Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-16/

Prof. Andrea Corradini
Department of Computer Science, Pisa

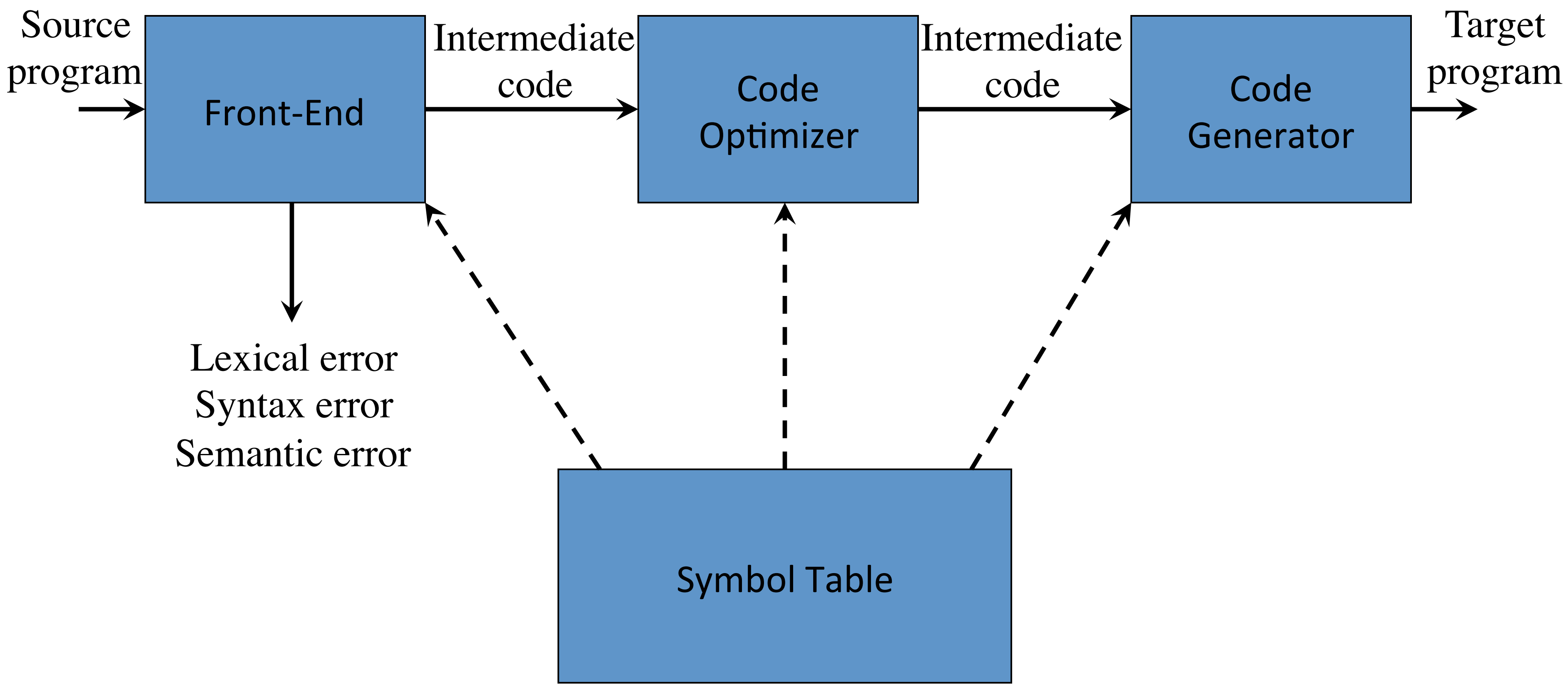
Lesson 12

- On code generation and local optimization

On Code Generation

- Code produced by compiler must be correct
  - Source-to-target program transformation should be *semantics preserving*
- Code produced by compiler should be of high quality
  - Effective use of target machine resources
  - Heuristic techniques should be used to generate good but suboptimal code, because generating optimal code is undecidable

Position of a Code Generator in the Compiler Model



Code Generation: tasks

- Code generation has three primary tasks:
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering
- The compiler can include an optimization phase (mapping IR to optimized IR) before the code generation
- We consider some simple optimizations only

Input of the Code Generator

- The input of code generation is the IR of the source program, with info in the symbol table
- Assumptions:
  - We assume that the IR is three-address code
  - Values and names in the IR can be manipulated directly by the target machine
  - The IR is free of syntactic and static semantic errors
  - Type conversion operators have been introduced where needed

Target Program Code

- The back-end code generator of a compiler may generate different forms of code, depending on the requirements:
  - Absolute machine code (executable code)
  - Relocatable machine code (object files for linker: allows separate compilation of subprograms)
  - Assembly language (facilitates debugging, but requires an assembly step)

Target Machine Architecture

- Defines the instruction-set, including addressing modes: high impact on the code generator
- **RISC** (*reduced instruction set computer*): single-clock instructions, many register, three address instructions, simple addressing modes
- **CISC** (*complex instruction set computer*): multi-clock instructions, complex addressing modes, several register classes, variable-length instructions operating in memory
- **Stack-based machines**: operands are put on the stack and operations act on top of stack (held in register). In general less efficient.
  - Revived thanks to bytecode forms for interpreters like the Java Virtual Machine

Our Target Machine

- We consider a RISC-like machine with some CISC-like addressing modes
- Assembly code as target language (for readability)
  - Variable names and constant are not translated
  - Absolute/relocatable target code requires to translate them using info from symbol table
- Our (hypothetical) machine:
  - Byte-addressable (word = 4 bytes)
  - Has *n* general purpose registers **R0**, **R1**, …, **R***n*-1
  - Simplified instruction-set: all operands are integer
  - Three-address instructions of the form

    op dest, *src1, src2*

The Target Machine: Instruction Set

- **`LD r, x`** (load operation: *r = x*)
- **`ST x, r`** (store operation: *x = r*)
- **`OP dst, src1, src2`** where OP = ADD, SUB, …: (apply *OP* to src1 and src2, placing the result in *dst*).
- **`BR`** ***`L`*** (unconditional jump: *goto L*)
- **`B`*****`cond`*** **`r, L`** (conditional jump: *if cond(r) goto L*) es: **`BLTZ r, L`** (*if (r < 0) goto L*)

The Target Machine: Addressing Modes

- Addressing modes and corresponding costs (*c* is an integer, *x* is a variable name, **`R`** is a register):

Mode	Form	Address (l-value)	Added Cost
Absolute	**`M`**	**`M`**	1
Variable name	*x*	location of *x* in memory	1
Register	**`R`**	**`R`**	0
Indexed address	*c*(**`R`**)	*c*+*contents*(**`R`**)	1
Indexed integer	*x*(**`R`**)	l-value of *x* +*contents*(**`R`**)	1
Indirect register	**`*R`**	*contents*(**`R`**)	0
Indirect indexed	**`*`***c*(**`R`**)	*contents*(*c*+*contents*(**`R`**))	1
Literal	**`#`***c*	N/A	1

Instruction Costs

- Machine is a simple, non-super-scalar processor with fixed instruction costs
- Realistic machines have deep pipelines, various kinds of caches, parallel instructions, etc.
- Define:

cost (**`OP dst, src1, src2`**) = 1
+ cost(*dst*-mode)
+ cost(*src1*-mode)
+ cost(*src2*-mode)

- Cost corresponds to length of instructions in memory

Examples

Instruction	Operation	Cost
`LD R0,R1`	Load *content*(**`R1`**) into register **`R0`**	1
`LD R0,M`	Load *content*(**`M`**) into register **`R0`**	2
`ST M,R0`	Store *content*(**`R0`**) into memory location **`R0`**	2
`BR 20(R0)`	Jump to address 20+*contents*(**`R0`**)	2
`ADD R0 R0 #1`	Increment **`R0`** by 1	2
`MUL R0,M,*12(R1)`	Multiply *contents*(**`M`**) by *contents*(12+*contents*(**`R1`**) and store the result in **`R0`**	3

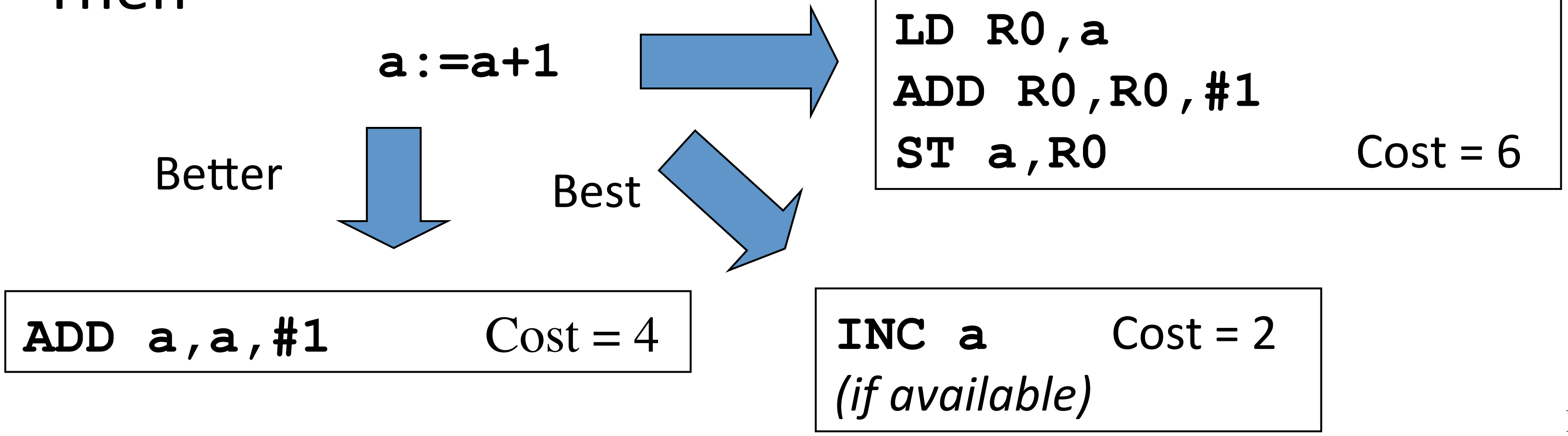
Instruction Selection

- Instruction selection depends on (1) the level of the IR, (2) the instruction-set architecture, (3) the desired quality (e.g. efficiency) of the generated code
- Suppose we translate three-address code

  x:=y+z to:

```
LD R0,y        \\ R0=y
ADD R0,R0,z    \\ R0=R0+z
ST x,R0        \\ x=R0
```

- Then



```
a:=a+1
```

```
LD R0,a
ADD R0,R0,#1
ST a,R0               Cost = 6
```

```
ADD a,a,#1       Cost = 4
```

```
INC a       Cost = 2
(if available)
```

Need for Global Machine-Specific Code Optimizations

- Suppose we translate three-address code
  x:=*y*+*z* to:

```
LD R0,y        \\ R0=y
ADD R0,R0,z    \\ R0=R0+z
ST x,R0        \\ x=R0
```

- Then, we translate
  `a:=b+c`
  `d:=a+e` to:

```
LD R0,b
ADD R0,R0,c
ST a,R0
LD R0,a
ADD R0,R0,e
ST d,R0
```

Redundant

- We can choose among several equivalent instruction sequences → *Dynamic programming* algorithms

Register Allocation and Assignment

- Efficient utilization of the limited set of registers is important to generate good code
- Registers are assigned by
  - *Register allocation* to select the set of variables that will reside in registers at a point in the code
  - *Register assignment* to pick the specific register that a variable will reside in
- Finding an optimal register assignment in general is NP-complete

Choice of Instruction Ordering

- When instructions are independent, their evaluation order can be changed

```
a+b-(c+d)*e
```

$\Longrightarrow$

```
t1:=a+b
t2:=c+d
t3:=e*t2
t4:=t1-t3
```

$\xRightarrow{\text{reorder}}$

```
t2:=c+d
t3:=e*t2
t1:=a+b
t4:=t1-t3
```

- The reordered sequence could lead to a better target code

Towards Flow Graphs

- In order to improve *instruction selection, register allocation and selection,* and *instruction ordering,* we structure the input three-address code as a *flow graph*
- This allows to make explicit certain dependencies among instructions of the IR
- Simple optimization techniques are based on the analysis of such dependencies
  - Better register allocation knowing how variables are defined and used
  - Better instruction selection looking at *sequences* of three-address code statements

Flow Graphs

- A *flow graph* is a graphical representation of a sequence of instructions with control flow edges
- A flow graph can be defined at the intermediate code level or target code level
- Nodes are *basic blocks*, sequences of instructions that are always executed together
- Arcs are execution order dependencies

Basic Blocks

- A *basic block* is a sequence of instructions s.t.:
  - Control enters through the first instruction only
  - Control leaves the block without branching, except possibly at the last instruction

```
2)      j=1
3)      t1=10*i
4)      t2=t1+j
5)      t3=8*t2
6)      t4=t3-88
7)      a[t4]=0.0
8)      j=j+1
9)      if j<=10 goto (3)
10)     i=i+1
11)     if i<=10 goto(2)
12)     i=1
```

```
2)      j=1
```

```
3)      t1=10*i
4)      t2=t1+j
5)      t3=8*t2
6)      t4=t3-88
7)      a[t4]=0.0
8)      j=j+1
9)      if j<=10 goto (3)
```

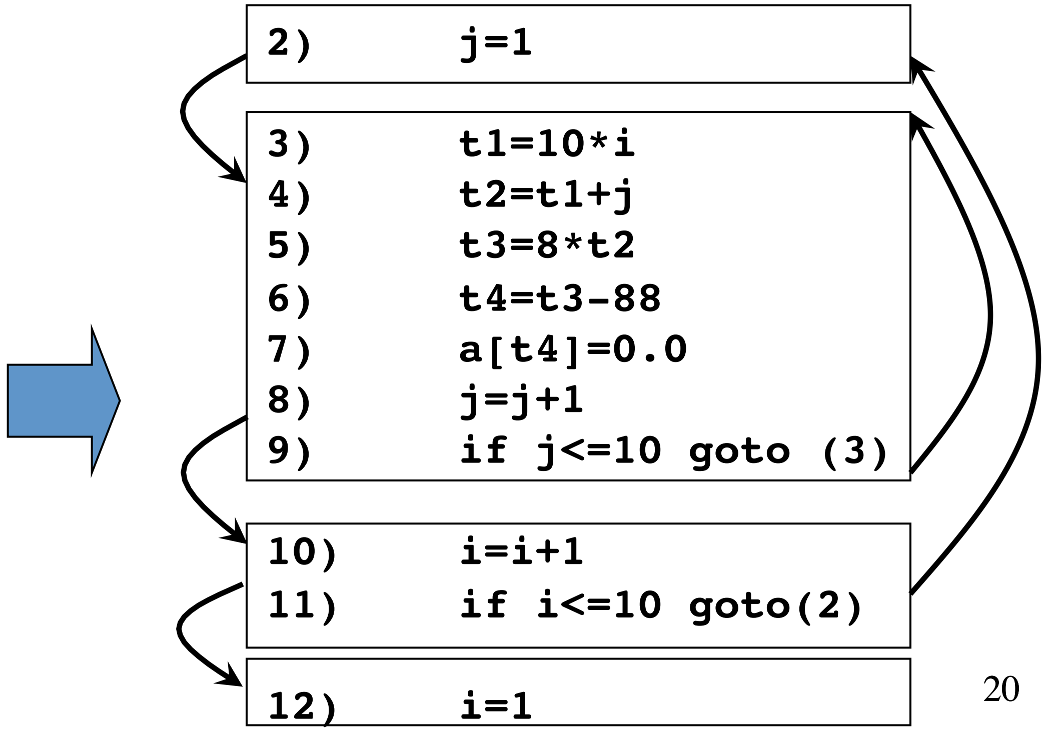
```
10)     i=i+1
11)     if i<=10 goto(2)
```

```
12)     i=1
```

Basic Blocks and Control Flow Graphs

- A control flow graph (CFG) is a directed graph with basic blocks B_i as vertices and with edges $B_i \to B_j$ iff B_j can be executed immediately after B_i
- Then B_i is a predecessor of B_j, B_j is a successor of B_i

```
2)      j=1
3)      t1=10*i
4)      t2=t1+j
5)      t3=8*t2
6)      t4=t3-88
7)      a[t4]=0.0
8)      j=j+1
9)      if j<=10 goto (3)
10)     i=i+1
11)     if i<=10 goto(2)
12)     i=1
```


```
2)      j=1
```
```
3)      t1=10*i
4)      t2=t1+j
5)      t3=8*t2
6)      t4=t3-88
7)      a[t4]=0.0
8)      j=j+1
9)      if j<=10 goto (3)
```
```
10)     i=i+1
11)     if i<=10 goto(2)
```
```
12)     i=1
```

Partition Algorithm for Basic Blocks

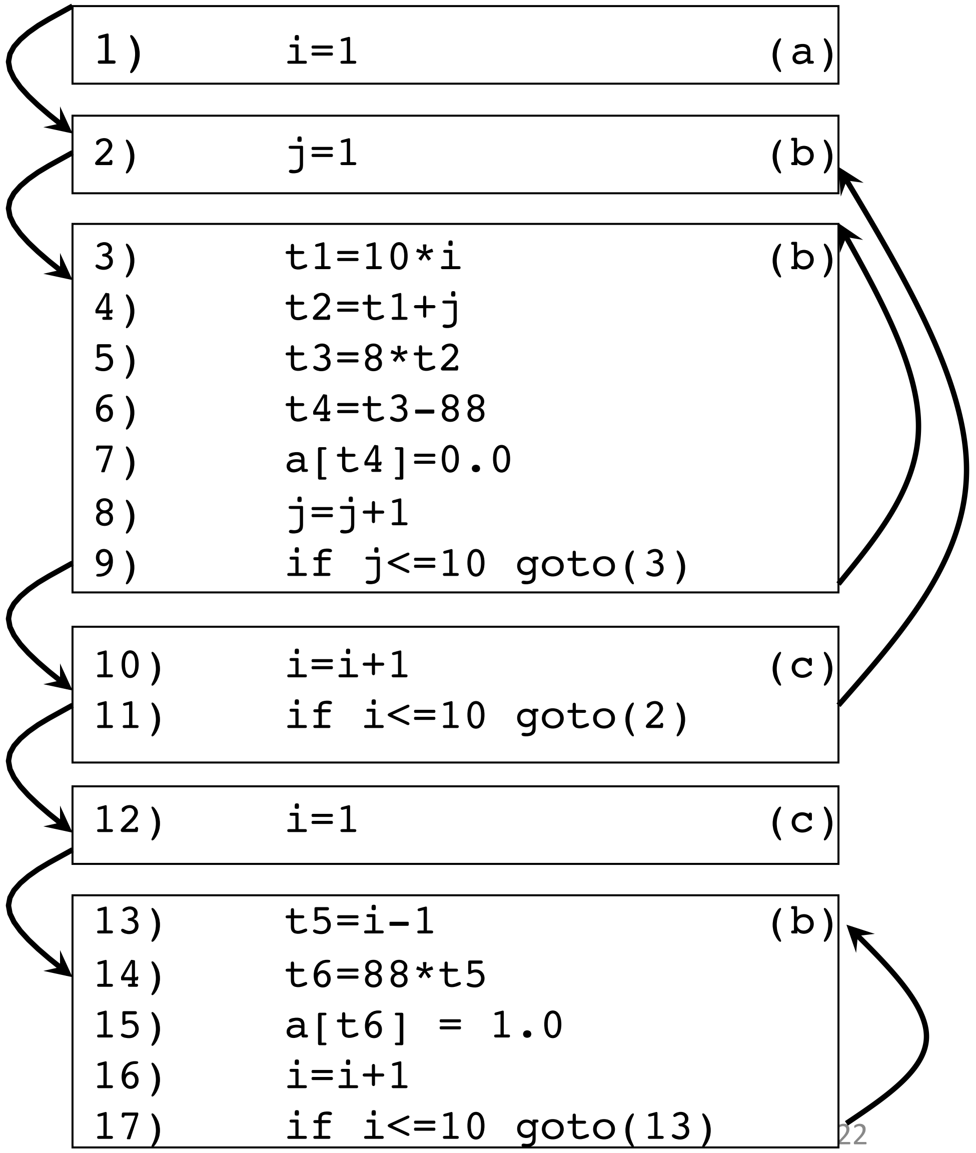
Input: A sequence of three-address statements
Output: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements in basic blocks
   a) The first statement is the leader
   b) Any statement that is the target of a *goto* is a leader
   c) Any statement that immediately follows a *goto* is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Partition Algorithm for Basic Blocks: Example

```
1)      i=1                  (a)
2)      j=1                  (b)
3)      t1=10*i              (b)
4)      t2=t1+j
5)      t3=8*t2
6)      t4=t3-88
7)      a[t4]=0.0
8)      j=j+1
9)      if j<=10 goto(3)
10)     i=i+1                (c)
11)     if i<=10 goto(2)
12)     i=1                  (c)
13)     t5=i-1               (b)
14)     t6=88*t5
15)     a[t6] = 1.0
16)     i=i+1
17)     if i<=10 goto(13)
```

Leaders

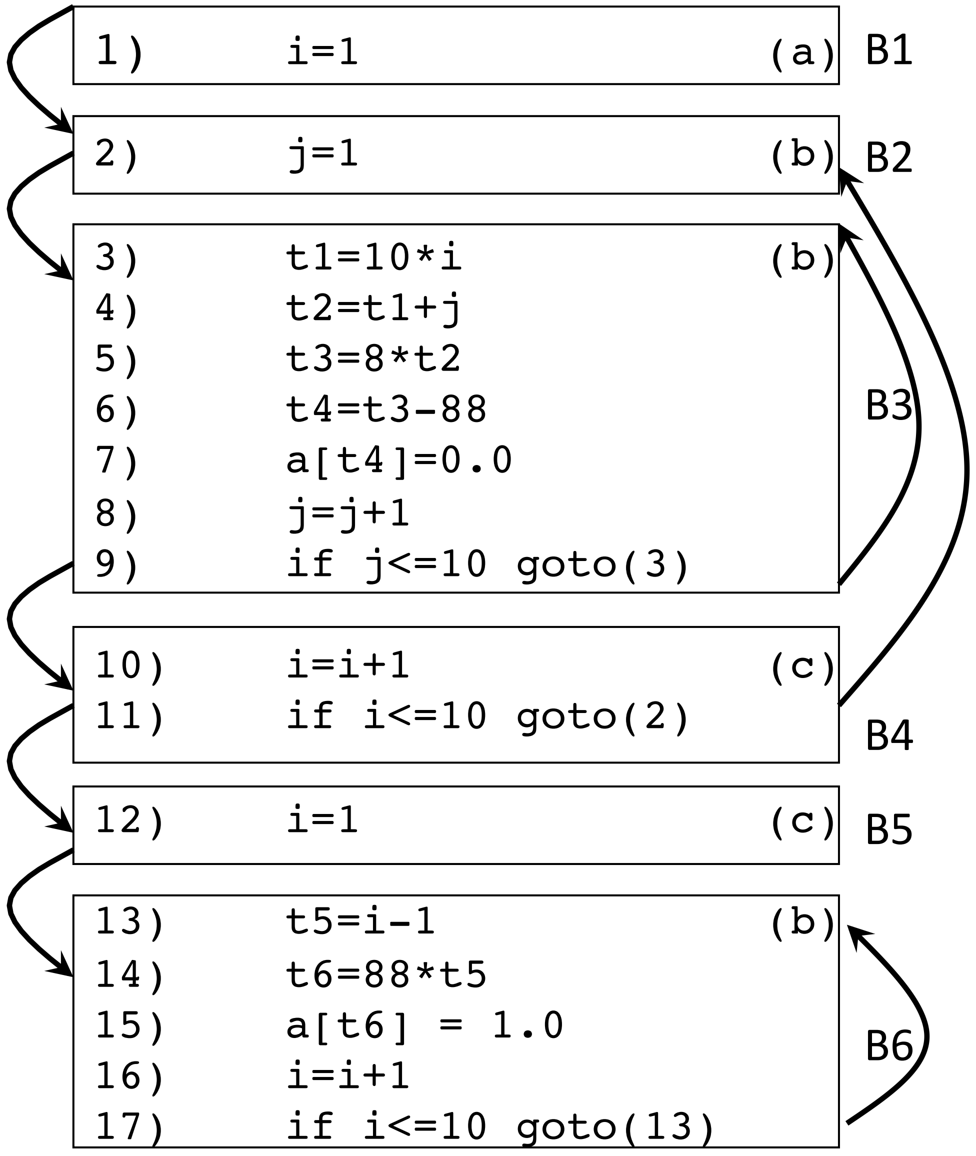


```
1)      i=1                  (a)

2)      j=1                  (b)

3)      t1=10*i              (b)
4)      t2=t1+j
5)      t3=8*t2
6)      t4=t3-88
7)      a[t4]=0.0
8)      j=j+1
9)      if j<=10 goto(3)

10)     i=i+1                (c)
11)     if i<=10 goto(2)

12)     i=1                  (c)

13)     t5=i-1               (b)
14)     t6=88*t5
15)     a[t6] = 1.0
16)     i=i+1
17)     if i<=10 goto(13)
```

Loops

- Programs spend most of the time executing loops
- Identifying and optimizing loops is important during code generation
- A *loop* is a collection of basic blocks, such that
  - All blocks in the collection are *strongly connected*
  - The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry

Loops (Example)



```
1)      i=1                         (a)

2)      j=1                         (b)

3)      t1=10*i                     (b)
4)      t2=t1+j
5)      t3=8*t2
6)      t4=t3-88
7)      a[t4]=0.0
8)      j=j+1
9)      if j<=10 goto(3)

10)     i=i+1                       (c)
11)     if i<=10 goto(2)

12)     i=1                         (c)

13)     t5=i-1                      (b)
14)     t6=88*t5
15)     a[t6] = 1.0
16)     i=i+1
17)     if i<=10 goto(13)
```

Strongly connected components:

SCC={ {B2,B3,B4},
{B3}, {B6} }

Entries:
B2, B3, B6

Local vs Global Optimization

- Code optimization techniques that work in the scope of a basic block are called *local optimizations*.
  - DAG based optimizations
  - Peephole optimizations
  - Local register allocation
- Code optimization techniques that need to analyze the entire control flow graph of a program are called *global optimizations*.
  - Dataflow-analysis based optimizations
  - (Global) register allocation

Equivalence of Basic Blocks

- Local optimization must ensure that the optimized block is equivalent to the original one
- Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions

```
b   := 0
t1  := a + b
t2  := c * t1
a   := t2
```

```
a   := c * a
b   := 0
```

```
a := c*a
b := 0
```

```
a := c*a
b := 0
```

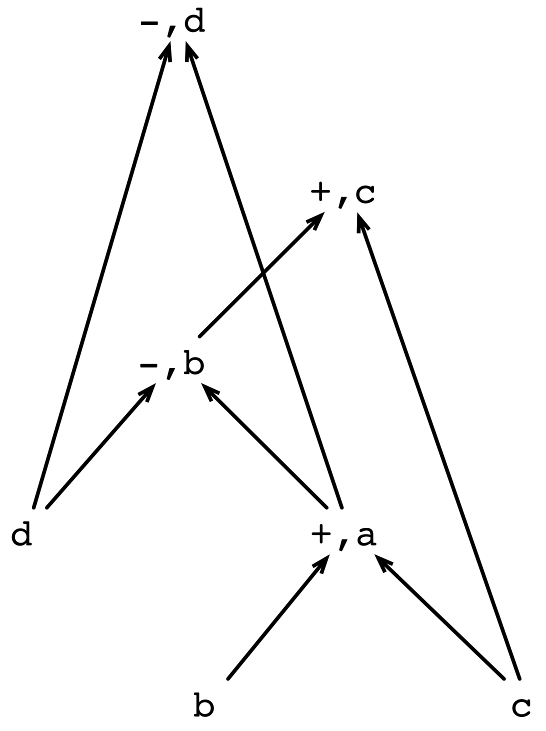
Blocks are equivalent, assuming **t1** and **t2** are *dead*: no longer used (no longer *live*)

DAG-Based Optimizations

- Some local optimizations relies on a directed acyclic representation of the instructions in the basic block. These DAGS are constructed as follows:
  - There is a node in the DAG for each *input value* appearing in the basic block.
  - There is a node associated with each instruction in the basic block.
  - If instruction S uses variables defined in statements S_1, …, S_n, then we have edges from each S_i to S.
  - If a variable is defined in the basic block, but is not used inside it, then we mark it as an *output value*.

Example of DAG Representation

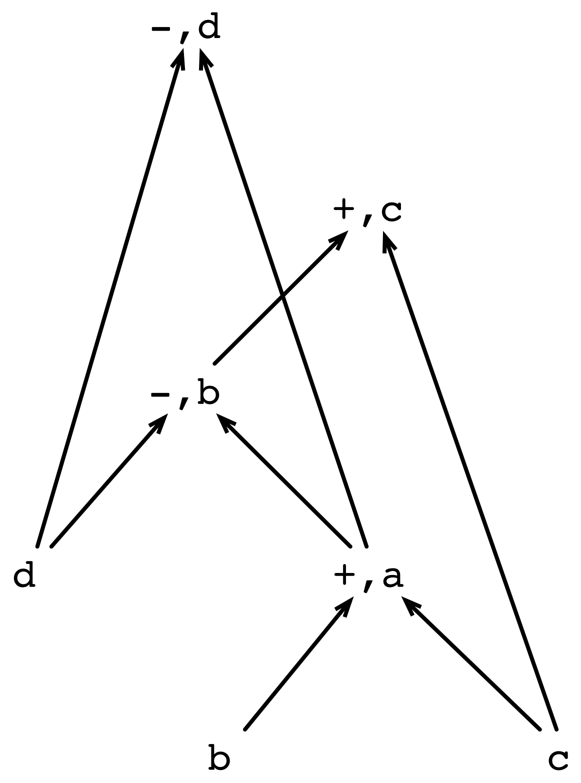
1: a = b + c
2: b = a – d
3: c = b + c
4: d = a – d



- In the basic block, the first occurrences of b, c and d are *input values*, because they are used in the block, but are not defined before the first use.
- The definitions of c and d, at lines 3 and 4, are *output values*, because they are not used in the basic block.

Algorithm for DAG Representation

1: a = b + c
2: b = a - d
3: c = b + c
4: d = a - d

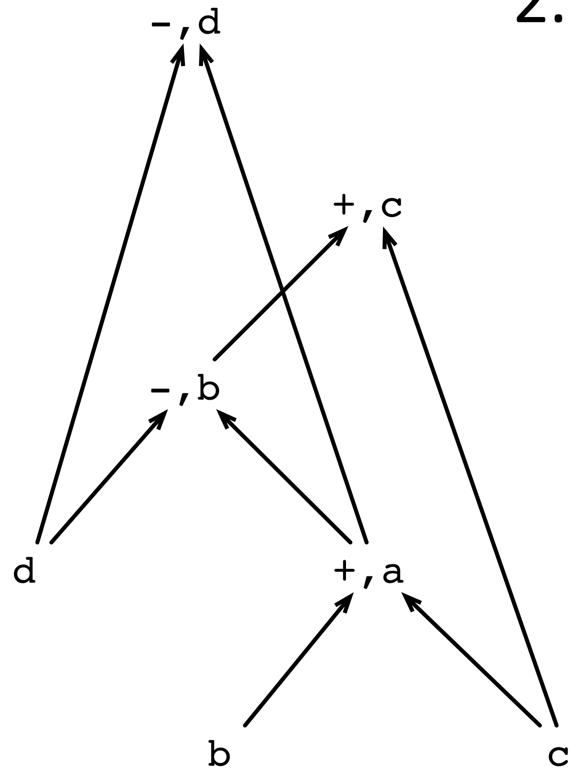


- For each input value v_i:
  - create a node v_i in the DAG
  - label this node with the tag ***in***
- For each statement $v = \boldsymbol{f}(v_1, ..., v_n)$, in the basic block:
  - create a node v in the DAG
  - create an edge (v_1, v) for each i, $1 \leq i \leq n$
  - label this node with the tag $\boldsymbol{f}$

How could we adapt our algorithm to reuse expressions that have already been created?

Finding Local Common Subexpressions

```
1: a = b + c
2: b = a - d
3: c = b + c
4: d = a - d
```

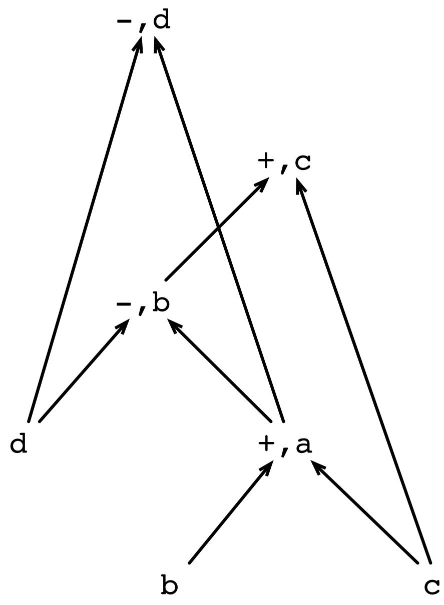


1. For each input value v_i:
    1. create a node v_i in the DAG
    2. label this node with the tag ***in***
2. For each statement $v = \boldsymbol{f}(v_1, ..., v_n)$, in the sequence defined by the basic block:
    1. If the DAG already **contains** a node v' labeled f, with all the children $(v_1, ..., v_n)$ in the order given by i, $1 \le i \le n$
        1. Let v' be an alias of v in the DAG
    2. else:
        1. create a node v in the DAG
        2. create an edge (v_1, v) for each i, $1 \le i \le n$
        3. label this node with the tag ***f***

Value Numbers

- We associate each node with a signature (lb, v_1, …., v_n), where lb is the label, and each v_i, $1 \leq i \leq n$ *is* a child
  - We use this signature as the key of a *hash-function*
  - The value produced by the hash-function is called the *value-number* of that node

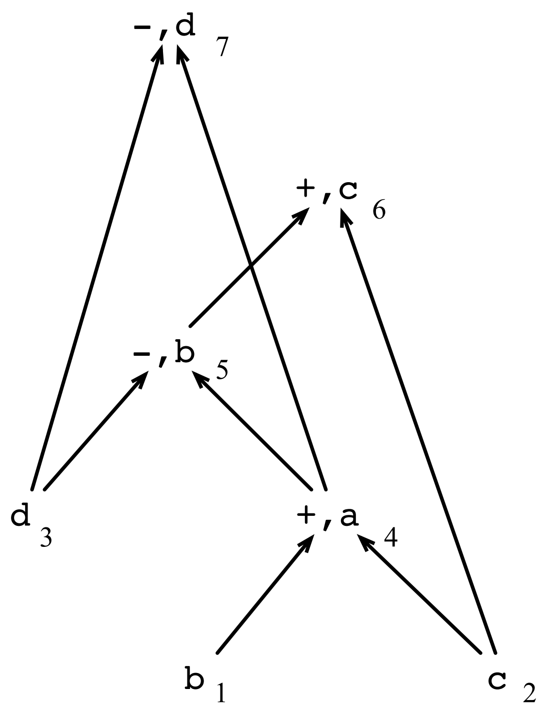
- Whenever we build a new node to our DAG, e.g., step 2.1 of our algorithm, we perform a search in the hash-table. If the node is already there, we simply return a reference to it.



What is the result of optimizing this DAG?

Value Numbers

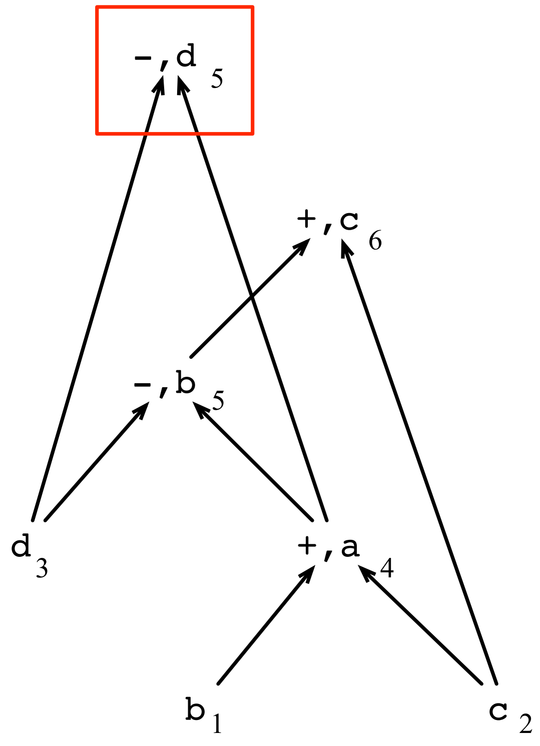
- Value numbers allows us to refer to nodes in our DAG by their semantics, instead of by their textual program representation.



Original DAG

1 (b)	`(in, _)`
2 (c)	`(in, _)`
3 (d)	`(in, _)`
4 (a = b + c)	`(+, 1, 2)`
5 (b = d - a)	`(-, 3, 4)`
6 (c = b + c)	`(+, 5, 2)`
7 (d = d - a)	`(-, 3, 4)`

Value-Number Table



Optimized DAG

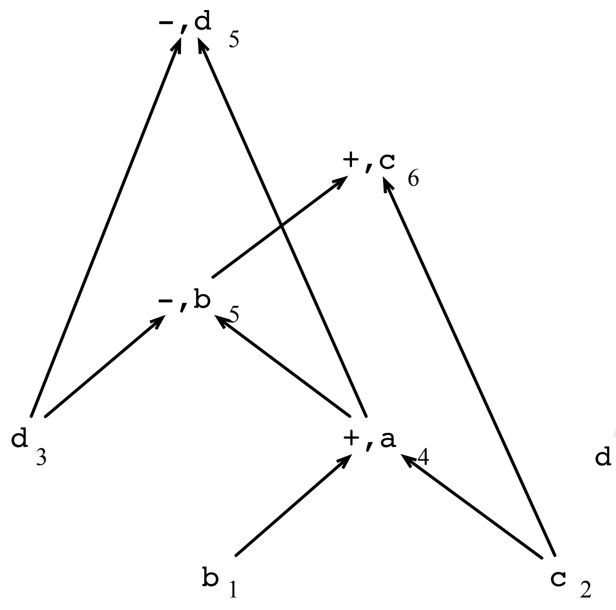
Finding more identities

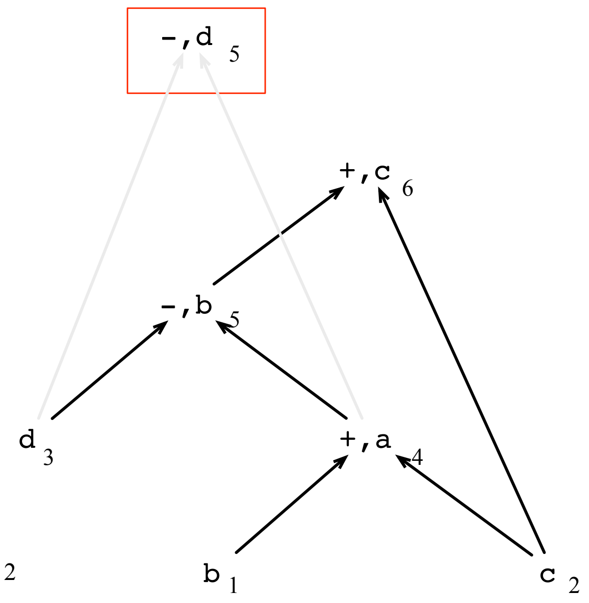
- We can use several tricks to find common subexpressions in DAGs
  - Commutativity: the value number of x + y and y + x should be the same.
  - Identities: the comparison x < y can often be implemented by t = x – y; t < 0
  - Associativity:

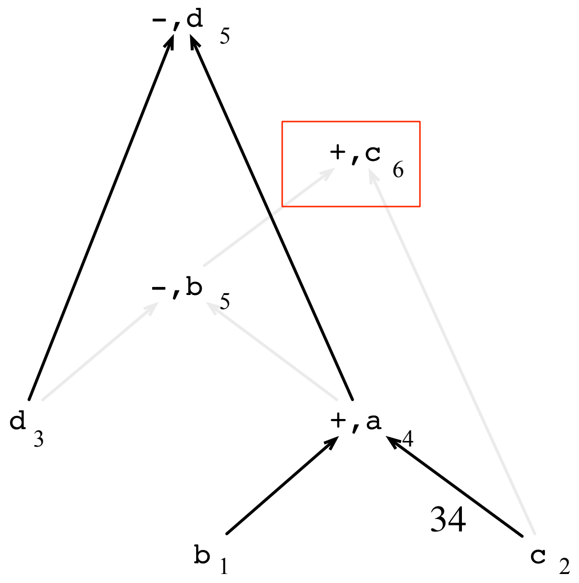
$$\begin{array}{l} a = b + c \\ t = c + d \\ e = t + b \end{array} \longrightarrow \begin{array}{l} a = b + c \\ e = a + d \end{array}$$

Dead Code Elimination

- We can eliminate any node if:
  - This node has no descendants, e.g., it is a *root node*.
  - This node is not marked as an *output node*.
- We can iterate this pattern of eliminations until we have no more nodes that we can remove.
- Examples if (-, d) (or (+,c)) were not an output node







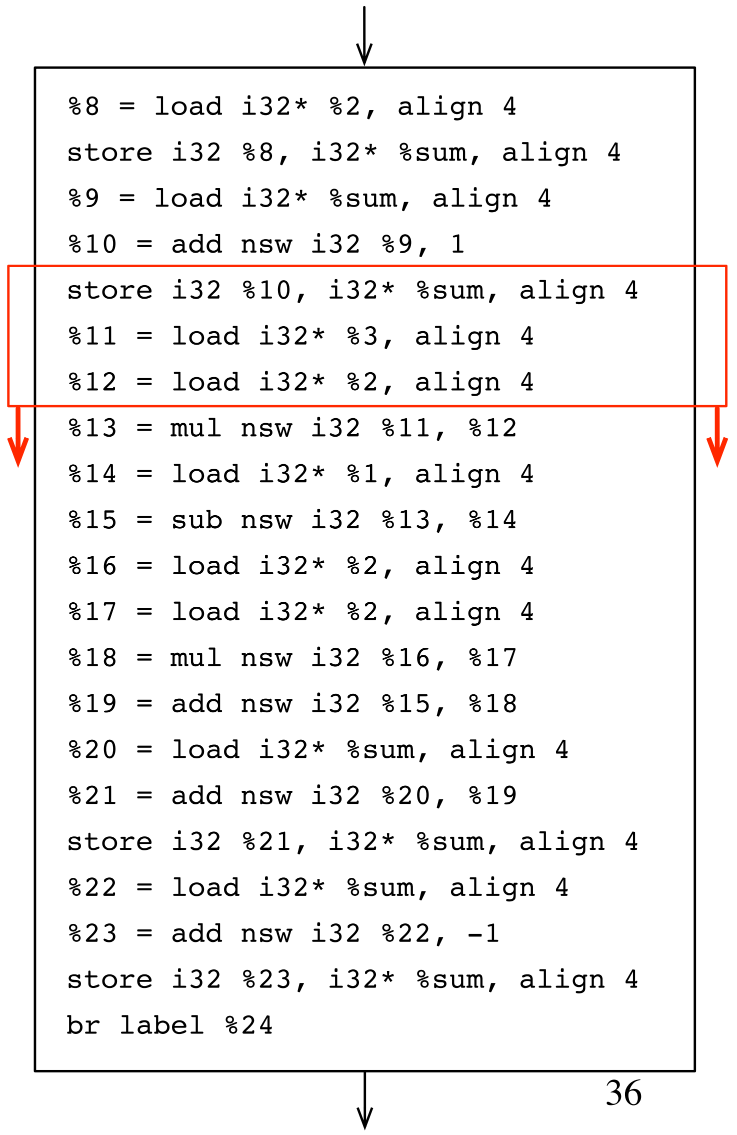
Algebraic Identities

- We can exploit algebraic identities to optimize DAGs
  - Arithmetic Identities:
    - $x + 0 = 0 + x = x$; $x * 1 = 1 * x = x$; $x – 0 = x$; $x/1 = x$
  - Reduction in strength:
    - $x^2 = x * x$; $2 * x = x + x$; $x / 2 = x * 0.5$; $4 * x = x << 2$

    Reduction in strength optimizes code by replacing some sequences of instructions by others, which can be computed more efficiently.
  - Constant folding:
    - evaluate expressions at compilation time, replacing each expression by its value

Peephole Optimizations

- Peephole optimizations are a category of local code optimizations
- The principle is simple:
  - the optimizer analyzes sequences of instructions.
  - only code that is within a small window of instructions is analyzed each time.
  - this window slides over the code.
  - once patterns are discovered inside this window, optimizations are applied.
- May go across the border of basic blocks



```
%8 = load i32* %2, align 4
store i32 %8, i32* %sum, align 4
%9 = load i32* %sum, align 4
%10 = add nsw i32 %9, 1
store i32 %10, i32* %sum, align 4
%11 = load i32* %3, align 4
%12 = load i32* %2, align 4
%13 = mul nsw i32 %11, %12
%14 = load i32* %1, align 4
%15 = sub nsw i32 %13, %14
%16 = load i32* %2, align 4
%17 = load i32* %2, align 4
%18 = mul nsw i32 %16, %17
%19 = add nsw i32 %15, %18
%20 = load i32* %sum, align 4
%21 = add nsw i32 %20, %19
store i32 %21, i32* %sum, align 4
%22 = load i32* %sum, align 4
%23 = add nsw i32 %22, -1
store i32 %23, i32* %sum, align 4
br label %24
```

Redundant Loads and Stores

- Some memory access patterns are clearly redundant:

```
load R0, m
store m, RO
```

- Patterns like this can be easily eliminated by a peephole optimizer.
    - Why is this optimization only safe inside a basic block?

Branch Transformations

- Some branches can be rewritten into faster code. For instance:

```
if debug == 1 goto L1
goto L2
L1: …
L2: …
```

```
if debug != 1 goto L2
L1: …
L2: …
```

- This optimization crosses the boundaries of basic blocks, but it is still performed on a small sliding window.

Jumps to Jumps

- How could we optimize this code sequence?

```
    goto L1
    ...
L1: goto L2
```

Jumps to Jumps

- How could we optimize this code sequence?

```
    goto L1
    ...
L1: goto L2
```

➡

```
    goto L2
    ...
L1: goto L2
```

Could we remove the third line altogether?

Jumps to Jumps

- How could we optimize this code sequence?

```
    goto L1
    ...
L1: goto L2
```

$\Rightarrow$

```
    goto L2
    ...
L1: goto L2
```

$\Rightarrow$

```
goto L2
...
```

- We can eliminate the second jump, provided that we know that there is no jump to L1 in the rest of the program.
- Notice that this peephole optimization requires some previous information gathering: we must know which instructions are targets of jumps.

Jumps to Jumps

- How could we optimize this code sequence?

We saw how to optimize the sequence on the right. Does the same optimization applies on the code below?

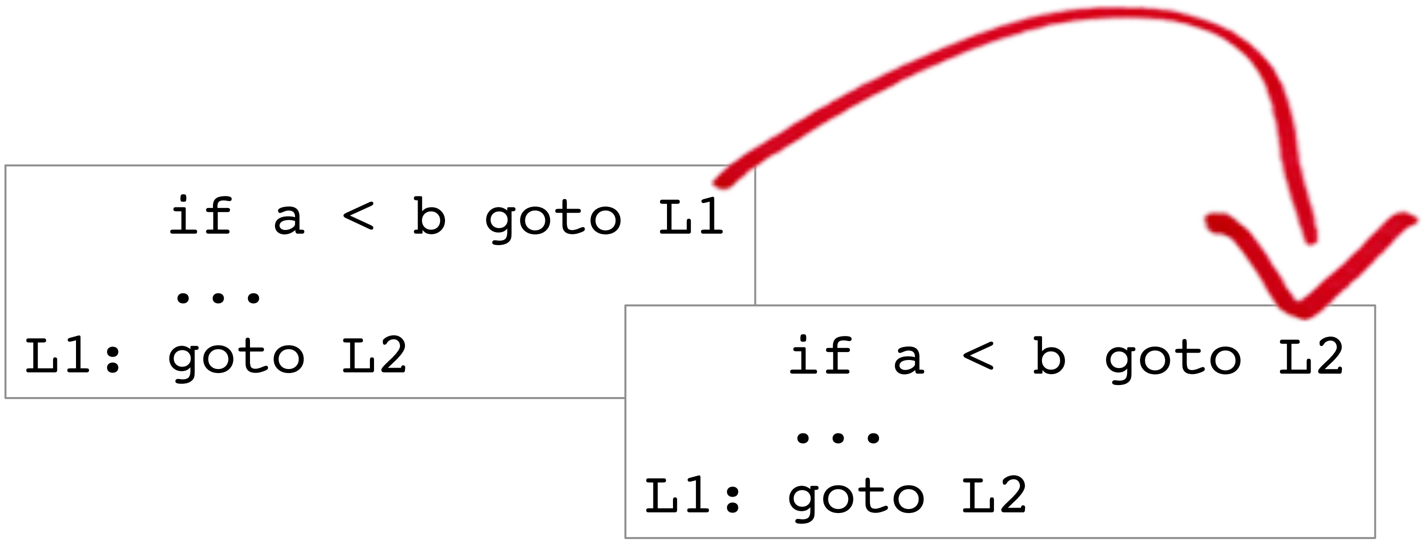
```
    goto L1
    ...
L1: goto L2
```

```
    if a < b goto L1
    ...
L1: goto L2
```

Jumps to Jumps

- How could we optimize this code sequence?

The optimization is the same as in the previous case. and if there is no jump to L1, we can remove it from the code, as it becomes dead-code after our transformation.



```
    if a < b goto L1
    ...
L1: goto L2
```

```
    if a < b goto L2
    ...
L1: goto L2
```

Renaming Temporary Variables

- Temporary variables that are dead at the end of a block can be safely renamed

```
t1 := b + c
t2 := a - t1
t1 := t1 * d
d := t2 + t1
```

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

Normal-form block

Interchange of Statements

- Independent statements can be reordered

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

$\Rightarrow$

```
t1 := b + c
t3 := t1 * d
t2 := a - t1
d := t2 + t3
```

Note that normal-form blocks permit all statement interchanges that are possible