CSE110A: Compilers

May 24, 2023

Topics:

- *Basic Blocks*
- *Local Value numbering*

Announcements

- HW 4 is due on Monday
  - No guaranteed help on weekends or holidays
- Working on grading HW 2 and HW 3: grades should be out soon.
- HW 5 is planned to be released on Monday

Quiz review

Quiz review

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

Quiz review

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);
```

Quiz review

loop unrolling is a _____ optimization

- ◯ local
- ◯ regional
- ◯ global

Optimization categories

Next category level is how much code we need to reason about for the optimization.

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

Quiz review

Describe some compiler optimizations you know of. Write one (or more) small example program on Godbolt and look at the llvm IR (using -emit-llvm on a clang compiler) or ISA code. You can also play with optimization flags (-O0, -O3, etc). Did the compiler do the optimization you thought of?

Describe your program and the optimization below. Feel free to share your experiment on piazza!

New material

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 {
  …
}
…
```

Two Basic Blocks

```
Label_x:
op1;
op2;
op3;

Label_y:
op4;
op5;
```

Single Basic Block

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

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 (3 address code)

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

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

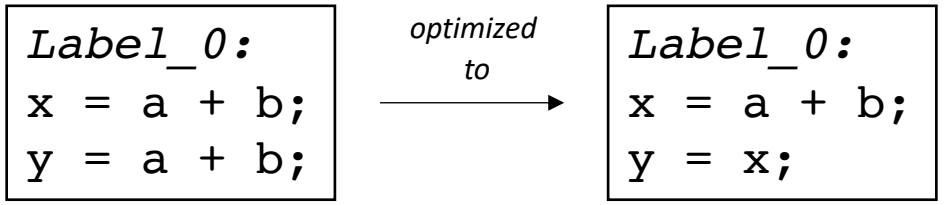
- **Local optimizations:**
  - Optimizes an individual basic block
- **Regional optimizations:**
  - Combines several basic blocks
- **Global optimizations:**
  - operates across an entire procedure
  - what about across procedures?

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
  - what about across procedures?

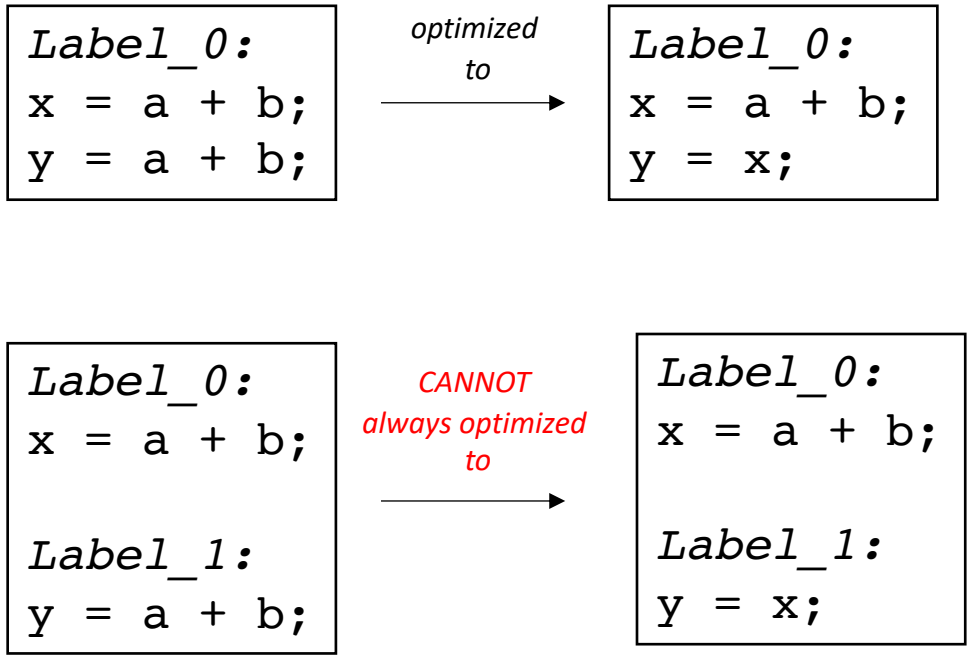
Optimization levels



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

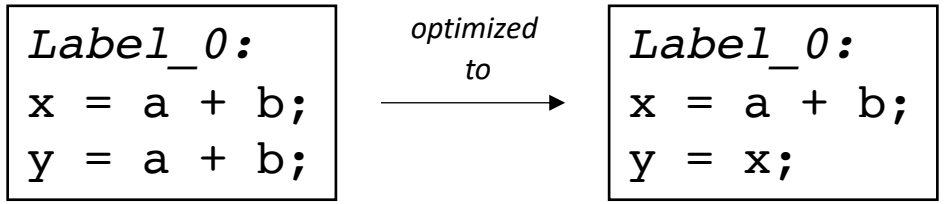
Optimization levels

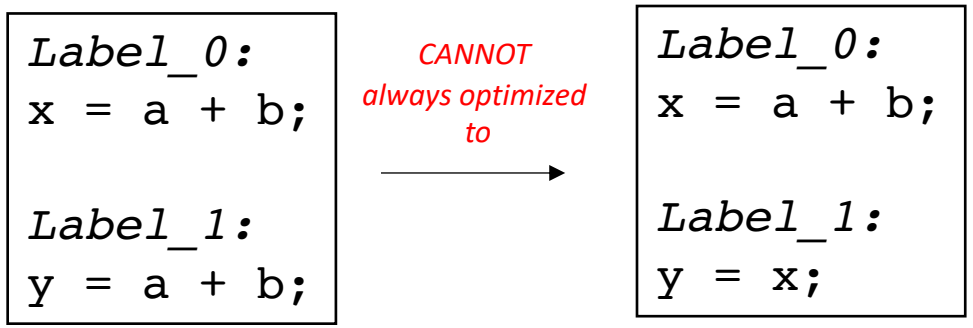
- **Local optimizations:**
  - Optimizes an individual basic block
- **Regional optimizations:**
  - Combines several basic blocks
- **Global optimizations:**
  - operates across an entire procedure
  - what about across procedures?



Optimization levels

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





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;

```
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;

This requires regional analysis and optimizations

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

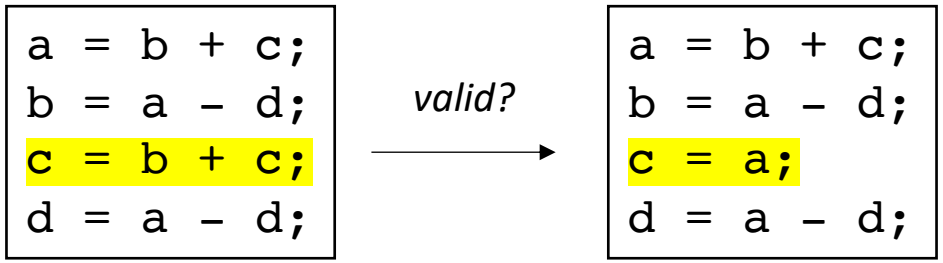
Local value numbering

- 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;
```

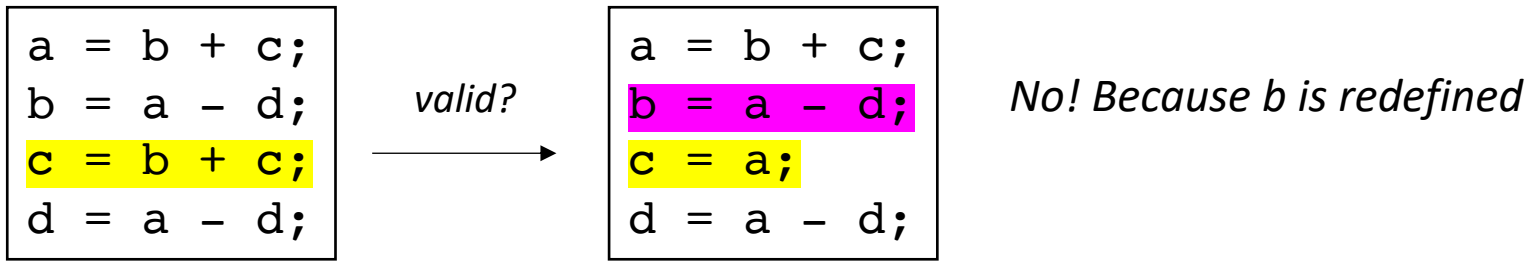
Local value numbering

- 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



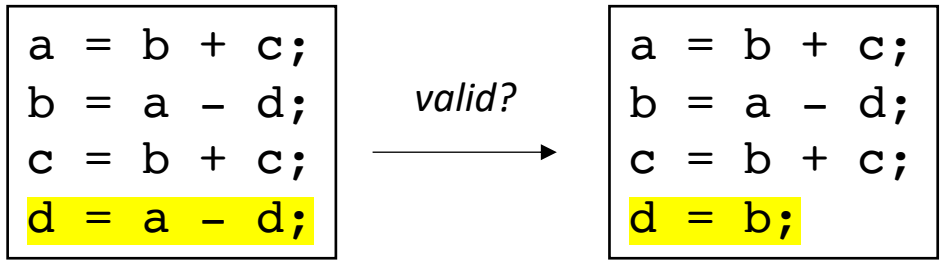
Local value numbering

- 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



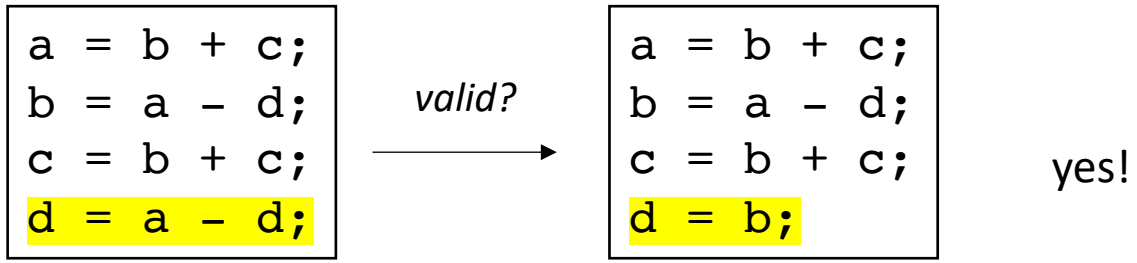
Local value numbering

- 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



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

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.

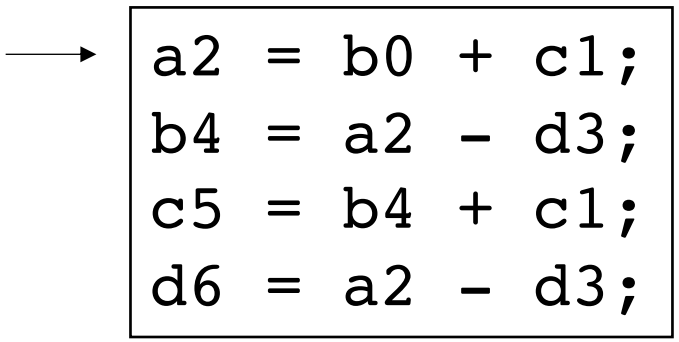
```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

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;
```

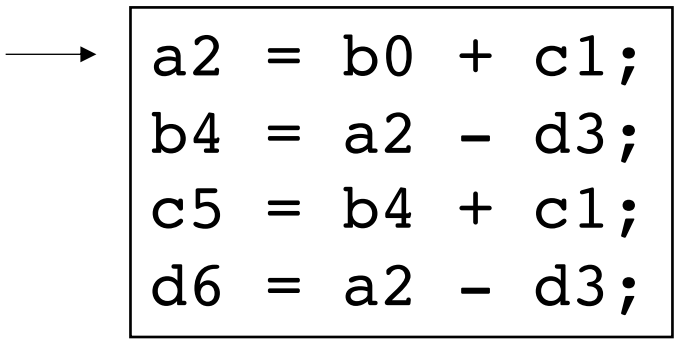
```
H = {
}
```

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;
```

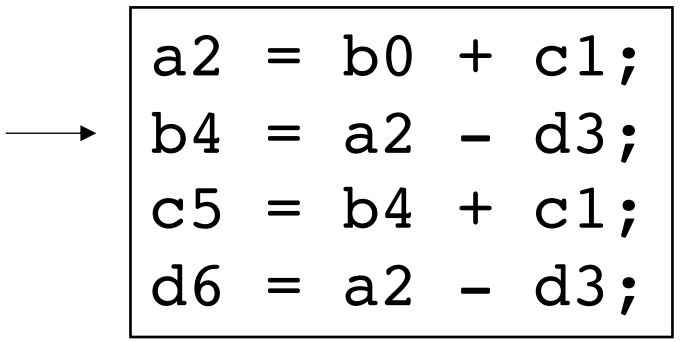
```
H = {
        “b0 + c1” : “a2”,
}
```

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.



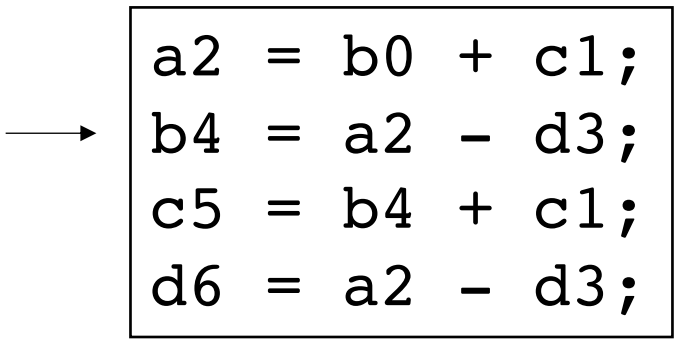

```
H = {
        “b0 + c1” : “a2”,
}
```

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.



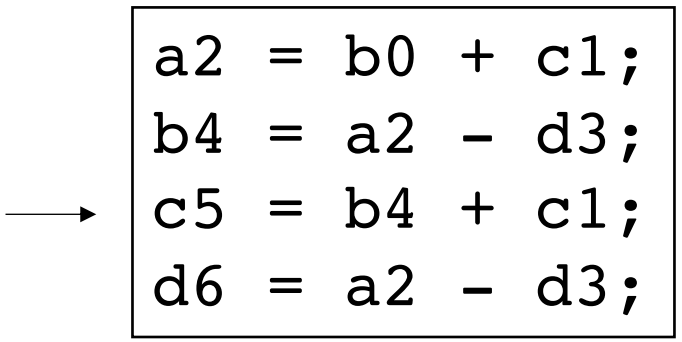

```
H = {
        “b0 + c1” : “a2”,
        “a2 - d3” : ”b4”,
}
```

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.




```
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",
}
```

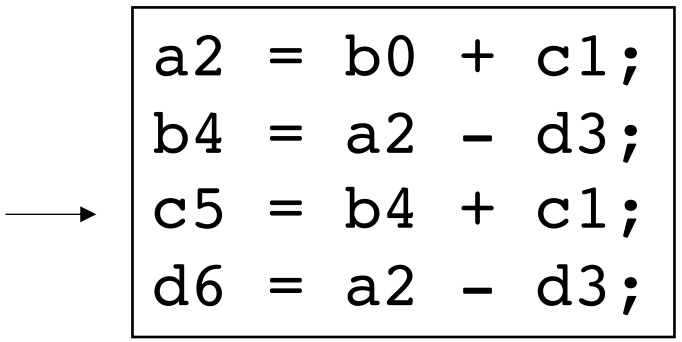
mismatch due to numberings!

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.




```
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

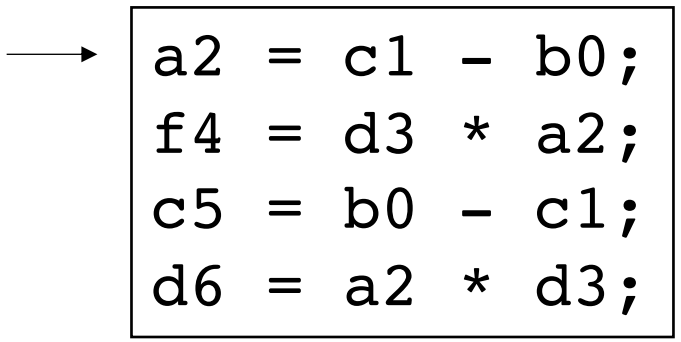
- For commutative operators (e.g. + *), the analysis should consider a deterministic order of operands.

- 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

→

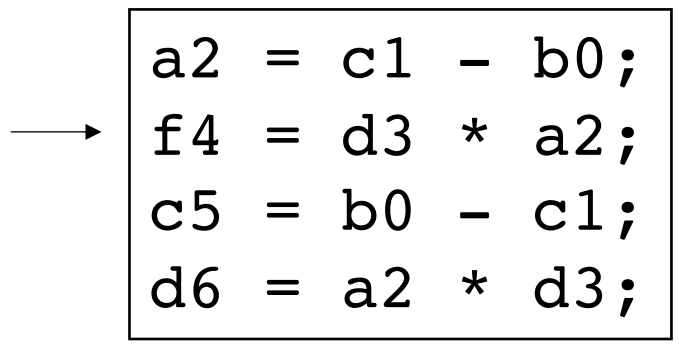
```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

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

Local value numbering: commutative operations

Algorithm optimization:

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




```
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

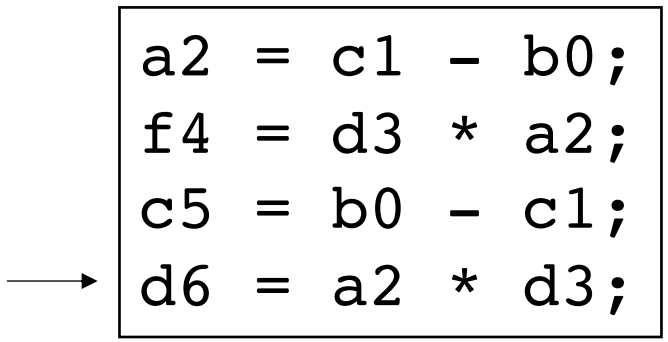
```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
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




```
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

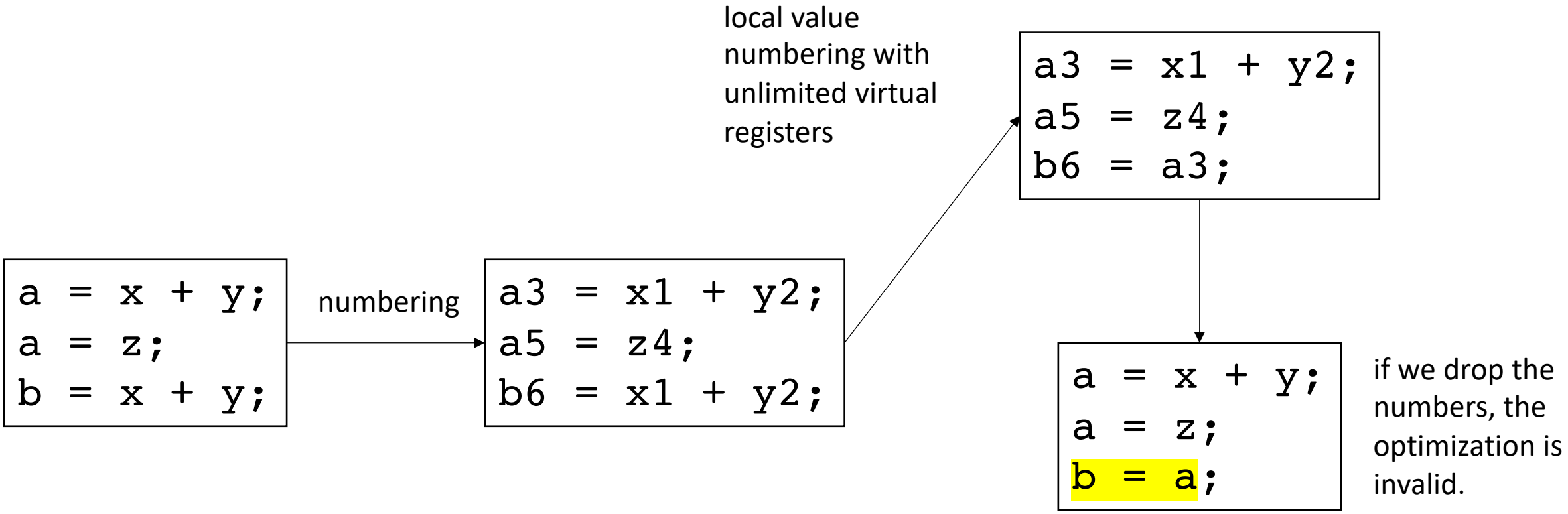
- We've assumed we have access to an unlimited number of virtual registers.

- 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.

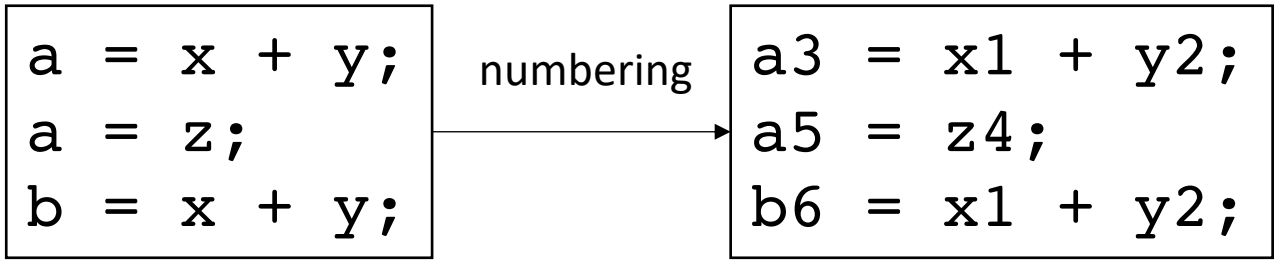
Local value numbering w/out adding registers

- Example:



Local value numbering w/out adding registers

- Solutions?



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

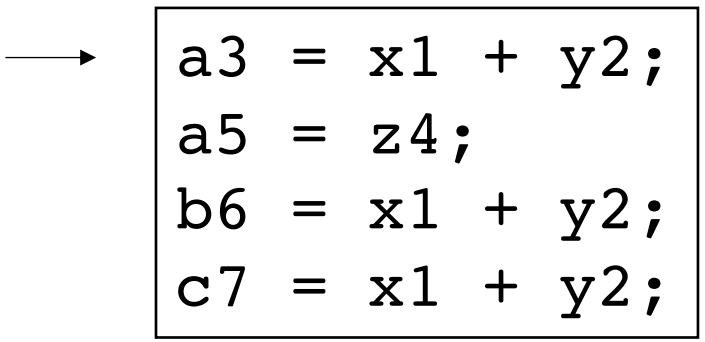
Local value numbering w/out adding registers

- 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

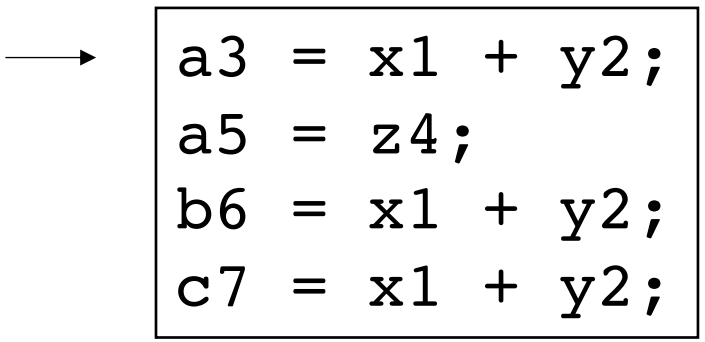



```
Current_val = {
}
```

```
H = {
}
```

Local value numbering w/out adding registers

- Keep another hash table to keep the current variable number



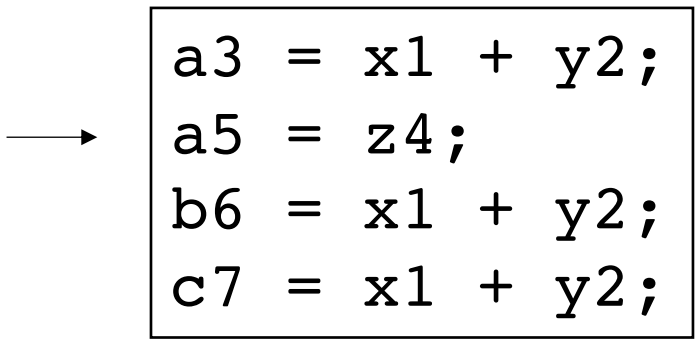
```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

```
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



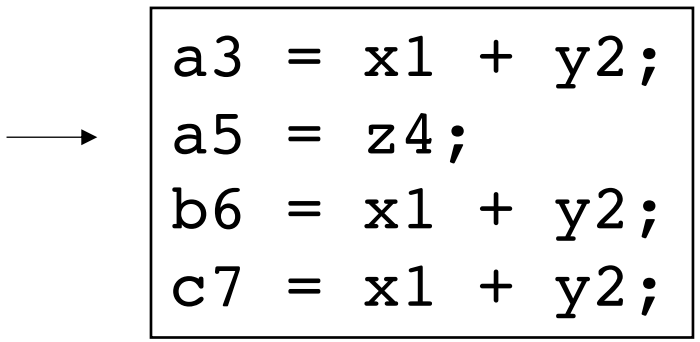
```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

```
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



```
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

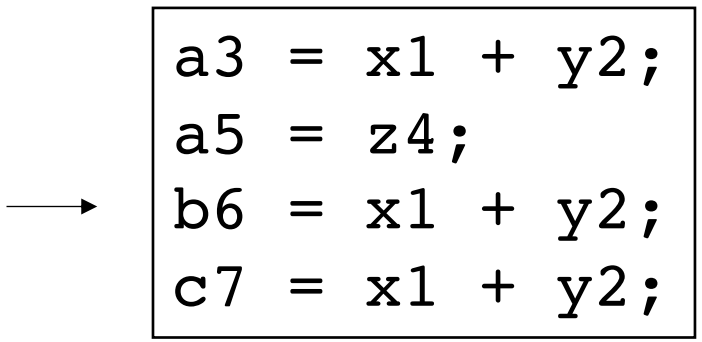
```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

```
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



```
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

```
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

```
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

```
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

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

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

H = {
      ”x1 + y2” : ”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

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

```
Current_val = {
}
```

```
H = {
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

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

```
Current_val = {
}
```

```
H = {
}
```

Work through example

Local value numbering: value sets

- Final heuristic: keep sets of possible values

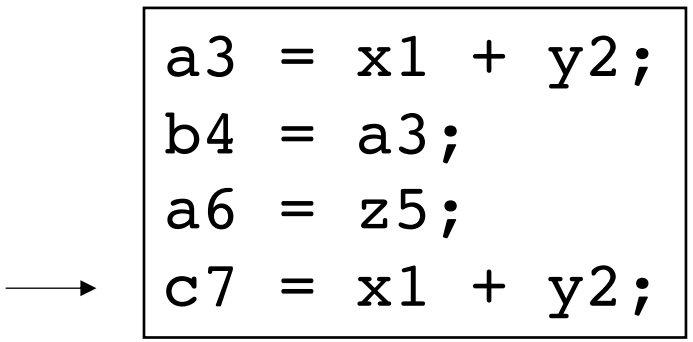
```
a3 = x1 + y2;
b4 = x1 + y2;
a6 = z5;
c7 = x1 + y2;
```

```
Current_val = {
}
```

```
H = {
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values

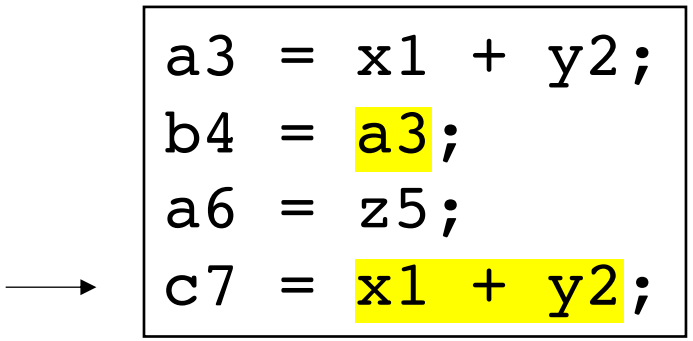



```
Current_val = {
                  “a” : 6,
                  “b” : 4
}


H = {
   “x1 + y2” : “a3”
}
```

Local value numbering: value sets

- Final heuristic: keep sets of possible values




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


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”
}
```

but we could have replaced it with b4!

Local value numbering: value sets

- Final heuristic: keep sets of possible values

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

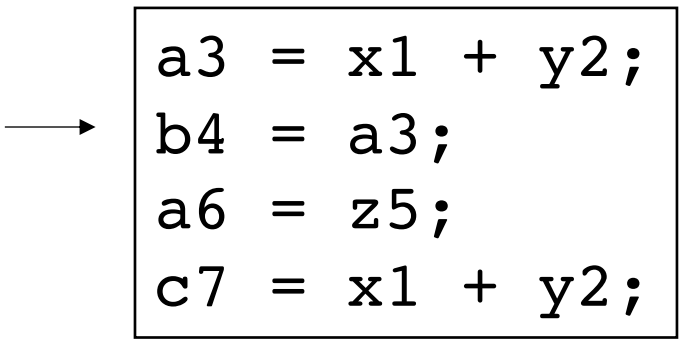
rewind to this point →

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

```
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

fast forward again →

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

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


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

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!

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

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

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 Friday

- More about optimizations!