".

3. Registers and types: The bytecode verifier checks the types of all values (in registers and stack). "`iload 4`; `fload 5`; `fadd`" will cause a type error before execution. Instead, we should use "`iload 4`; `i2f`; `fload 5`; `fadd`".

4. Static fields are manipulated with `getstatic fieldname type` and `putstatic` instructions. Example:

   ```
   getstatic java/lang/System.out Ljava/io/PrintStream
   ```

   The "`getstatic fieldname type`" instruction takes 3 bytes: one

for the operator and the remaining two form an index of an entry in the constant pool. That entry will contain the `fieldname` and `type`.

(A compiler will generate the correct `type` in a `getstatic` field. What if a hacker modifies the `type`? How and will the bytecode verifier check this?)

`pustatatic` pops a value off the stack and stores it into the specified location.

5. Instance fields are similarly manipulated with `getfield` and `putfield` instructions. "`getfield x I`" needs to use the entry on the stack top, which is an object reference.

   "`puttfield x I`" stores the value on stack top to the specified field. It also needs to use the entry on the stack top, which is an object reference.

6. Branching: `goto delta` takes three byes. The last two bytes forms a 16-bit offset from the location of the `goto` instruction.

   There is also a 5-byte version: `goto_w delta`, which accommodates a 4-byte offset.

There are also conditional jump instructions. Some conditional jumps combine comparison with jump. Others make use of two instructions: one for comparison and the other for conditional jump. Examples: `if_icmpeq` (compare two values on stack top), `if_icmpne`, `if_icmplt`, `if_icmple`, `if_icmtgt`, `if_icmtge`, `ifeq` (compare the number on stack top with zero), `ifne`, `iflt`, `ifle`, `ifgt`, `ifge`.

7. Calling a static method:

```
invokestatic java/lang/Math/pow(DD)D
```

pops two double values from the stack, calculates the exponentiation, and pushes the result back on stack top. The parameters are pushed from left to right. The method name and signature are stored in the constant pool. The `invokestatic` instruction uses an ordinal index into the constant pool.

8. Calling an instance method:

```
invokevirtual java/io/PrintStream/print(Z)V
```

The parameters and the implicit object reference are taken from the stack.

9. All constructors are named `<init>`, which are invoked with the `invokespecial` instruction.

10. stack operations: `dup` makes a copy of the value on stack top. `new t` leaves a reference on TOS. Then a constructor will be invoked to use that reference. A `dup` instruction is used to duplicate that reference. Note the bizarre behavior of `dup_x1`.

Case study. Javac compiler.

The intermediate form in `javac` is a tree, which consists of 50 kinds of nodes. The root is a `JCCompilationUnit` node.

Figure 8 is the structure of the javac compiler. It consists of 7 parts: `parseFiles`, `enterTrees`, `proecssAnnotations`, `attribute`, `flow`, `desugar`, and `generate`.

`parseFiles` reads the input Java program and produces an intermediate form, which is a tree.

`enterTrees` identifies and puts symbol into the symbol table. `attribute` determines the attributes of the nodes in the tree. It is the semantic analyzer. `flow` performs data flow analysis in the tree.

Java 5.0 adds the new *annotation* feature. `proecssAnnotations` will process the annotations. `desugar` translates all the new features of Java 5.0 in the intermediate form into old constructs so that `generate` can generate class files that are compatible with existing Java virtual machine.

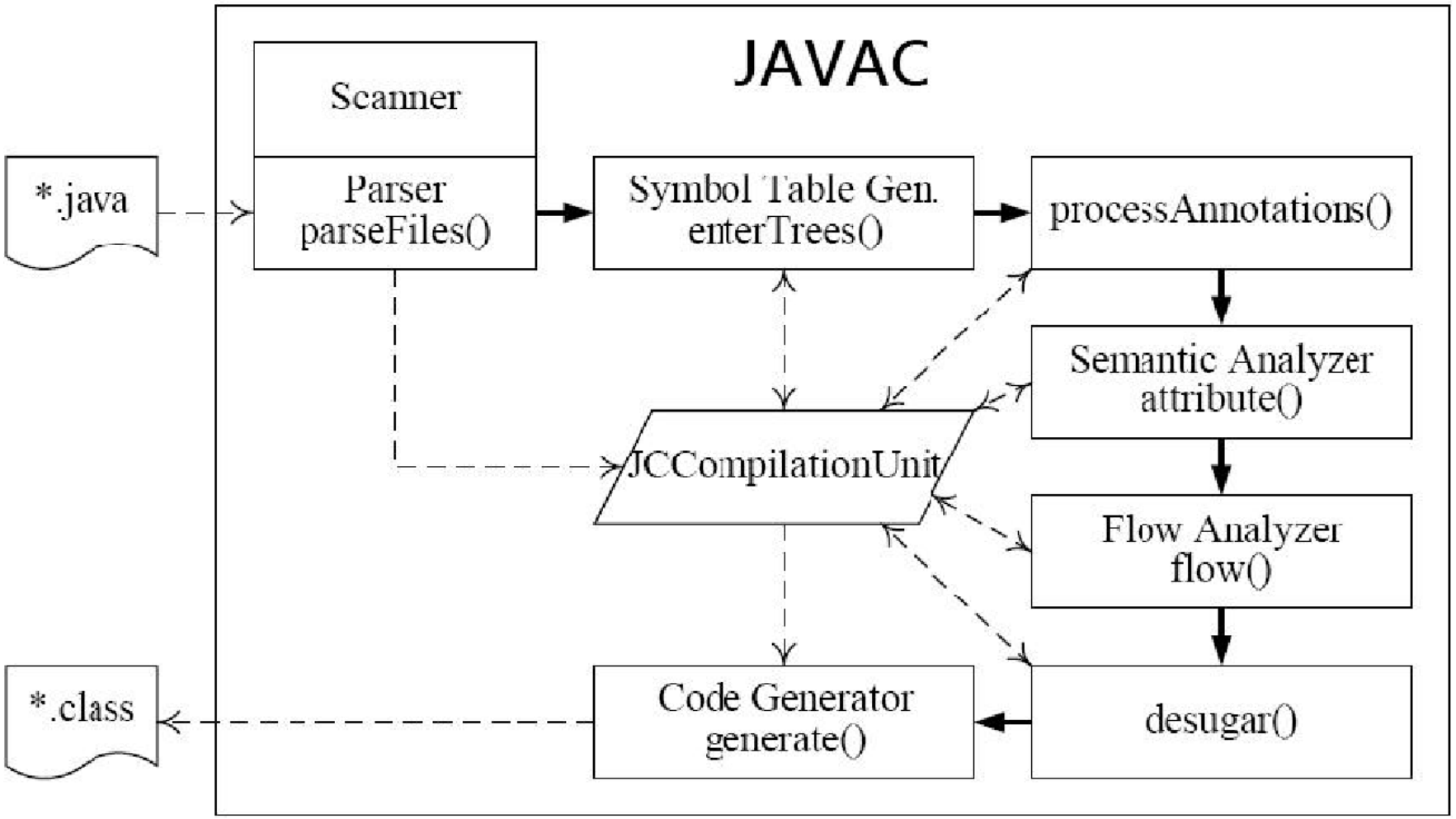


Figure 8: Javac Compiler

data flow analysis:

- u-d-chain: use-def chain.
  For the use of `s` at B3, the list of reaching definitions of `s`: B1, B3, and B4.
- d-u-chain: def-use chain.
  For the definition of `s` at B3, the list of uses of `s`: B3 and B4.

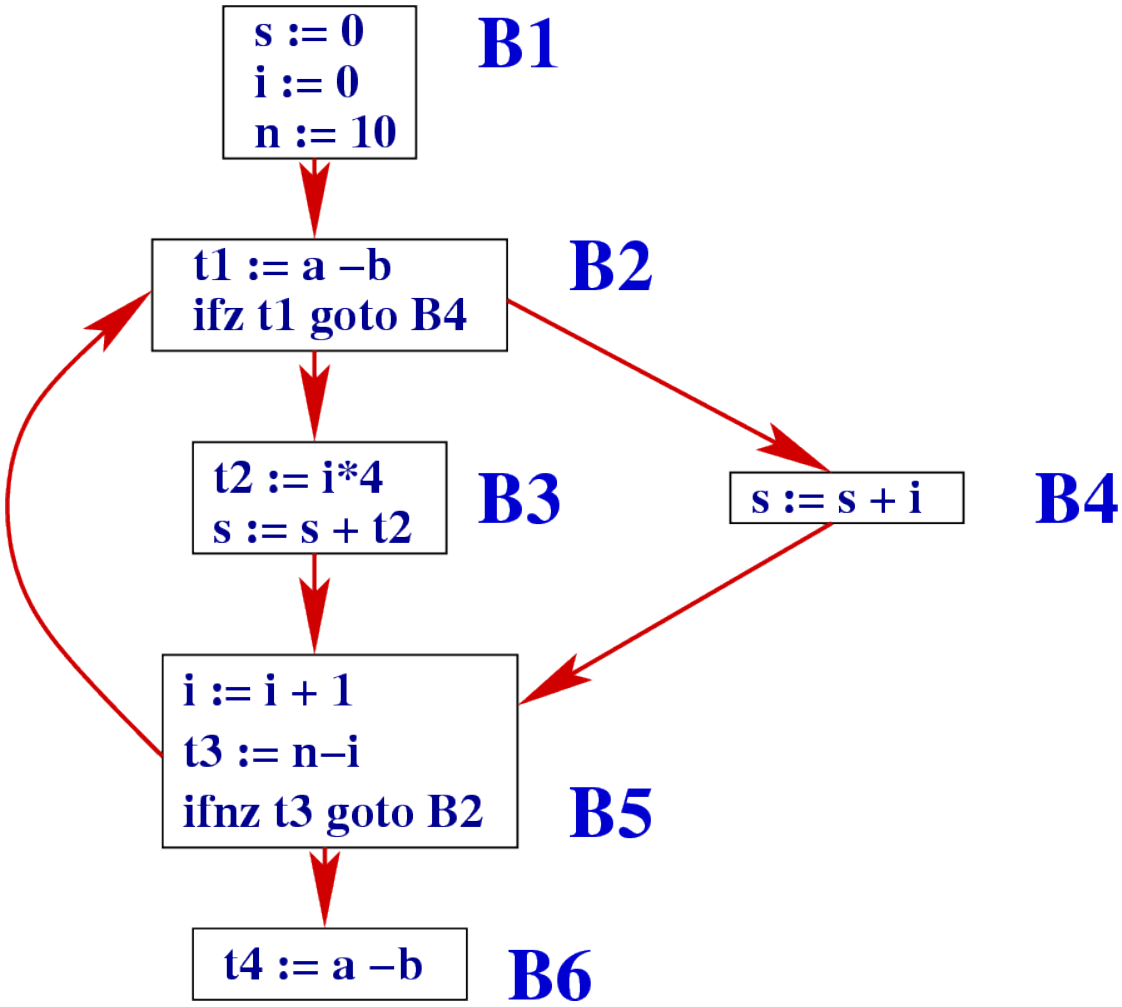


Figure 9: data flow analysis

Two uses of a variable `x` with disjoint definitions are essentially distinct variables. We may give them different names, such as `x1` and `x2`. This process is called *live range splitting*. For example,

```
x := 10;
y := x + 20;
x := 30;
z := x + 40;
```

SSA: Each variable is assigned exactly once in the program text.

Method: Rename variables and add ϕ assignments.

Easy for structured flow graphs (`if` and `while`).

Purpose: Make ud-chains and du-chains more explicit.

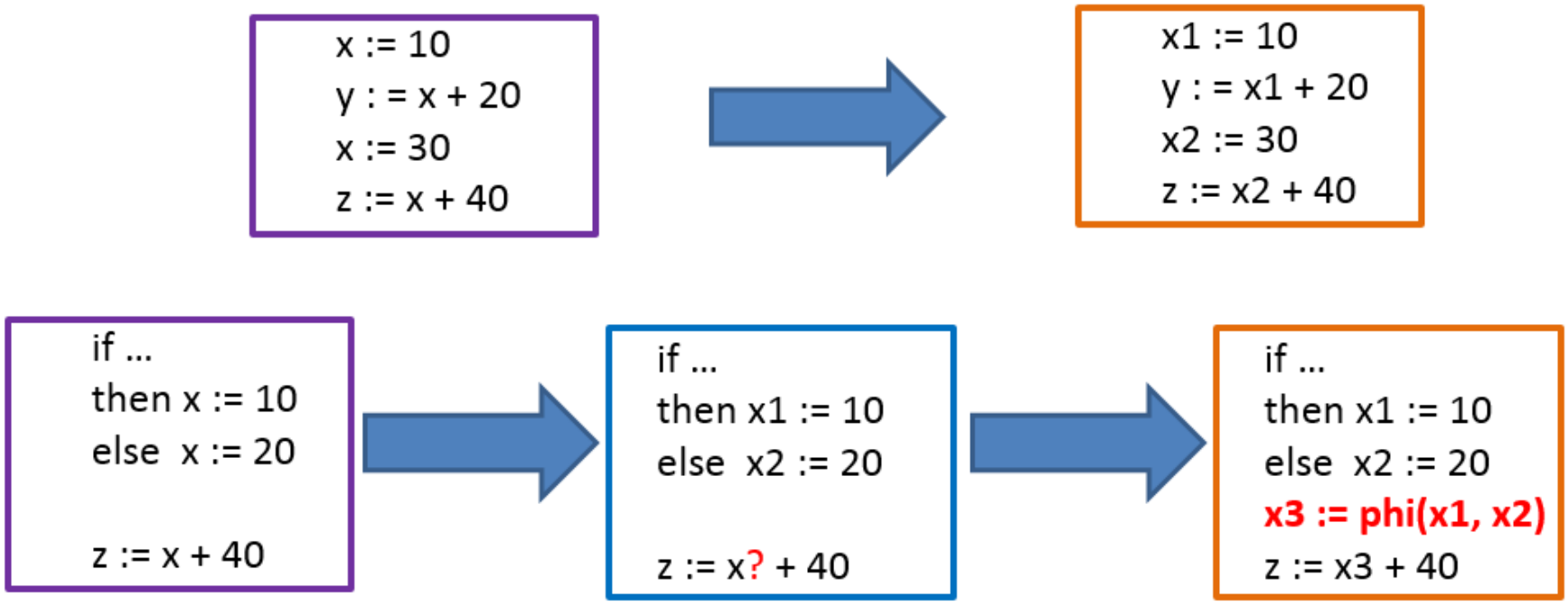


Figure 10: SSA form

How to place the ϕ statement in a general control flow graph?

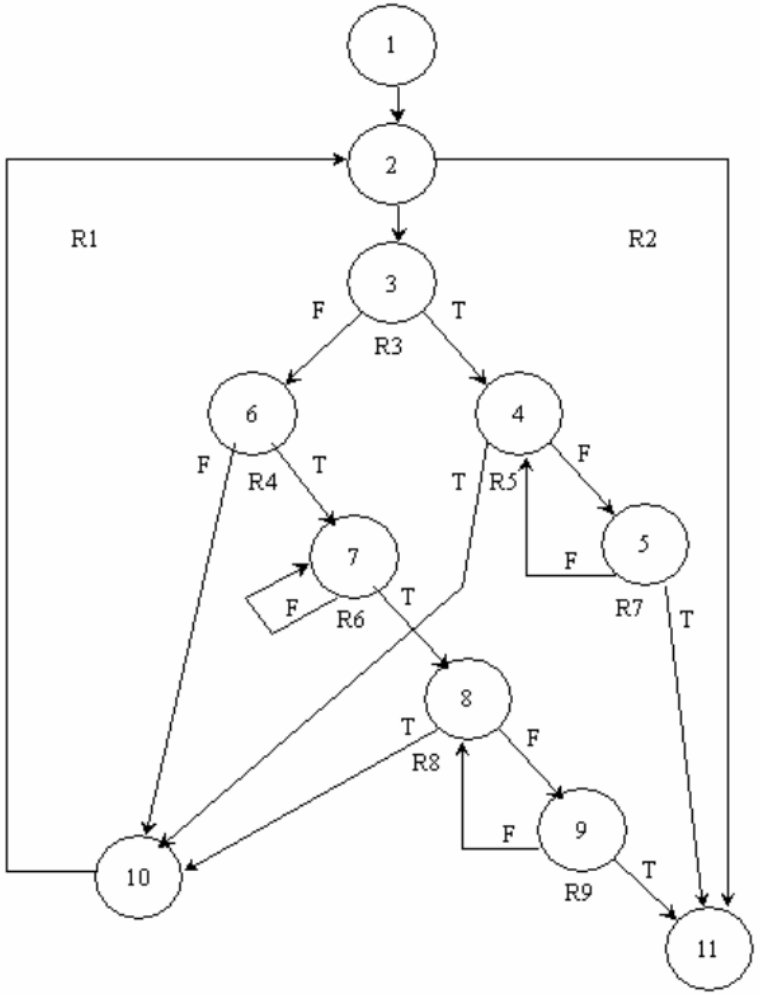


Figure 11: A general control flow graph

§10.3 Static single assignment form

Background. AST ⇒ CFG (control flow graph, structured or unstructured) ⇒ SSA form

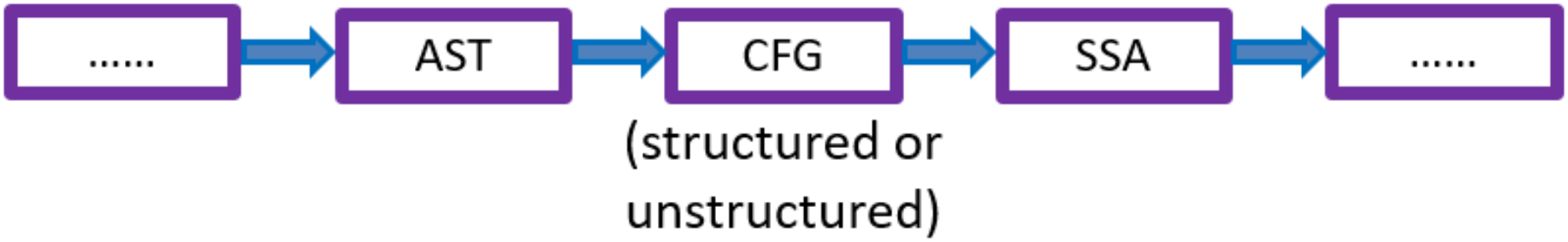


Figure 12: Current progress of this course.

§10.3 Static single assignment form

Charcteristic: each use is reachable from exactly one definition.

Static single assignment form makes a program's data flow (def-use relation) explicit from the names of variables. Many optimizations are thus simplified.

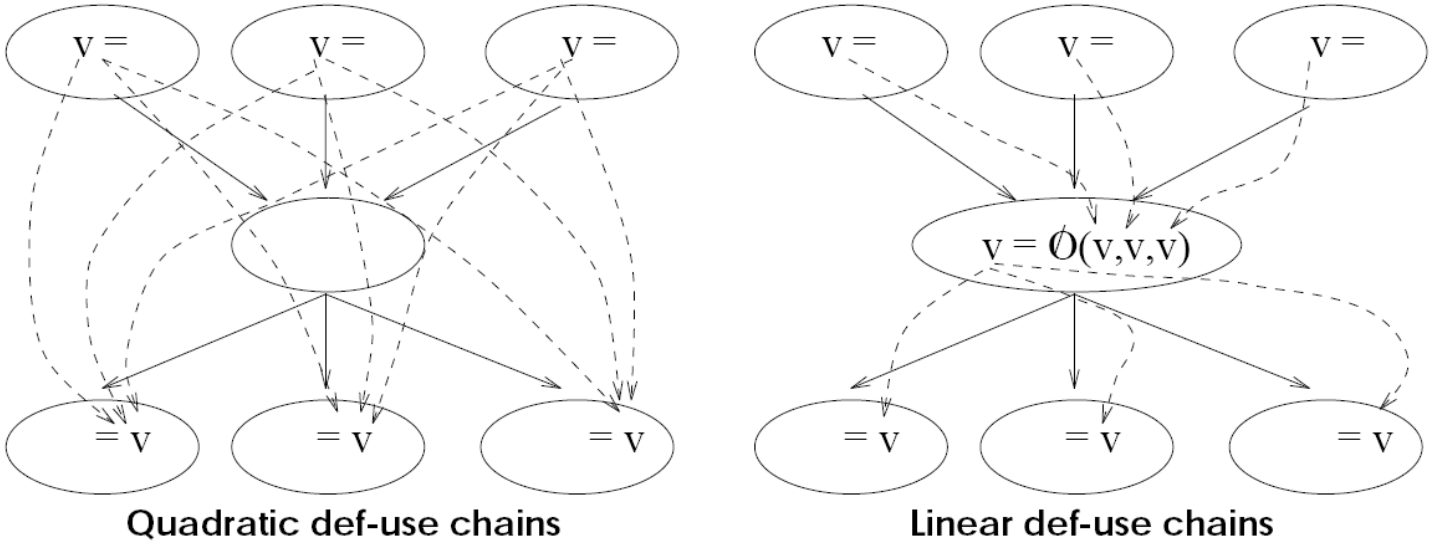
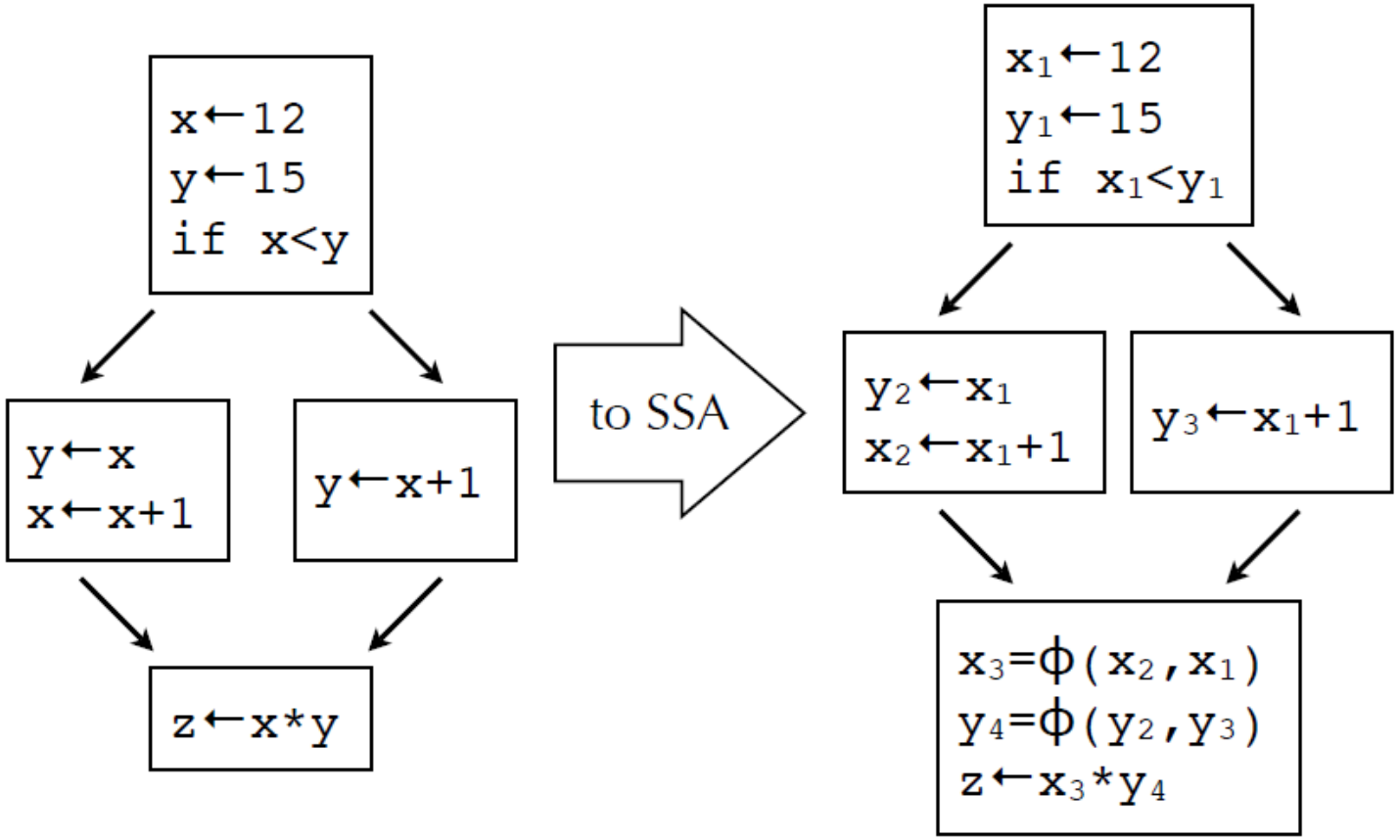


Figure 13: Motivation of SSA form (from R. Cytron, PLDI Tutorial Notes-part6.pdf)

Figure 14: Explicit vs implicit.


x←12
y←15
if x<y
y←x
x←x+1
y←x+1
z←x*y
to SSA
x1←12
y1←15
if x1<y1
y2←x1
x2←x1+1
y3←x1+1
x3=ϕ(x2,x1)
y4=ϕ(y2,y3)
z←x3*y4

```
i ← 1
j ← 1
k ← 1
l ← 1
repeat
    if p
    then
        j ← i
        if q
        then  l ← 2
        else  l ← 3
        k ← k + 1
    else  k ← k + 2
    call      (i, j, k, l)
    repeat
        if r
        then
            l ← l + 4
    until s
    i ← i + 6
until t
```

(a)

```
i1 ← 1
j1 ← 1
k1 ← 1
l1 ← 1
repeat
    i2 ← φ(i3, i1)
    j2 ← φ(j4, j1)
    k2 ← φ(k5, k1)
    l2 ← φ(l9, l1)
    if p
    then
        j3 ← i2
        if q
        then  l3 ← 2
        else  l4 ← 3
        l5 ← φ(l3, l4)
        k3 ← k2 + 1
    else  k4 ← k2 + 2
    j4 ← φ(j3, j2)
    k5 ← φ(k3, k4)
    l6 ← φ(l2, l5)
    call      (i2, j4, k5, l6)
    repeat
        l7 ← φ(l9, l6)
        if r
        then
            l8 ← l7 + 4
        l9 ← φ(l8, l7)
    until s
    i3 ← i2 + 6
until t
```

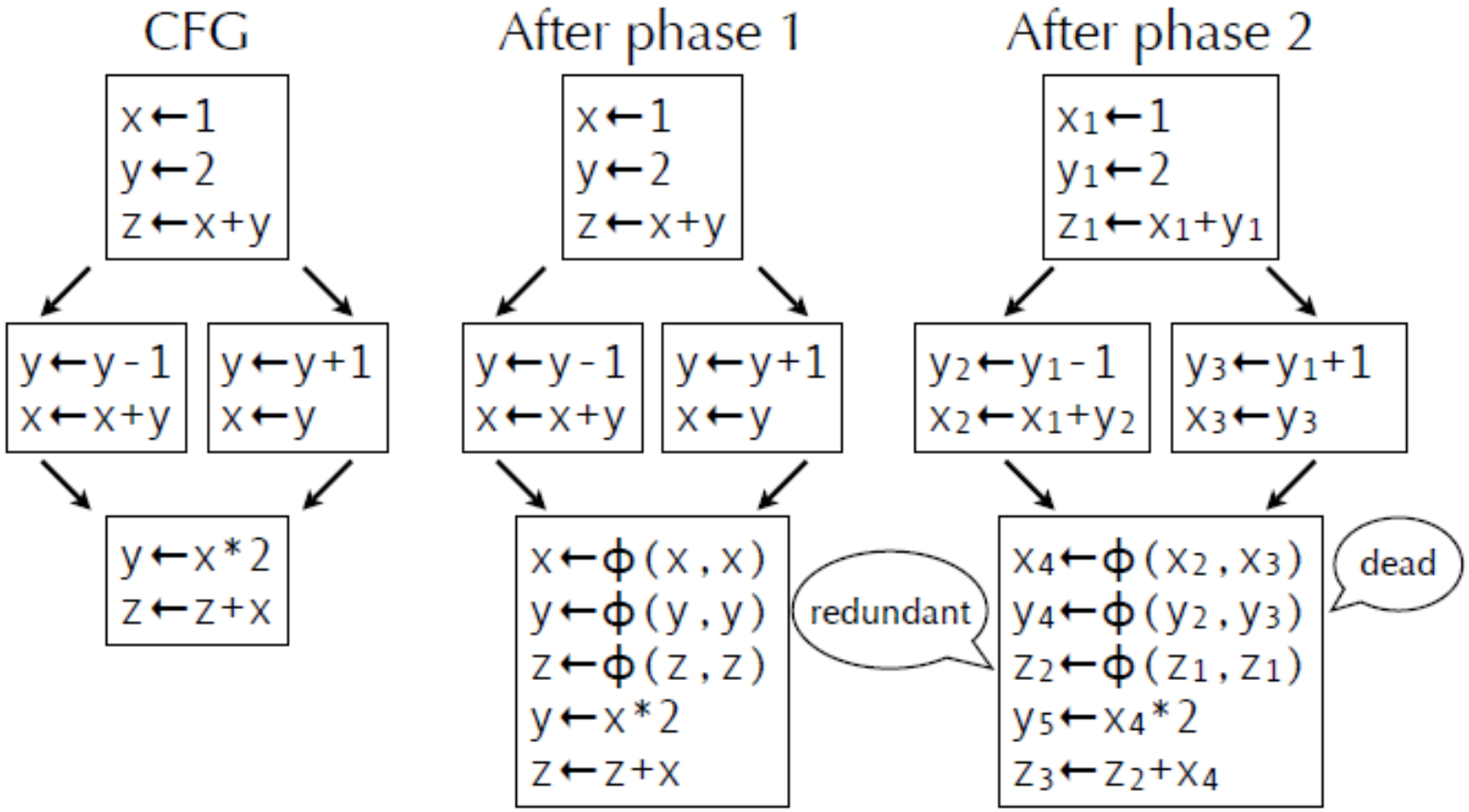
(b)

Figure 10.5: SSA Form example taken from [CFR+91]. Program (b) shows the SSA Form for program (a).

SSA construction: a naive way

1. For each variable x and at each joint point in the control flow graph, insert a $x := \phi(x, x)$ statement.
2. Add an index to each variable on the left-hand side of an assignment $x_i := ...$
3. Compute the reaching definitions to x that may reach each ϕ assignment.
4. Remove redudnat ϕ assignments, that is, assignments of the form $x_i := \phi(x_j, x_j)$.
5. Remove dead ϕ assignments.

(Naïve) building of SSA form



More sophisticated SSA construction

Definition. In a (control-flow) graph, a node m *dominates* another node n if and only if every path from *start* to n must pass through m.

Dominance is a relexive and transitive relation. In particular, every node m dominates itself.

Definition. We say m *strictly dominates* n if and only if m dominates n and $m \neq n$.

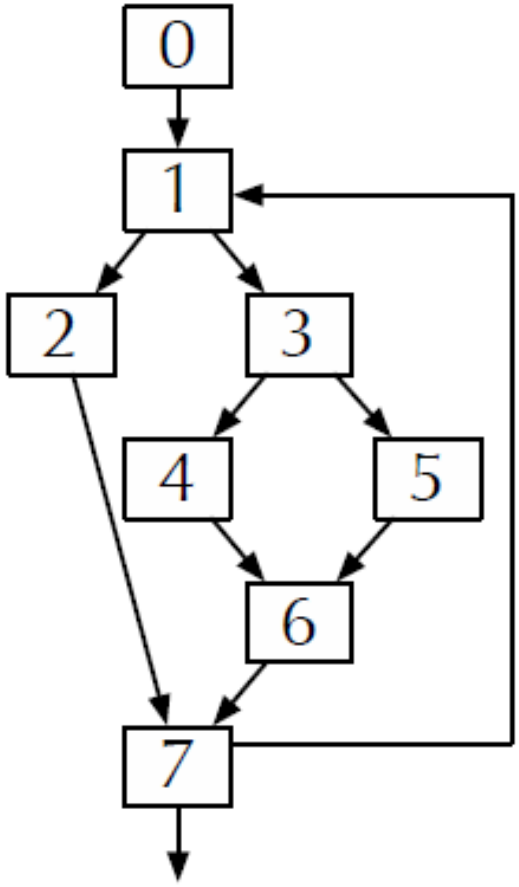
Note that m never strictly dominates itself.

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

Lemma. $n \neq$ immediate-dominator of n.

Example

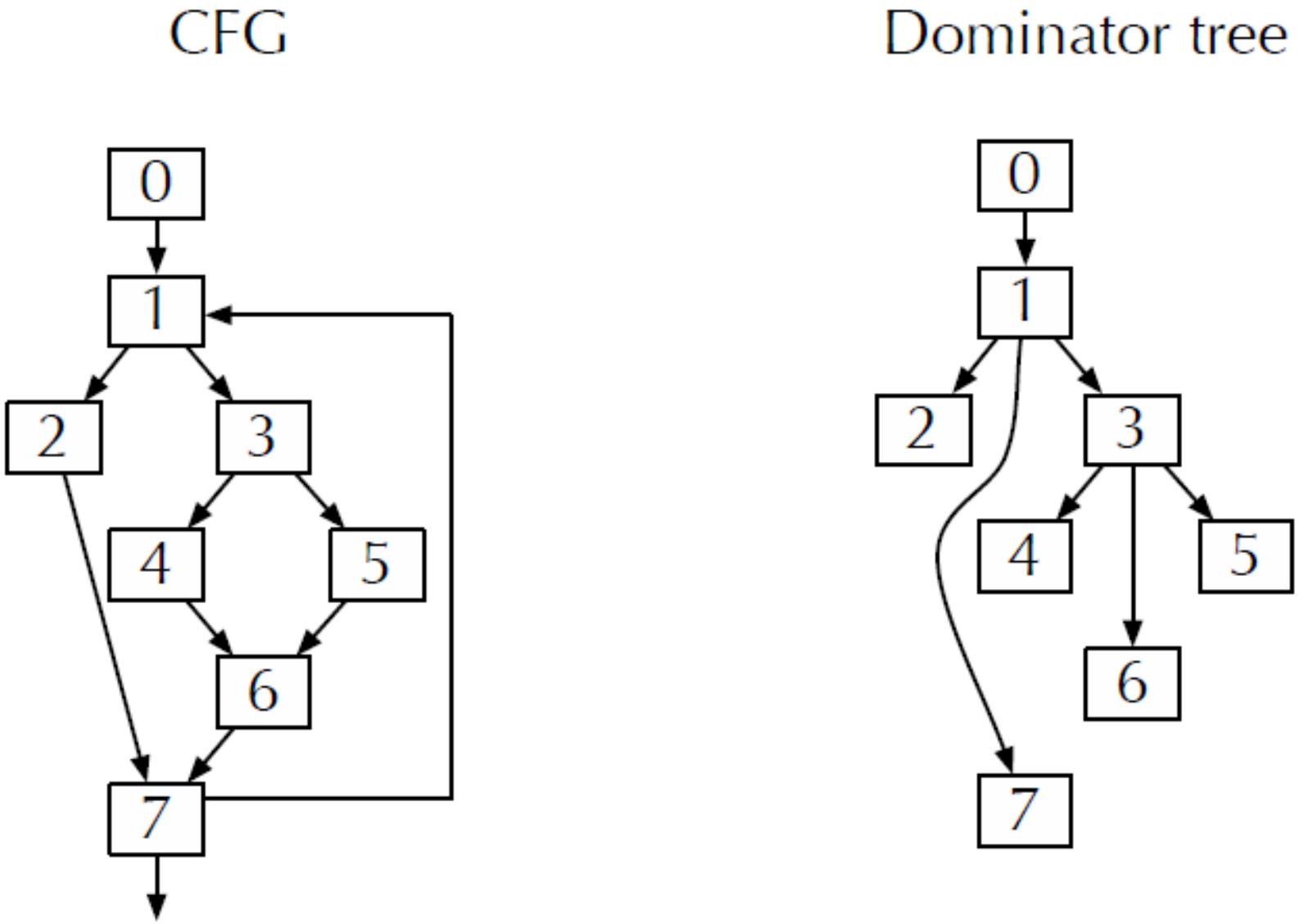
CFG



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 }

The immediate-dominance relation can be represented as a dominator tree.

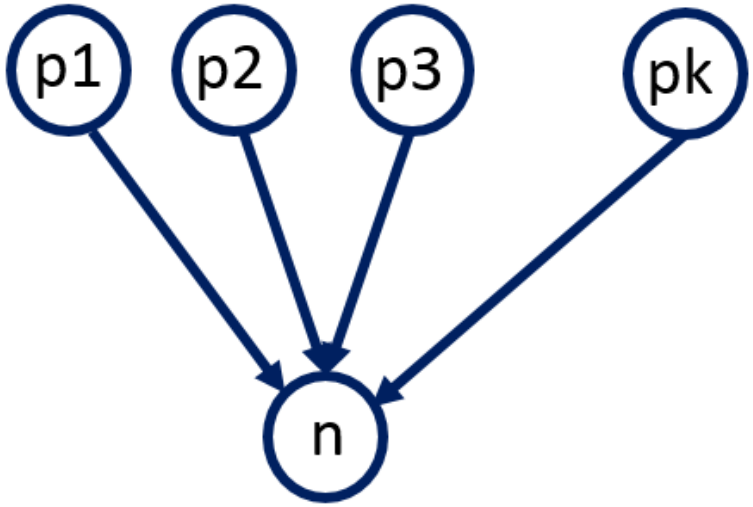


The dominance relation can be computed with data-flow analysis.

Algorithm. For each node n in the control flow graph, let v_n denote the set of n's dominators.

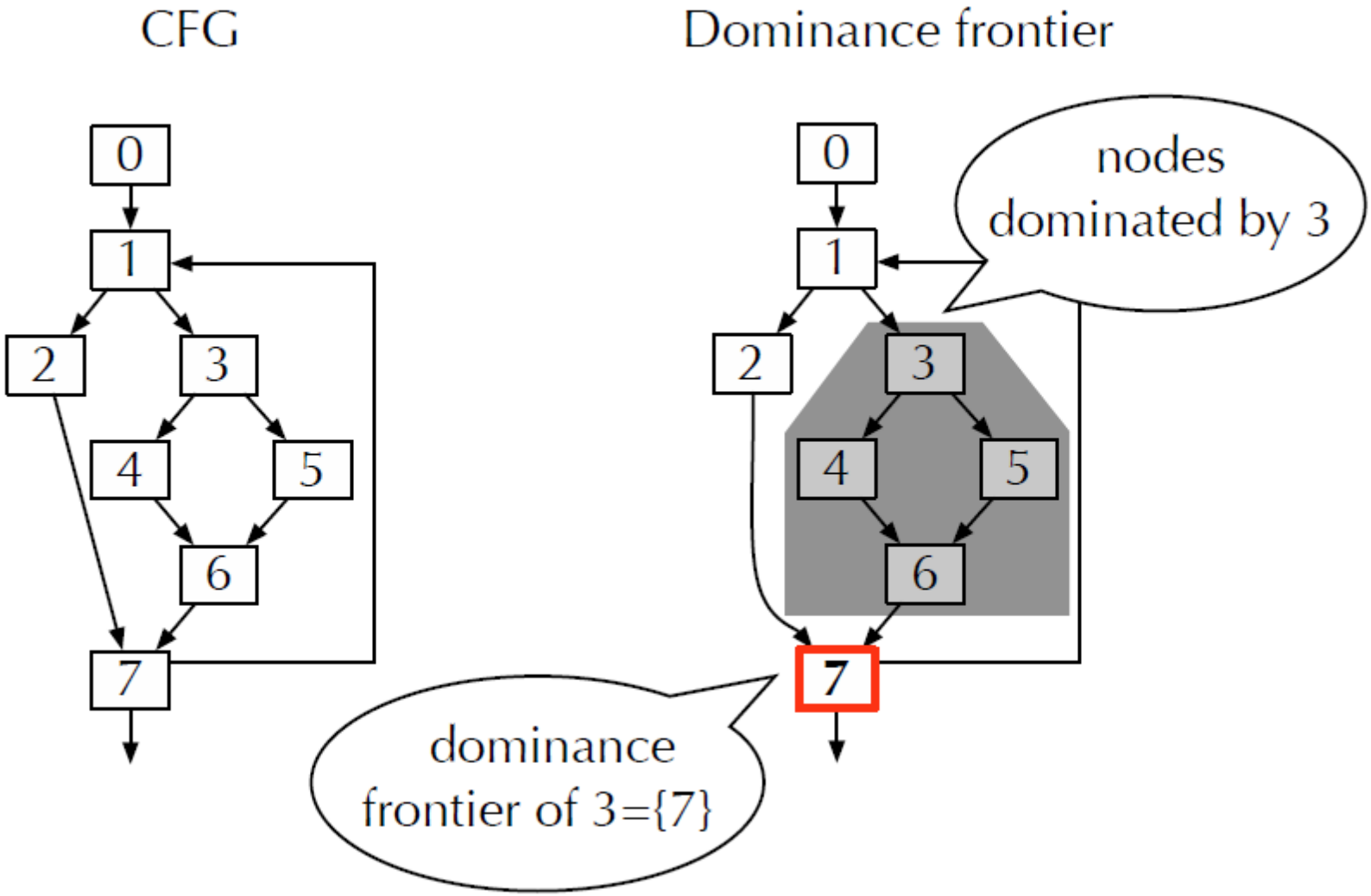
$$v_n = \{n\} \cup (v_{p1} \cap v_{p2} \cap \ldots \cap v_{pk})$$

where $p1, p2, \ldots, pk$ are n's predecssors in the CFG. Initially, $v_n = \emptyset$, for every node n.



Definition. The *dominance frontier* of a node m, $DF(m)$, is the set of nodes q such that m dominates a predecessor of q but m does not strictly dominate q.

It is possible that $m \in DF(m)$, for some node m in the CFG.


CFG
Dominance frontier
0
1
2
3
4
5
6
7
nodes dominated by 3
dominance frontier of 3={7}

The general idea of the dominance-frontier set.

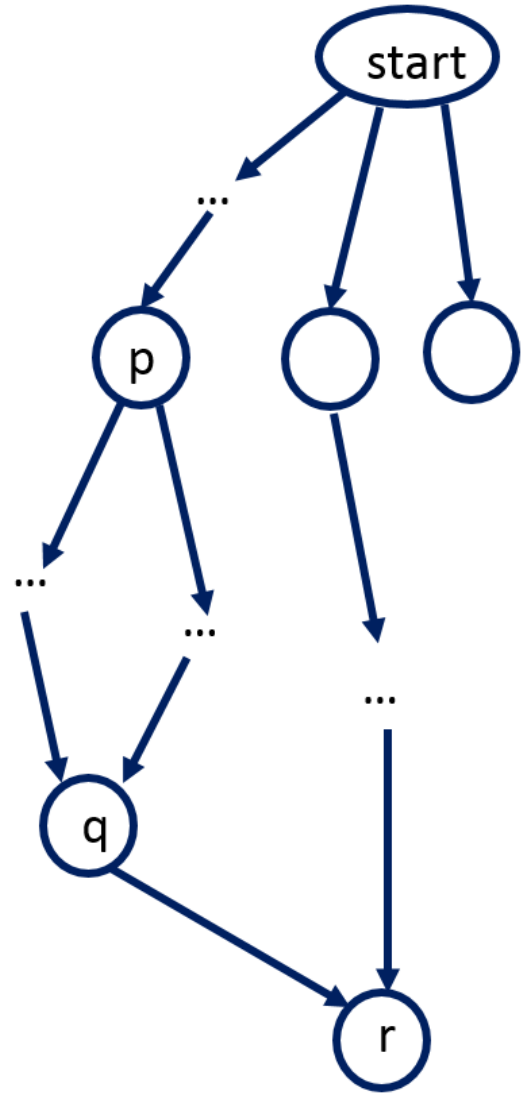


Figure 15: p dominates q but p does not dominate r. Hence, $r \in DF(p)$.

It is possible that $m \in DF(m)$, for some node m in the CFG.

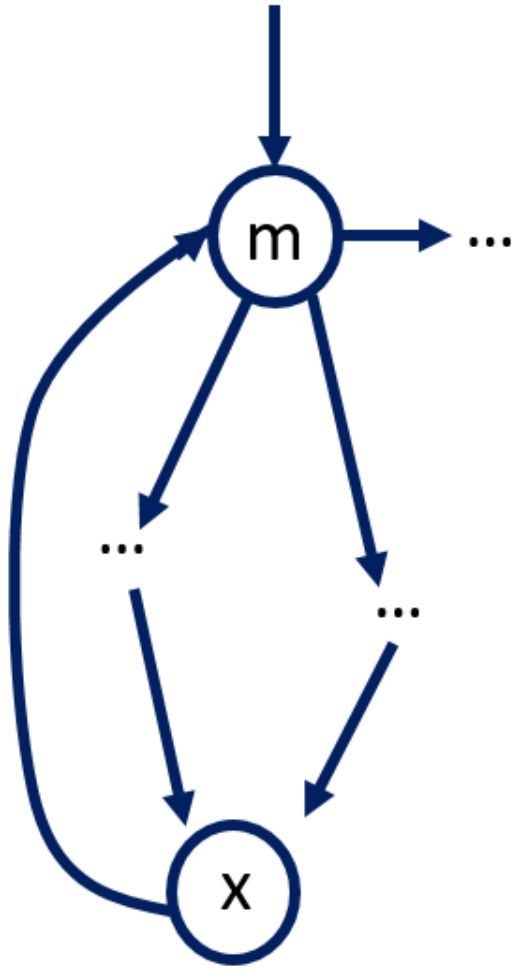


Figure 16: m dominates x, which is a predecessor of m itself. But m does not strictly dominate m itself. Hence, $m \in DF(m)$.

One algorithm for computing the dominance frontier set from the immediate-dominance relation is: (note that the traversal need not be in the pre-order traversal of CFG)

```
foreach node b do DF(b) := empty_set; end;
foreach node b do
  if the number of immediate predecessors of b >= 2 then
    foreach immediate predecessor, p, of b do
      runner := p;
      while runner != idom(b) do
        // runner is an ancestor of b,
        // but runner does not dominate b
        // Therefore, add b to DF(runner).
        DF(runner) := DF(runner) union { b };
        runner := idom(runner);
      end
    end
end
```

The minimal SSA form of a program: one with the fewest ϕ nodes.

Principle: for each definition of a variable x in a node n, insert a ϕ node for x in every node in $DF(n)$.

A ϕ node is also considered as a definition. Thus, a ϕ node may cause more ϕ nodes to be inserted.

Finally, remove all dead ϕ nodes.

There are several improvements to the above basic method.

First, a variable/definition that is live only in a single node will not need ϕ nodes. These variables/definitions can be removed beforehand.

The resulting SSA form is called the *semi-pruned SSA form*.

We call the set of variables/definitions that are live in more than one node the *global names* G.

Step 1: insert ϕ nodes.

```
for each global name x in G do
  worklist := all nodes in which x is defined
  foreach node n in worklist do
    for each node m in DF(n) do
      insert ‘‘x := phi(x, x)’’ to node m
      worklist := worklist union { m }
    end
  end
end
```

Step 2: rename variables. Renaming is perfomed with a pre-order traversal of the dominator tree.

For each node n that is visited in the pre-order traversal, do two things:

1. definitions and uses of variables in n are renamed.
2. The parameter corresponding to n in all ϕ nodes of all successors of n in CFG is renamed.

Code may be generated from the SSA form after optimizations are performed on the SSA form. The issue is to remove the ϕ nodes as they cannot be implemented on real machines.

A ϕ function has the form $x_i := \phi(x_1, x_2, \ldots, x_n)$. It is replaced by assignments of the form $x_i := x_j$ in all the predecessors of the ϕ node.

Most of these assignments will be removed later during register allocation, thanks to register coalescing.

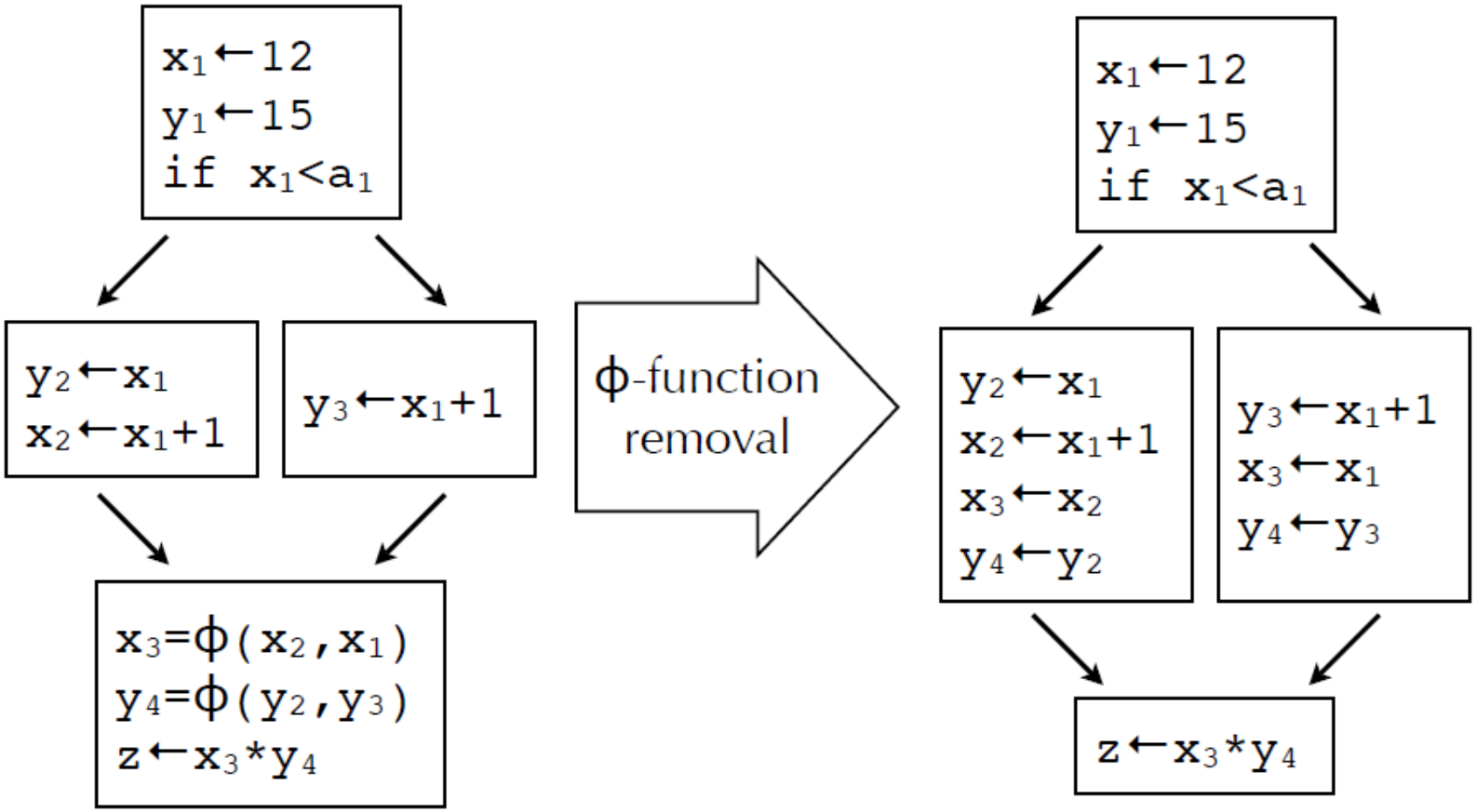


Figure 17: Remove ϕ functions.

Edge splitting (after removing ϕ functions)

In CFG, edges that go from a node m with multiple successors to a node n with multiple predecessors are called *critical edges*. See Figure 19.

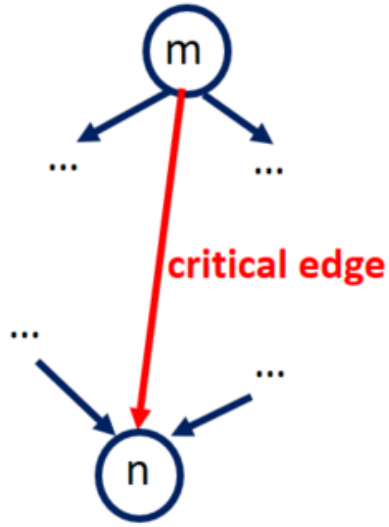


Figure 18: A critical edge $m \to n$.

For critical edges $m \to n$, the assignments inserted into node m is executed only if control reaches n later. That is, it is not clear if the assignment should be executed when m is executed.

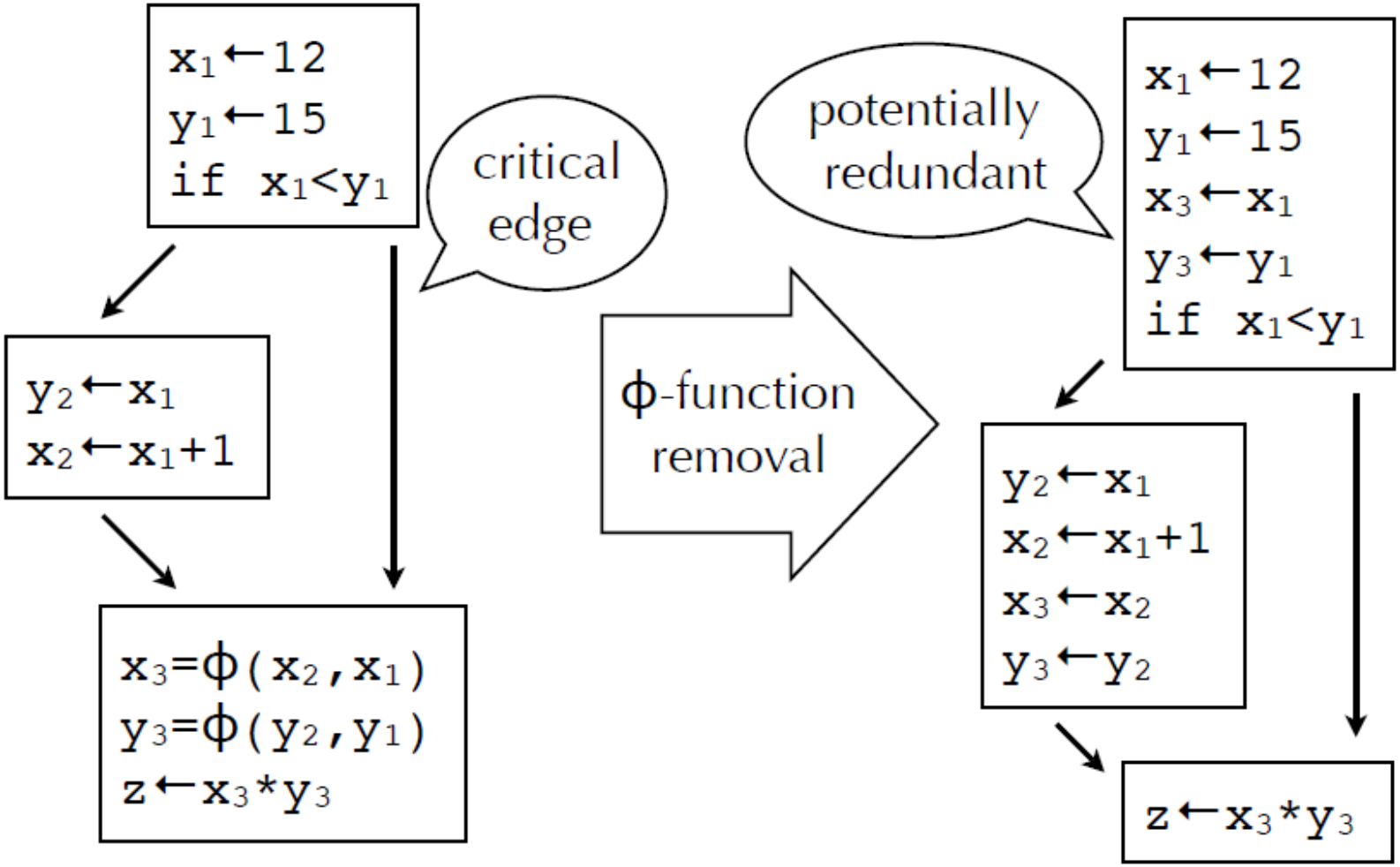


Figure 19: No edge splitting.

This may be avoided by adding a new empty node p and replacing the critical edge $m \to n$ with two edges $m \to p$ and $p \to n$. This process is called *edge splitting*. See Figure 20.

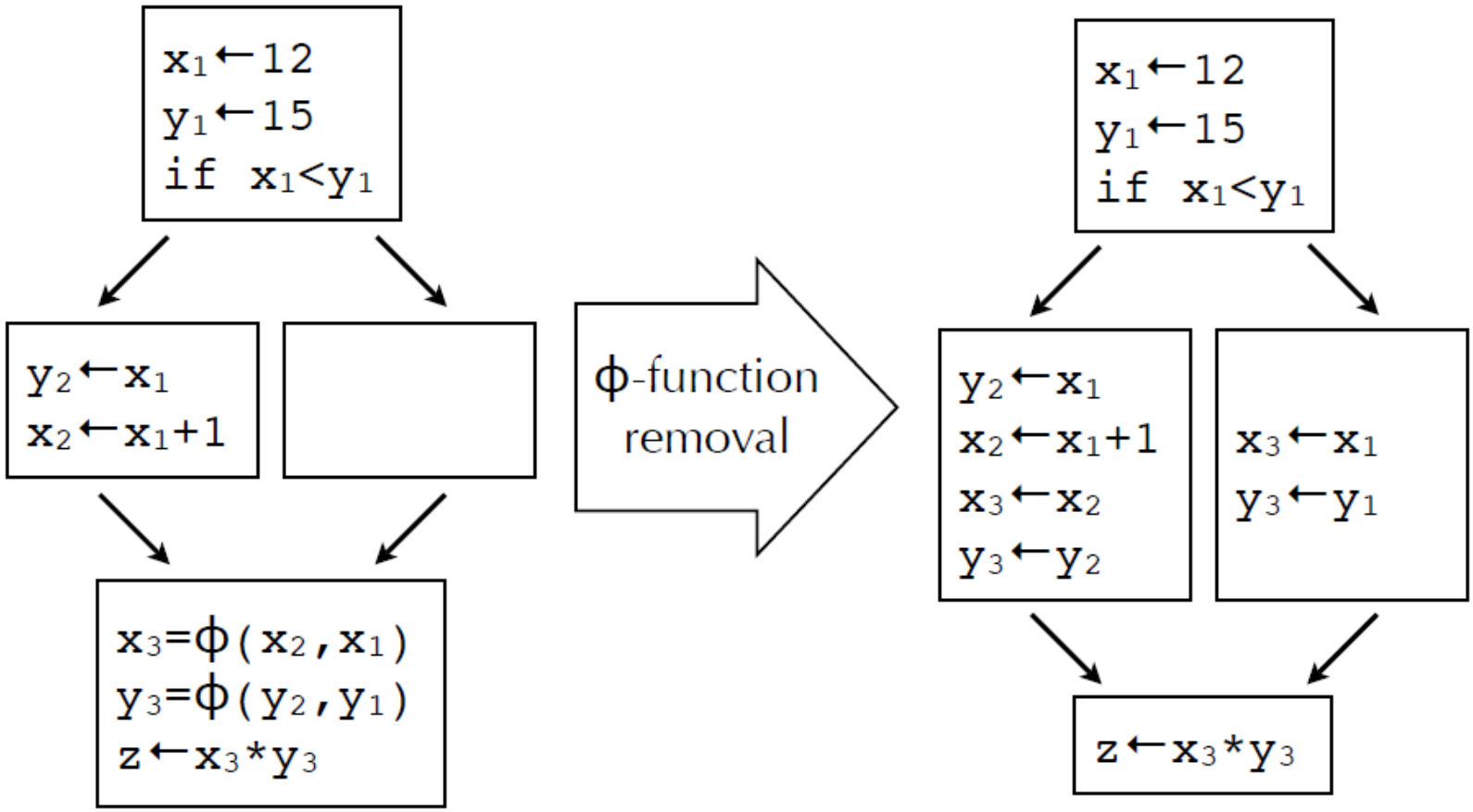


Figure 20: With edge splitting.

Applications of SSA:

1. SSA based constant propagation and value numbering, see "R. Cytron, PLDI Tutorial Notes-part6.pdf".
2. SSA based register allocation, see "SSA-register-allocation.pdf". Register allocation can be formulated as graph coloring. General graph coloring is NP-complete. However, programs in SSA form have chordal interference graphs. Chordal interference graphs can be colored in linear time.

Example. Dead code elimination with SSA form. Since data flow in SSA form is explicit, it is easy to identify dead code: If a variable x_i is not used in any expressions, its definition (either $x_i := j_j$ or $x_i := \phi(x_k, x_m)$) can be deleted.

Deleting a definition may make other definitions become dead code in which case these other defitinitions may be deleted as well.

Example. Liveness analysis with SSA form. SSA form simplifies liveness analysis. Hence the construction of interference graph for register allocation is also easier with SSA form.

To compute the region in which a variable x_i is live in SSA form, we start from all uses of x_i and walk backwards in the CFG until the definition of x_i. The statements encountered during the walk form the region in which x_i is live.

Example. SSA constant propagation. Every variable has the $\top$ value initially.

SSA Form

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
  $i_2 \leftarrow \phi(i_1, i_5)$
  $j_2 \leftarrow \phi(j_1, j_3)$
  $k_2 \leftarrow \phi(k_1, k_4)$
  if $(i_2 = 6)$ **then**
    $k_3 \leftarrow 0$
  else
    $i_3 \leftarrow i_2 + 1$
  fi
  $i_4 \leftarrow \phi(i_2, i_3)$
  $k_4 \leftarrow \phi(k_3, k_2)$
  $i_5 \leftarrow i_4 + k_4$
  $j_3 \leftarrow j_2 + 1$
until $(i_5 = j_3)$

Pass 1

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
  $i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge \top) = 6$
  $j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge \top) = 1$
  $k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = 1$
  if $(i_2 = 6)$ **then**
    $k_3 \leftarrow 0$
  else
    ★ /⋆ Not executed ⋆/
  fi
  $i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
  $k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
  $i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
  $j_3 \leftarrow j_2 + 1 \Rightarrow (1 + 1) = 2$
until $(i_5 = j_3 \Rightarrow (6 = 2) =$ **false**$)$

Pass 1

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
  $i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge \top) = 6$
  $j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge \top) = 1$
  $k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = 1$
  if $(i_2 = 6)$ **then**
    $k_3 \leftarrow 0$
  else
    /⋆ Not executed ⋆/
  fi
  $i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
  $k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
  $i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
  $j_3 \leftarrow j_2 + 1 \Rightarrow (1 + 1) = 2$
until $(i_5 = j_3 \Rightarrow (6 = 2) =$ **false**$)$

Pass 2

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
  $i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge 6) = 6$
  $j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge 2) = \bot$
  $k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = \bot$
  0
  if $(i_2 = 6)$ **then**
    $k_3 \leftarrow 0$
  else
    /⋆ Not executed ⋆/
  fi
  $i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
  $k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
  $i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
  $j_3 \leftarrow j_2 + 1 \Rightarrow (\bot + 1) = \bot$
until $(i_5 = j_3 \Rightarrow (6 = \bot) = \bot)$

Pass 3

$i_1 \leftarrow 6$
$j_1 \leftarrow 1$
$k_1 \leftarrow 1$
repeat
  $i_2 \leftarrow \phi(i_1, i_5) = (6 \wedge 6) = 6$
  $j_2 \leftarrow \phi(j_1, j_3) = (1 \wedge \bot) = \bot$
  $k_2 \leftarrow \phi(k_1, k_4) = (1 \wedge \top) = \bot$
  0
  if $(i_2 = 6)$ **then**
    $k_3 \leftarrow 0$
  else
    /⋆ Not executed ⋆/
  fi
  $i_4 \leftarrow \phi(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
  $k_4 \leftarrow \phi(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
  $i_5 \leftarrow i_4 + k_4 \Rightarrow (6 + 0) = 6$
  $j_3 \leftarrow j_2 + 1 \Rightarrow (\bot + 1) = \bot$
until $(i_5 = j_3 \Rightarrow (6 = \bot) = \bot)$

Pass 3 is identical to pass 2. Hence the algorithm terminates.

Example. Value numbering.
Consider the following code:

```
a := read()
v := a + 2
c := a
w := c + 2
t := a + 2
x := t - 1
```

Even though we do not know the value of `a` at compile time, but we can infer that `v` and `w` have the same value and the addition `c + 2` can be replaced with `v` directly. For this situation we use *value numbering*.

Value numbering gives each value a label (i.e., numbering) so that identical computations are given identical label.

There are several ways to implement value numbering.

First we may compare program text. The program text of an expression (or a subexpression) is hashed to a value. Text comparison cannot discover the equivalence of `v` and `w` in the above example. We also need to consider redfinitions to the related variables between two expressions.

Simple value numbering is confined within a single basic block.

An alternative approach is *partitioning*. Initially assume all values are equal and gradually partition them. First we partition the values according to the functions that are used to compute them.

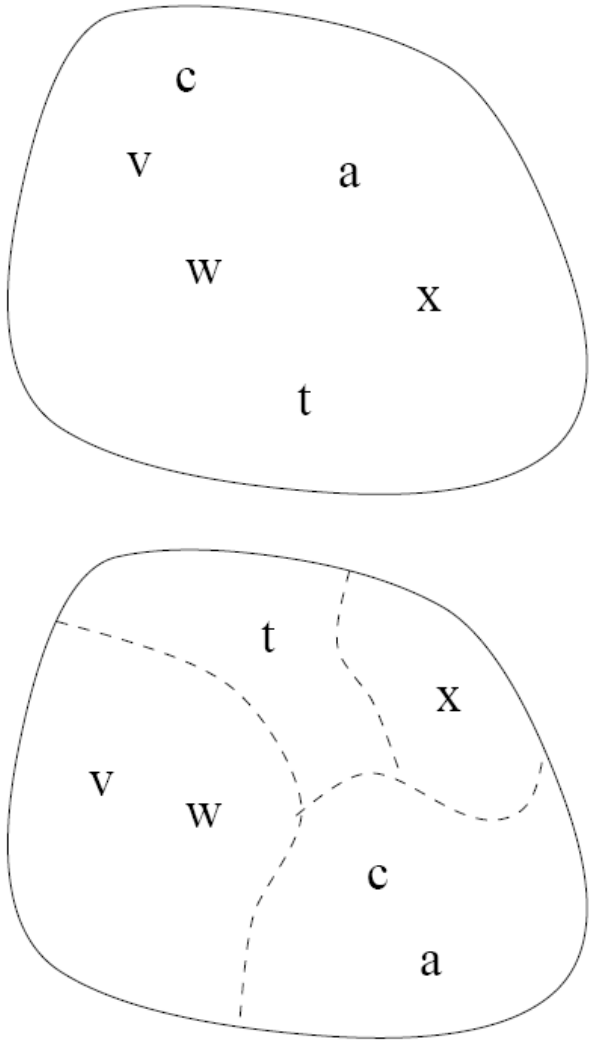


Figure 21: First partition

Partition may be based on the SSA form. Note that ϕ functions at different locations are considered different.

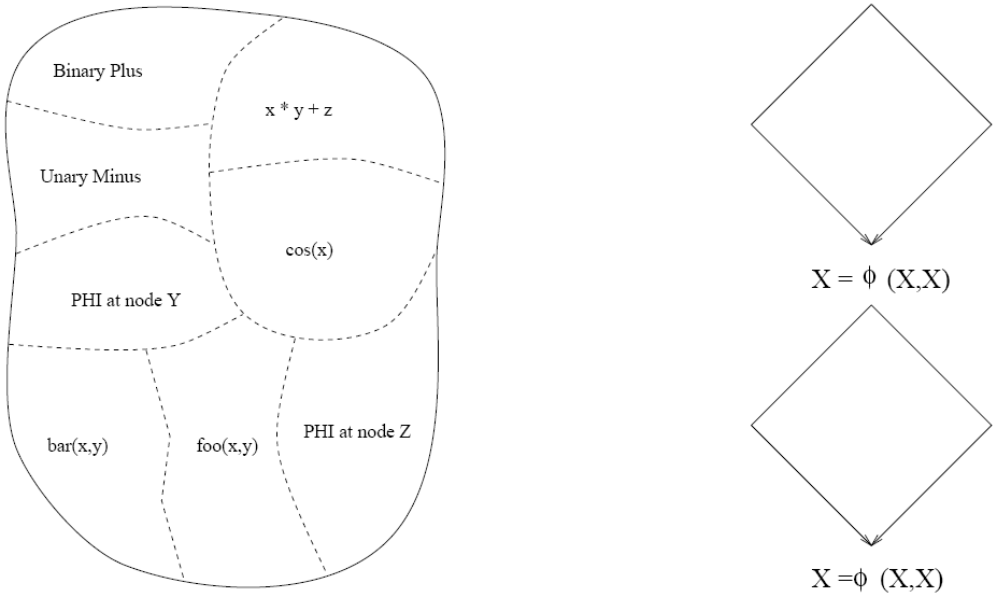


Figure 22: First partition based on the SSA form

if $(condA)$ **then**
  $a_1 \leftarrow \alpha$
  if $(condB)$ **then**
    $b_1 \leftarrow \alpha$
  else
    $a_2 \leftarrow \beta$
    $b_2 \leftarrow \beta$
  fi
  $a_3 \leftarrow \phi(a_1, a_2)$
  $b_3 \leftarrow \phi(b_1, b_2)$
  $c_2 \leftarrow \star a_3$
  $d_2 \leftarrow \star b_3$
else
  $b_4 \leftarrow \gamma$
fi
$a_5 \leftarrow \phi(a_1, a_0)$
$b_5 \leftarrow \phi(b_0, b_4)$
$c_3 \leftarrow \star a_5$
$d_3 \leftarrow \star b_5$
$e_3 \leftarrow \star a_5$

Figure 23: Third step

Consider the example in Figure 23. Assume α and β are different expressions. c_2 and d_2 have the same value while c_3 and d_3 have different values. Thus, we can use the value of c_2 for d_2. This saves one memory fetch.

If variable b is declared `volatile` in C, we have to make a safer (and more conservative) assumption: all b's are different.

Figure 24 shows the initial partition. We assume all variables have the same initial values.

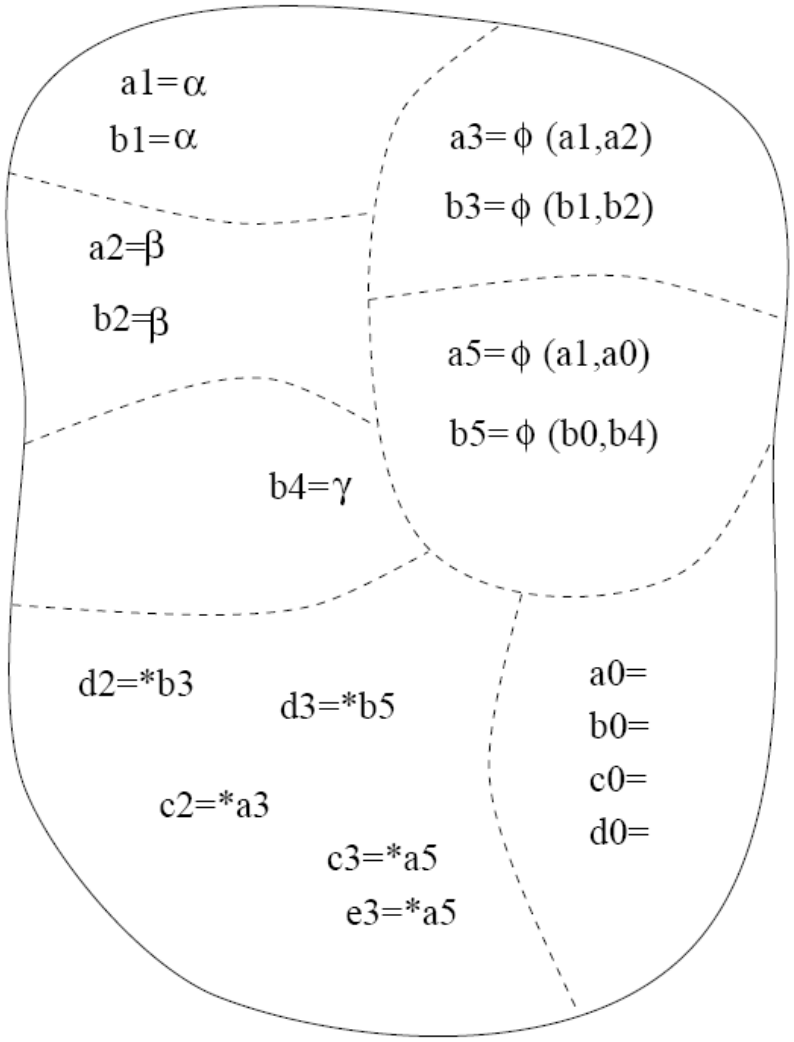


Figure 24: 4th step

Then the block with five names is split because $a5$ and $a3$ are different (i.e., in different blocks). Then the block with $a5$ and $b5$ is split because $a0$ and $b4$ are different.

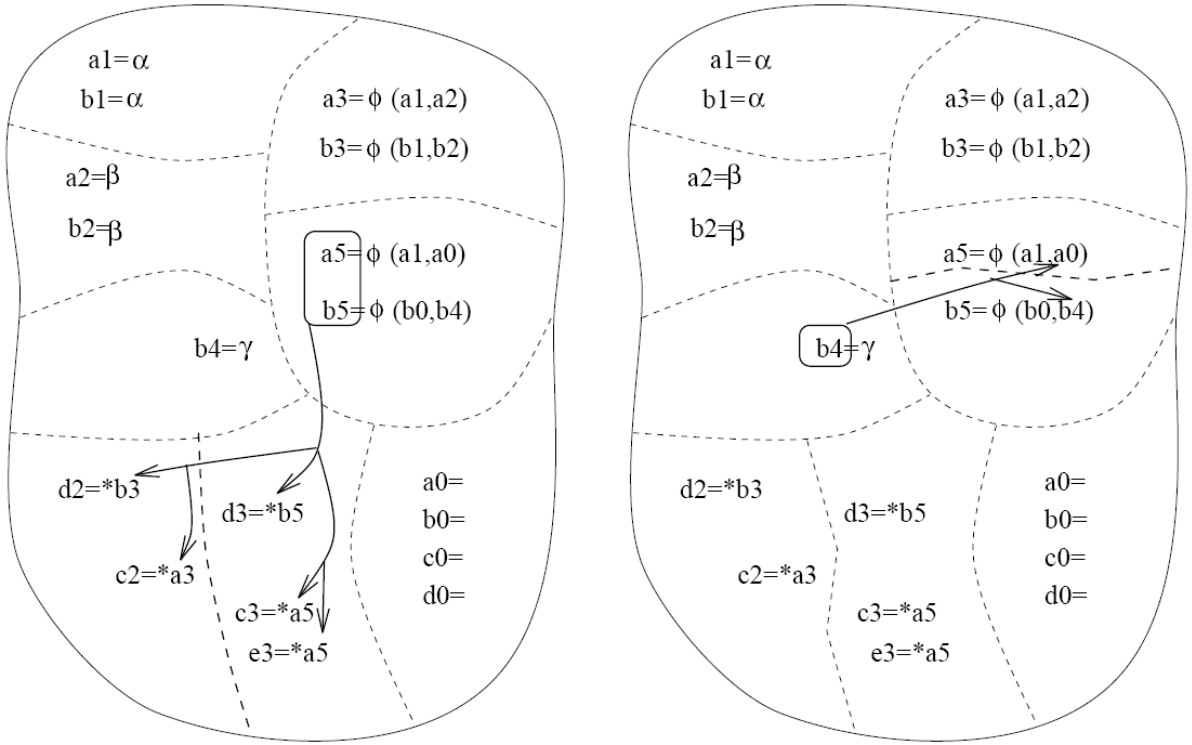


Figure 25: 5th step

Finally the block with $d3$ and $c3$ is split because $a5$ and $b5$ are different.

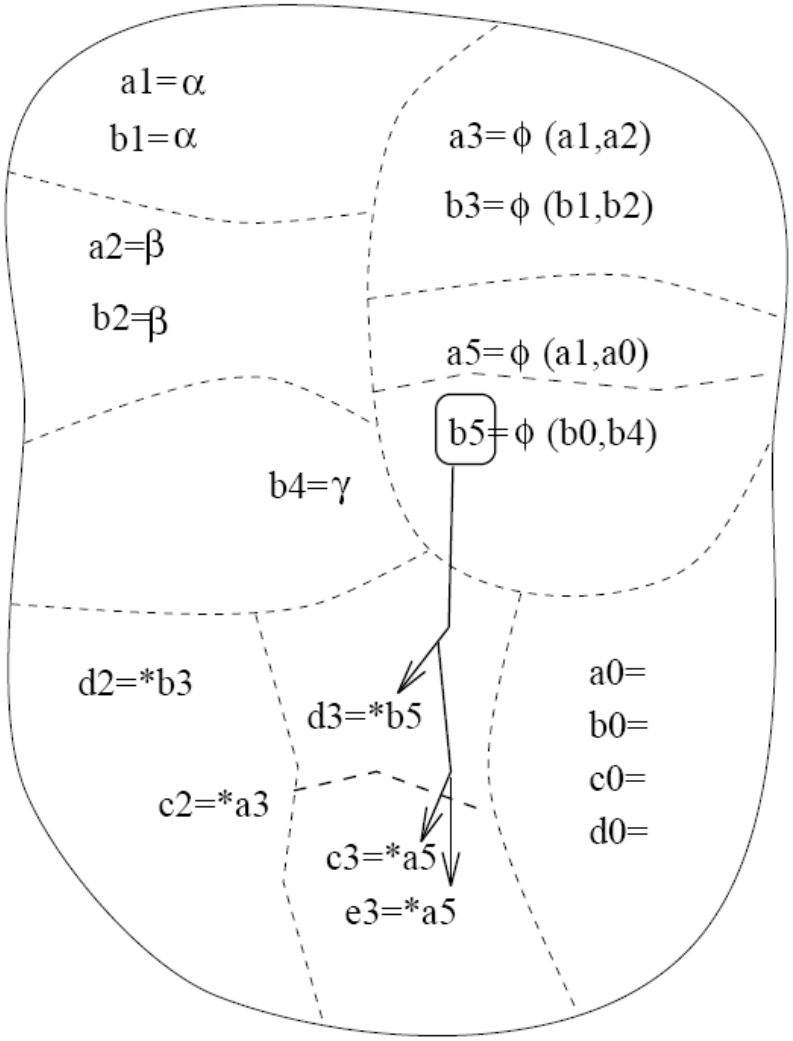


Figure 26: 6th step

The above algorithm can be summarized as follows:

- Find the SSA form.
- Find the initial partition.
- Choose a block to partition according to existing blocks.
- Repeat the above step until no more blocks can be partitioned.

Constant propagation with SSA:

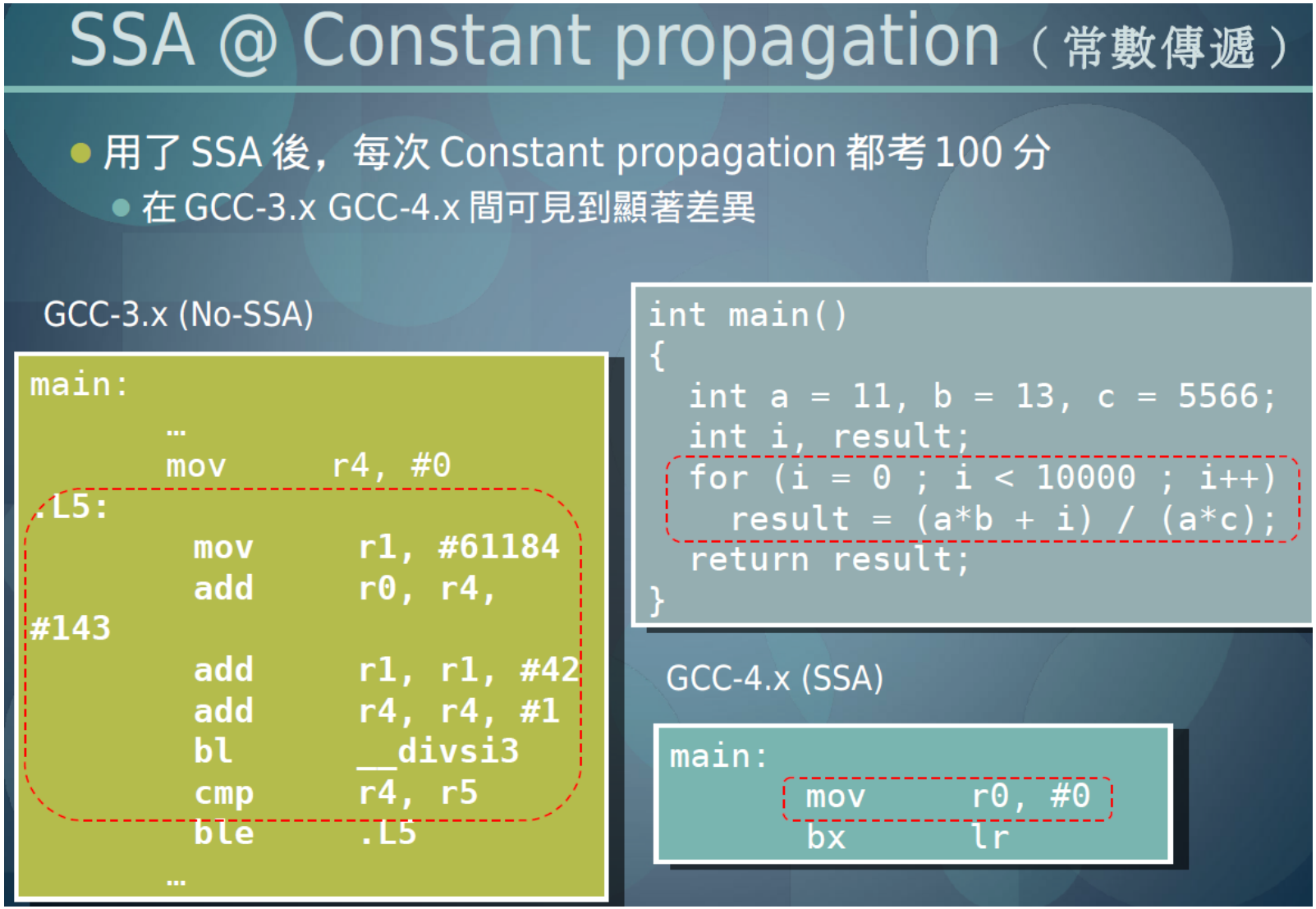


Figure 27: Constant propagation with SSA.

Example. Register allocation.

Current status:

- Optimal register allocation is NP-complete (based on graph coloring).
- A common approach is to begin with as many *virtual registers* as needed. Then map these virtual registers to the real registers. We can use an interferene graph as the data structure for register allocation. Spilling might be necessary.
- We can also compute the maximum live variables at any instant. If real registers are more than maximum live variables, there should be appripriate allocation schemes. However, some variable may need to be relocated to other registers during its life time.

Figure 28: LLVM logo.

Example. LLVM IR in Figure 29.

```
define internal fastcc i32 @L_f7fcd438_() {
L_f7fcd438_:
  br label %L_f7fcd438_1
L_f7fcd438_1:
  store i32 48, i32* @ext_R3
  br label %L_f7fcd43c_3
 ......
L_f7fcd450_14:
  store i32 -134425516, i32* @ext_LR
  store i32 -134436108, i32* @ext_PC
  %17 = tail call i32 @75()
  ret i32 %17
}
```

Figure 29: A dynamically translated block (an LLVM function). Note that there is type information.

```
define void @f(%mytype2** %myobj) {
  %obj = load %mytype2** %myobj
  %get = getelementptr inbounds %mytype2* %obj, i32 0, i32 1
  %ml = mul i32 4, 4
  %naj = call noalias i8* @_Znaj(i32 %ml)
  %cast = bitcast i8* %naj to %mytype**
  store %mytype** %cast, %mytype*** %get
  %get2 = getelementptr %mytype** %get, i32 0
  %ld = load %mytype** %get2
  ret void
}
```

Expected %mytype**, found %mytype***

1 quick fix available:

Insert bitcast conversion for %get

Press 'F2' for focus

Figure 30: A sample LLVM IR.

Case study.

IR in GCC:

- high Level : GENERIC (Syntax Tree Style IR)
- middle Level : Gimple (Tree Style IR, SSA form)
- low Level : RTL (List Style IR, Register Based)

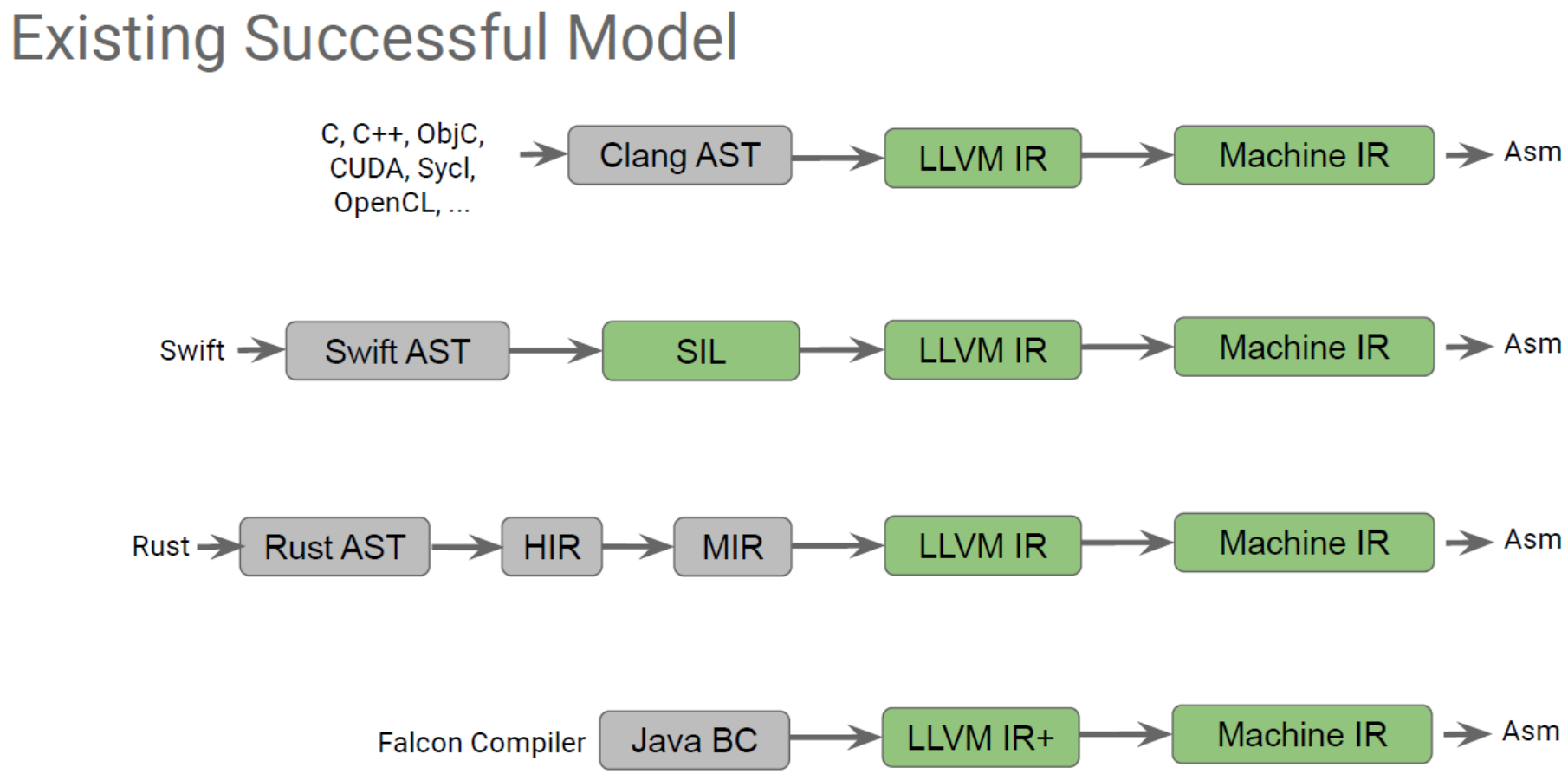


Figure 31: Various IRs

multi-level IR

Figure 32: MLIR

Why multi-level?

- Some optimizations should be done in the lower leve; some (such as matrix optimizatization or language-specific optimizations) should be done in a higher level (e.g., Swift, Rust, Julia).
- We factor out the common parts so we do not need to the same things several times.

All operations from different levels are mixed together, which is quite confusing. Therefore, we use dialects to group related operations. Dialects (and classes and operations) are specified/generated with

Tblgen.