# Three-Address-Code (TAC) Optimizations

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

December 2012 Compiler Lecture Notes

# Structure of a Compiler

Program text
↓
Lexical analysis
↓
Symbol sequence
↓
Syntax analysis
↓
Syntax tree
↓
Typecheck
↓
Syntax tree
↓
Intermediate code generation ⟶ Interm code + Optimizations
↑
Machine code generation
↑
Symbolic machine code
↑
Register allocation
↑
Ditto with named registers
↑
Assembly and linking
↑
Binary machine code

1. Optimizations: Bird-Eye View
2. Recovering Program Structure from TAC
   - Basic Blocks
   - Control-Flow Graph (CFG)
   - Identifying Loops
   - Control-Flow-Graph Reducibility
3. Examples of Optimizations
4. Data-Flow Analysis
   - Reaching Definitions
   - Copy Propagation
   - Common-Subexpression Elimination (CSE)

# Code-Optimization Motivation and Requirements

Compiler's target code is still not as good as the hand-tuned code of the assembly-language expert, but is better than what most programmers would generate, even if they wanted to.

Achieved via code transformations, albeit the result is hardly optimal.

# Code-Optimization Motivation and Requirements

Compiler's target code is still not as good as the hand-tuned code of the assembly-language expert, but is better than what most programmers would generate, even if they wanted to.

Achieved via code transformations, albeit the result is hardly optimal.

An optimization, a.k.a., code transformation, needs to:

- preserve the observable behavior of the original prg (semantics),
- speed-up the program by a measurable amount (on average),
- prg's size not an issue but may affect instr-cache performance,
- be worth the effort, e.g., compile time, maintainability, etc.

# High-Level Optimization Strategy

The compiler effort for a certain code fragment is relative to:

- the number of times the prg will be run (user sets optim level);
- the amount of time spent in that code fragment relative to the progrm's runtime.

# High-Level Optimization Strategy

The compiler effort for a certain code fragment is relative to:

- the number of times the prg will be run (user sets optim level);
- the amount of time spent in that code fragment relative to the progrm's runtime.

Code optimization problems are NP-complete or undecidable:

- in many cases even NP-complete to aproximate, hence
- cannot expect to find a global optimum in a reasonable time.

# High-Level Optimization Strategy

The compiler effort for a certain code fragment is relative to:

- the number of times the prg will be run (user sets optim level);
- the amount of time spent in that code fragment relative to the progrm's runtime.

Code optimization problems are NP-complete or undecidable:

- in many cases even NP-complete to aproximate, hence
- cannot expect to find a global optimum in a reasonable time.

Strategy: spend the compiler effort in hot areas, e.g., loop nests:

- programers can write clear code in high-level languages
- while still getting efficient execution.

# Optimization Break Down

Source Code $\Rightarrow$ $\boxed{\text{Front End}}$ $\Rightarrow$ Interm Code $\Rightarrow$ $\boxed{\text{Code Generator}}$ $\Rightarrow$ Target Code

| User can: | Compiler Can: | Compiler Can: |
|---|---|---|
| profile program | improve loops | use register |
| change algorithm | re-order/pipeline loops | select/reorder instrs |
| transform loops | optimize procedure calls | peephole transfs |
| | eliminate procedure calls | |
| | straighten code | |
| | recognize common subexps | |
| | compute constant subexps | |
| | propagate constants | |
| | ... and lots more | |

1. Optimizations: Bird-Eye View

2. Recovering Program Structure from TAC
   - Basic Blocks
   - Control-Flow Graph (CFG)
   - Identifying Loops
   - Control-Flow-Graph Reducibility

3. Examples of Optimizations

4. Data-Flow Analysis
   - Reaching Definitions
   - Copy Propagation
   - Common-Subexpression Elimination (CSE)

# Optimizations at Three-Address-Code (TAC) Level

Interm-lang optimizations, e.g., TAC, portable to various backends:

- TAC is more flexible than AbSyn for local code transformations,
- we can rebuild the control-flow or global structure from TAC!

# Optimizations at Three-Address-Code (TAC) Level

Interm-lang optimizations, e.g., TAC, portable to various backends:

- TAC is more flexible than AbSyn for local code transformations,
- we can rebuild the control-flow or global structure from TAC!

A basic block is a TAC sequence where the flow of control enters at the begining and leaves at the end:

**Example of a Basic Block (straight-line code):**

```
L3: t5 := a  * b
    t6 := t5 + c
    d  := t2 * t2
    if(d = 0) goto L4
```

Local Optimizations: reordering TAC instructions in a basic block.Need to worry about the effects caused to the values of the variables at the start/end of the block.

# Identifying Basic Blocks (BB)

- Find the statements that start a basic block (BB):
  - first statement of any function
  - any labeled statement that is the target of a branch
  - any statement following a branch (conditional or unconditional)
- for each statement starting a BB, the BB consists of all stmts up to, but excluding, the start of a BB or the end of the program!

**Example: Finding Basic Blocks and The Control-Flow Graph (CFG)**

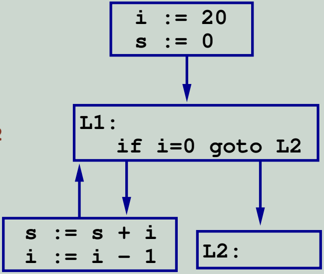
```
    i := 20
    s := 0
L1:
    if i=0 goto L2
    s := s + i
    i := i - 1
    goto L1
L2:
```

# Control-Flow Graph (CFG)

- Find the statements that start a basic block (BB):
  - first statement of any function
  - any labeled statement that is the target of a branch
  - any statement following a branch (conditional or unconditional)
- for each statement starting a BB, the BB consists of all stmts up to, but excluding, the start of a BB or the end of the program!

## Example: Finding Basic Blocks and The Control-Flow Graph (CFG)

```
    i := 20
    s := 0
L1:
    if i=0 goto L2
    s := s + i
    i := i - 1
    goto L1
L2:
```




Control-Flow Graph:
BBs are nodes in CFG.

Place an arrow from node $A$ to node $B$ if it is possible for control to "flow" from $A$ to $B$.

# Identifying Loops, Preliminaries

**Motivation**: loops is where most of the time is spend!

# Identifying Loops, Preliminaries

**Motivation**: loops is where most of the time is spend!

**Defition**: A loop, $L$, is a subgraph of the $\mathrm{CFG}$ such that:

- all nodes of the loop are strongly connected, i.e., the loop contains a path between any two loop nodes.
- the loop has an unique entry point, named header, such that
- the only way to reach a loop node is through the entry point.

A loop that contains no other loop is called an *inner loop*.

# Identifying Loops, Preliminaries

**Motivation**: loops is where most of the time is spend!

**Defition**: A loop, *L*, is a subgraph of the CFG such that:

- all nodes of the loop are strongly connected, i.e., the loop contains a path between any two loop nodes.
- the loop has an unique entry point, named header, such that
- the only way to reach a loop node is through the entry point.

A loop that contains no other loop is called an *inner loop*.

**Dominator Definition**: a node *p* dominates node *q* if all paths from the start of the program to *q* go through *p*.

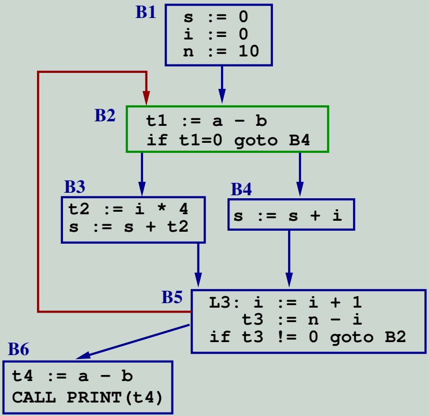
**Identifying** loops requires finding their "back edges":

- edges in the program in which the destination node dominates the source node.
- a loop must have an unique header, and one or more backedges.
- header dominates all blocks in the loop, otherwise not unique.

# Example of a Loop CFG

## Example: Finding Basic Blocks and The Control-Flow Graph (CFG)

```
    s := 0
    i := 0
    n := 10
L1:
    t1 := a - b
    if t1 = 0 goto L2
    t2 := i * 4
    s := s + t2
    goto L3
L2: s := s + i
L3: i := i + 1
    t3 := n - i
    if t3 != 0 goto L1
    t4 := a - b
    CALL PRINT(t4)
```



# Identifying Loops

**Algorithm for Dominators. $D(n)$ is the set of dominators of block $n$.**

**Input**: $\mathrm{CFG}$ with node set $N$, initial node $n_0$. **Output**: $D(n), \forall n \in N$

$D(n_0) := \{n_0\}$

for $n \in N - \{n_0\}$ do $D(n) := N$

while changes to any $D(n)$ occur do

  for $n \in N - \{n_0\}$ do

    $D(n) := \{n\} \cup (\cap_{p \in pred(n)} D(p))$

# Identifying Loops

**Algorithm for Dominators. $D(n)$ is the set of dominators of block $n$.**

**Input**: CFG with node set $N$, initial node $n_0$. **Output**: $D(n), \forall n \in N$

$D(n_0) := \{n_0\}$

for $n \in N - \{n_0\}$ do $D(n) := N$

while changes to any $D(n)$ occur do

  for $n \in N - \{n_0\}$ do

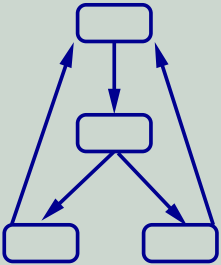
    $D(n) := \{n\} \cup (\cap_{p \in pred(n)} D(p))$

**High-Level Algorithm**: With each backedge $n \rightarrow d$ ($d$ is the loop header), we associate a natural loop (of $n \rightarrow d$) consisting of node $d$ and all nodes that can reach $n$ without going through $d$.

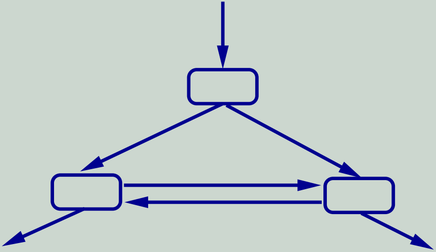
**Intuition**: since $d$ is the only entry to the loop, a path from any block outside the loop must pass through $d$.

# Reducible Control-Flow Graphs

**Which one is the innermost?**



**Improper Loop**



**A CFG is reducible** if it can be partitioned in forward and backward edges, where the forward edges form a directed-acyclic graph.

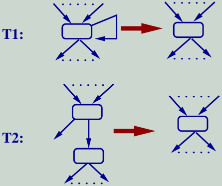
Non-reducible CFG: from unstructured use of GOTO & from allowing jumps into loops from outside the loop.

**Key property**: if a CFG is reducible then all cycles are (regular) loops, and identifying the backedges is enough to find all loops.

# Testing and Solving Ireducible CFG

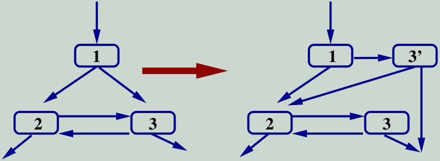
## Alg for Testing Reducibility

In a copy of the CFG, apply `T1` and `T2` to a fixpoint. If the result is a single node than the CFG is reducible.




## Node Splitting

Irreducible CFGs are difficult to optimize. It is always possible to solve irreducibility, but, in the worst case, at the cost of an exponential-code explosion:



# Example of Optimizations

We have already seen **liveness analysis & register allocation**.

**Common-Subexpression Elimination (CSE)**: if the same expression $e$ is computed twice, replace if possible the second occurrence of $e$ with the temporary that holds the value of the first computation of $e$.

**Copy Propagation (CP)**: after a statement `x := y`, we know that `x` and `y` have the same value, hence we can replace all occurrences of `x` with `y` between this assignment and the next definition of `x` or `y`.

**Dead-Code Elimination (DC)** (one reason is copy propagation):

- can safely remove any statement that defines a dead variable,
- a branch to dead code moved to whatever follows the dead code,
- if a branch-condition value is statically known $\Leftrightarrow$ merge two BBs.

**Constant Folding and Propagation (CtF/P)**: if possible, expressions should be computed at compile time, and the constant result should be propagated. **And these are just a few!**

# Example: Common-Subexpression Elimination (CSE) and Copy Propagation (CP)

| Original | After CSE1 | After CP1 | After CSE2 & CP2 |
|---|---|---|---|
| t1 := 4 - 2 | t1 := 4 - 2 | t1 := 4 - 2 | t1 := 4 - 2 |
| t2 := t1 / 2 | t2 := t1 / 2 | t2 := t1 / 2 | t2 := t1 / 2 |
| t3 := a * t2 | t3 := a * t2 | t3 := a * t2 | t3 := a * t2 |
| t4 := t3 * t1 | t4 := t3 * t1 | t4 := t3 * t1 | t4 := t3 * t1 |
| t5 := t4 + b | t5 := t4 + b | t5 := t4 + b | t5 := t4 + b |
| t6 := t3 * t1 | t6 := t4 | t6 := t4 | t6 := t4 |
| t7 := t6 + b | t7 := t6 + b | t7 := t4 + b | t7 := t5 |
| c := t5 * t7 | c := t5 * t7 | c := t5 * t7 | c := t5 * t5 |

Copy propagation makes further common-subexpression elimination possible and the reverse.

# Example Continuation: Constant Folding (CFP) & Copy Propagation (CP) & Dead Code Elim (DCE)

| Original | After CtFP | After CP | After DCE |
|---|---|---|---|
| t1 := 4 - 2 | t1 := 2 | t1 := 2 | |
| t2 := t1 / 2 | t2 := 1 | t2 := 1 | |
| t3 := a * t2 | t3 := a | t3 := a | |
| t4 := t3 * t1 | t4 := t3 * 2 | t4 := a * 2 | t4 := a * 2 |
| t5 := t4 + b | t5 := t4 + b | t5 := t4 + b | t5 := t4 + b |
| t6 := t4 | t6 := t4 | t6 := t4 | |
| t7 := t5 | t7 := t5 | t7 := t5 | |
| c := t5 * t5 | c := t5 * t5 | c := t5 * t5 | c := t5 * t5 |

One of the main difficulties is deciding in which order to apply these optimizations, and how many times to apply them...

1. Optimizations: Bird-Eye View
2. Recovering Program Structure from TAC
   - Basic Blocks
   - Control-Flow Graph (CFG)
   - Identifying Loops
   - Control-Flow-Graph Reducibility
3. Examples of Optimizations
4. Data-Flow Analysis
   - Reaching Definitions
   - Copy Propagation
   - Common-Subexpression Elimination (CSE)

# Data-Flow Analysis

Global analysis is used to derive information that can drive optimizations. Example: *Liveness analysis.*

Information can flow forwards or backwards through the program. Typical Structure:

- Successor/predecessor on basic blocks ($succ[B_i]$ or $pred[B_i]$).
- Define $gen[i]$ and $kill[i]$ sets.
- Define equations for $in[i]$ and $out[i]$.
- Initialize $in[i]$ and $out[i]$.
- Iterate to a fix point.
- Use $in[i]$ or $out[i]$ for optimizations.

# Reaching Definitions at Basic-Block Level

What variable definitions might reach a certain use of the variable?

**TAC Statement** $\gamma$, of shape `a := b op c`, generates a new definition for `a` and kills all previous definitions of `a`:

$$gen_r[\gamma] = \{\gamma\}, kill_r[\gamma] = D_a - \{\gamma\}, \text{ where } D_a = \text{set of all defs of a.}$$

**Intuitively, at basic-block level**, we just add (union) together the generated definitions for all statements in the basic lock, except for the case when a following statement kills a previous definition!

# Reaching Definitions at Basic-Block Level

What variable definitions might reach a certain use of the variable?

**TAC Statement** $\gamma$, of shape `a := b op c`, generates a new definition for `a` and kills all previous definitions of `a`:

$gen_r[\gamma] = \{\gamma\}, kill_r[\gamma] = D_a - \{\gamma\}$, where $D_a$ = set of all defs of a.

**Intuitively, at basic-block level**, we just add (union) together the generated definitions for all statements in the basic lock, except for the case when a following statement kills a previous definition!

**Formally, for basic block B**, denoting $\beta > \gamma$ if stmt $\beta$ follows $\gamma$:

$$gen_r[B] = \cup_{\gamma \in B}(gen_r[\gamma] - \cup_{\beta \in B}^{\beta > \gamma} kill_r[\beta])$$

$$kill_r[B] = \cup_{\gamma \in B}(kill_r[\gamma] - \cup_{\beta \in B}^{\beta > \gamma} gen_r[\beta])$$

# Reaching Definitions at Program Level (Global)

## Global-Reaching-Definitions Example

$pred_{B_i}$ : the set of blocks $B_j$ s.th. $\exists$ edge from $B_j$ to $B_i$.

**Initialization**: $\forall B_i$:

$in_r[B_i] = \emptyset$

$out_r[B_i] = gen_r[B_i]$.

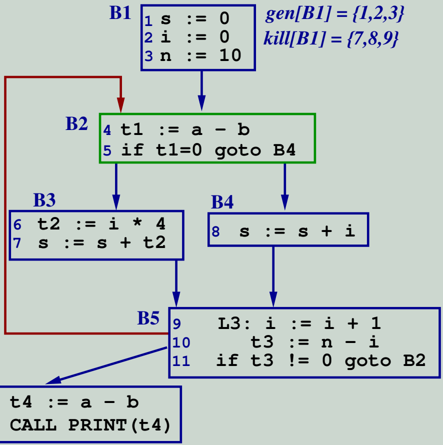
**Data-Flow Equations:**

$in_r[B_i] = \cup_{B_j \in pred(B_i)}(out_r[B_j])$

$out_r[B_i] =$

$(in_r[B_i] - kill_r[B_i]) \cup gen_r[B_i]$

Solve it $\Rightarrow$



# Copy Propagation at Basic-Block Level

Many optimizations and user code introduce copy stmts *s*: `x := y`

Possible to eliminate such copy stmts *s* if we determine all stmts *u* where `x` is used, by substituting `y` for `x`, provided that:

- *s* is the only definition of `x` reaching *u* (done that already!)
- any path from *s* to *u* exhibits no assignments to `y` (how to do?).

# Copy Propagation at Basic-Block Level

Many optimizations and user code introduce copy stmts $s$: `x := y`

Possible to eliminate such copy stmts $s$ if we determine all stmts $u$ where `x` is used, by substituting `y` for `x`, provided that:

- $s$ is the only definition of `x` reaching $u$ (done that already!)
- any path from $s$ to $u$ exhibits no assignments to `y` (how to do?).

**For each basic-block B**:

- $gen_c(B)$ is the set of copy stmts `x := y` for which there is no subsequent assignment to `y` (or `x`) within $B$.
- $kill_c(B)$ consists of all copy stmts `x := y`, anywhere in the program, that are killed within $B$ by a redefinition of `y` or `x`.

# Copy Propagation at Basic-Block Level

Many optimizations and user code introduce copy stmts $s$: `x := y`

Possible to eliminate such copy stmts $s$ if we determine all stmts $u$ where `x` is used, by substituting `y` for `x`, provided that:

- $s$ is the only definition of `x` reaching $u$ (done that already!)
- any path from $s$ to $u$ exhibits no assignments to `y` (how to do?).

**For each basic-block B**:

- $gen_c(B)$ is the set of copy stmts `x := y` for which there is no subsequent assignment to `y` (or `x`) within $B$.
- $kill_c(B)$ consists of all copy stmts `x := y`, anywhere in the program, that are killed within $B$ by a redefinition of `y` or `x`.

**Initialization**: we start with the *out* set containing all copy statements in the program other than the statements killed in the block itself (except for the entry block, which dominates all others).

# Copy Propagation at Program Level (Global)

## Global-Copy-Propagation Example

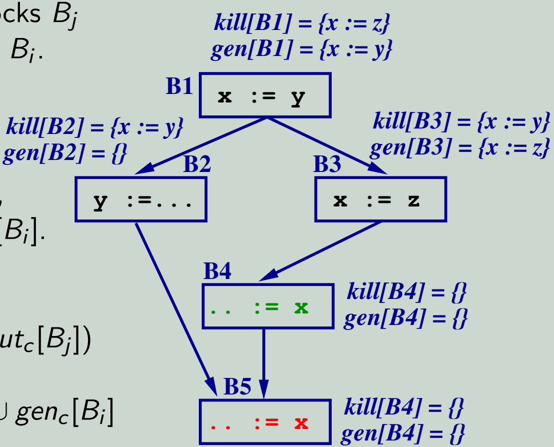
$C$ = set of all copy stmts in the prg. $pred_{B_i}$ = set of blocks $B_j$ s.th. $\exists$ edge from $B_j$ to $B_i$.

**Initialization**: $\forall B_i$:

$in_c[B_i] = \emptyset$,
$out_c[B_1] = gen_c[B_1]$,
$out_c[B_i] = C - kill_c[B_i]$.

**Data-Flow Equations:**

$in_c[B_i] = \cap_{B_j \in pred(B_i)}(out_c[B_j])$
$out_c[B_i] =$
$(in_c[B_i] - kill_c[B_i]) \cup gen_c[B_i]$




Solve it $\Rightarrow$

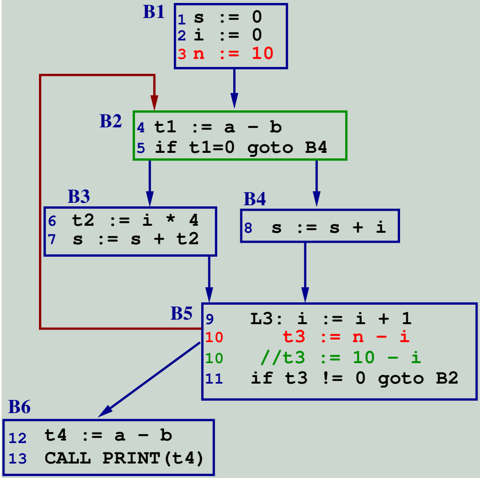
# Global Copy Propagation Algorithm

**Algorithm.** For each copy stmt $\gamma$, of shape `b` $:=$ `c` in program do:

- Find all uses of `b` that are reached by $\gamma$, i.e., references to `b` in block $B$ such that $\gamma \in in_r[B]$ and $B$ does not contain any earlier redefinition of `b` or `c`,
- For each such use, if furthermore $\gamma \in in_c[B]$ then replace `b` by `c`,
- If replacement succeeds in all the above cases, then remove $\gamma$.

# Demonstrating Copy Propagation

## Copy-Propagation Example




Copy stmt $1 \in in_r[B_3]$ and $1 \in in_r[B_4]$. However $1 \notin in_c[B_3]$ and $1 \notin in_c[B_4]$, hence replacement fails!

Similar for copy stmt 2.

Copy stmt $3 \in in_r[B_5] \cap in_c[B_5]$ and reaches the reference of `n` in stmt 10.
Hence it is safe to delete stmt 3 and to replace `n` with 10 in stmt 10!

# Common-Subexpression Elimination (CSE)

We wish to find two expressions that would always yield the same value, and to replace the second one with the result of the first!

We say that `b op c` is an available expression at stmt $S$ iff:

- all routes from the start of the program to $S$ must pass through the evaluation of `b op c`, and
- after the **last** eval of `b op c` there are no redefinitions of `b` or `c`.

# Common-Subexpression Elimination (CSE)

We wish to find two expressions that would always yield the same value, and to replace the second one with the result of the first!

We say that `b op c` is an available expression at stmt $S$ iff:

- all routes from the start of the program to $S$ must pass through the evaluation of `b op c`, and
- after the **last** eval of `b op c` there are no redefinitions of `b` or `c`.

This means that in $S$ we can replace `b op c` with the result of its previous evaluation. **Looks very similar to Copy Propagation!**

Intuitively, we can think again of killing and generating expressions:

- `b op c` is generated by the statement where it is evaluated,
- `b op c` is killed by a statement that redefines `b` or `c`.

# CSE at Basic-Block and Global Level

**Basic-Block (B) Level**: Initially set $gen_e[B] = \emptyset$, then

- Go through $B$'s stmts in sequence, and for each `a := b op c`
  - add `b op c` to $gen_e[B]$
  - delete any expression containing `a` from $gen_e[B]$
- $kill_e[B]$ = the set of all expressions anywhere in the program that are killed in $B$ by a redefinition of expression's operands.

**Init**: $in_e[B_1] = \emptyset$, $out_e[B_1] = gen_e[B_1]$, $out_e[B_i] = E - kill_e[B_i]$, where $E$ is the set of all available expressions in the program.

**Data-Flow Equations**: for each basic block $B_i$, with $i \neq 1$:

- $in_e[B_i] = \cap_{B_j \in pred(B_i)}(out_e[B_j])$,
- $out_e[B_i] = (in_e[B_i] - kill_e[B_i]) \cup gen_e[B_i]$.

# CSE Algorithm

For each basic block $B$ do:

(1) Find all statements $\gamma$ of shape `a := b op c` in $B$, such that `b op c` $\in in_e[B]$, and $B$ contains no earlier definitions of `b` or `c`.

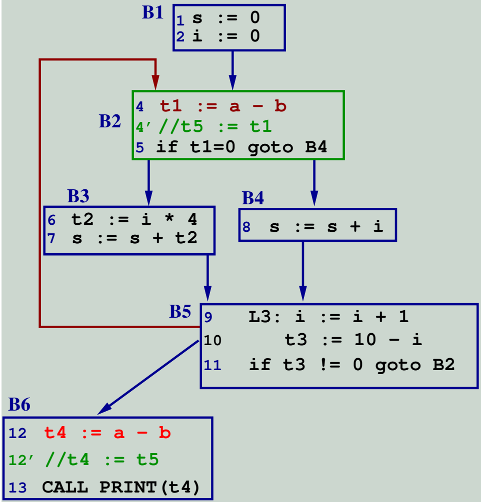
(2) For each stmt $\gamma$ found in (1), allocate fresh temporary `t`.

(3) From $B$, search backwards along all possible paths all previous statements `d := b op c`, and immediately after them insert `t := d`.

(4) Replace statement $\gamma$ by `a := t`.

# Demonstrating CSE

## Common-Subexpression-Elimination Example




Only stmt 12, i.e., `t4 := a-b` is found in stage (1) of Alg.

Searching backwards, the unique previous evaluation is in stmt 4, hence insert `t5 := t1` as stmt 4'.

Stage (4) of Alg replaces stmt 12 by `t4 := t5`

Subsequent copy propagation and dead-code elimination will eliminate `t4` and `t5`, i.e., both will be replaced by `t1`!