CSE110A: Compilers

Topics:

- *Start of Module 4 - Optimizations*

Module 4: Optimizations

Discussion

- What are compiler optimizations?
- Why do we want compiler optimizations?

Discussion

- What are compiler optimizations?
  - automated program transforms designed to make code more optimal
  - optimal can mean different things
    - code optimized for one system might be different for code optimized for a different system
    - we can optimize for speed, for energy efficiency, or for code size. What else?
- Why do we want the compiler to help us optimize?
  - So we can write more maintainable/portable code
  - So we don’t have to worry about learning nuanced details about every possible system

Discussion

- What are some compiler optimizations you know about?

Discussion

- What are some compiler optimizations you know about?

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
}
```

loop unrolling

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
  i++;
  x = x + 1;
}
```

Discussion

- What are some compiler optimizations you know about?

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
}
```

loop unrolling

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
  i++;
  x = x + 1;
}
```

```
int foo() {
  int i,j,k;
  i = 10;
  j = i;
  k = j;
  return k;
}
```

constant propagation

```
int foo() {
  int i,j,k;
  return 10;
}
```

Discussion

- What are some compiler optimizations you know about?

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
}
```

loop unrolling

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
  i++;
  x = x + 1;
}
```

What does this save us?

Discussion

- What are some compiler optimizations you know about?

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
}
```

loop unrolling

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
  i++;
  x = x + 1;
}
```

What does this save us?

optimizations at one stage can enable optimizations at another stage:

```
for (int i = 0; i < 10; i+=2) {
  x = x + 2;
}
```

provides a bigger window for local analysis

Discussion

- What are some compiler optimizations you know about?

let’s do a few more

Function inlining

```
int add(int x, int y) {
  return x + y;
}

int foo(int x, int y, int z) {
  return add(x,y);
}
```

```
int foo(int x, int y, int z) {
  return x + y;
}
```

What does this save us?
code size? speed? the ability to debug? local regions to optimize more?

Discussion

- What are some compiler optimizations you know about?

There are many more! This is an active area of research and development

For a rough metric:

`git effort` shows activities on different files and directories

clang C++/C parser: 3.5K commits
clang AST: 8.7K commits
LLVM transforms/optimizations: 30K commits

The LLVM project has been under active development since **2000**, and Clang since about **2007–2008**.

The transformation part of the code base has the most activity by far

Discussion

- How do you enable compiler optimizations?

Discussion

- How do you enable compiler optimizations?

- most C/++ compilers
  - optimizing for speed
    - -O0, -O1, -O2, -O3
    - what about O4?
  - optimizing for size
    - -Os, -Oz
  - relax some constraints (especially around floating point):
    - -Ofast
    - Godbolt example

https://stackoverflow.com/questions/15548023/clang-optimization-levels

Discussion

- How do you enable compiler optimizations?

  - most C/++ compilers
    - optimizing for speed
    - -O0, -O1, -O2, -O3, -O4
  - optimizing for size
    - -Os, -Oz
  - relax some constraints (especially around floating point):
    - -Ofast
    - We can use Godbolt to test various optimizations.

https://stackoverflow.com/questions/15548023/clang-optimization-levels

Discussion

STABILIZER: Statistically Sound Performance Evaluation

Charlie Curtsinger   Emery D. Berger
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{charlie,emery}@cs.umass.edu

2013 research paper

“the performance impact of -O3 over -O2 optimizations is indistinguishable from random noise.”

Discussion

- What are some of the biggest improvements you’ve seen from compiler optimizations?

Discussion

- What are some of the biggest improvements you’ve seen from compiler optimizations?
- compiler optimizations are great at well-structured, regular loops and arrays

Discussion

- What kind of transforms on your code is the compiler allowed to do?

Discussion

- What kind of transforms on your code is the compiler allowed to do?
- many_add example
- Why did we get such a dramatic increase?

Discussion

- Extreme example

```
void foo(int * arr, int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
         for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1]) {
                tmp = arr[j];
                arr[j] = arr[j + 1]);
                arr[j + 1] = tmp;
            }
}
```

```
int p(int arr[], int start, int end)
{

    int pivot = arr[start];

    int count = 0;
    for (int i = start + 1; i <= end; i++) {
        if (arr[i] <= pivot)
            count++;
    }

    int pivotIndex = start + count;
    swap(arr[pivotIndex], arr[start]);

    int i = start, j = end;
    while (i < pivotIndex && j > pivotIndex) {

        while (arr[i] <= pivot) {
            i++;
        }

        while (arr[j] > pivot) {
            j--;
        }

        if (i < pivotIndex && j > pivotIndex) {
            swap(arr[i++], arr[j--]);
        }
    }

    return pivotIndex;
}

void foo(int *arr, int n)
{

    if (start >= end)
        return;

    int p = p(arr, m, n);

    foo(arr, start, p - 1);

    foo(arr, p + 1, end);
}
```

is this transform legal?

code from https://www.geeksforgeeks.org/

Discussion

- Extreme example

bubble sort

```
void foo(int * arr, int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
         for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1]) {
                tmp = arr[j];
                arr[j] = arr[j + 1]);
                arr[j + 1] = tmp;
            }
}
```

code from https://www.geeksforgeeks.org/

Yes this transform
would be legal!

Could any compiler figure it out?
currently unlikely..

This is a technique called
“super optimizing” and it is
getting more and more interest

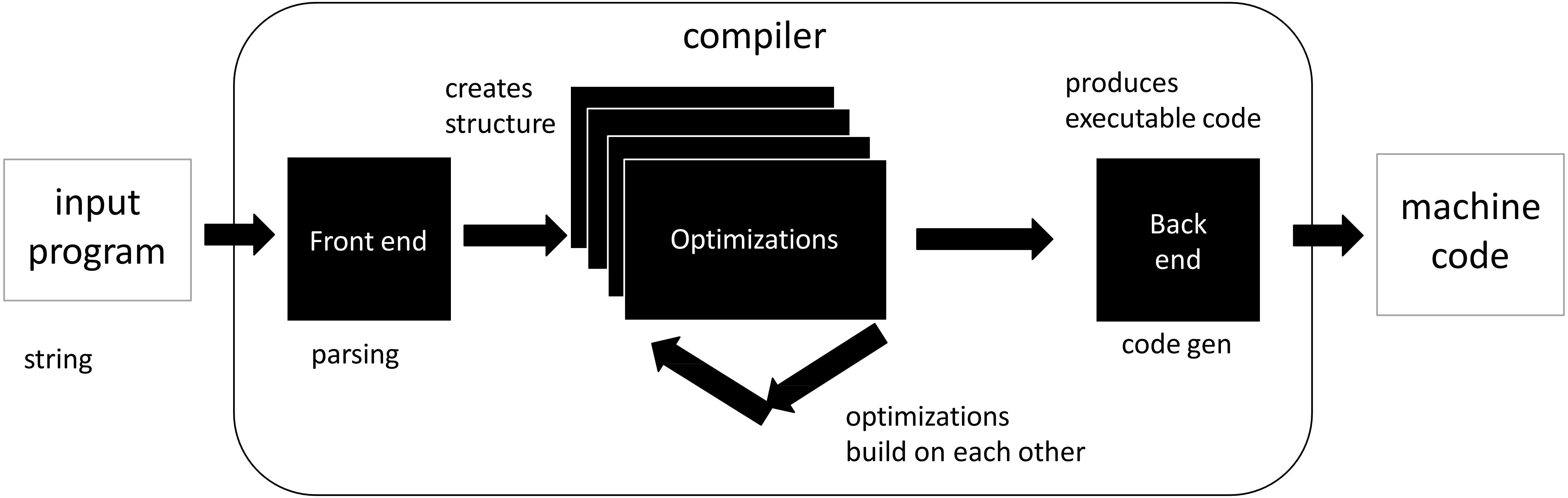
```
int p(int arr[], int start, int end)
{

    int pivot = arr[start];

    int count = 0;
    for (int i = start + 1; i <= end; i++) {
        if (arr[i] <= pivot)
            count++;
    }

    int pivotIndex = start + count;
    swap(arr[pivotIndex], arr[start]);

    int i = start, j = end;
    while (i < pivotIndex && j > pivotIndex) {

        while (arr[i] <= pivot) {
            i++;
        }

        while (arr[j] > pivot) {
            j--;
        }

        if (i < pivotIndex && j > pivotIndex) {
            swap(arr[i++], arr[j--]);
        }
    }

    return pivotIndex;
}

void foo(int *arr, int n)
{

    if (start >= end)
        return;

    int p = p(arr, m, n);

    foo(arr, start, p - 1);

    foo(arr, p + 1, end);
}
```

quick sort

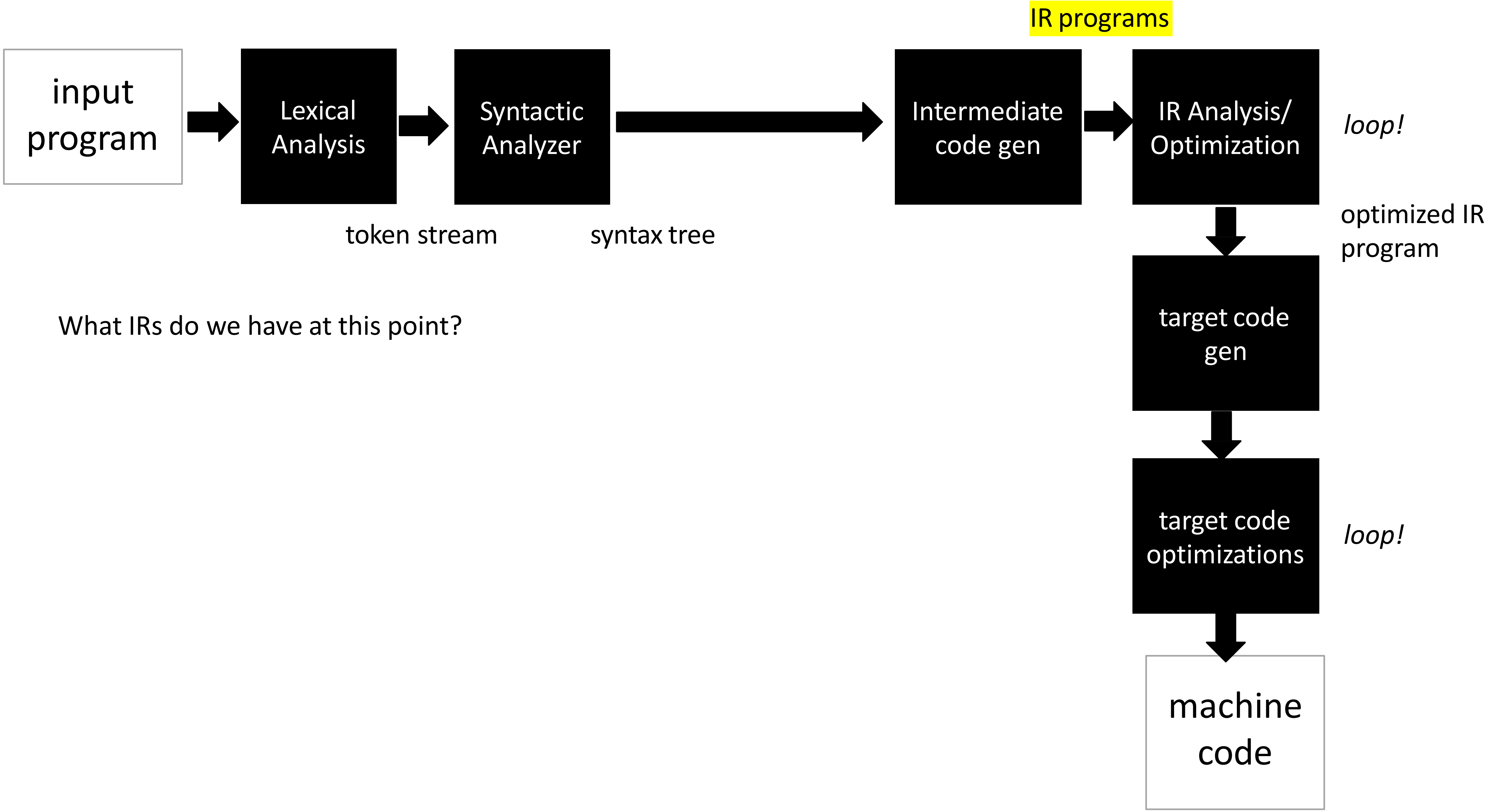
is this transform legal?

Moving on

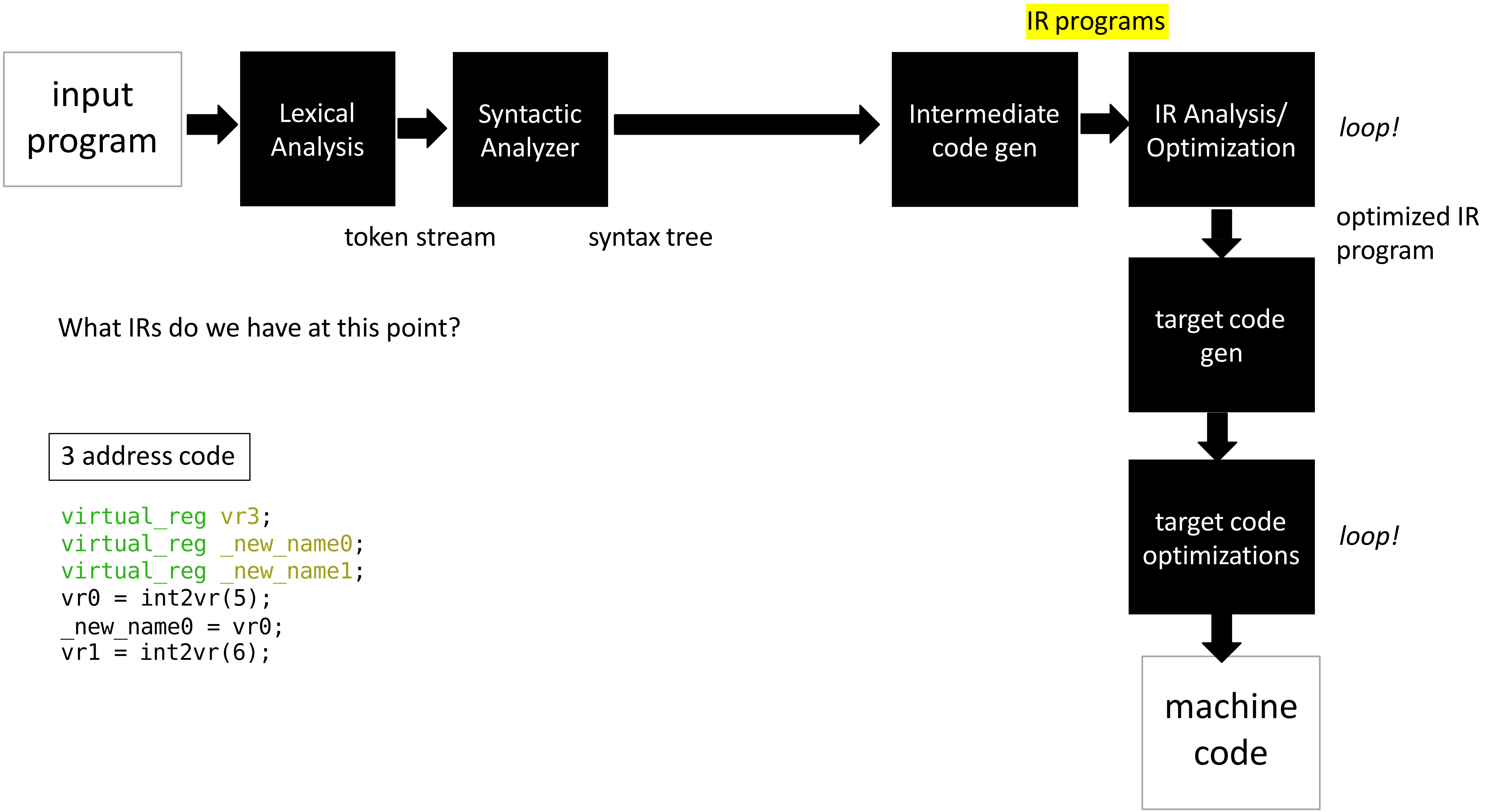
Zooming out again: Compiler Architecture



IRs and type inference type inference are at the boundary of parsing and optimizations



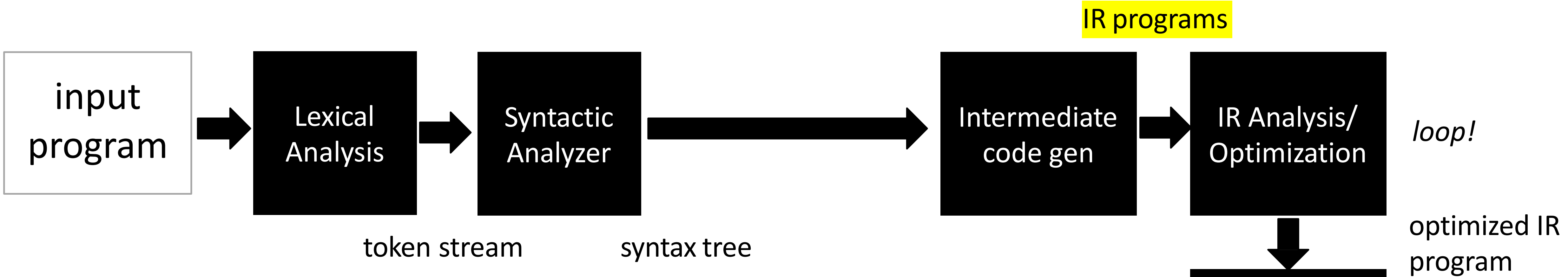
What IRs do we have at this point?



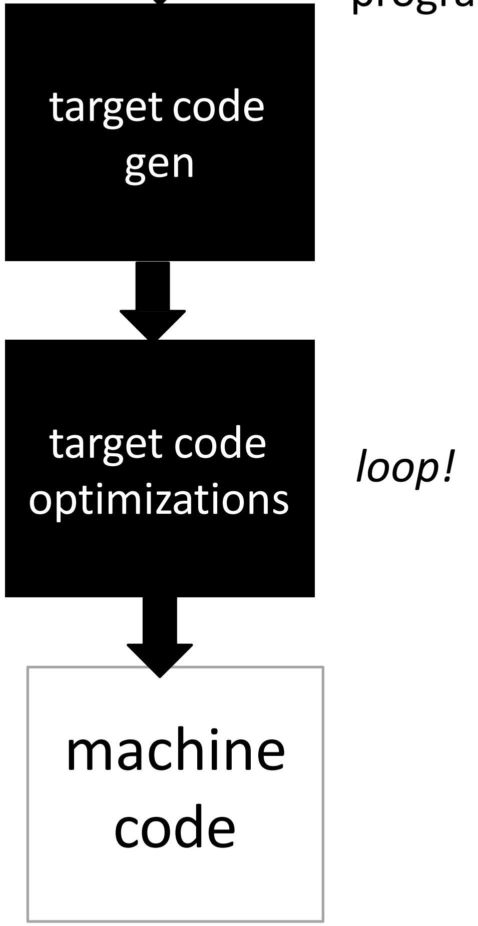
What IRs do we have at this point?

3 address code

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



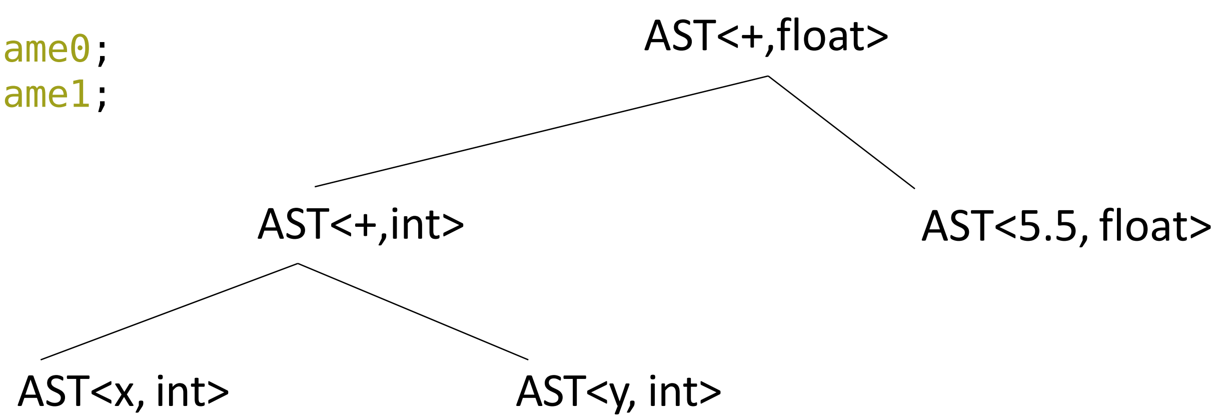
What IRs do we have at this point?



3 address code

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

AST



Implicit parse tree

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

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

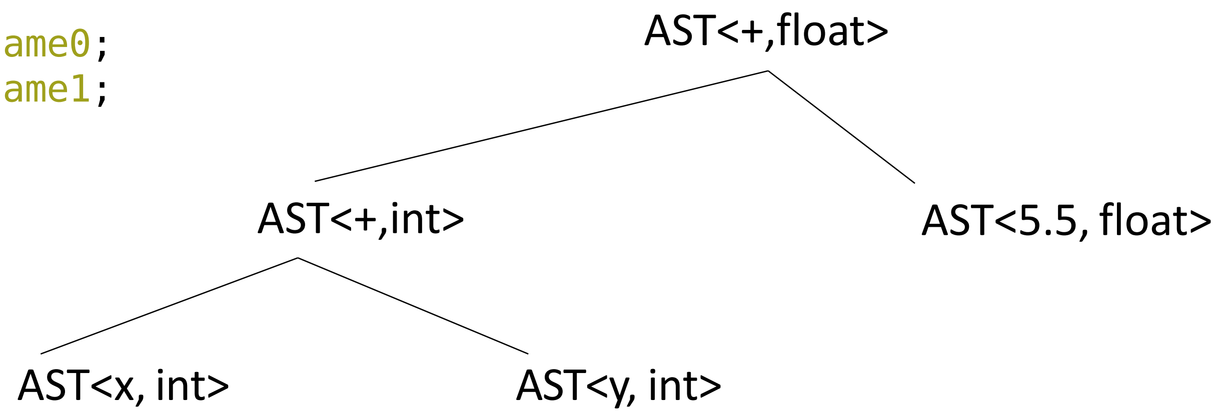
We have several structures to utilize to analyze and optimize programs!

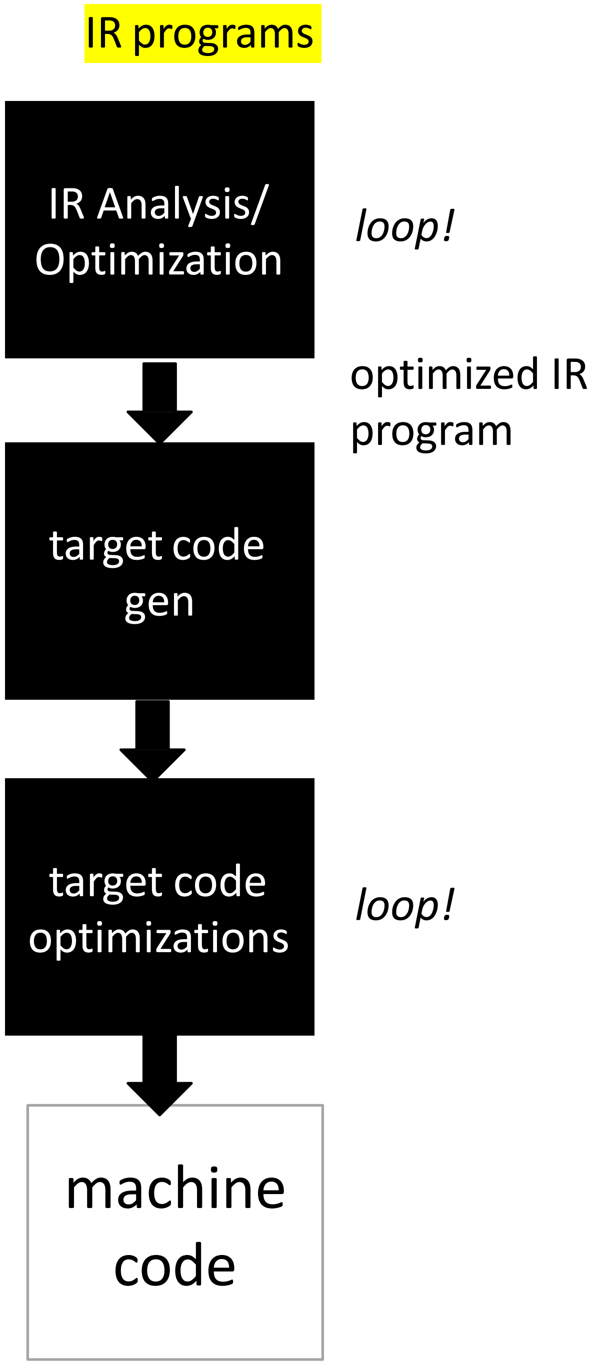
What IRs do we have at this point?

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

- Machine-independent - these optimizations should work well across many different systems
  - Examples?

- Machine dependent - these optimizations start to optimize the code for a given system
  - Examples?

Optimization categories

- Machine-independent - these optimizations should work well across many different systems
  - Examples?
  - All the examples we looked at before seem like they will help across many systems

- Machine dependent - these optimizations start to optimize the code for a given system
  - Examples?
  - loop chunking for cache line size and vectorization.
  - instruction re-orderings to take advantage of processor pipelines.
  - fused multiply-and-add instructions

Optimization categories

- **Machine-independent** - these optimizations should work well across many different systems
  - Examples?
  - All the examples we looked at before seem like they will help across many systems

- In this module we will be looking at machine-independent optimizations.

- What are the pros of machine-independent optimizations?

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

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
- **IDFA Inter-Procedural Data Flow Analysis** : Optimizes across functions

Discussion:

- What are the pros and cons of each?
- Going across functions is less common, why?

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

For this module:

- We will look at two optimizations in detail:
- A local optimization: Local value numbering
- A regional optimization: Loop unrolling
- We will implement both as homework
- We will discuss several other optimizations and analysis

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 4:

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.

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

```
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;
…
```

we cannot replace:

y = a + b.

with

y = x;

Regional Optimization: Data Flow Analysis

```
…
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;
…
```

Can x = a + b be hoisted out of the if then else?

In that 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

```
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

```
a =  b0 + c1;
b =  a  - d;
c =  b  + c;
d =  a  - d;
```

Global_counter = 2

b and c have not been seen, give them a number

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;
b  =  a2 - d3;
c  =  b  + c;
d  =  a  - d;
```

Global_counter = 4

a2 have been updated use the last defined number for a on RHS a.   i.e. a2

Local value numbering

Algorithm:

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

- 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

So … for LHS variable always increment and update number, for right hand side use, always append last number

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;
```

If we remember to declare every one of these variable names, the IR is still valid !!!

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!

i.e. it is no longer just

c = b + c

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

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

```
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

```
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

```
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

```
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

```
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
}
```

fast forward again

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

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

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
    - We may have time to discuss this later in the module

Next:

- More optimizations!