CSE110A: Compilers

May 31, 2023

Topics:

- *Local value numbering continued*

Announcements

- Grades:
  - Expect HW 2 grades by the end of the day
  - HW 3 grades are coming shortly after
- HW 5 is out
  - It is another big assignment, so get started early!
  - Due on Sunday June 11
    - Try to turn it in my Friday so that you have time to study.
  - For HW 4, our office hours the first week were almost empty! Please take advantage of them
- Final
  - Monday June 12 at 8 AM ☹
  - 3 pages of notes allowed
  - Comprehensive

Quiz

Quiz

It's the parser's job to perform local value numbering

- ○ True
- ○ False

Discussion

- Local value numbering operates over 3 address code
- The parser produces 3 address code
- In some cases, the parser might use LVN, but it is independent

→

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

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

Quiz

Local value numbering can only work in just one basic block.

- ○ True
- ○ False

Discussion

- Reminder on a basic block

Discussion

How might they appear in a high-level language?

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

How many basic blocks?

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

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

Two Basic Blocks

```
Label_x:
op1;
op2;
op3;

Label_y:
op4;
op5;
```

Single Basic Block

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

Discussion

```
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;
```

```
br Label_1;

Label_0:
x = a + b;

Label_1:
y = a + b;
```

Quiz

Local value numbering can only work in just one basic block.

- ○ True
- ○ False

True!

Although you might imagine algorithms that can work on more basic blocks.
This is called *superlocal value numbering*

Quiz

After perform local value numbering on the following program, how many operations can you save?

```
a = b + c;

d = e * f;

b = b + c;

c = c + b ;

g = f * e;
```

- ○ 1
- ○ 2
- ○ 3
- ○ 4

Discussion

```
a = b + c;
d = e * f;
b = b + c;
c = c + b ;
g = f * e;
```

```
H = {
          ,
}
```

Discussion

```
a = b + c;
d = e * f;
b = b + c;
c = c + b ;
g = f - e;
```

```
H = {
          ,
}
```

What if we changed this?

Quiz

What is a good order of performing the following optimizations (left to right):

1) Local value numbering

2) Loop unrolling

3) Constant propagation

Discussion

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

loop unrolling

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

how might this influence other optimizations?

Discussion

```
a = 16;
b = a + c;
d = 16;
e = d + c;
```

How might constant propagation change this program

Discussion

```
a = 16;
b = a + c;
d = 16;
e = d + c;
```

How might constant propagation change this program

```
a = 16;
b = 16 + c;
d = 16;
e = 16 + c;
```

Discussion

```
a = 16;
b = a + c;
d = 16;
e = d + c;
```

How might constant propagation change this program

```
a = 16;
b = 16 + c;
d = 16;
e = 16 + c;
```

LVN can now replace the bottom one

It's a little more difficult to apply
to ClassIeR. Do people have any ideas?

Quiz

What is a good order of performing the following optimizations (left to right):

1) Local value numbering

2) Loop unrolling

3) Constant propagation

Loop unrolling -> Constant propagation -> Local value numbering

Review

- Basic blocks
  - A piece of 3 address code that has one entry and one exit
  - Any line of code can assume that all lines before it have been executed
  - Allows “local” reasoning
  - pycfg example

- Local value numbering
  - Local optimization
  - Simple algorithm that can be built on:
    - initial version just used string comparison
    - next we added commutativity
    - lastly we extended the algorithm to not add any new registers

Review

- Algorithm for applying LVN:
  - split 3 address code into basic blocks
  - for each basic block
    - number the variables
    - try to remove expensive arithmetic operations

Today

- Adding constant folding to LVN
- Discussing memory and functions in LVN
- How to add optimized code blocks back into the IR

Constant propagation and constant folding

- Colloquially, they are often used interchangeably
- Technically (e.g. according to the books)
  - Constant propagation is replacing variables with constants
  - Constant folding is compile-time evaluation when constants are known

Constant propagation and constant folding

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

from: https://en.wikipedia.org/wiki/Constant_folding#Constant_propagation

Constant propagation and constant folding

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

constant propagation

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

from: https://en.wikipedia.org/wiki/Constant_folding#Constant_propagation

Constant propagation and constant folding

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

constant propagation

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

constant folding

```
int x = 14;
int y = 0;
return y * (4);
```

from: https://en.wikipedia.org/wiki/Constant_folding#Constant_propagation

Constant propagation and constant folding

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

```
int x = 14;
int y = 0;
return 0 * (4);
```

constant propagation

constant propagation

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

constant folding

```
int x = 14;
int y = 0;
return y * (4);
```

from: https://en.wikipedia.org/wiki/Constant_folding#Constant_propagation

Constant propagation and constant folding

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

constant propagation

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

constant folding

```
int x = 14;
int y = 0;
return y * (4);
```

constant propagation

```
int x = 14;
int y = 0;
return 0 * (4);
```

constant folding

```
int x = 14;
int y = 0;
return 0;
```

from: https://en.wikipedia.org/wiki/Constant_folding#Constant_propagation

Typically performed at the same time

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

constant propagation and folding second line

```
int x = 14;
int y = 0;
return y * (28 / x + 2);
```

```
int x = 14;
int y = 0;
return 0;
```

constant propagation and folding third line

from: https://en.wikipedia.org/wiki/Constant_folding#Constant_propagation

Adding constant folding to LVN

```
b = 5;
c = 3;

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

```
H = {
}
```

```
Known_values = {

}
```

Adding constant folding to LVN

numbering

```
b0 = 5;
c1 = 3;

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

```
H = {
}
```

```
Known_values = {

}
```

Adding constant folding to LVN

As you are iterating through code, add any constant mappings to Known_values:

```
b0 = 5;
c1 = 3;

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

```
H = {
}
```

```
Known_values = {

}
```

Adding constant folding to LVN

As you are iterating through code, add any constant mappings to Known_values:

```
b0 = 5;
c1 = 3;

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

```
H = {
}
```

```
Known_values = {
"b0" : 5
"c1" : 3

}
```

Adding constant folding to LVN

When you find an arithmetic operation, first check if operands are known

```
b0 = 5;
c1 = 3;

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

```
H = {
}
```

```
Known_values = {
"b0" : 5
"c1" : 3

}
```

Adding constant folding to LVN

When you find an arithmetic operation, first check if operands are known

```
b0 = 5;
c1 = 3;

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

5 + 3

evaluate and add to known values

```
H = {
}
```

```
Known_values = {
"b0" : 5
"c1" : 3

}
```

Adding constant folding to LVN

When you find an arithmetic operation, first check if operands are known

```
b0 = 5;
c1 = 3;

a2 = 8;
b4 = a2 - d3;
c5 = a2 + c1;
d6 = a2 - d3;
```

5 + 3

evaluate and add to known values

```
H = {
}
```

```
Known_values = {
"b0" : 5
"c1" : 3
"a2" : 8

}
```

Adding constant folding to LVN

When you find an arithmetic operation, first check if operands are known

```
b0 = 5;
c1 = 3;

a2 = 8;
b4 = 8 - d3;
c5 = a2 + c1;
d6 = a2 - d3;
```

propagate constant (if IR allows it)

```
H = {

}
```

```
Known_values = {
"b0" : 5
"c1" : 3
"a2" : 8

}
```

Adding constant folding to LVN

When you find an arithmetic operation, first check if operands are known

```
b0 = 5;
c1 = 3;

a2 = 8;
b4 = 8 - d3;
c5 = a2 + c1;
d6 = a2 - d3;
```

add to H

why do we want to store 8 here rather than a2?

```
H = {
 "8 - d3" : b4
}
```

```
Known_values = {
"b0" : 5
"c1" : 3
"a2" : 8

}
```

continue on.

Arithmetic identities

```
b0 = 0;
d3 = 1;
f7 = 4;

a2 = b0 + c1;
b4 = a2 * d3;
d6 = e5 * f7;
```

```
H = {
}
```

```
Known_values = {
}
```

Arithmetic identities

what can we do here?

```
b0 = 0;
d3 = 1;
f7 = 4;

a2 = b0 + c1;
b4 = a2 * d3;
d6 = e5 * f7;
```

```
H = {
}
```

```
Known_values = {
"b0":0, "d3":1, "f7":4
}
```

Arithmetic identities

what can we do here?
add a special rule for +
that if any side is 0, you can
just drop the 0.

```
b0 = 0;
d3 = 1;
f7 = 4;

a2 = b0 + c1;
b4 = a2 * d3;
d6 = e5 * f7;
```

```
H = {
}
```

```
Known_values = {
"b0":0, "d3":1, "f7":4
}
```

Arithmetic identities

what can we do here?
add a special rule for +
that if any side is 0, you can
just drop the 0.

```
b0 = 0;
d3 = 1;
f7 = 4;

a2 = c1;
b4 = a2 * d3;
d6 = e5 * f7;
```

```
H = {
}
```

```
Known_values = {
"b0":0, "d3":1, "f7":4
}
```

What other rules could we have?

Other considerations in LVN

- Memory and functions

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?

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

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

- Consider a 3 address code that allows memory accesses

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

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

Example, initially:

```
i = j
a = x
y[k] = 1
x[j] = 1
```

What does b[i] equal at the end of each computation?

Local value numbering: Memory

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

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

Does this help at all?

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,6) = (x[j],4) + (y[k],5);
(c,7) = (x[j],4) + (y[k],5);
```

Does this help at all?

If there is no memory writes between an assignment to a variable then we can do a replacement

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,6) = (x[j],4) + (y[k],5);
(c,7) = (b,6);
```

Does this help at all?

If there is no memory writes between an assignment to a variable then we can do a replacement

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);
```

A compiler analysis might try to determine that addresses can't alias

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

A compiler analysis might try to determine that addresses can't alias

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

Warning: the compiler does not enforce this!

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);
```

Local value numbering: functions

Local value numbering: functions

How to number?

```
a = foo(x);
x = b;
c = foo(x);
```

Local value numbering: functions

How to number?

```
a = foo(x);
x = b;
c = foo(x);
```

the same way

```
a1 = foo(x0);
x3 = b2;
c4 = foo(x3);
```

What if you had
first class functions?

Local value numbering: functions

How to number?

```
a = foo(x);
x = b;
c = foo(x);
```

the same way

```
a1 = foo(x0);
x3 = b2;
c4 = foo(x3);
```

Can we replace?

Local value numbering: functions

How to number?

```
a = foo(x);
c = foo(x);
```

the same way

```
a1 = foo(x0);
c2 = foo(x0);
```

How about now?

Local value numbering: functions

How to number?

```
a = foo(x);
c = foo(x);
```

the same way

```
a1 = foo(x0);
c2 = foo(x0);
```

How about now?

```
int count = 0;
int foo(int x) {
  count += 1;
  return 0;
};
```

What if foo had
this implementation?

Local value numbering: functions

How to number?

```
a = foo(x);
c = foo(x);
```

the same way

```
a1 = foo(x0);
c2 = foo(x0);
```

How about now?

side effects!

```
int count = 0;
int foo(int x) {
  count += 1;
  return 0;
};
```

What if foo had
this implementation?

Local value numbering: functions

```
a = foo(x);
c = foo(x);
print(count);
```

```
a1 = foo(x0);
c2 = a1;
print(count);
```

are these two programs the same?

```
int count = 0;
int foo(int x) {
  count += 1;
  return 0;
};
```

Local value numbering: functions

- In C/++, functions are assumed to have side effects
- A function that does not have side effects is called “pure”
  - You can annotate a function as pure
  - `__attribute__((pure))`
  - warning: compiler does not check this and you can introduce subtle bugs
- Functional languages tend to have a pure-by-default design. Allows more compiler optimizations, but less control to the programmer.

How to stitch optimized code back into the program

How to stitch optimized code back into the program

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

How to stitch optimized code back into the program

split into basic blocks

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

```
a = b + c;
d = e + f;
g = b + c;
```

```
label_0:
h = g + a;
k = a + g;
```

How to stitch optimized code back into the program

number

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

```
a = b + c;
d = e + f;
g = b + c;
```

```
label_0:
h = g + a;
k = a + g;
```

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = b0 + c1;
```

```
label_0:
h2 = g0 + a1;
k3 = a1 + g0;
```

How to stitch optimized code back into the program

move code on slide to make room

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = b0 + c1;
```

```
label_0:
h2 = g0 + a1;
k3 = a1 + g0;
```

How to stitch optimized code back into the program

optimize

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = b0 + c1;
```

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = a1 + g0;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

How to stitch optimized code back into the program

optimize

put together?

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = b0 + c1;
```

```
label_0:
h2 = g0 + a1;
k3 = a1 + g0;
```

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
label_0:
h2 = g0 + a1;
k3 = h2;
```

What are the issues?

How to stitch optimized code back into the program

optimize

put together?

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = b0 + c1;
```

```
label_0:
h2 = g0 + a1;
k3 = a1 + g0;
```

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
label_0:
h2 = g0 + a1;
k3 = h2;
```

undefined!

What are the issues?

How to stitch optimized code back into the program

optimize

stitch
part 1: *assign original variables their latest values*

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = b0 + c1;
```

```
label_0:
h2 = g0 + a1;
k3 = a1 + g0;
```

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
g = g6;
d = d5
a = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
h = h2;
k = k3;
```

How to stitch optimized code back into the program

make room on slide

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
g = g6;
d = d5
a = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
h = h2;
k = k3;
```

what else needs to be done?

How to stitch optimized code back into the program

stitch part 2: drop numbers from first use of variables

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
g = g6;
d = d5
a = a2;
```

```
a2 = b + c;
d5 = e + f;
g6 = a2;
g = g6;
d = d5
a = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
h = h2;
k = k3;
```

```
label_0:
h2 = g + a;
k3 = h2;
h = h2;
k = k3;
```

How to stitch optimized code back into the program

Now they can be combined

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
g = g6;
d = d5
a = a2;
```

```
a2 = b + c;
d5 = e + f;
g6 = a2;
g = g6;
d = d5
a = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
h = h2;
k = k3;
```

```
label_0:
h2 = g + a;
k3 = h2;
h = h2;
k = k3;
```

```
a2 = b + c;
d5 = e + f;
g6 = a2;
g = g6;
d = d5
a = a2;
label_0:
h2 = g + a;
k3 = h2;
h = h2;
k = k3;
```

How to stitch optimized code back into the program

original

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

new

```
a2 = b + c;
d5 = e + f;
g6 = a2;
g = g6;
d = d5
a = a2;
label_0:
h2 = g + a;
k3 = h2;
h = h2;
k = k3;
```

is it really optimized?

It looks a lot longer...

How to stitch optimized code back into the program

original

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

new

```
a2 = b + c;
d5 = e + f;
g6 = a2;
g = g6;
d = d5
a = a2;
label_0:
h2 = g + a;
k3 = h2;
h = h2;
k = k3;
```

is it really optimized?

Common pattern for code to get larger, but it will contain patterns that are easier optimize away

later passes will minimize copies

Loop optimizations

For loops

- How do they look in different languages
  - C/C++
  - Python
  - Numpy

- How do Python and Numpy look under the hood?

- The more constrained the for loops are, the more assumptions the compiler can make, but less flexibility for the programmer

For loops

- The compiler can optimize For loops if they fit a certain pattern

- When developing a compiler optimization, we start with strict constraints and then slowly relax them and make the optimization more general.
  - Sometimes it is not worth relaxing the constraints (code gets too complex)
  - If you know the constraints, then often you can write code such that the compiler can recognize the pattern and optimize!

For loops terminology

- Loop body:
  - A series of statements that are executed each loop iteration
- Loop condition:
  - the condition that decides whether the loop body is executed
- Iteration variable:
  - A variable that is updated exactly once during the loop
  - The loop condition depends on the iteration variable

Loop unrolling

- Executing multiple instances of the loop body without checking the loop condition.

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

unrolled by a factor of 2

```
for (int i = 0; i < 128; i++) {
  // body
}
```

```
for (int i = 0; i < 128; i++) {
  // body
  i++
  // body
}
```

could we unroll more?

Loop unrolling conditions

- Under what conditions can we unroll?

FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement

```
for (int i = 0; i < 128; i++) {
  // body
}
```

Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

Validate that we actually have an iteration variable

1. lhs of assignment statement
2. no assignment to variable in body
3. lhs of loop condition
4. lhs of assignment_statement

Do these guarantee we will find an iteration variable?
What happens if we don't find one?

Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

Validate that we actually have an iteration variable

1. lhs of assignment statement
2. no assignment to variable in body
3. lhs of loop condition
4. lhs of assignment_statement

How does C-simple help us here?

Do these guarantee we will find an iteration variable?
What happens if we don't find one?

Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

Validate properties of iteration variable

1. ?

Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

Validate properties of iteration variable

1. identify an iteration range (start and end)
2. increment by 1

See everyone on Friday

- Topics: Continue Loop unrolling and other loop optimizations!