Introduction to CS 293S Advanced Compiler Technology --- Code Optimizations for Scalar and Parallel Programs

Yufei Ding

Review of Last Lecture

- Course overview
- Code optimization
  - Metric & importance of program quality
  - Who makes the enhancement?
    - Compiler is the primary engine
  - How to do it?
    - Program analysis
    - Program transformations
- Consideration of optimization
- Sources of inefficiency

This Lecture

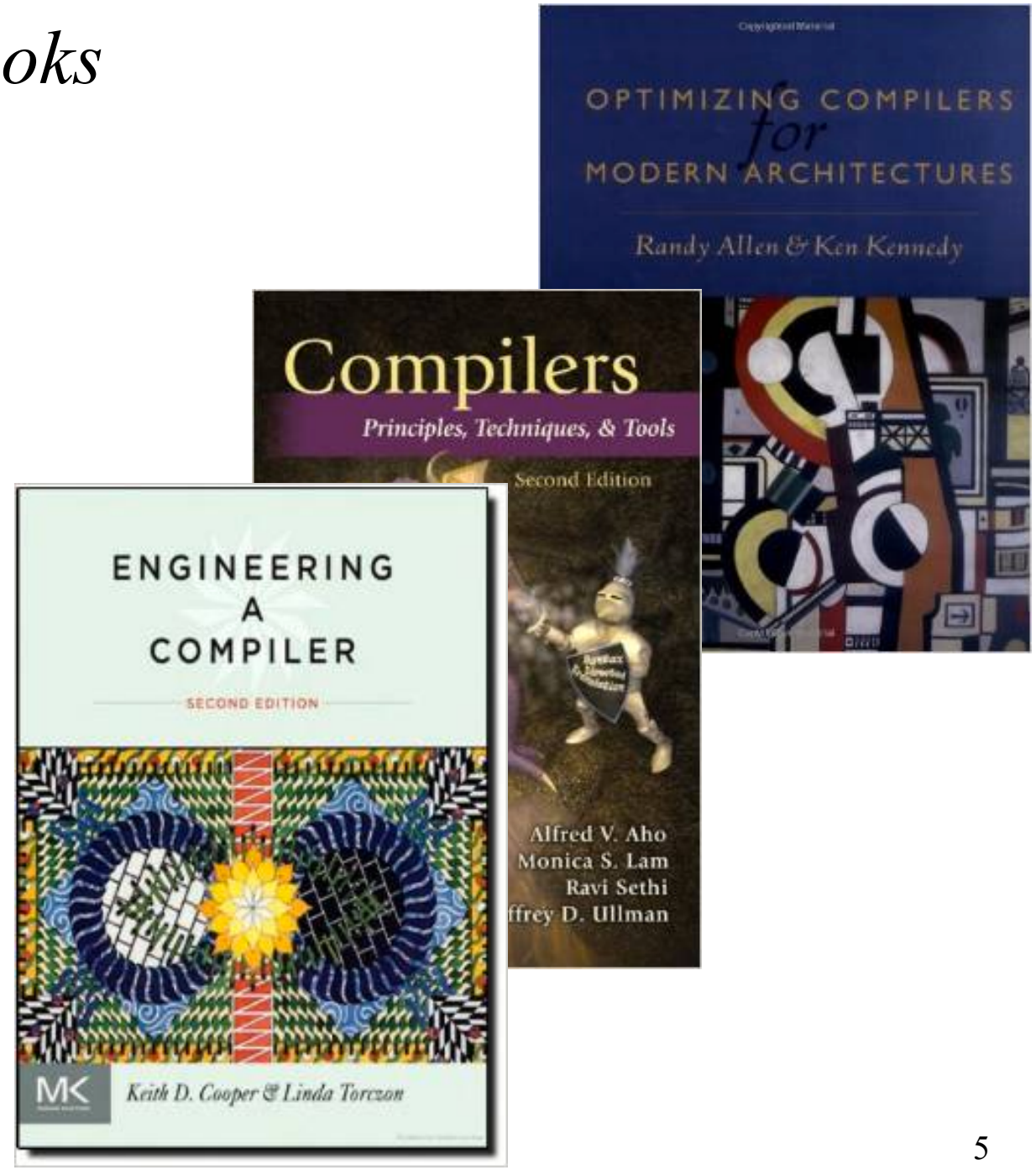
- Redundancy Elimination

Topics

- Types of intermediate representations of a program
- Removing redundant expressions
  - DAG: version tracking
  - Linear representation: value numbering

Reference books

CT book



Intermediate Representations (Details in Ch. 5 of CT book)

Why use an intermediate representation?

Key insight: An intermediate representation is a compile-time data structure

- break the compiler into manageable pieces – good software engineering technique
- simplifies handling of "poly-architecture" problem

  m languages, n targets ⇒ m+n components

Intermediate Representations (Details in Ch. 5 of CT book)

What are the good properties of an intermediate representation?

- ease of generation
- ease of manipulation
- cost of manipulation
- level of abstraction
- freedom of expression
- size of typical procedure

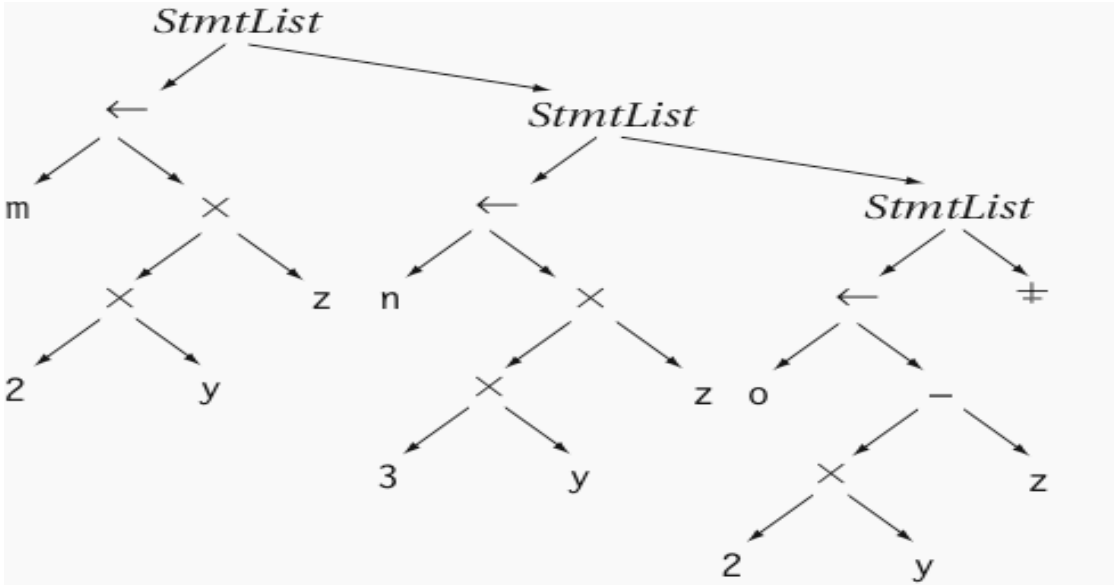
Level of exposed detail is a crucial consideration. The selection of IR would affect the speed and effectiveness of the compiler.

Types of Intermediate Representations (Details in Ch. 5 of CT book)

Three major categories

- I: Structural
  - Graphically oriented
  - Heavily used in source-to-source translators
  - Tend to be large

Examples:
Trees,
Directed Acyclic Graphs (DAGs),
Dependence graph




```
m <-- 2 x y x z
n <--  3 x y x z
o <--  2 x y - z
```

Types of Intermediate Representations (Details in Ch. 5 of CT book)

Three major categories

- 2: Linear
  - Pseudo-code for an abstract machine
  - Simple, compact data structures
  - Easier to rearrange

Examples:
Three-address code
X <- y op z

```
z ← x - 2 * y
```

becomes

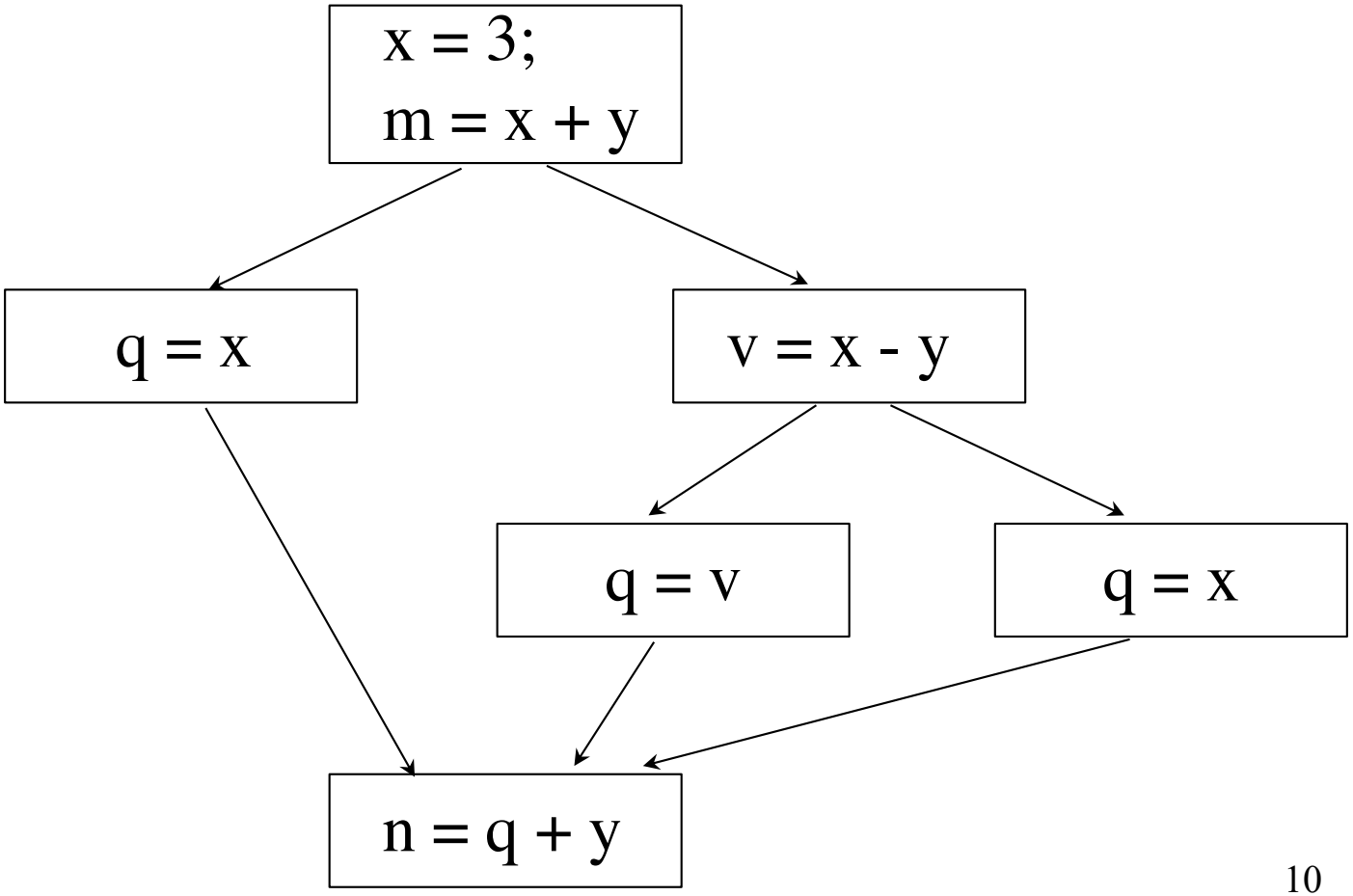
```
t ← 2 * y
z ← x - t
```

Types of Intermediate Representations (Details in Ch. 5 of CT book)

Three major categories

- 3: Hybrid
  - Combination of graphs and linear code

Example:
Control-flow graph



Redundancy Removal: A Deep Look

Redundancy Removal

- Replace redundant expressions by their values
- Example

```
m <-- 2 x y x z
n <-- 3 x y x z
o <-- 2 x y - z
```

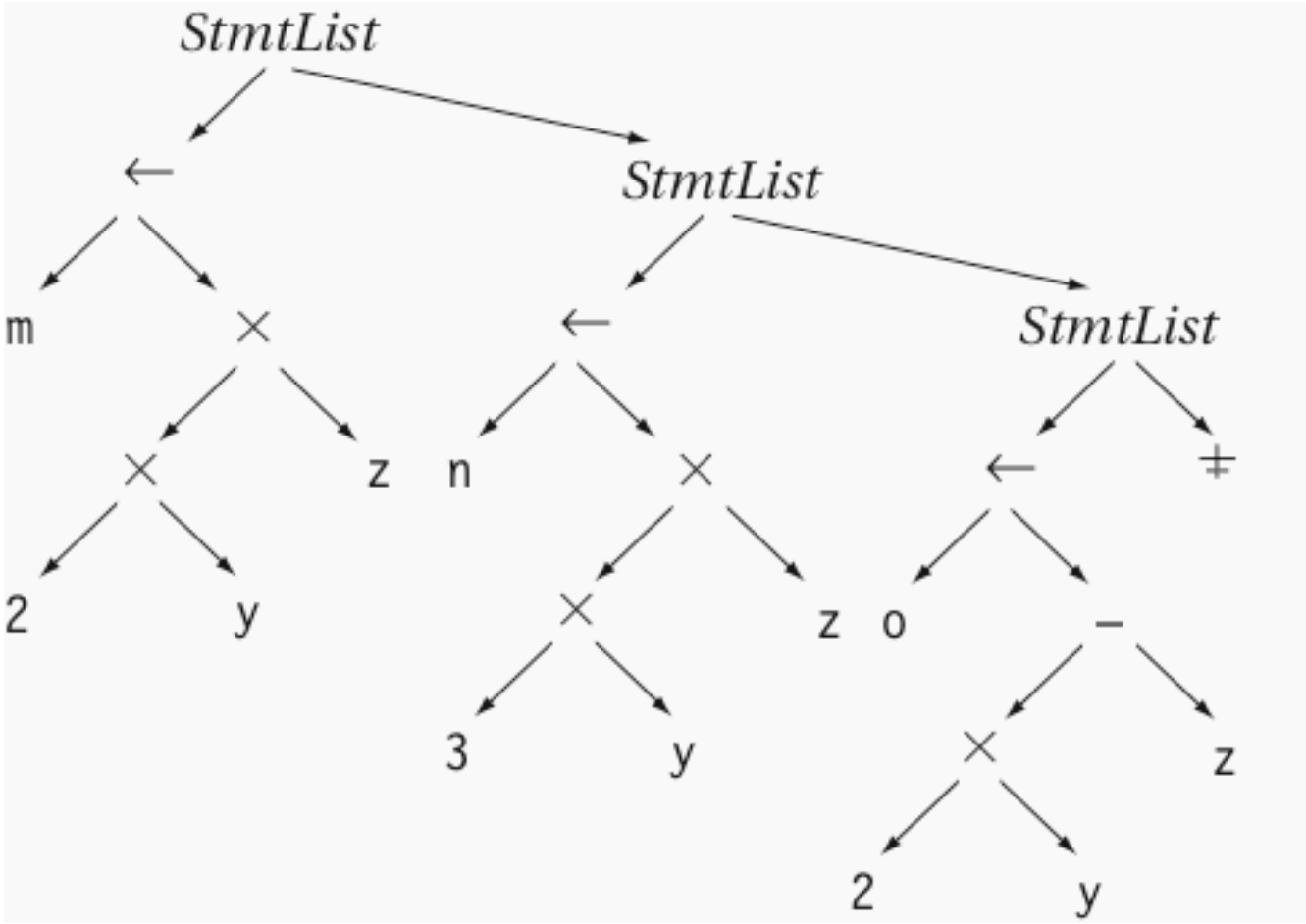
```
t <-- 2 x y
m <-- t x z

n <-- 3 x y x z

o <-- t - z
```

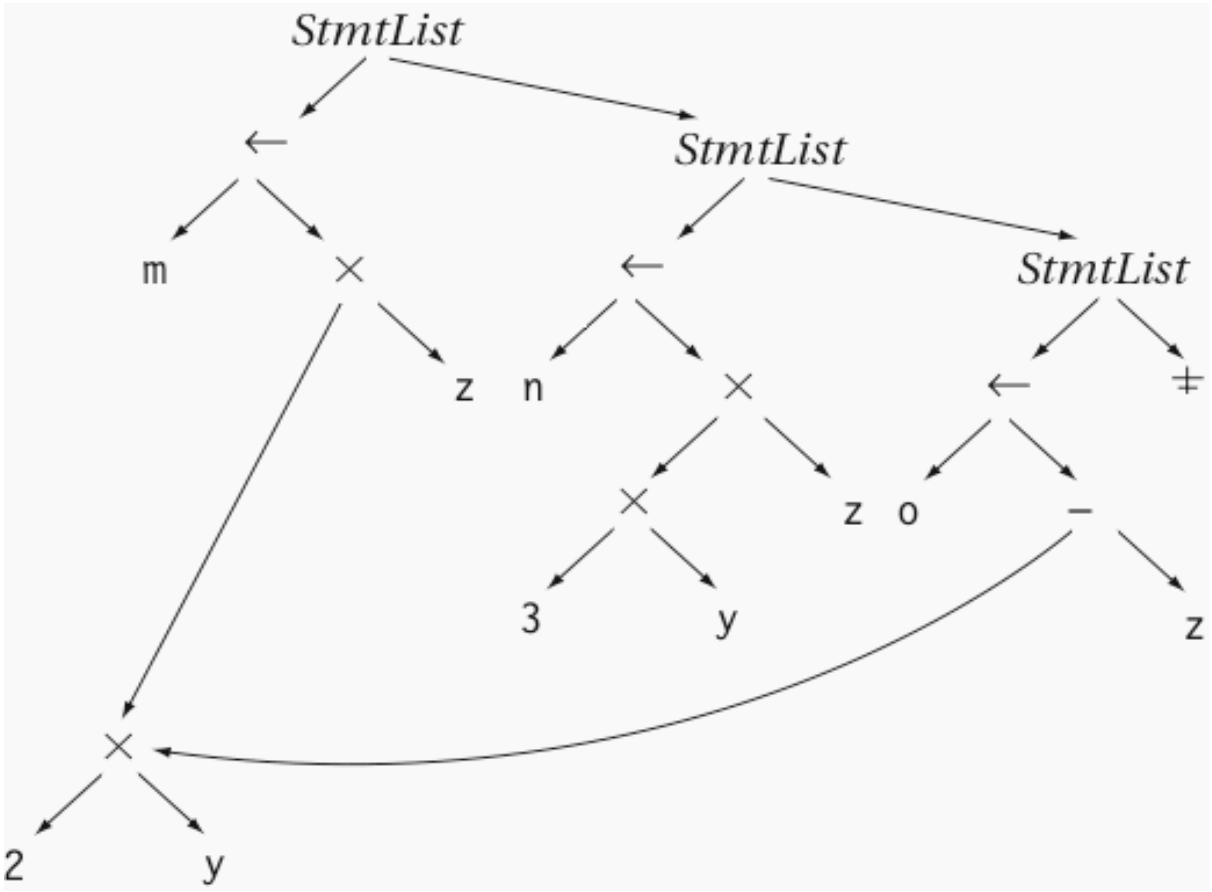
Method 1: Finding Redundancy in Directed Acyclic Graph (DAG)

```
m <-- 2 x y x z
n <--  3 x y x z
o <--  2 x y - z
```



A directed acyclic graph (DAG) is an AST with a unique node for each value.

DAG: Eliminating Identical Subtrees

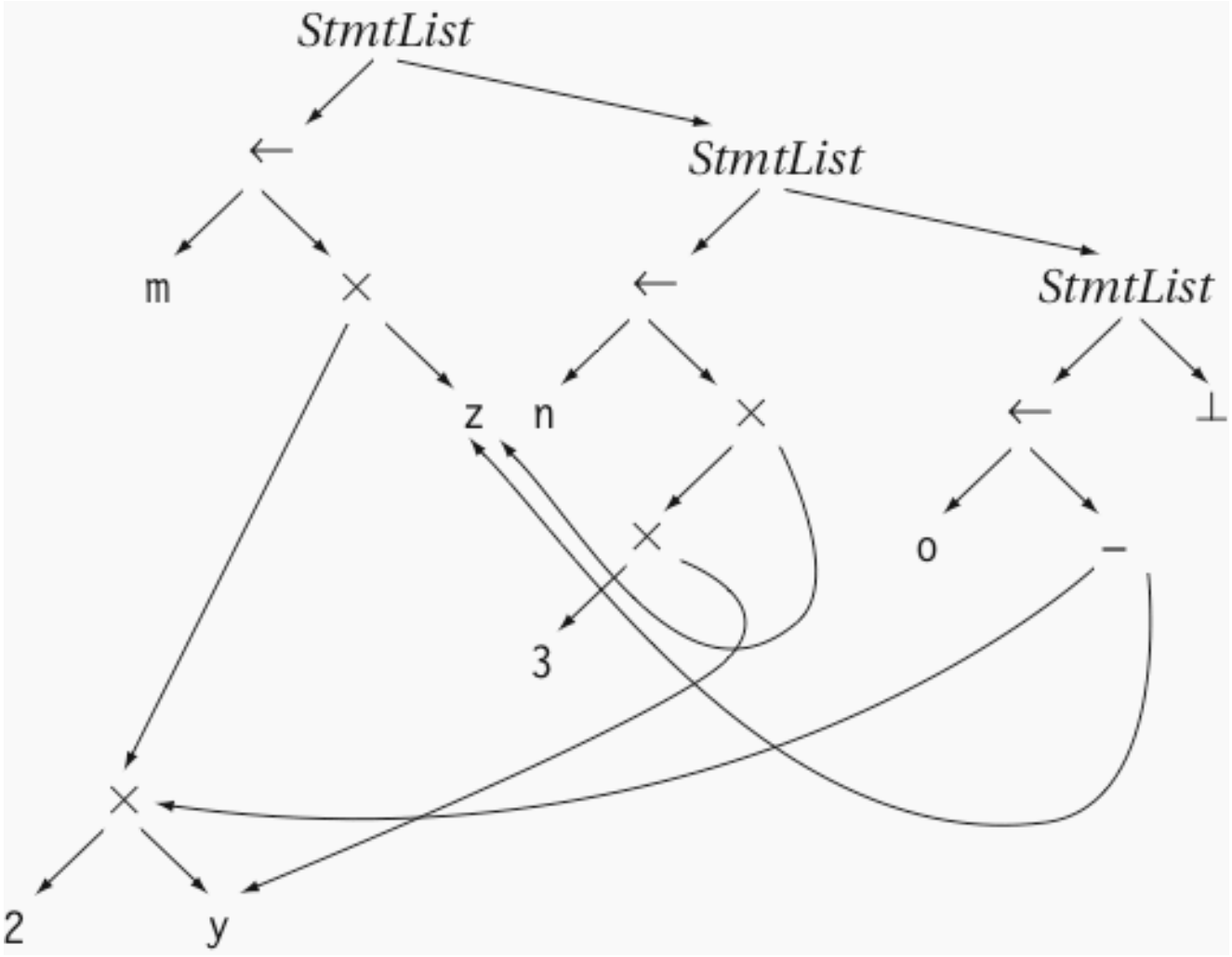



```
m <-- 2 x y x z
n <--  3 x y x z
o <--  2 x y - z
```

Using hash-based constructor:

- key is the textual notion of operators and operands;
- value is the node number or address.

Finding Redundancy in DAG




```
m <-- 2 x y x z
n <--  3 x y x z
o <--  2 x y - z
```

Using hash-based constructor:

- key is the textual notion of operators and operands;
- value is the node number or address.

Limitations of the method?

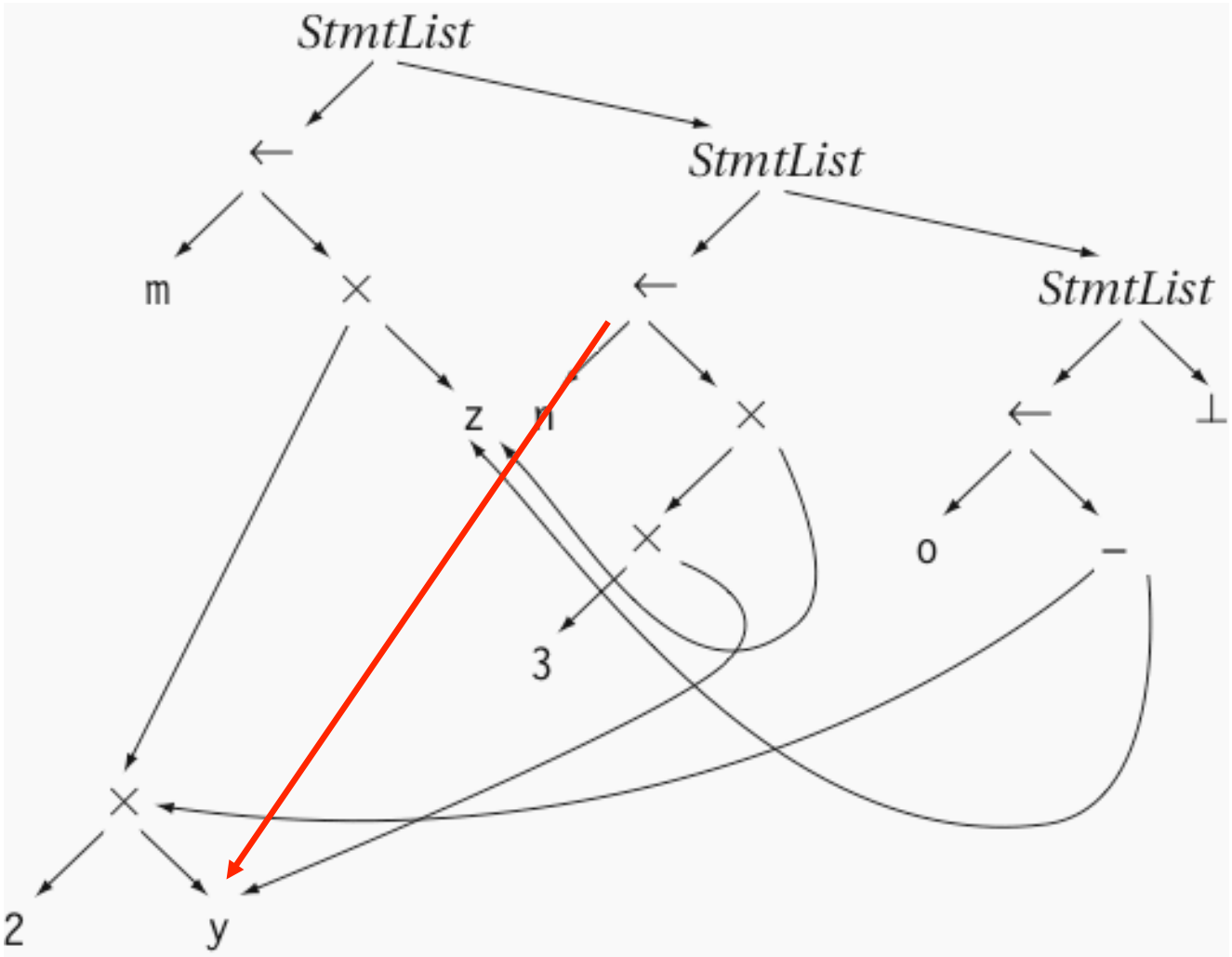
Original Code

```
m <-- 2 x y x z
y <--  3 x y x z
o <--  2 x y - z
```

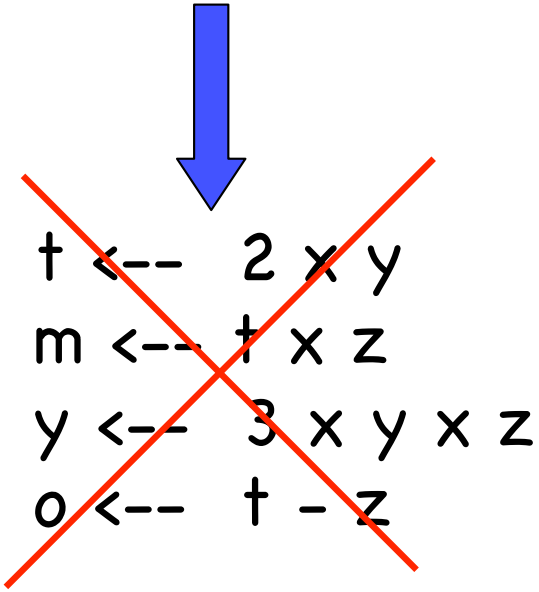
Using hash-based constructor:

- key is the textual notion of operators and operands;
- value is the node number or address.

Problem




```
m <-- 2 x y x z
y <-- 3 x y x z
o <-- 2 x y - z
```



```
t <-- 2 x y
m <-- t x z
y <-- 3 x y x z
o <-- t - z
```

Version Tracking

```
m <-- 2 x y x z
y <--  3 x y x z
o <--  2 x y - z
```

Associate a counter with each variable for version tracking

```
m1 <-- 2 x y1 x z1
y2 <--  3 x y1 x z1
o1 <--  2 x y2 - z1
```

Missing Any Opportunities?

Missing Any Opportunities?

Original Code

a ← x + y

z ← y

d ← 17

c ← x + z

- Is there any redundant expression?
- Can the DAG-based approach recognize it?

The hash-based constructor uses a textual notion of "equality",
- so y equals y, independent of its value.
- z does not equal y, independent of its value.

Value Numbering

- An old idea: Balke 1968 or Ershov 1954
  - Value instead of textual notation
- This technique was invented for low-level, linear IRs
- Equivalent methods exist for trees (build a DAG)

Linear Representation:

Linear IR Example: Three Address Code

- Statement form:

$$x \leftarrow y \ \underline{op} \ z$$

With 1 operator (op) and, at most, 3 names (x, y, z)

Example:

```
z ← x - 2 * y        becomes        t ← 2 * y
                                    z ← x - t
```

Statements are executed sequentially.

Local Value Numbering

Basic Algorithm

For each expression e

(assuming in the form $result_e$<-- o_1 op o_2)

- Get value numbers for operands from hash lookup table, represented by $VN(o_1)$, $VN(o_2)$. If no such key exists, insert these keys to the hash table with new value numbers;

- Search with hash key <op,$VN(o_1)$,$VN(o_2)$>,

  If no such key exists,

  1. insert the key to the hash table with a new value number;
  2. use the same value number for $result_e$ and insert them to hash table.

  else,

  get the value of this key, use the same value number for $result_e$, and insert them to hash table.

Local Value Numbering to Find Redundancy

An example

Original Code
$a \leftarrow x + y$
$z \leftarrow y$
$d \leftarrow 17$
$c \leftarrow x + z$

With VNs	Hash Table for VN
$a^3 \leftarrow x^1 + y^2$	{<x,1>, <y,2>, <<+,1,2>,3>, <a,3>}
$z^2 \leftarrow y^2$	{..., <z,2>}
$d^4 \leftarrow 17$	{..., <z,2>, <17,4>, <d,4>}
$c^3 \leftarrow x^1 + z^2$	{..., <z,2>, <17,4>, <d,4>, <c,3>}

Compare Value Num with Version Num. (The way they get increase; why the latter is insufficient for the example.

Rewritten for Redundancy Elimination

An example

Original Code

$a \leftarrow x + y$

$z \leftarrow y$

$d \leftarrow 17$

$c \leftarrow x + z$

With VNs	Hash Table for VN
$a^3 \leftarrow x^1 + y^2$	{<x,1>, <y,2>, <<+,1,2>,3>, <a,3>}
$z^2 \leftarrow y^2$	{..., <z,2>}
$d^4 \leftarrow 17$	{..., <z,2>, <17,4>, <d,4>}
$c^3 \leftarrow x^1 + z^2$	{..., <z,2>, <17,4>, <d,4>, <c,3>}

Rewritten

$a \leftarrow x + y$

$z \leftarrow y$

$d \leftarrow 17$

$c \leftarrow a$

Hash Table for Rewritten
{<1,x>, <2,y>, <3,a>}
{<1,x>, <2,y>, <3,a>}
{<1,x>, <2,y>, <3,a>, <4,17>}
{<1,x>, <2,y>, <3,a>, <4,17>}

Resolving Overwritten Issue

An example

Original Code	With VNs	Rewritten
$a \leftarrow x + y$	$a^3 \leftarrow x^1 + y^2$	$a \leftarrow x + y$
$z \leftarrow y$	$z^2 \leftarrow y^2$	$z \leftarrow y$
* $b \leftarrow x + y$	* $b^3 \leftarrow x^1 + y^2$	* $b \leftarrow a$
$a \leftarrow 17$	$a^4 \leftarrow 17$	$a \leftarrow 17$
* $c \leftarrow x + y$	* $c^3 \leftarrow x^1 + y^2$	* $c \leftarrow a$

Two redundancies marked by *

Hash Table for Rewritten

{<1,x>, <2,y>, <3,a>}

{<1,x>, <2,y>, <3,a>}

{<1,x>, <2,y>, <3,a>, <4,17>}

{<1,x>, <2,y>, <3,a>, <4,17>}

Optional Solutions:
- Use $c^3 \leftarrow b^3$
- Save a^3 in t^3
- **Rename around it (best)**

Renaming in Value Numbering

Renaming:
- Give each value a unique name

Example (continued)

Original Code	With VNs	Rewritten
$a_0 \leftarrow x_0 + y_0$	$a_0^3 \leftarrow x_0^1 + y_0^2$	$a_0 \leftarrow x_0 + y_0$
$z_0 \leftarrow y_0$	$z^2_0 \leftarrow y_0^2$	$z_0 \leftarrow y_0$
* $b_0 \leftarrow x_0 + y_0$	* $b_0^3 \leftarrow x_0^1 + y_0^2$	* $b_0 \leftarrow a_0$
$a_1 \leftarrow 17$	$a_1^4 \leftarrow 17$	$a_1 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$	* $c_0^3 \leftarrow x_0^1 + y_0^2$	* $c_0 \leftarrow a_0$

Hash Table for Rewritten

{<1,x_0>, <2,y_0>, <3,a_0>}

{<1,x_0 >, <2,y_0>, <3,a_0>}

{<1,x_0 >, <2,y_0>, <3,a_0>, <4,17>}

{<1,x_0 >, <2,y_0>, <3,a_0>, <4,17>}

Result:
- a_0 is available
- Rewriting just works

Reordering based on associativity

- The order in which expressions are written matters
- Example: either 2 x y or y x z will be missed in left- or right-associative treatment.

```
m <-- 2 x y x z
n <--  3 x y x z
o <--  2 x y - z
```

- Example: 3 x a x 5

A Hard Problem: Pointer Assignments

- *p = 17 could force every variable in a program to increase their version number.
- A major motivation for pointer analysis

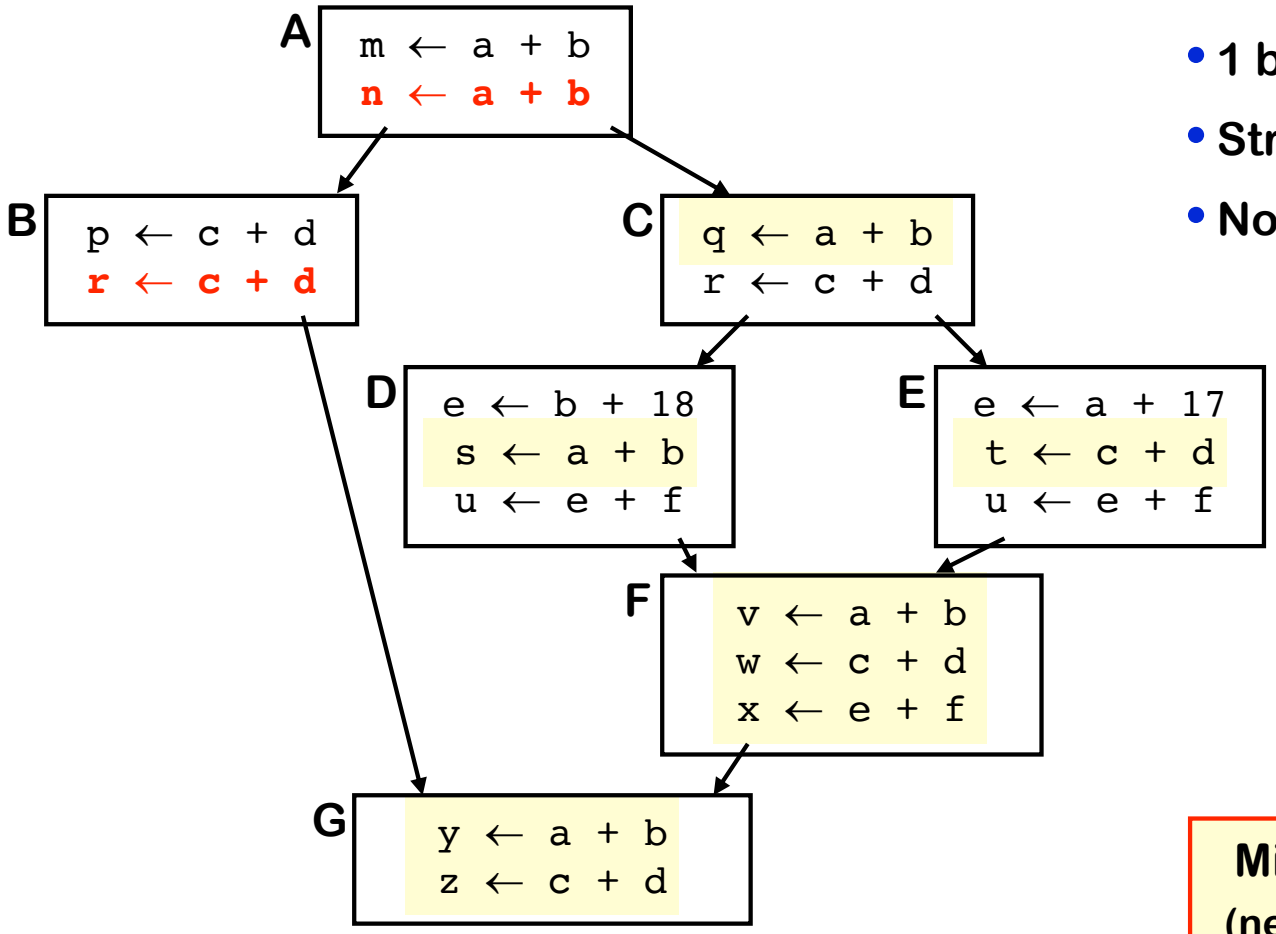
Original Code

a ← x + y
b ← w + v
*p ← 17
c ← x + y
d ← w + v

So far…

- Removing redundant expressions
  - DAG: version tracking
  - Linear representation: value numbering

Local Value Numbering <-> Linear IR



Local Value Numbering

- **1 block at a time**
- **Strong local results**
- **No cross-block effects**

Missed opportunities
(need stronger methods)