CSE110A: Compilers

May 16, 2022

Topics:

- *Basic blocks*
- *Local value numbering*

Announcements

- New grades:
  - HW 2 posted
  - Please let us know within 1 week if there are any issues!

- Pending grades
  - Midterm (expect by next Friday)

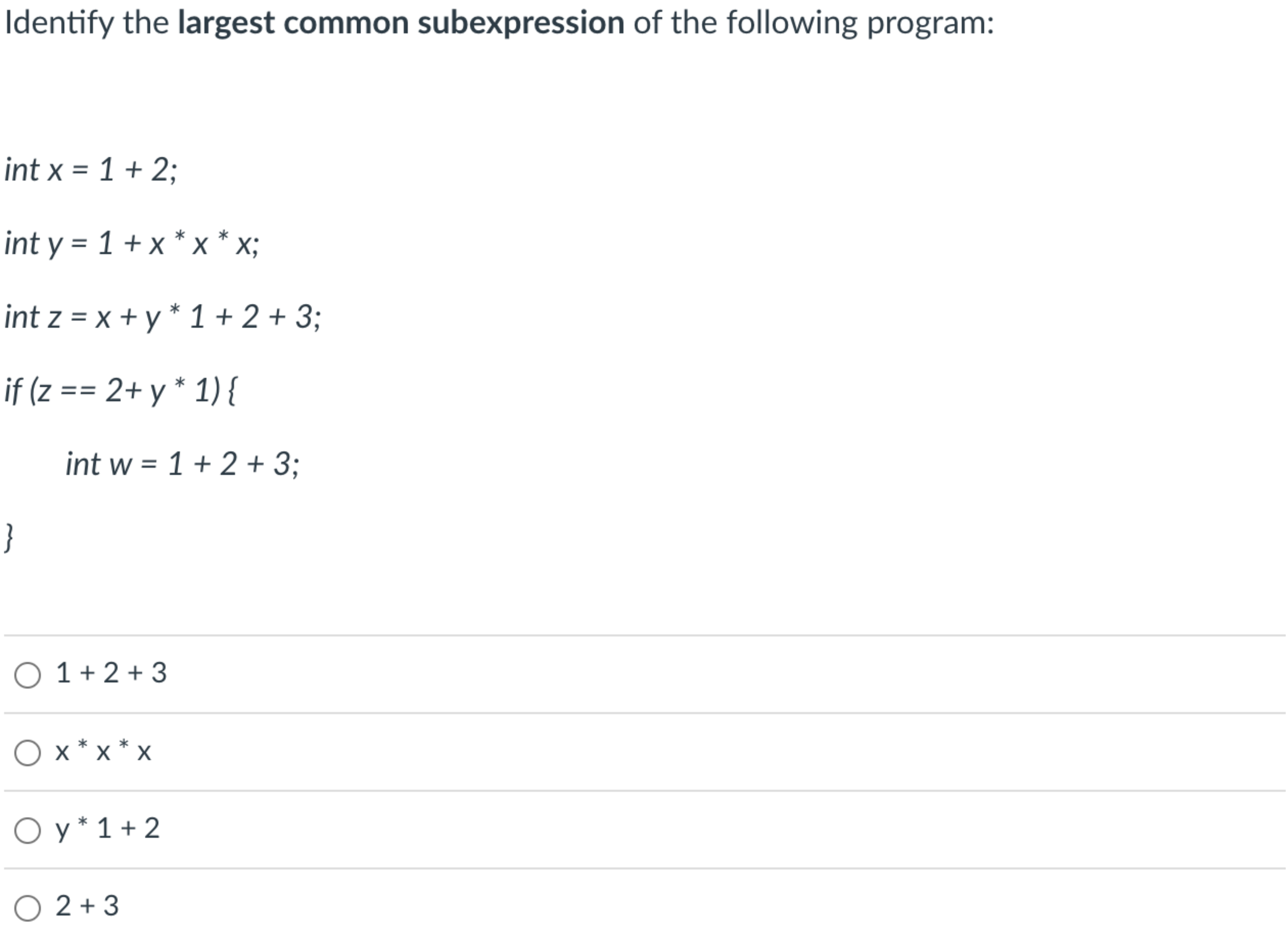
- HW 3 is released
  - Due in two weeks from release date
  - Get started early; you have all the material you need!
  - Packet updated (hopefully for the last time). Just updated the path to classir.h in ir_compiler.py.
  - Keep your eye on piazza for this assignment!

Announcements

- HW 4 should be released by May 23
  - This will give you 2 weeks to get it in before the final date (June 7)
  - You cannot turn this homework in after June 7
    - This is not my policy, it is the department policy!

Quiz

Quiz

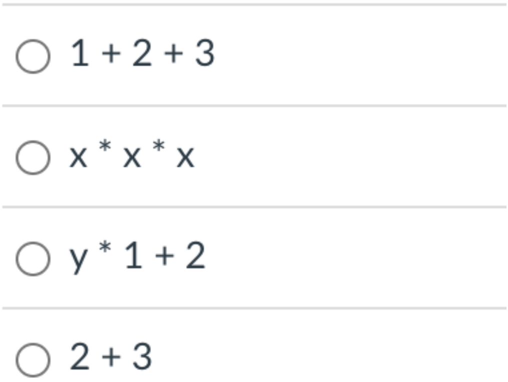


Identify the **largest common subexpression** of the following program:

```
int x = 1 + 2;

int y = 1 + x * x * x;

int z = x + y * 1 + 2 + 3;

if (z == 2+ y * 1) {

     int w = 1 + 2 + 3;

}
```

- ○ 1 + 2 + 3
- ○ x * x * x
- ○ y * 1 + 2
- ○ 2 + 3

Discussion



○ 1 + 2 + 3

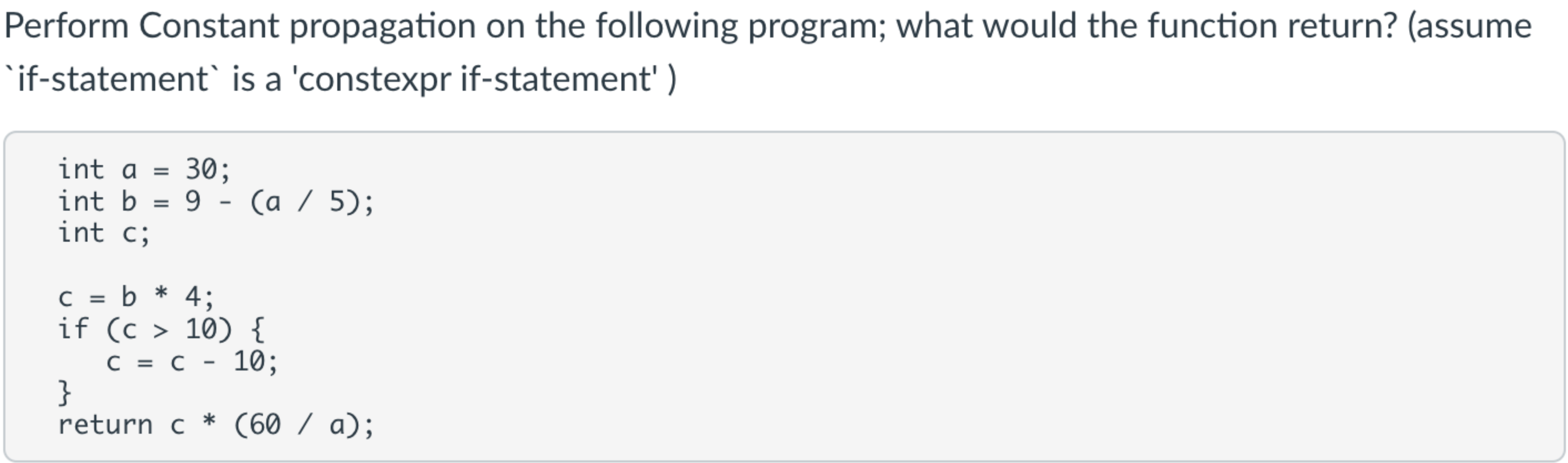
○ x * x * x

○ y * 1 + 2

○ 2 + 3

```
int x = 1 + 2;
int y = 1 + x * x * x;
int z = x + y * 1 + 2 + 3;
if (z == 2+ y * 1) {
        int w = 1 + 2 + 3;
}
```

Quiz



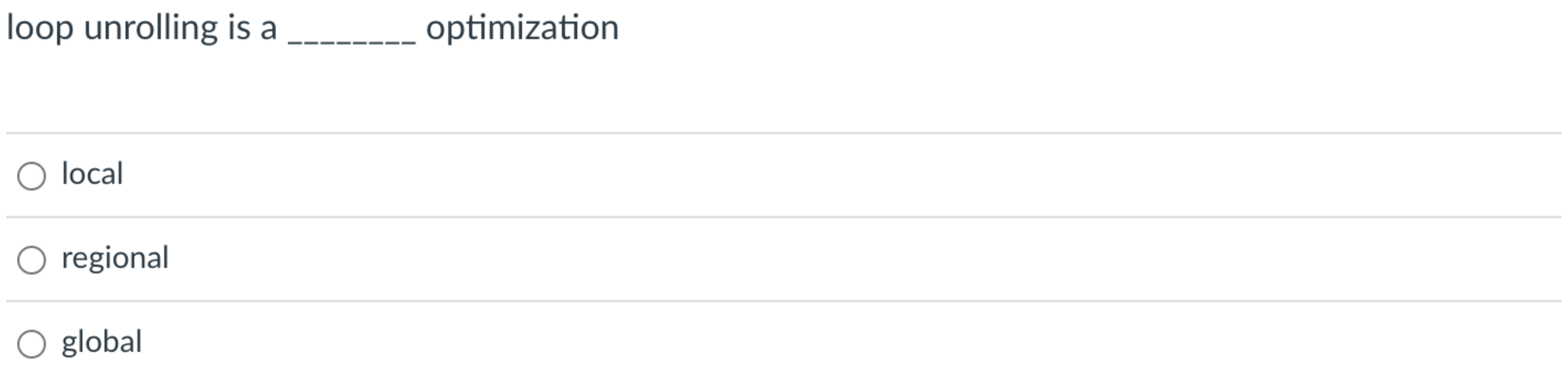
Perform Constant propagation on the following program; what would the function return? (assume `if-statement` is a 'constexpr if-statement')

```
int a = 30;
int b = 9 - (a / 5);
int c;

c = b * 4;
if (c > 10) {
   c = c - 10;
}
return c * (60 / a);
```

Discussion

```
int a = 30;
int b = 9 - (a / 5);
int c;
c = b * 4;
if (c > 10) { c = c - 10; }
return c * (60 / a);
```

Quiz



loop unrolling is a ________ optimization

○ local

○ regional

○ global

Optimization categories

- **local optimizations**: examine a ”basic block”, i.e. a small region of code with no control flow.
- **Regional optimizations**: several basic blocks with simple control flow.
- **Global optimization:** optimizes across an entire function

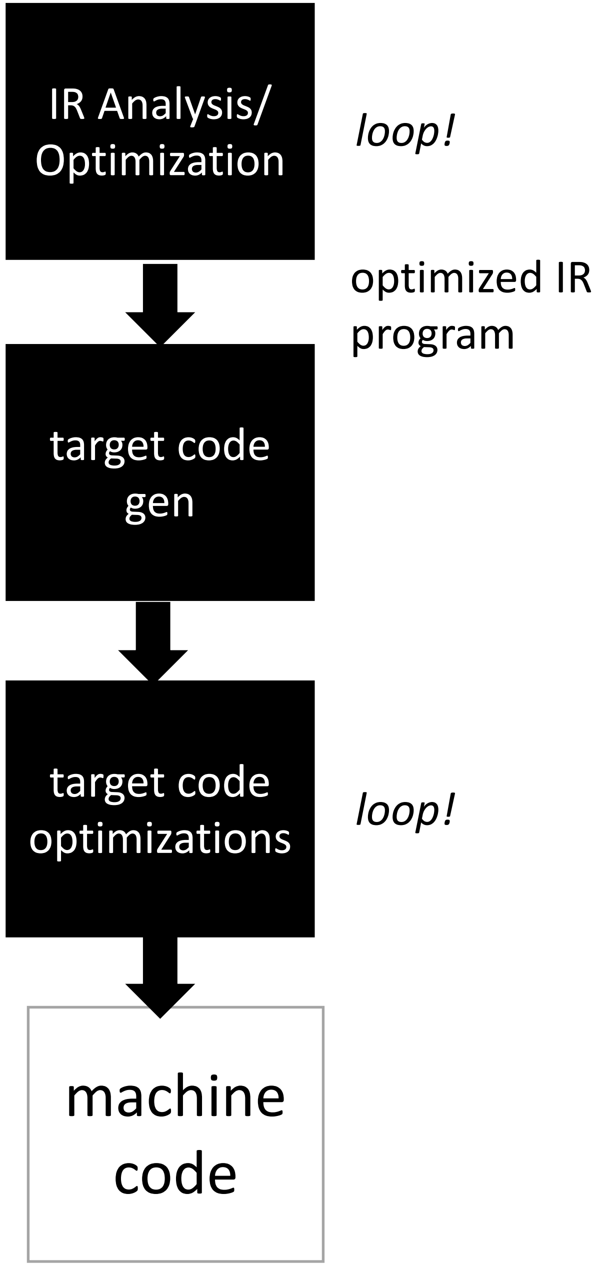
Implicit parse tree

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement
```

```
if (program0) {
   program1
}
else {
   program2
}
```

We have several structures to utilize to analyze and optimize programs!

IR programs



IR Analysis/ Optimization

loop!

optimized IR program

target code gen

target code optimizations

loop!

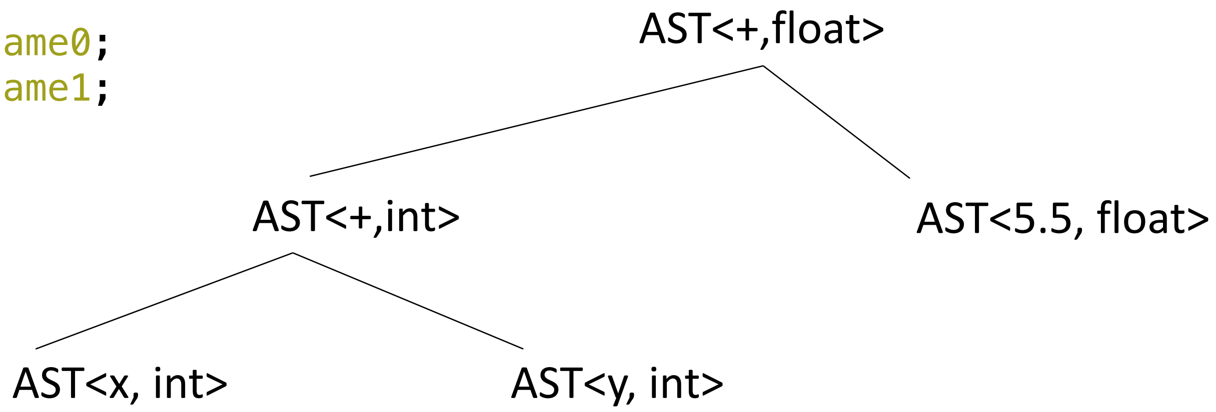
machine code

What IRs do we have at this point?

3 address code

```
virtual_reg vr3;
virtual_reg _new_name0;
virtual_reg _new_name1;
vr0 = int2vr(5);
_new_name0 = vr0;
vr1 = int2vr(6);
```

AST



Optimization categories

- **local optimizations**: examine a ”basic block”, i.e. a small region of code with no control flow.
- **Regional optimizations**: several basic blocks with simple control flow
- **Global optimization:** optimizes across an entire function

Discussion:

- What are the pros and cons of each?
- Why don’t we go further than functions?

Basic blocks

IR Program structure

- A sequence of 3 address instructions

- Programs can be split into **Basic Blocks:**
  - A sequence of 3 address instructions such that:
  - There is a single entry, single exit

- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

Single Basic Block

```
Label_x:
op1;
op2;
op3;
br label_z;
```

IR Program structure

- A sequence of 3 address instructions

- Programs can be split into **Basic Blocks:**
  - A sequence of 3 address instructions such that:
  - There is a single entry, single exit

- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

Single Basic Block

```
Label_x:
op1;
op2;
op3;
br label_z;
```

Two Basic Blocks

```
Label_x:
op1;
op2;
op3;

Label_y:
op4;
op5;
```

IR Program structure

How might they appear in a high-level language? What are some examples?

- A sequence of 3 address instructions
- Programs can be split into **Basic Blocks:**
  - A sequence of 3 address instructions such that:
  - There is a single entry, single exit
- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

Single Basic Block

```
Label_x:
op1;
op2;
op3;
br label_z;
```

Two Basic Blocks

```
Label_x:
op1;
op2;
op3;

Label_y:
op4;
op5;
```

IR Program structure

- A sequence of 3 address instructions

- Programs can be split into **Basic Blocks:**
  - A sequence of 3 address instructions such that:
  - There is a single entry, single exit

- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

How might they appear in a high-level language?

How many basic blocks?

```
…
if (x) {
    …
}
else {
   …
}
…
```

Single Basic Block

```
Label_x:
op1;
op2;
op3;
br label_z;
```

Two Basic Blocks

```
Label_x:
op1;
op2;
op3;

Label_y:
op4;
op5;
```

Converting 3 address code into basic blocks

- Let’s try an example: test 4 in HW 3:

Converting 3 address code into basic blocks

- Simple algorithm:
  - keep a list of basic blocks
  - a basic block is a list of instructions

  - Iterate over the 3 address instructions
  - if you see a branch or a label, finalize the current basic block and start a new one.
  - otherwise just add the current instruction to the current basic block

Converting 3 address code into basic blocks

pseudo code

```
basic_blocks = []
bb = []
for instr in program:
    if instr type is in [branch, label]:
        bb.append(instr)
        basic_blocks.append[bb]
        bb = []
    else:
        bb.append(instr)
```

Optimization levels

- **Local optimizations:**
    - Optimizes an individual basic block

- **Regional optimizations:**
    - Combines several basic blocks

- **Global optimizations:**
    - operates across an entire procedure

Optimization levels

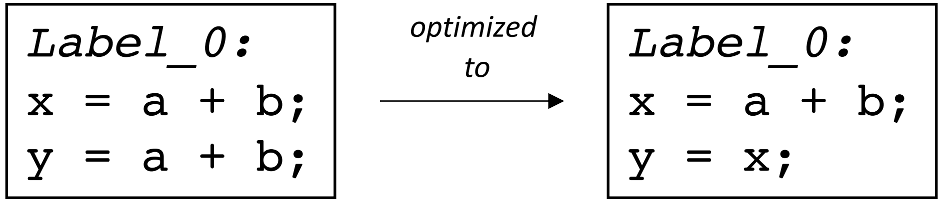
```
Label_0:
x = a + b;
y = a + b;
```

- **Local optimizations:**
  - Optimizes an individual basic block

- **Regional optimizations:**
  - Combines several basic blocks

- **Global optimizations:**
  - operates across an entire procedure

Optimization levels



```
Label_0:
x = a + b;
y = a + b;
```

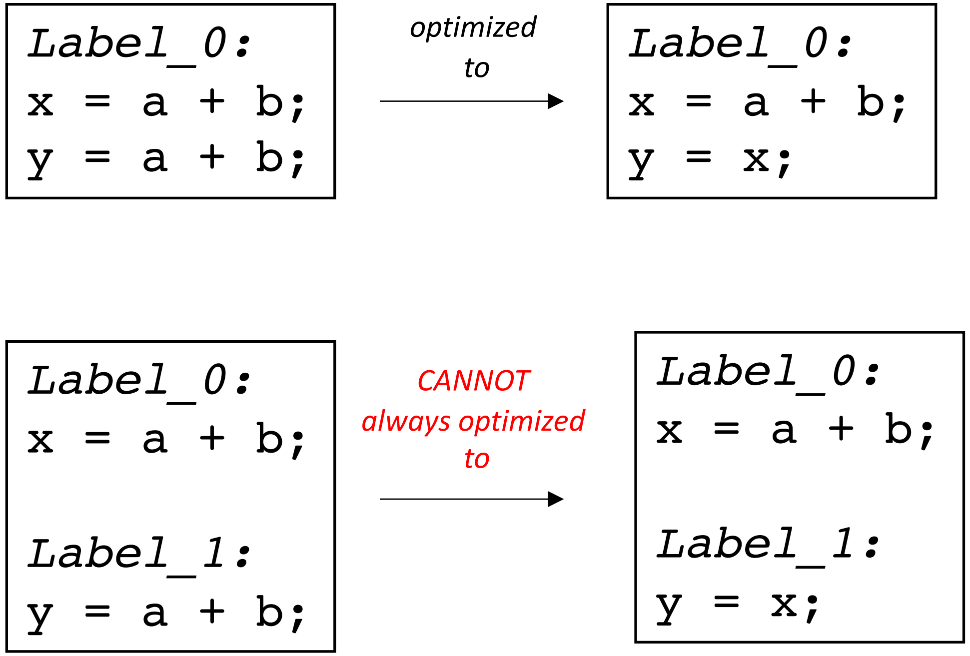
optimized to

```
Label_0:
x = a + b;
y = x;
```

- **Local optimizations:**
  - Optimizes an individual basic block
- **Regional optimizations:**
  - Combines several basic blocks
- **Global optimizations:**
  - operates across an entire procedure

Optimization levels

- **Local optimizations:**
  - Optimizes an individual basic block
- **Regional optimizations:**
  - Combines several basic blocks
- **Global optimizations:**
  - operates across an entire procedure



```
Label_0:
x = a + b;
y = a + b;
```

optimized to

```
Label_0:
x = a + b;
y = x;
```

```
Label_0:
x = a + b;

Label_1:
y = a + b;
```

CANNOT always optimized to

```
Label_0:
x = a + b;

Label_1:
y = x;
```

Optimization levels

- **Local optimizations:**
  - Optimizes an individual basic block

```
Label_0:
x = a + b;
y = a + b;
```

optimized to

```
Label_0:
x = a + b;
y = x;
```

- **Regional optimizations:**
  - Combines several basic blocks

```
Label_0:
x = a + b;

Label_1:
y = a + b;
```

CANNOT always optimized to

```
Label_0:
x = a + b;

Label_1:
y = x;
```

- **Global optimizations:**
  - operates across an entire procedure

code could skip Label_0, leaving x undefined!

```
br Label_1;

Label_0:
x = a + b;

Label_1:
y = a + b;
```

Regional Optimization

```
…
if (x) {
   …
}
else {
  x = a + b;
}
y = a + b;
…
```

we cannot replace:

y = a + b.

with

y = x;

Regional Optimization

```
…
if (x) {
   …
}
else {
  x = a + b;
}
y = a + b;
…
```

we cannot replace:

y = a + b.

with

y = x;

This requires regional analysis

```
x = a + b;
if (x) {
   …
}
else {
  …
}
y = a + b;
…
```

But in this case, we can check if a and b are not redefined, then

y = a + b;

can be replaced with

y = x;

Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis

Local value numbering

- A local optimization over 3 address code

- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)

- Can be extended to a regional optimization using flow analysis

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

valid? →

```
a = b + c;
b = a - d;
c = a;
d = a - d;
```

Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

valid? →

```
a = b + c;
b = a - d;
c = a;
d = a - d;
```

No! Because b is redefined

Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

valid? →

```
a = b + c;
b = a - d;
c = b + c;
d = b;
```

Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

valid? →

```
a = b + c;
b = a - d;
c = b + c;
d = b;
```

yes!

Local value numbering

Algorithm:

- Provide a number to each variable. Update the number each time the variable is updated.
- Keep a global counter; increment with new variables or assignments

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

Global_counter = 0

Local value numbering

Algorithm:

- Provide a number to each variable. Update the number each time the variable is updated.

- Keep a global counter; increment with new variables or assignments

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

Global_counter = 7

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
→ a2 = b0 + c1;
  b4 = a2 - d3;
  c5 = b4 + c1;
  d6 = a2 - d3;
```

```
H = {
        “b0 + c1” : “a2”,
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
   a2 = b0 + c1;
→  b4 = a2 - d3;
   c5 = b4 + c1;
   d6 = a2 - d3;
```

```
H = {
      “b0 + c1” : “a2”,
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
  a2 = b0 + c1;
→ b4 = a2 - d3;
  c5 = b4 + c1;
  d6 = a2 - d3;
```

```
H = {
        “b0 + c1” : “a2”,
        “a2 - d3” : ”b4”,
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
  a2 = b0 + c1;
  b4 = a2 - d3;
→ c5 = b4 + c1;
  d6 = a2 - d3;
```

```
H = {
        “b0 + c1” : “a2”,
        “a2 - d3” : ″b4″,
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
   a2 = b0 + c1;
   b4 = a2 - d3;
→  c5 = b4 + c1;
   d6 = a2 - d3;
```

```
H = {
      “b0 + c1” : “a2”,
      “a2 - d3” : ”b4”,
}
```

mismatch due to numberings!

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
   a2 = b0 + c1;
   b4 = a2 - d3;
→  c5 = b4 + c1;
   d6 = a2 - d3;
```

```
H = {
      “b0 + c1” : “a2”,
      “a2 - d3” : ”b4”,
      “b4 + c1” : “c5”,
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
        “b0 + c1” : “a2”,
        “a2 – d3” : ”b4”,
        “b4 + c1” : “c5”,
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = b4;
```

```
H = {
        “b0 + c1” : “a2”,
        “a2 - d3” : ”b4”,
        “b4 + c1” : “c5”,
}
```

match!

What else can we do?

What else can we do?

Consider this snippet:

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

Commutative operations

What is the definition of commutative?

Commutative operations

What is the definition of commutative?

```
x OP y == y OP x
```

What operators are commutative? Which ones are not?

Adding commutativity to local value numbering

- For commutative operators (e.g. + *), the analysis should consider a deterministic order of operands.
- You can use variable numbers or lexigraphical order

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

cannot re-order because - is not commutative

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
        “c1 - b0” : “a2”,
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
  a2 = c1 - b0;
→ f4 = d3 * a2;
  c5 = b0 - c1;
  d6 = a2 * d3;
```

```
H = {
        “c1 - b0” : “a2”,
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

re-ordered because a2 < d3 lexigraphically

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
        “c1 - b0” : “a2”,
        ”a2 * d3” : “f4”,
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
  a2 = c1 - b0;
  f4 = d3 * a2;
→ c5 = b0 - c1;
  d6 = a2 * d3;
```

```
H = {
        “c1 - b0” : “a2”,
        ”a2 * d3” : “f4”,
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
   a2 = c1 - b0;
   f4 = d3 * a2;
→  c5 = b0 - c1;
   d6 = a2 * d3;
```

```
H = {
       “c1 - b0” : “a2”,
       ”a2 * d3” : “f4”,
       ”b0 - c1” : “c5”,
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
   a2 = c1 - b0;
   f4 = d3 * a2;
   c5 = b0 - c1;
→  d6 = a2 * d3;
```

```
H = {
        “c1 - b0” : “a2”,
        ”a2 * d3” : “f4”,
        ”b0 - c1” : “c5”,
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = f4;
```

```
H = {
        “c1 - b0” : “a2”,
        ”a2 * d3” : “f4”,
        ”b0 - c1” : “c5”,
}
```

Other considerations?

Local value numbering w/out adding registers

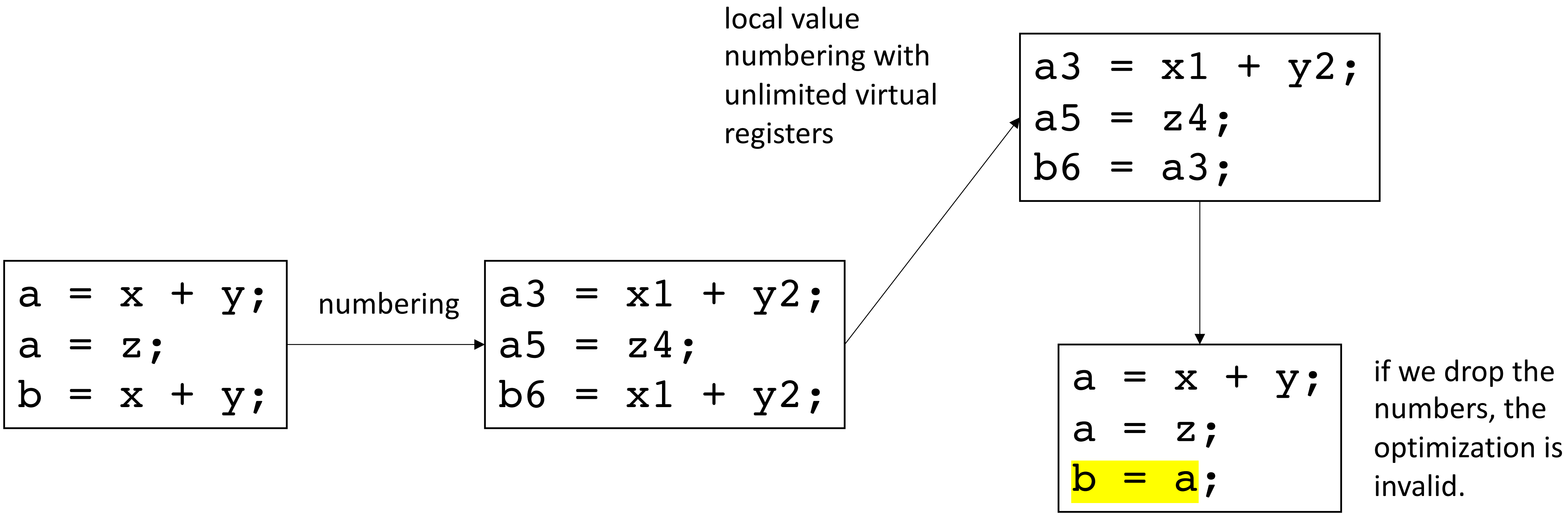
- We’ve assumed we have access to an unlimited number of virtual registers.

- In some cases we may not be able to add virtual registers
  - If an expensive register allocation pass has already occurred.

- New constraint:
  - We need to produce a program such that variables without the numbers is still valid.

Local value numbering w/out adding registers

- Example:



```
a = x + y;
a = z;
b = x + y;
```

numbering

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
```

local value numbering with unlimited virtual registers

```
a3 = x1 + y2;
a5 = z4;
b6 = a3;
```

```
a = x + y;
a = z;
b = a;
```

if we drop the numbers, the optimization is invalid.

Local value numbering w/out adding registers

- Solutions?

```
a = x + y;
a = z;
b = x + y;
```

numbering

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a = x + y;
a = z;
b = x + y;
c = x + y;
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a = x + y;
a = z;
b = x + y;
c = x + y;
```

We cannot optimize the first line, but we can optimize the second

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a = x + y;
a = z;
b = x + y;
c = x + y;
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a = x + y;
a = z;
b = x + y;
c = x + y;
```

First we number

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {
}
```

```
→  a3 = x1 + y2;
   a5 = z4;
   b6 = x1 + y2;
   c7 = x1 + y2;
```

```
H = {
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
→   a3 = x1 + y2;
    a5 = z4;
    b6 = x1 + y2;
    c7 = x1 + y2;
```

```
Current_val = {
                  ″a″ : 3,
}


H = {
       ″x1 + y2″ : ″a3″,
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

```
Current_val = {
                  ”a” : 3,
}
```

```
H = {
       ”x1 + y2” : ”a3”,
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {
                    ″a″ : 5,
}
```

```
H = {
        ″x1 + y2″ : ″a3″,
}
```

```
   a3 = x1 + y2;
→  a5 = z4;
   b6 = x1 + y2;
   c7 = x1 + y2;
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {
                   ″a″ : 5,
}
```

```
H = {
       ″x1 + y2″ : ″a3″,
}
```

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
  a3 = x1 + y2;
  a5 = z4;
→ b6 = x1 + y2;
  c7 = x1 + y2;
```

```
Current_val = {
                 "a" : 5,
}


H = {
      "x1 + y2" : "a3",
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {
                  ”a” : 5,
                  ”b” : 6
}
```

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

```
H = {
       ”x1 + y2” : ”b6”,
}
```

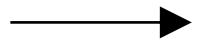
Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {
                  ″a″ : 5,
                  ″b″ : 6
}
```

```
H = {
       ″x1 + y2″ : ″b6″,
}
```

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```



Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
  a3 = x1 + y2;
  a5 = z4;
  b6 = x1 + y2;
→ c7 = x1 + y2;
```

```
Current_val = {
                 ″a″ : 5,
                 ″b″ : 6
}

H = {
       ″x1 + y2″ : ″b6″,
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {
                    ″a″ : 5,
                    ″b″ : 6
}
```

```
H = {
        ″x1 + y2″ : ″b6″,
}
```

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = b6;
```

Anything else we can add to local value numbering?

Anything else we can add to local value numbering?

- Final heuristic: keep sets of possible values

Local value numbering: value sets

- Final heuristic: keep sets of possible values

```
Current_val = {
}
```

```
a = x + y;
b = x + y;
a = z;
c = x + y;
```

```
H = {
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

```
Current_val = {
}
```

```
a3 = x1 + y2;
b4 = x1 + y2;
a6 = z5;
c7 = x1 + y2;
```

```
H = {
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

```
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = x1 + y2;
```

```
Current_val = {
                   “a” : 6,
                   “b” : 4
}
```

```
H = {
   “x1 + y2” : “a3”
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

```
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = x1 + y2;
```

```
Current_val = {
                 "a" : 6,
                 "b" : 4
}
```

```
H = {
    "x1 + y2" : "a3"
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

```
Current_val = {
                  "a" : 6,
                  "b" : 4
}
```

```
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = x1 + y2;
```

```
H = {
    "x1 + y2" : "a3"
}
```

but we could have replaced it with b4!

Local value numbering: value sets

- Final heuristic: keep sets of possible values

rewind to this point →

```
a3 = x1 + y2;
b4 = x1 + y2;
a6 = z5;
c7 = x1 + y2;
```

```
Current_val = {
                  “a” : 3,
}
```

```
H = {
   “x1 + y2” : “a3”
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

```
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = x1 + y2;
```

```
Current_val = {
                  “a” : 3,
                  ”b” : 4
}
```

```
H = {
   “x1 + y2” : [“a3”, “b4”],
}
```

hash a list of possible values

Local value numbering: value sets

- Final heuristic: keep sets of possible values

```
Current_val = {
                 “a” : 6,
                 “b” : 4
}
```

```
H = {
    “x1 + y2” : [“a3”, ”b4”],
}
```

fast forward again

```
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = x1 + y2;
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

fast forward again →

```
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = b4;
```

```
Current_val = {
                  “a” : 6,
                  “b” : 4
}


H = {
   “x1 + y2” : [“a3”, ”b4”],
}
```

Local value numbering: Memory

- Consider a 3 address code that allows memory accesses

```
a[i] = x[j] + y[k];
b[i] = x[j] + y[k];
```

is this transformation allowed?
No!

only if the compiler can prove that `a` does not alias `x` and `y`

```
a[i] = x[j] + y[k];
b[i] = a[i];
```

In the worst case, every time a memory location is updated, the compiler must update the value for all pointers.

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair

```
(a[i],3) = (x[j],1) + (y[k],2);
b[i] = x[j] + y[k];
```

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

compiler analysis:

can we trace `a,x,y` to
`a = malloc(…);`
`x = malloc(…);`
`y = malloc(…);`

// `a,x,y` are never overwritten

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],1) + (y[k],2);
```

in this case we do not have to update the number

compiler analysis:

can we trace `a,x,y` to

```
a = malloc(…);
x = malloc(…);
y = malloc(…);
```

// `a,x,y` are never overwritten

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

programmer annotations can also tell
the compiler that no other pointer
can access the memory pointed to by a

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

in this case we do not have to update the number

`restrict a`

programmer annotations can also tell the compiler that no other pointer can access the memory pointed to by a

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (a[i],3);
```

Optimizing over wider regions

- Local value numbering operated over just one basic block.
- We want optimizations that operate over several basic blocks (a region), or across an entire procedure (global)
- For this, we need Control Flow Graphs and Flow Analysis
  - We may have time to discuss this later in the module

See everyone on Wednesday

- Topics: Loop unrolling