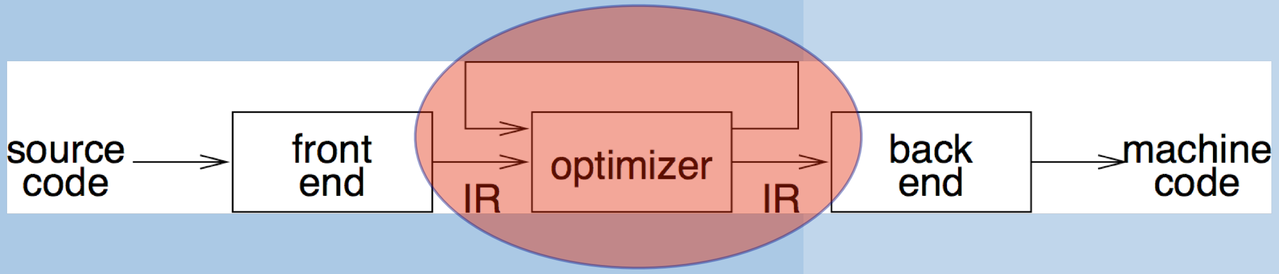
6. Optimization

Oscar Nierstrasz



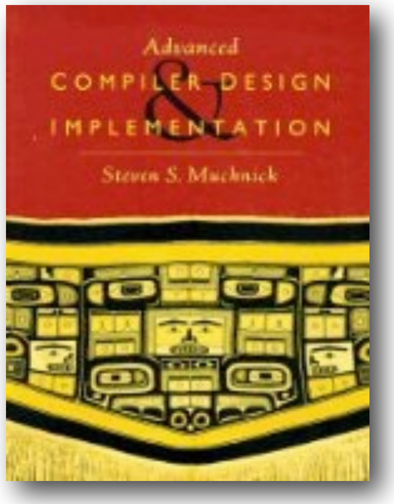
Lecture notes courtesy
Marcus Denker

Roadmap

> Introduction
> Optimizations in the Back-end
> The Optimizer
> SSA Optimizations
> Advanced Optimizations

Literature

> Muchnick: *Advanced Compiler Design and Implementation*
  — *>600 pages on optimizations*



> Appel: *Modern Compiler Implementation in Java*
  — *The basics*

Roadmap

> **Introduction**
> Optimizations in the Back-end
> The Optimizer
> SSA Optimizations
> Advanced Optimizations

Optimization: The Idea

> Transform the program to improve efficiency

> **Performance**: faster execution
> **Size**: smaller executable, smaller memory footprint

Tradeoffs:

1) **Performance** vs. **Size**

2) **Compilation speed** and **memory**

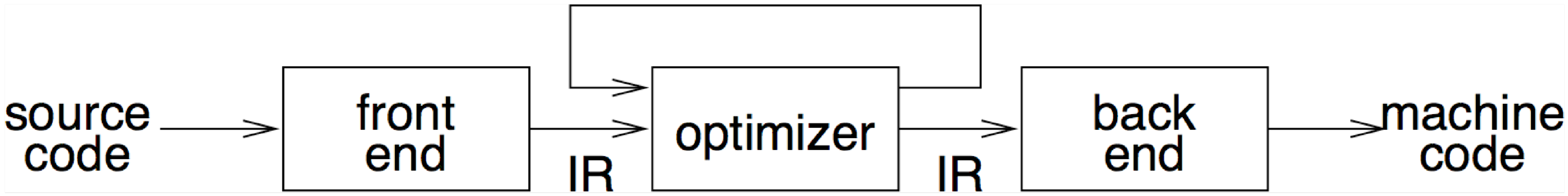
No Magic Bullet!

> Rice (1953): *For every compiler there is a modified compiler that generates shorter code.*

> ***Proof:*** Assume there is a compiler U that generates the shortest optimized program Opt(P) for all P.
  - —Assume P to be a program that does not stop and has no output
  - —Opt(P) will be L1 goto L1
  - —Halting problem. Thus: U does not exist.

> There will be always a better optimizer!
  - —Job guarantee for compiler architects :-)

Optimization at many levels



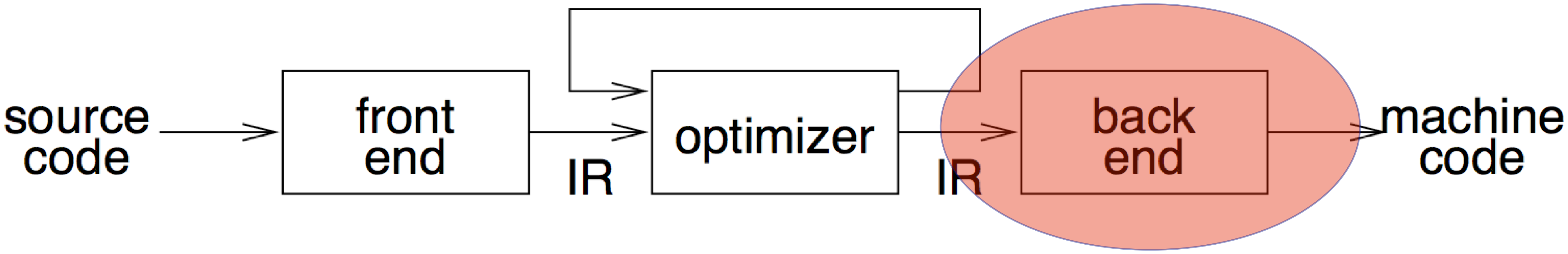
> Optimizations both in the optimizer and back-end

Back-end optimizations may focus on how the machine code is optimally generated.

Roadmap

> Introduction
> **Optimizations in the Back-end**
> The Optimizer
> SSA Optimizations
> Advanced Optimizations

Optimizations in the Backend



> Register Allocation
> Instruction Selection
> Peep-hole Optimization

Register Allocation

> Processor has only finite amount of registers
  —Can be very small (x86)

> Temporary variables
  —non-overlapping temporaries can share one register

> Passing arguments via registers

> Optimizing register allocation very important for good performance
  —Especially on x86

There are various problems with the x86 architecture: few registers, overlapping register classes, irregular access to registers (some encoded in instructions), fixed registers for multiplication/division, ...

Source: http://news.ycombinator.com/item?id=276418

Instruction Selection

> For every expression, there are many ways to realize them for a processor

> Example: Multiplication*2 can be done by bit-shift

Instruction selection is a form of optimization

Peephole Optimization

> Simple local optimization
> Look at code “through a hole”
  - replace sequences by known shorter ones
  - table pre-computed

```
store R,a;     →     store R,a;
load a,R

imul 2,R;      →     ashl 1,R;
```

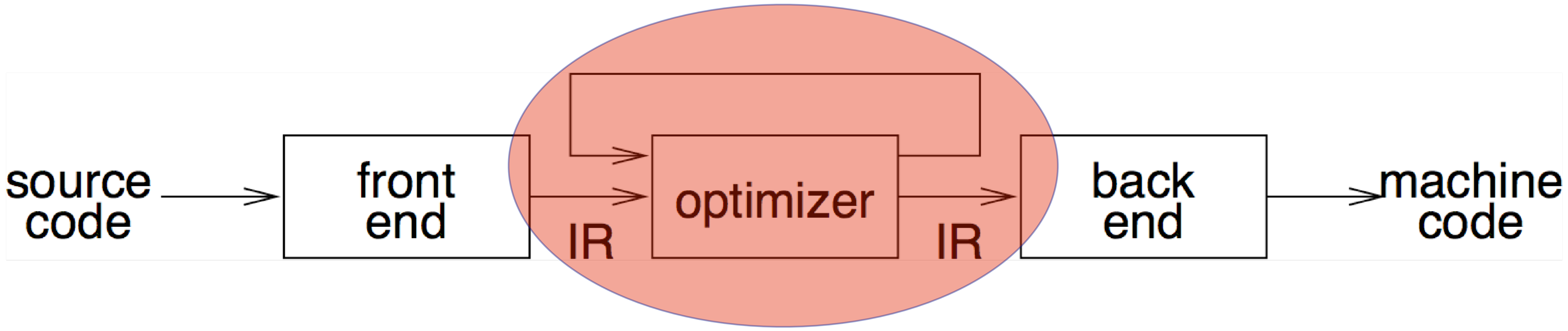
Important when using simple instruction selection!

Peephole optimization typically considers just 2-3 consecutive lines of generated code. It works well for simple compilers. For longer instructions sequences use graph matching.

In the first example, we eliminate a redundant loading of a variable from memory into a register, since it is already there.

In the second example, instead of multiplying a number by two, we simply bit shift (ashl) it left by one position.

Optimization at many levels



Most optimization is done in a special phase

Roadmap

> Introduction
> Optimizations in the Back-end
> **The Optimizer**
> SSA Optimizations
> Advanced Optimizations

Examples for Optimizations

- Constant Folding / Propagation
- Copy Propagation
- Algebraic Simplifications
- Strength Reduction
- Dead Code Elimination
  - Structure Simplifications
- Loop Optimizations
- Partial Redundancy Elimination
- Code Inlining

Constant Folding

> Evaluate constant expressions at compile time
> Only possible when side-effect freeness guaranteed

`c:= 1 + 3` $\longrightarrow$ `c:= 4`

`true not` $\longrightarrow$ `false`

Caveat: Floats — implementation could be different between machines!

Constant folding is a form of partial evaluation. Constant expressions are recognized and evaluated at compile time. Some of this can be done early while generating IR from the AST.

Constant Propagation

> Variables that have constant value, e.g. c := 3
    - Later uses of c can be replaced by the constant
    - If no change of c between!

```
b := 3
c := 1 + b
d := b + c
```

$\Longrightarrow$

```
b := 3
c := 1 + 3
d := 3 + c
```

Analysis needed, as b can be assigned more than once!

Constant propagation entails recognizing that certain “variables” actually have constant values, and then propagating these constants to expressions where they are used. Later we will see SSA is ideal to analyze this.

Copy Propagation

> for a statement x := y
> replace later uses of x with y, if x and y have not been changed.

```
x := y
c := 1 + x
d := x + c
```

```
x := y
c := 1 + y
d := y + c
```

Analysis needed, as y and x can be assigned more than once!

Algebraic Simplifications

> Use algebraic properties to simplify expressions

`-(-i)` ⟶ `i`

`b or: true` ⟶ `true`

Important to simplify code for later optimizations

Strength Reduction

- Replace expensive operations with simpler ones
- Example: Multiplications replaced by additions

$$\begin{tikzcd}\boxed{\texttt{y := x * 2}} \arrow[r] & \boxed{\texttt{y := x + x}}\end{tikzcd}$$

Peephole optimizations are often strength reductions

Actually here a bit shift would be even better.

Dead Code

> Remove *unnecessary* code
—e.g. variables assigned but never read

```
b := 3
c := 1 + 3
d := 3 + c
```

$\Longrightarrow$

```
c := 1 + 3
d := 3 + c
```

> Remove code never reached

```
if (false)
{a := 5}
```

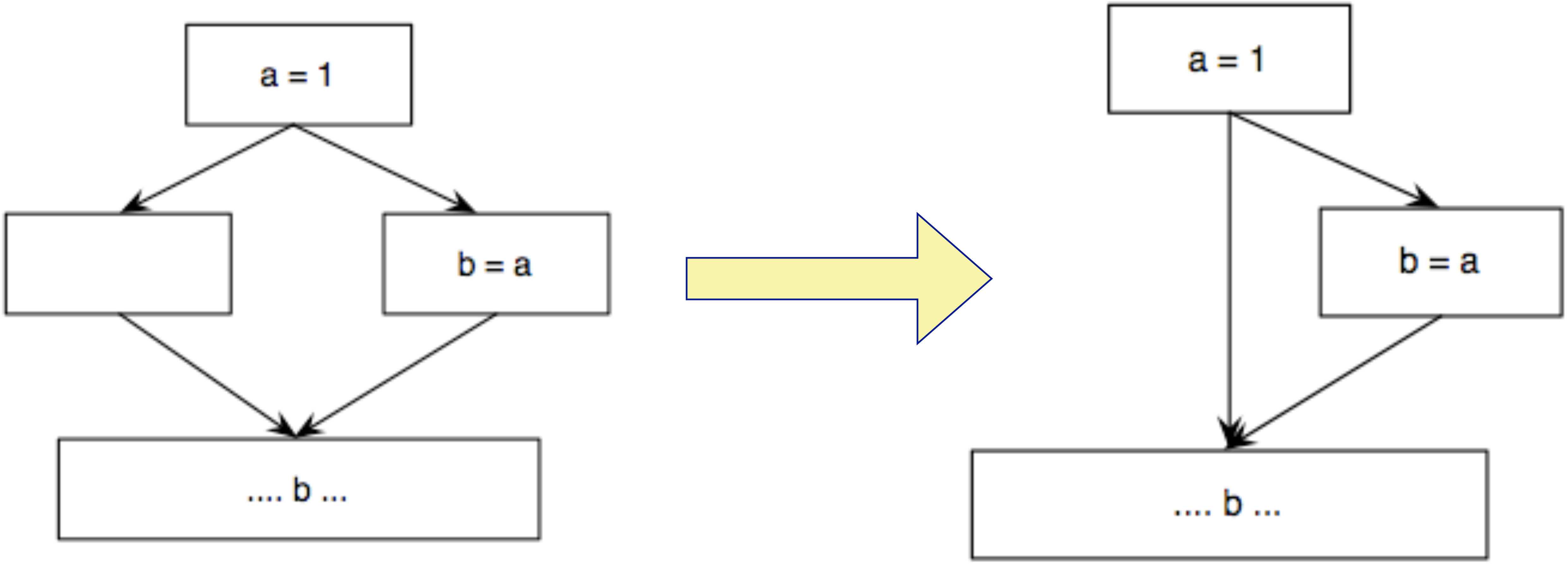
$\Longrightarrow$

```
if (false)
{}
```

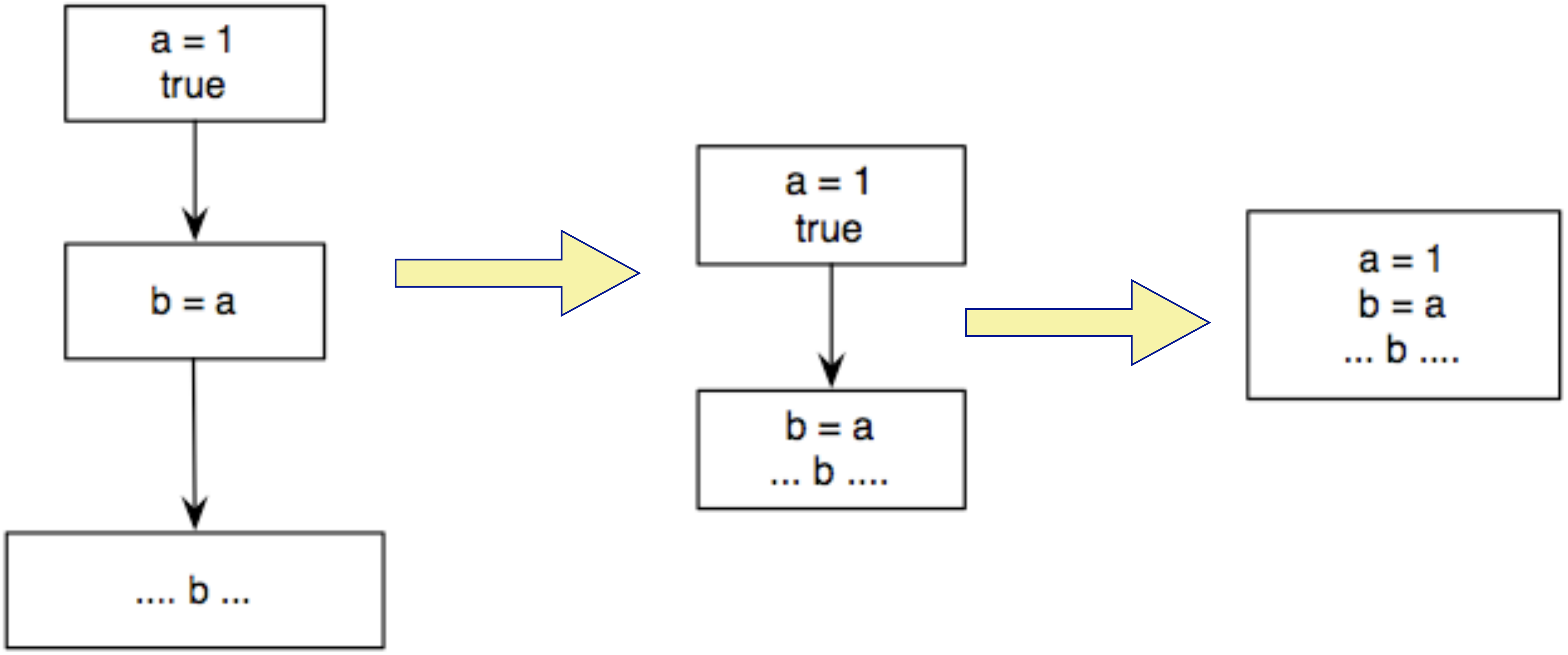
Simplify Structure

- > Similar to dead code: Simplify CFG Structure
  - – Eg delete empty basic blocks, fuse basic blocks (next slides)
- > Optimizations will degenerate CFG
  - – Needs to be cleaned to simplify further optimization!

Delete Empty Basic Blocks



Fuse Basic Blocks



Here we have “conditional” jumps between basic blocks, where the conditions are always true. We can fuse together these basic blocks and eliminate the jumps.

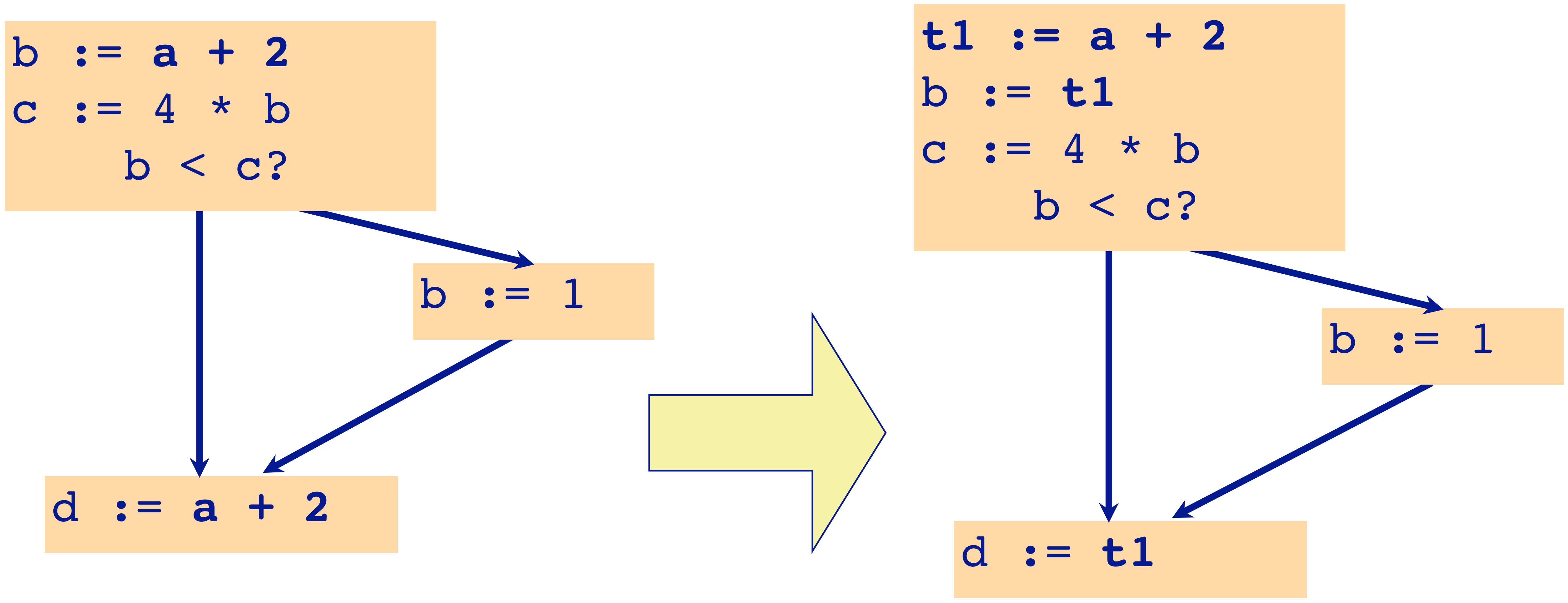
Common Subexpression Elimination (CSE)

> **Common Subexpression:**
  - There is another occurrence of the expression whose evaluation always precedes this one
  - operands remain unchanged

> **Local** (inside one basic block): When building IR

> **Global** (complete flow-graph)

Example CSE



Note that we need to verify that a has not changed in between!

Loop Optimizations

> Optimizing code in loops is important
  - often executed, large payoff

> Various techniques
  - fission/fusion: split/combine loops to improve locality or reduce overhead
  - scheduling: run parts in multiple processors
  - unrolling: duplicate body several times to decrease test cost
  - loop-invariant code motion: move invariant code out of loop
  - ...

http://en.wikipedia.org/wiki/Loop_optimization

The idea of loop unrolling is to replicate the loop body some number of times to reduce the number of iterations:

```
 for (i = 0; i < 100; i++)
   g();
```

Becomes:

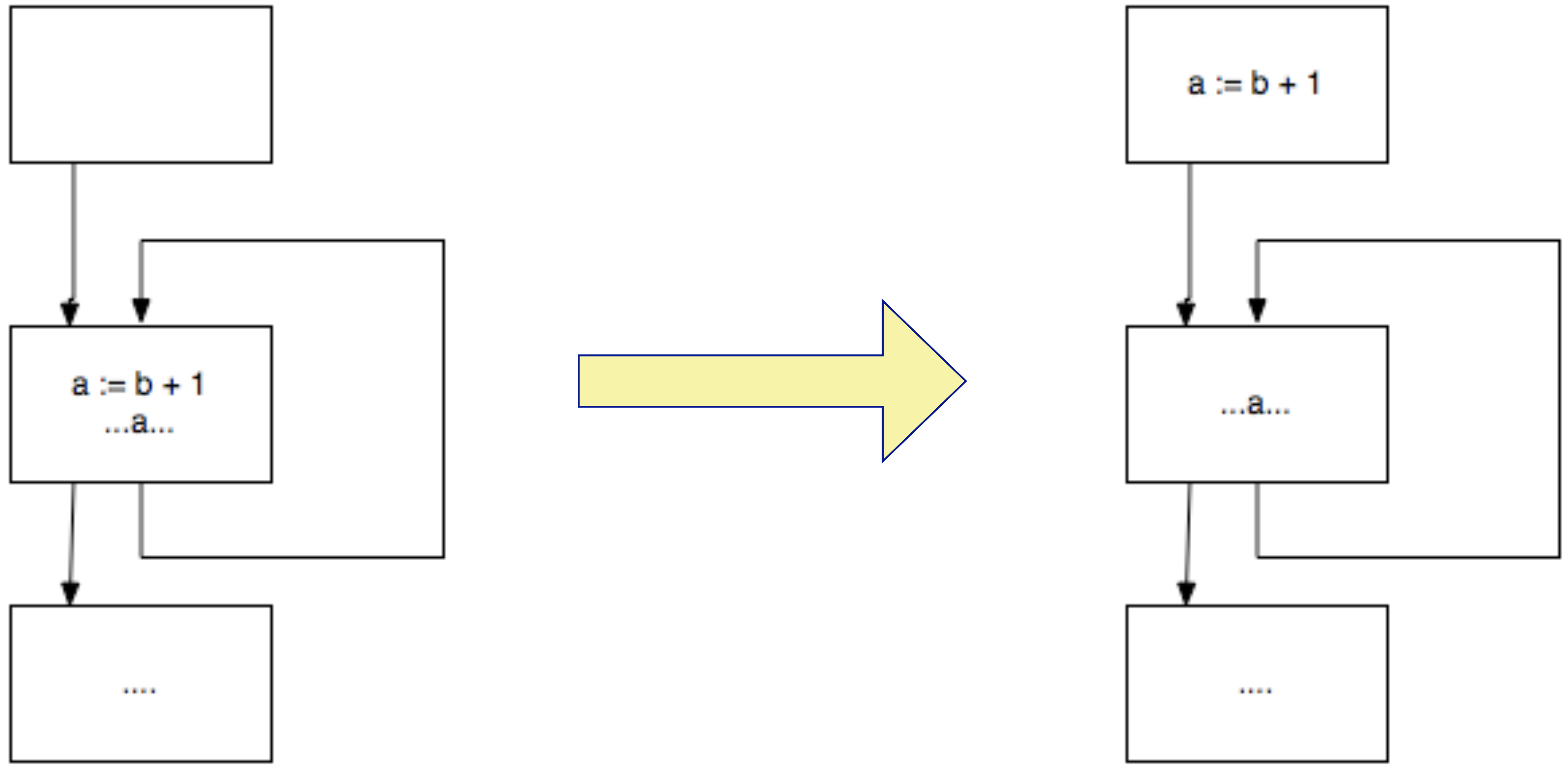
```
 for (i = 0; i < 100; i+=5) {
   g(); g(); g(); g(); g();
 }
```

Pros: reduces testing cost

Cons: bloats code; uses more registers

Loop Invariant Code Motion

> Move expressions that are constant over all iterations out of the loop



NB: This does not generally work for expressions with side effects.

Induction Variable Optimizations

> Values of variables form an arithmetic progression

```
integer a(100)
do i = 1, 100
 a(i) = 202 - 2 * i
endo
```

value assigned to *a* decreases by 2

```
integer a(100)
t1 := 202
do i = 1, 100
  t1 := t1 - 2
  a(i) = t1
endo
```

uses *Strength Reduction*

FORTRAN example. Instead of repeatedly computing 202-2*i, we simply subtract 2 from an temporary variable each time through the loop.

Automatically finding such optimizations is rather complicated (see chapter in Muchnick).

Partial Redundancy Elimination (PRE)

> Combines multiple optimizations:

—global common-subexpression elimination

—loop-invariant code motion

> **Partial Redundancy:** computation done more than once on some path in the flow-graph

> PRE: insert and delete code to minimize redundancy.

Partial Redundancy Elimination

```
if (some_condition) {
   // some code
   y = x + 4;
}
else {
   // other code
}
z = x + 4;
```

```
if (some_condition) {
   // some code
   t = x + 4;
   y = t;
}
else {
   // other code
   t = x + 4;
}
z = t;
```

http://en.wikipedia.org/wiki/Partial_redundancy_elimination

In the original code, x+4 is computed twice if we follow the first branch. In the rewritten version it is always computed only once.

Code Inlining

> All optimizations up to now were local to one procedure

> ***Problem:*** procedures or functions are very short
>
> —Especially in good OO code!

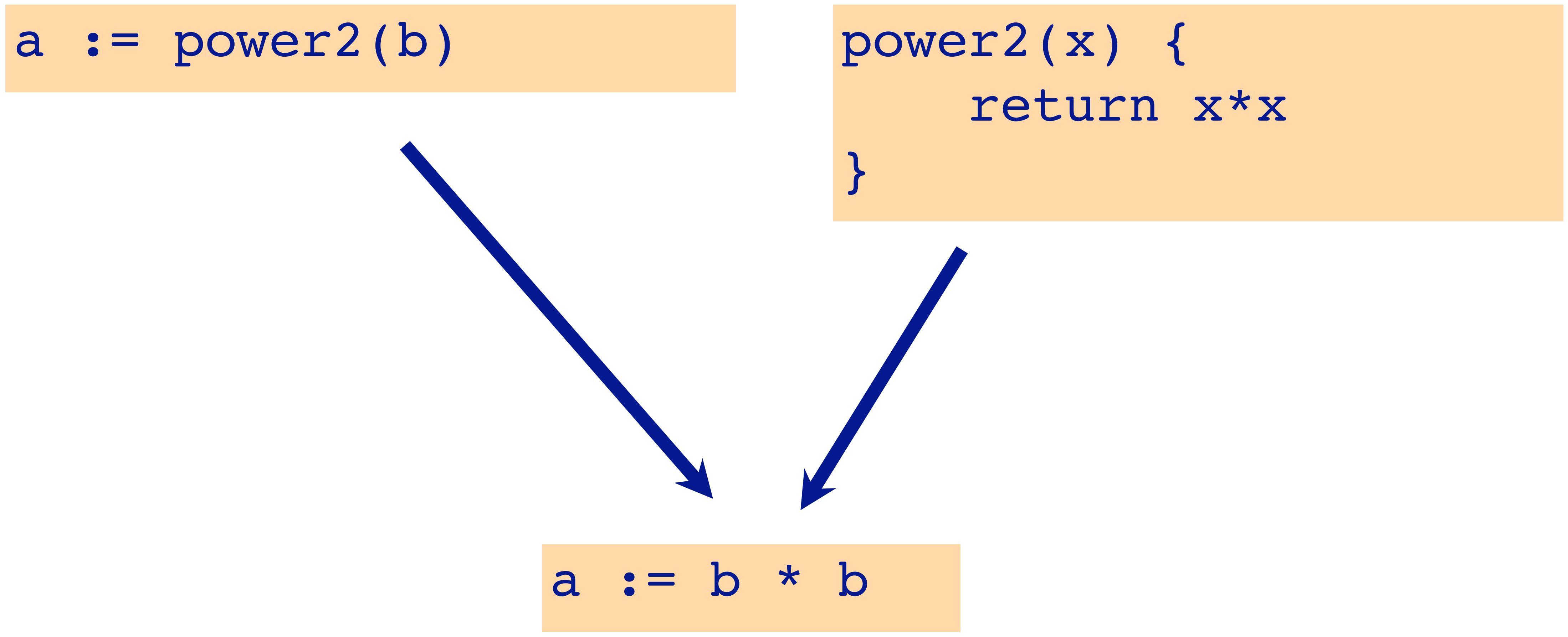
> ***Solution:*** Copy code of small procedures into the caller
>
> —OO: Polymorphic calls. Which method is called?

en.wikipedia.org/wiki/Inline_caching

Good OO code classically has small methods — these are great to inline if possible (e.g. if there is only one implementor).

With polymorphic inline caching, a limited number of possible methods are cached. If that limit is exceeded, the code reverts to “megamorphic” (i.e. non-inlined) mode.

Example: Inlining



```
a := power2(b)
```

```
power2(x) {
      return x*x
}
```

```
a := b * b
```

NB: inlining can bloat the generated code. C++ added the `inline` keyword as a hint, but modern compilers ignore this hint.

Roadmap

Recall: SSA

> SSA: Static Single Assignment Form

> **Definition**: Every variable is only assigned once

Properties

- Definitions of variables (assignments) have a list of all uses
- Variable uses (reads) point to the one definition
- CFG of Basic Blocks

Examples: Optimization on SSA

> We take three simple ones:
  - —Constant Propagation
  - —Copy Propagation
  - —Simple Dead Code Elimination

Recall: Constant Propagation

> Variables that have constant value, e.g. c := 3
  - Later uses of c can be replaced by the constant
  - If no change of c between!

```
b := 3
c := 1 + b
d := b + c
```

$\Longrightarrow$

```
b := 3
c := 1 + 3
d := 3 + c
```

Analysis needed, as b can be assigned more than once!

Constant Propagation and SSA

- Variables are assigned once
- We know that we can replace all uses by the constant!

```
b1 := 3
c1 := 1 + b1
d1 := b1 + c1
```

$\Longrightarrow$

```
b1 := 3
c1 := 1 + 3
d1 := 3 + c1
```

Note that this example shows that you must optimize *iteratively*, since now c1 will also be a constant.

We also now get dead code (b1 := 3)

Recall: Copy Propagation

- for a statement x := y
- replace later uses of x with y, if x and y have not been changed.

```
x := y
c := 1 + x
d := x + c
```

$\Longrightarrow$

```
x := y
c := 1 + y
d := y + c
```

Analysis needed (without SSA), as y and x can be assigned more than once!

Copy Propagation and SSA

> for a statement x1 := y1
> replace later uses of x1 with y1

```
x1 := y1
c1 := 1 + x1
d1 := x1 + c1
```

$\Longrightarrow$

```
x1 := y1
c1 := 1 + y1
d1 := y1 + c1
```

Copy propagation can be useful as a “clean up” optimization after other optimizations have been performed.

Dead Code Elimination and SSA

> Variable is *live* if the list of uses is not empty.

> Dead definitions can be deleted
—(If there is no side-effect)

```
b1 := 3
c1 := 1 + 3
d1 := 3 + c
```

Roadmap

> Introduction
> Optimizations in the Back-end
> The Optimizer
> SSA Optimizations
> **Advanced Optimizations**

Profile-guided optimization

- Approach:
  - Generate code,
  - profile it in a typical scenario,
  - then use that information to optimize it

- Problem:
  - usage scenarios can change in deployment, there is no way to react to that as profile is generated at compile time.

Dynamic optimization

- Re-optimize at run time in the VM
  - uses profile information gathered at run time
  - for both hardware and language VM
  - good way to exploit unused CPU cycles or unused CPUs (multi-core)

Multicore

- Optimizing for using multiple processors
  - —Auto parallelization
  - —Very active area of research (again)

JIT compilation

Dynamic Translation: Compilation done during execution of a program – *at run time* – rather than prior to execution

Improve time and space efficiency of programs using:

> portable and space-efficient byte-code

> run-time information → feedback directed optimizations

> speculative optimization

JIT compilation was introduced by McCarthy in a 1960 paper on dynamic translation of LISP code.

JIT compilation on demand was pioneered in Smalltalk. These techniques were later transferred to Self and Java.

McCarthy, “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”

http://www-formal.stanford.edu/jmc/recursive.pdf

See also:

https://en.wikipedia.org/wiki/Just-in-time_compilation

Iterative Process

> There is no general “right” order of optimizations
> One optimization generates new opportunities for a preceding one.
> Optimization is an iterative process

Compile Time vs. **Code Quality**

What you should know!

- ✎ *Why do we optimize programs?*
- ✎ *Is there an optimal optimizer?*
- ✎ *Where in a compiler does optimization happen?*
- ✎ *Can you explain constant propagation?*

Can you answer these questions?

- ✎ *What makes SSA suitable for optimization?*
- ✎ *When is a definition of a variable live in SSA Form?*
- ✎ *Why don’t we just optimize on the AST?*
- ✎ *Why do we need to optimize IR on different levels?*
- ✎ *In which order do we run the different optimizations?*