# CSE211: Compiler Design
Oct. 20, 2021

- **Topic**: Local optimizations (Local Value Numbering)

- **Questions**:
  - How can you optimize arithmetic expressions?

# Announcements

- Homework 1 is out
  - Due on the 25th
  - No extensions

- Get your paper reading approved by me by Monday
  - No extensions, 5% of your grade

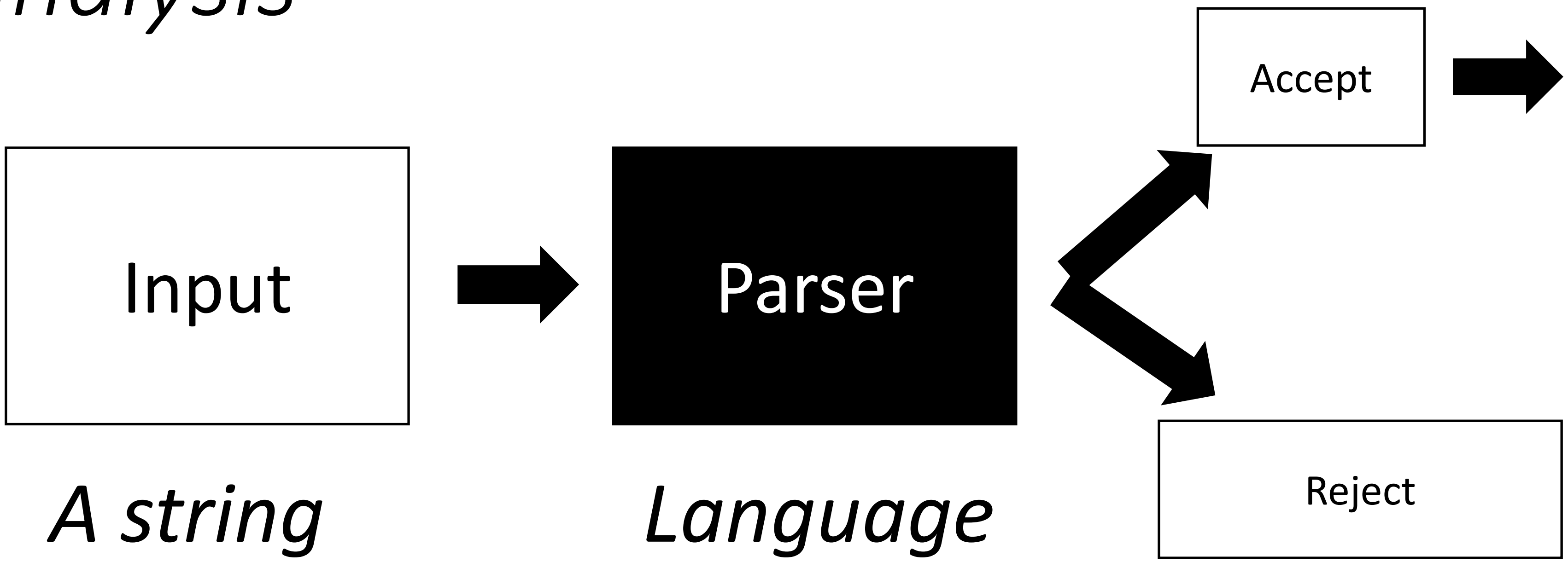
- Ask questions on Piazza if you have questions

# Announcements

- IF I AM FEELING WELL ENOUGH: I will be gone Monday and Wednesday next week to attend a khronos group meeting.
  - The schedule is still in flux:
    - either I will hold class synchronously on Zoom
    - Or provide asynchronous lectures
    - Maybe a combination, stay tuned

# Review

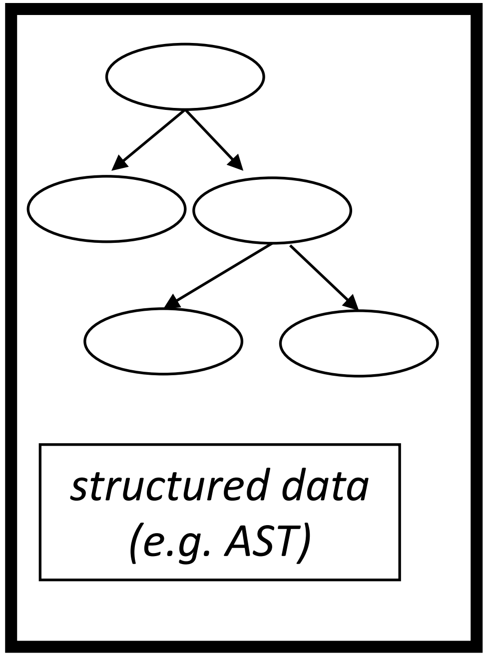
# On to Module 2! *Optimizations and analysis*




*A string*

*Language Recognizer for language L*

*continue to the rest of compilation*




**Where most optimizations and flow analysis happens!**

# Different IRs

Many different IRs, each have different purposes

- Trees
  - Abstract syntax trees
  - Data-dependency trees
  - **Good for instruction scheduling**

- Textual
  - 3 address code
  - **Good for removing redundant expressions**

- Graphs
  - Control flow graphs
  - **Good for data flow analysis (finding uninitialized variables)**

# Abstract Syntax Trees

- Easier to see bigger trees, e.g. quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

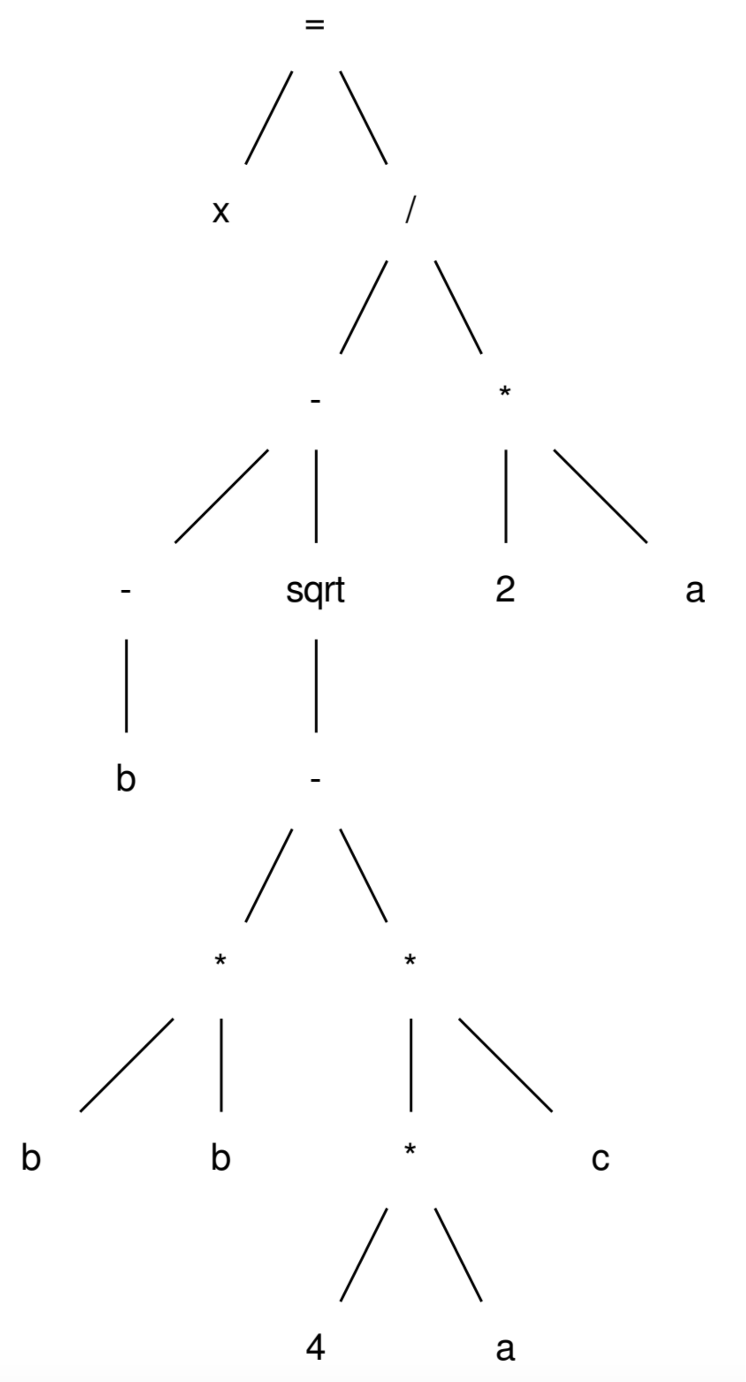
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)

Thanks to Sreepathi Pai for the example!

Convert this code to 3 address code

post-order traversal, creating virtual registers for each node

```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 - r3;
r5 = sqrt(r4);
r6 = r0 - r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



# What about control flow?

- 3 address code typically contains a conditional branch:

```
br <reg>, <label0>, <label1>
```

if the value in <reg> is true, branch to <label0>, else branch to <label1>

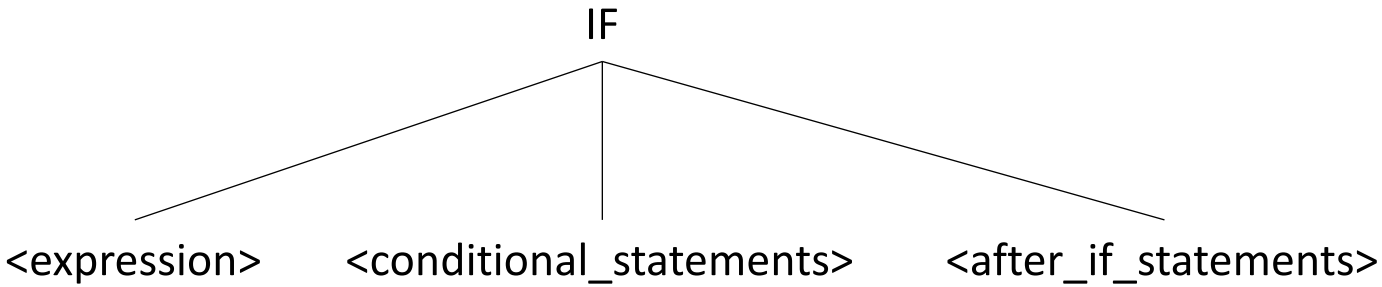
```
br <label0>
```

unconditional branch

# What about control flow?

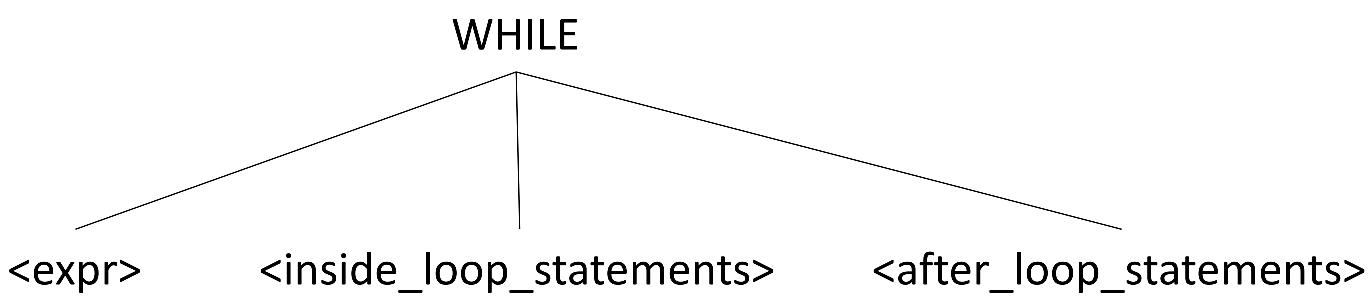
```
if (expr) {
 // conditional statements
}
// after if statements
```

```
r0 = <expression>;
br r0, conditional_stmts, after_if;

conditional_stmts:
<conditional_statements>;


after_if:
<after_if_statements>;
```



# What about control flow?

```
while (expr) {
 // inside_loop_statements
}
// after_loop_statements
```




```
beginning_label:
r0 = <expr>

br r0, inside_loop, after_loop;

inside_loop:
<inside_loop_statements>
br beginning_label;

after_loop:
<after_loop_statements>
```

# New material

# 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?

Four 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;
```

# 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

```
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
  - 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?

```
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
- **Regional optimizations:**
  - Combines several basic blocks
- **Global optimizations:**
  - operates across an entire procedure
  - what about across procedures?

```
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;
```

*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;
…
```

*at a higher-level,*
*we cannot replace:*
*y = a + b.*
*with*
*y = x;*

# Regional Optimization

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

*at a higher-level,*
*we cannot replace:*
*y = a + b.*
*with*
*y = x;*

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

*But if a and b are*
*not redefined, then*
*y = a + b;*
*can be replaced with*
*y = x;*

# Today’s lecture: A local optimization

# 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
  - We will cover in later lectures.

# 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
  - We will cover in later lectures.

```
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
  - We will cover in later lectures.

```
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
  - We will cover in later lectures.

```
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
  - We will cover in later lectures.

```
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
  - We will cover in later lectures.

```
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

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

Global_counter = 6

# 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;
b44 = 4;
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 = f4;
```

# 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

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

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

```
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

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

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

```
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

```
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

```
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

```
Current_val = {
}
```

```
H = {
}
```

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

# 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

```
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

```
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

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

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

fast forward again

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