CSE110A: Compilers

June 1, 2022

Topics:

- *Control flow graphs*
- *Live variable analysis*

Announcements

- Pending grades:
  - HW 3
  - We will try to get feedback (and hopefully grades) ASAP
- Homework 4 is out
  - ***Get started now if you haven't already***
  - Due on the date of the final (June 7 by midnight). No late days for this HW
- SETs are out:
  - please take some time to fill them out
  - It really helps make the classes better in the future

Announcements

- Final is on June 7 (less than 1 week away)
- Similar to Midterm
  - Major difference: only 1 day to do it: it will be assigned by 8 AM on June 7 and due by midnight on June 7.
  - No time limit enforced during those hours
  - Open note, slides, internet, etc.
    - Do not discuss any aspect of the final with classmates while it is out
    - Do not discuss (or ask questions about) the test on an online forum; we do monitor these things!
  - Similar length to Midterm
    - Designed to take 2-3 hours assuming ~6 hours of studying
    - As you saw with the midterm: it is common to spend longer on take home tests
  - Cumulative material: Anything discussed in class if fair game.

Announcements

- Final is on June 7 (less than 1 week away)
- We will keep a piazza note with clarification questions
- Ask any clarifications as a private piazza post
- Not guaranteed help outside of business hours
- Help will be guaranteed 7:30 PM to 10:30 PM (the scheduled time of the test)
  - sad hours I know ☹

Announcements

- Final grades:
  - No curve planned
  - Follow standard scale
  - Grades are rounded up to nearest whole number
  - C- is rounded up to a C
    - Need a 69.1% to pass

Letter Grade	Percentage	GPA
A+	97–100%	4.0
A	93–96%	3.9
A–	90–92%	3.7
B+	87–89%	3.3
B	83–86%	3.0
B–	80–82%	2.7
C+	77–79%	2.3
C	73–76%	2.0
C–	70–72%	1.7
D+	67–69%	1.3
D	63–66%	1.0
D–	60–62%	0.7
F	0–59%	0.0

from: https://en.wikipedia.org/wiki/Academic_grading_in_the_United_States

Announcements

- Dealing with missing HW 5
  - Originally all HW and midterm were worth 10% each (total 60%)
  - I will redistribute HW 5 points either to your midterm grade or your average HW grade. Whichever one is higher
  - If your HW average is higher than your midterm, then HW average is 50% of your grade
  - If your midterm average is higher, then your midterm is worth 20% of your grade

No quiz from last time

Review

DOALL Loops

DOALL Loops

- requires that loop iterations are independent
  - You can do the loop iterations in any order and get the same result

are these independent?

```
for (int i = 0; i < 2; i++) {
    counter += 1;
  }
```

vs

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

adds two arrays

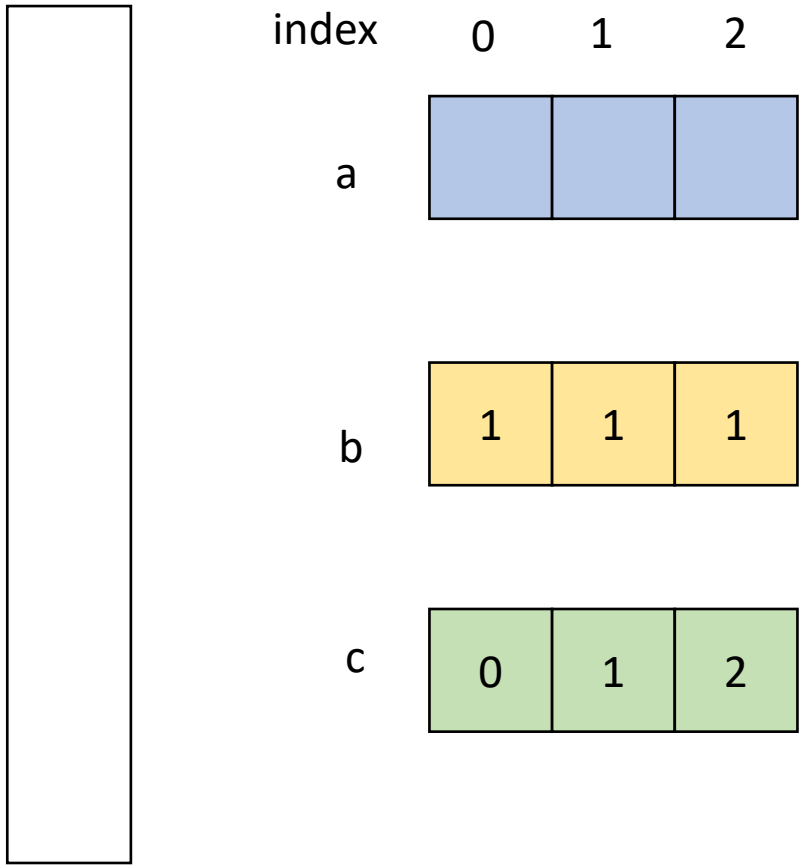
```
for (int i = 0; i < 3; i++) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < 2; i++) {
  a[i] += a[i+1]
}
```

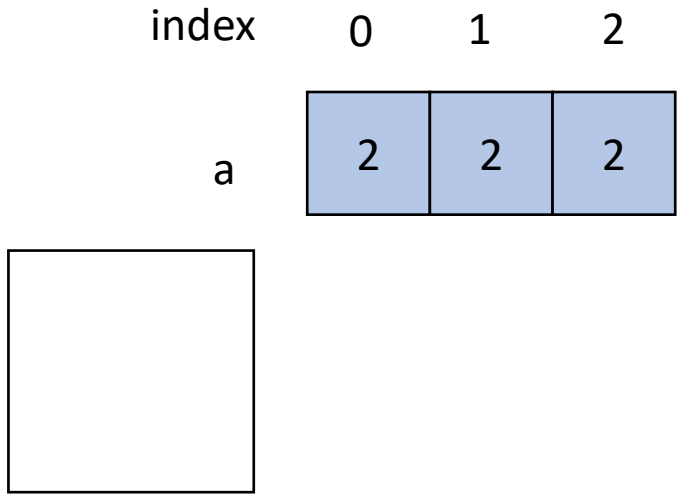
adds two arrays

```
for (int i = 0; i < 3; i++) {
  a[i] = b[i] + c[i];
}
```



adds elements with neighbors

```
for (int i = 0; i < 2; i++) {
  a[i] += a[i+1]
}
```



```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

When loop iterations are independent, they are called **DOALL** loops:

- You can do them in ANY order and get the same results
- If a compiler can find a DOALL loop then there are lots of optimizations to apply!

Safety Criteria: independent iterations

- How do we check this?
  - If the property doesn't hold then there exists 2 iterations, such that if they are re-ordered, it causes different outcomes for the loop.
  - **Write-Write conflicts**: two distinct iterations write different values to the same location
  - **Read-Write conflicts**: two distinct iterations where one iteration reads from the location written to by another iteration.

Motivation:

Image processing

Taken from Halide:
A project out of MIT

pretty straight
forward computation
for brightening

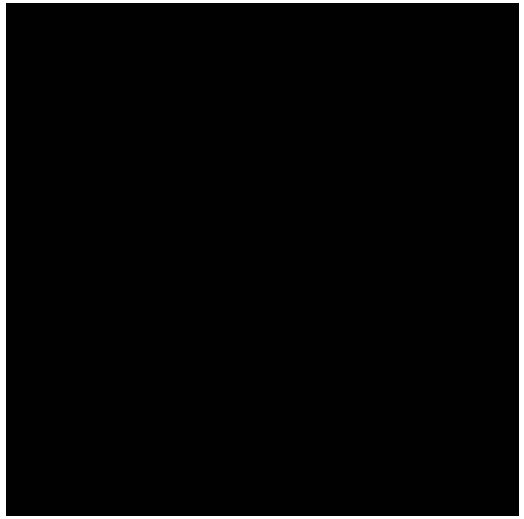
(1 pass over all pixels)

This computation is known as the “Local Laplacian Filter”. Requires visiting all pixels 99 times

We want to be able to do this
fast and efficiently!

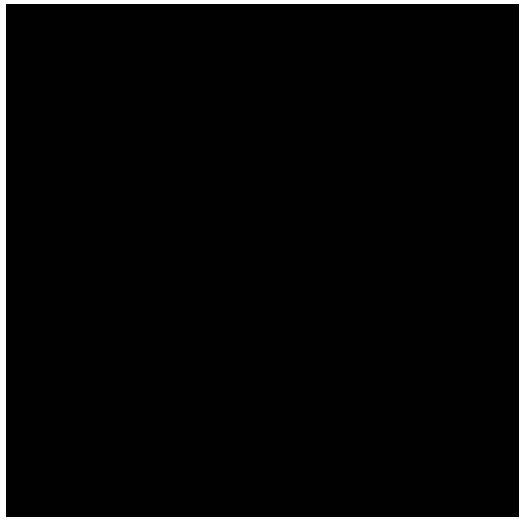
Main results in from an image DSL show
a 1.7x speedup with 1/5 the LoC
over hand optimized versions at Adobe

from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/



```
for (int y = 0; y < 4; y++) {
    for (int x = 0; x < 4; x++) {
        output[y,x] = x + y;
    }
}
```

you can compute the pixels in any order you want, you just have to compute all of them!



```
for (int x = 0; x < 4; x++) {
    for (int y = 0; y < 4; y++) {
        output[y,x] = x + y;
    }
}
```

What is the difference here? What will the difference be?

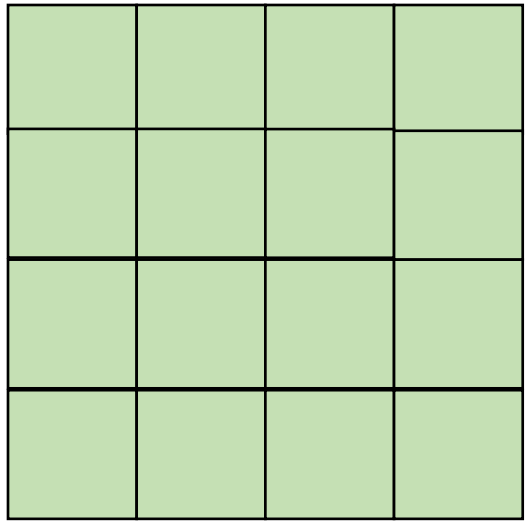
from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

Adding 2D arrays together

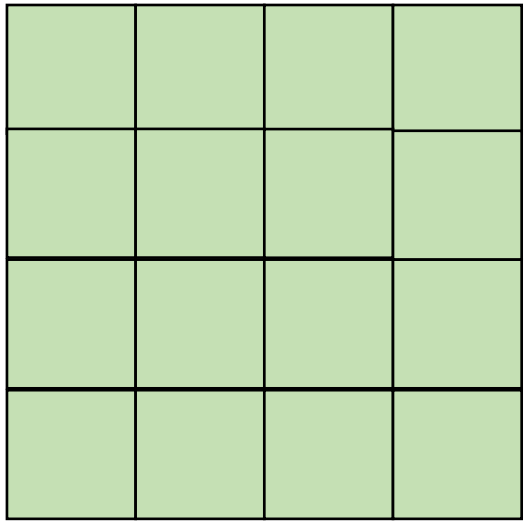
- Memory accesses

$$A = B + C$$

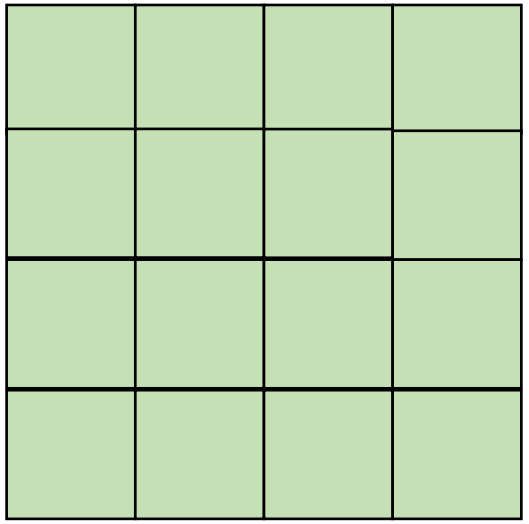
A



B



C



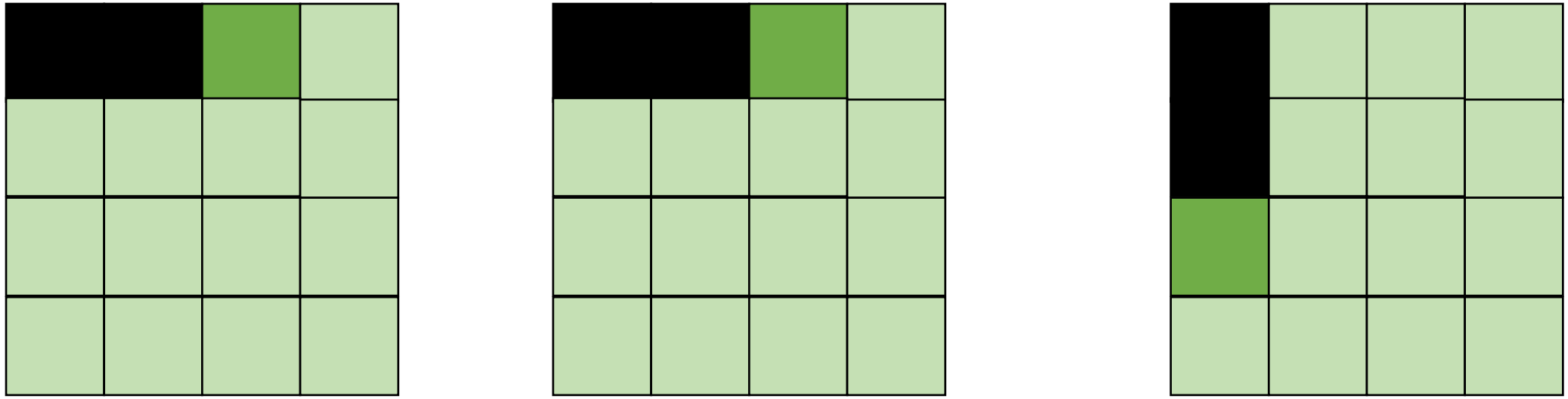
But sometimes there isn’t a good ordering

transposed arrays

- In some cases, there might not be a good nesting order for all accesses:

$$A = B + C^T$$

A B C



Hit on A and B. Miss on C

```
for (int y = 0; y < 4; y++) {
    for (int x = 0; x < 4; x++) {
        output[y,x] = x + y;
    }
}
```

Loop splitting:

```
for (int y = 0; y < 4; y++) {
    for (int x_outer = 0; x_outer < 4; x_outer+=2) {
        for (int x = x_outer; x < x_outer+2; x++) {
            output[y,x] = x + y;
        }
    }
}
```

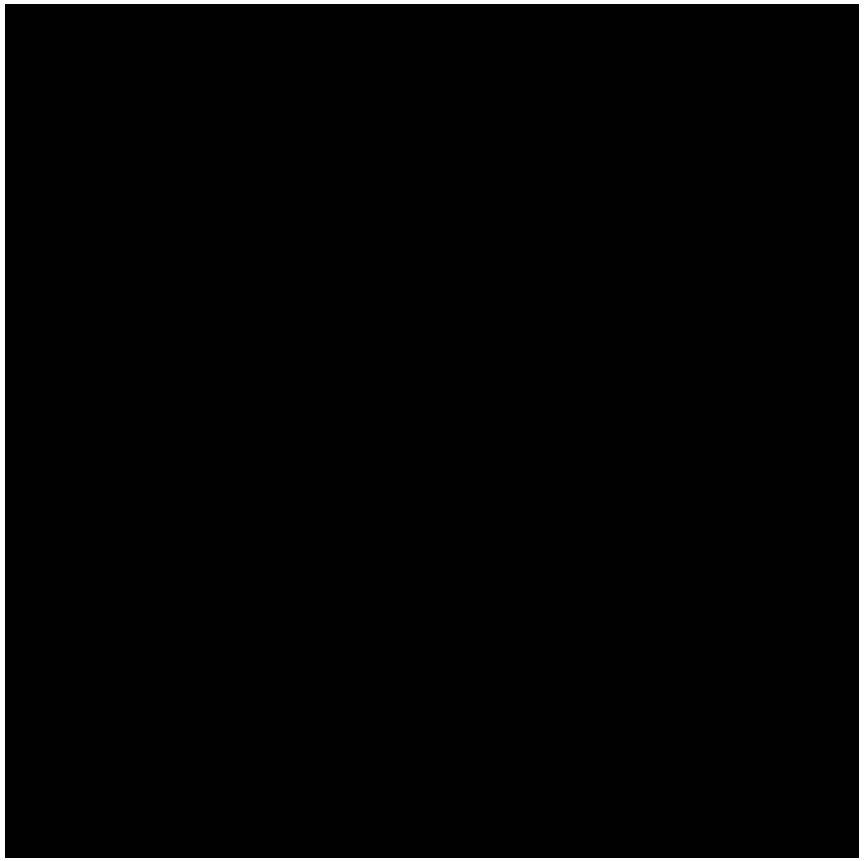
What is the difference here?

from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

Chaining optimizations

- First split the loops then reorder

Our new schedule looks like this:



Why is this beneficial?

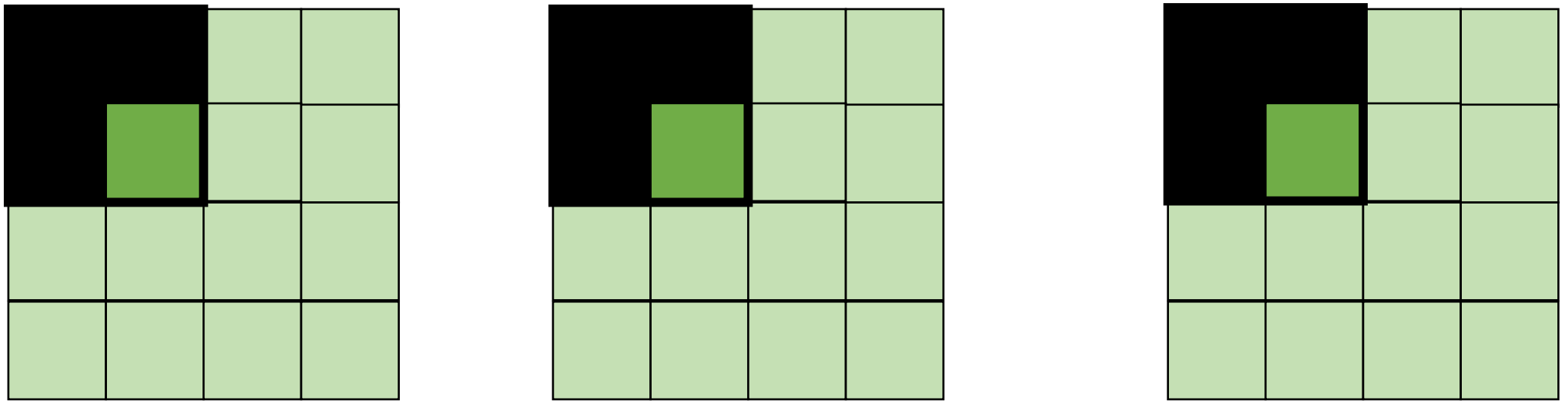
from: https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

blocking

- Blocking operates on smaller chunks to exploit locality in column increment accesses. Example 2x2

$$A = B + C^T$$

A B C



Hit on all!

Loop transformation summary

- If the compiler can prove different properties about your loops, you can automatically make code go a lot faster

- It is hard in languages like C/C++. But in constrained languages (often called domain specific languages (DSLs) it is easier!
  - Hot topic right now for Machine learning, graphics, graph analytics, etc!

Main results in from an image DSL show a 1.7x speedup with 1/5 the LoC over hand optimized versions at Adobe

from: https://people.csail.mit.edu/sparis/publi/2011/siggraph/

Global Optimization (analysis)

- Loop transforms are a regional analysis
  - Compiler works hard to show that code fits a certain pattern
- Global analysis must account for arbitrary patterns
- Generality costs us! Lots of times these optimizations are not as effective or precise.
- But they can still help...

To finish up the class: Live variable analysis

- Not an analysis to make your code go faster
- An analysis to help warn programmers about potential bugs
- Optimizations that make code go faster are really fun but the reality i that programmers often spend ~70% of their time debugging and testing.
- Compilers can help!!

A new data structure for 3 address code:

- Control flow graph

Control flow graphs

A graph where:

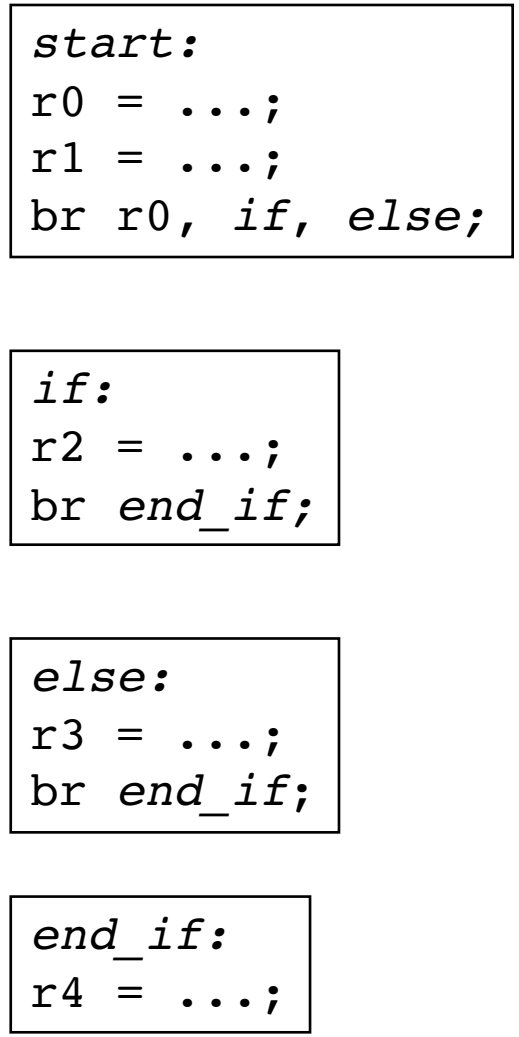
- nodes are basic blocks
- edges mean that it is possible for one block to branch to another

```
start:
r0 = ...;
r1 = ...;
br r0, if, else;

if:
r2 = ...;
br end_if;

else:
r3 = ...;

end_if:
r4 = ...;
```

Control flow graphs

A graph where:

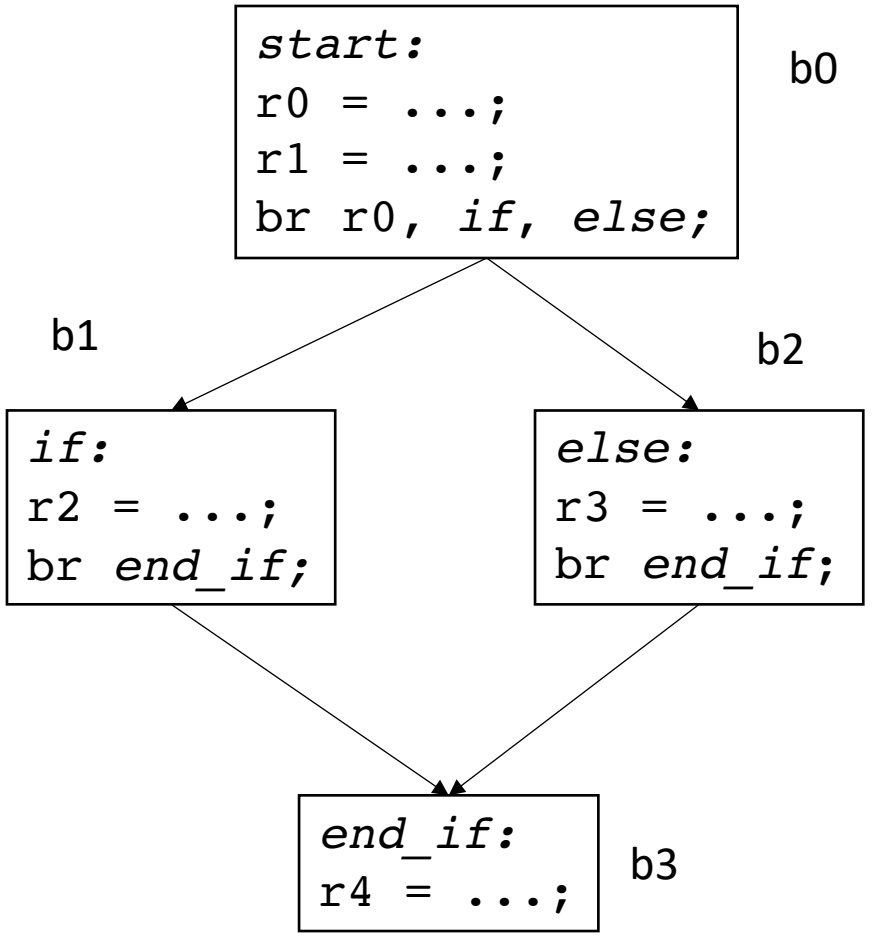
- nodes are basic blocks
- edges mean that it is possible for one block to branch to another



Control flow graphs

A graph where:

- nodes are basic blocks
- edges mean that it is possible for one block to branch to another

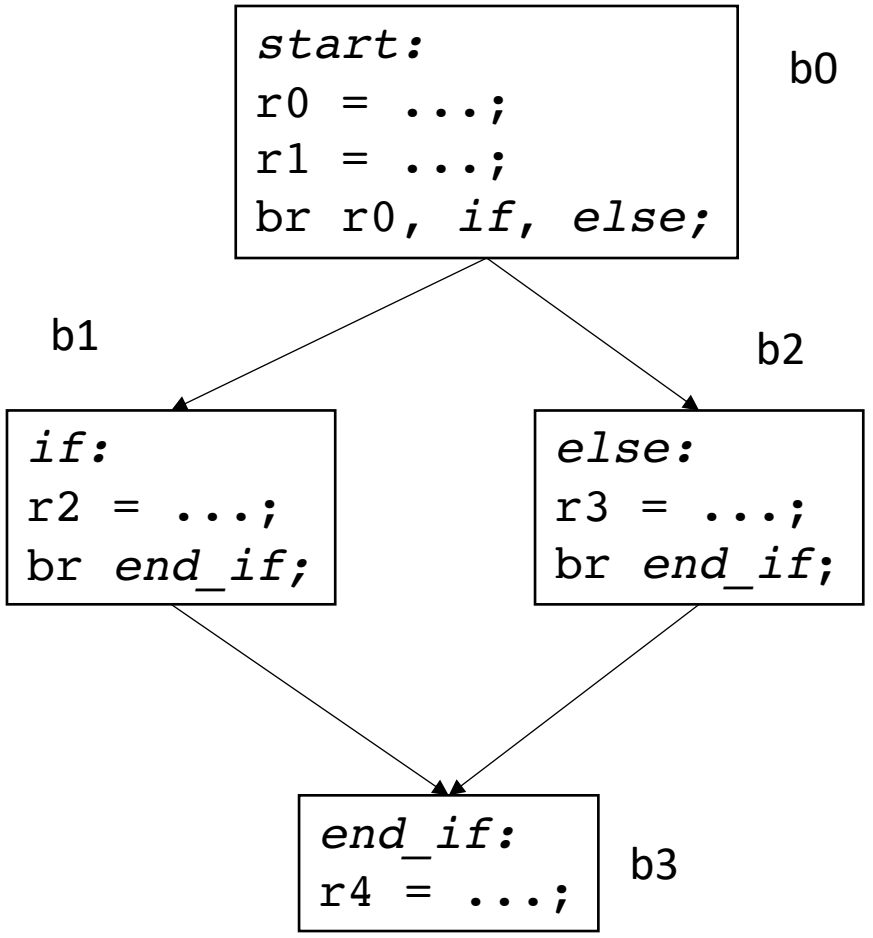


Interesting CFGs

CFGs are easiest to construct over 3 address code.

Labels are explicit and it is easy to partition code into basic blocks

But we can think about CFG patterns from high level code



if/else pattern

Interesting CFGs

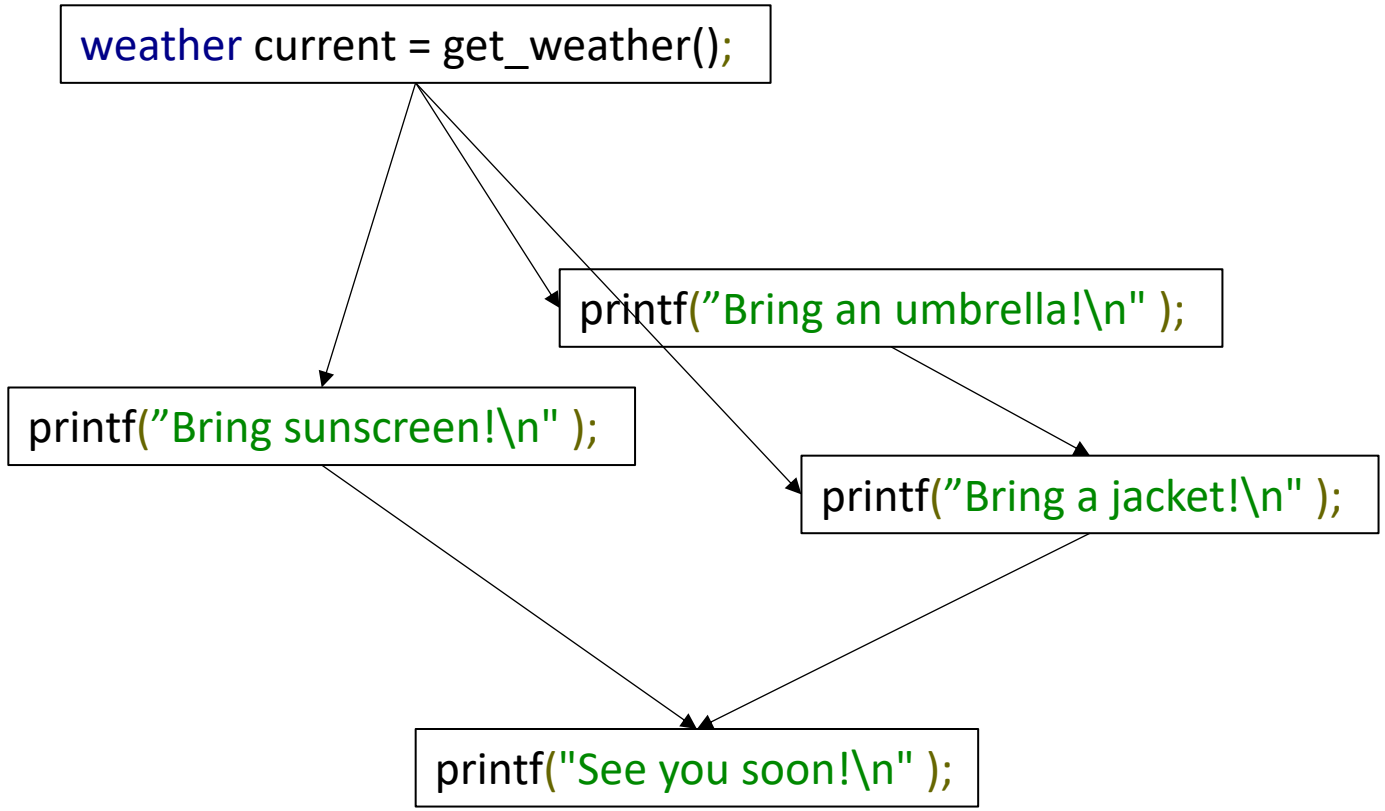
Case statements

```
weather current = get_weather();
switch(current) {
   case SUNNY :
        printf(”Bring sunscreen!\n" );
        break;
  case RAIN :
       printf(”Bring an umbrella!\n" );
  case CLOUDY :
        printf(" Bring a jacket!\n" );
        break;
}
printf("See you soon!\n" );
```

Interesting CFGs

Case statements

```
weather current = get_weather();
switch(current) {
  case SUNNY :
      printf(”Bring sunscreen!\n" );
      break;
  case RAIN :
     printf(”Bring an umbrella!\n" );
  case CLOUDY :
      printf(" Bring a jacket!\n" );
      break;
}
printf("See you soon!\n" );
```



Interesting CFGs

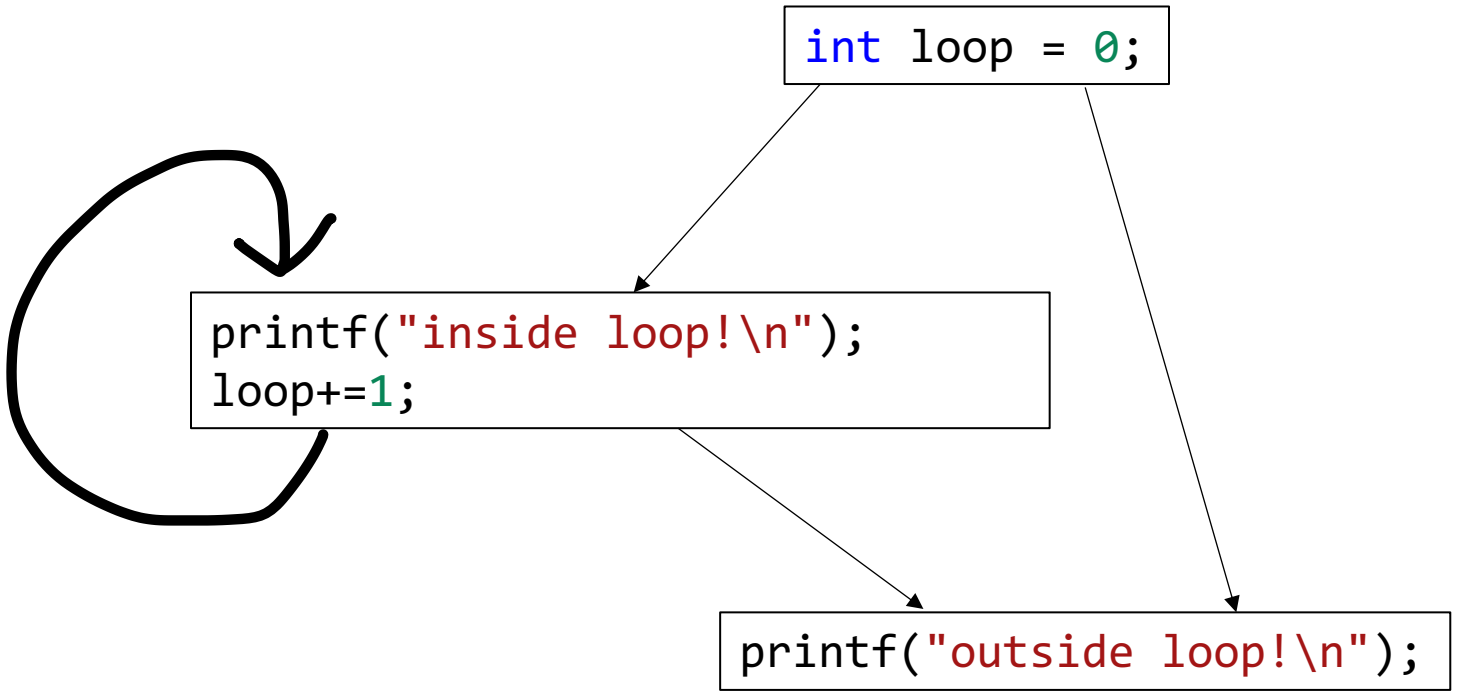
Loops

```
int loop = x;
while(loop!=100) {
  printf("inside loop!\n");
  loop+=1;
}
printf("outside loop!\n");
```

Interesting CFGs

Loops

```
int loop = x;
while(loop!=100) {
  printf("inside loop!\n");
  loop+=1;
}
printf("outside loop!\n");
```



Interesting CFGs

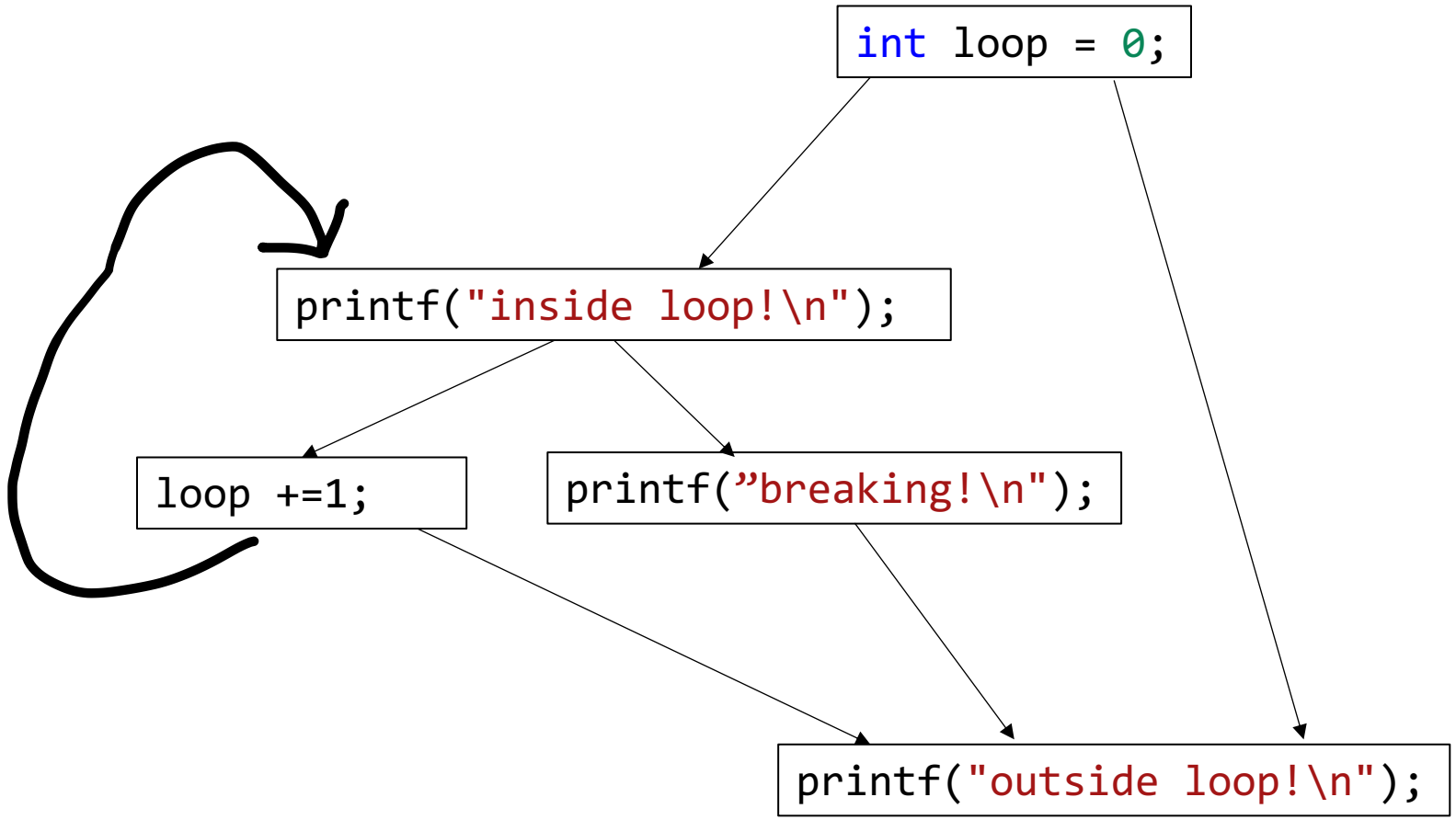
Loops with a break statement

```
int loop = x;
while(loop!=100) {
  printf("inside loop!\n");
  if (loop < 0) {
    printf(”breaking!\n");
    break;
  }
  loop+=1;
}
printf("outside loop!\n");
```

Interesting CFGs

Loops with a break statement

```
int loop = x;
while(loop!=100) {
  printf("inside loop!\n");
  if (loop < 0) {
    printf(”breaking!\n");
    break;
  }
  loop+=1;
}
printf("outside loop!\n");
```



Interesting CFGs

- Other constructs to think about:
  - Exceptions
  - Storing labels in variables

CFG demo

- python demo

Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

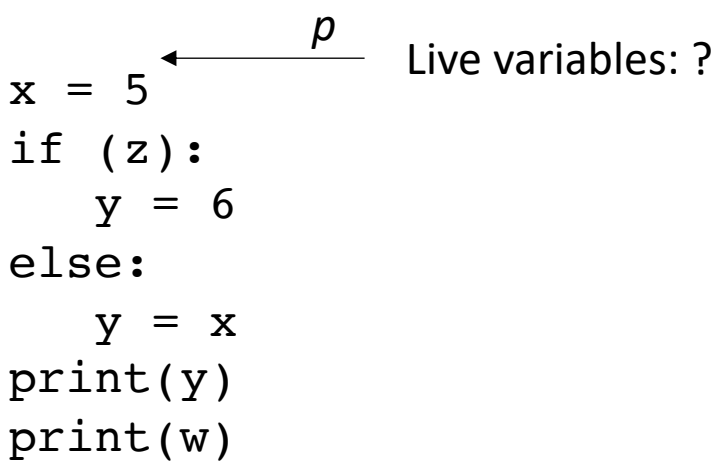
- examples:

```
x = 5
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

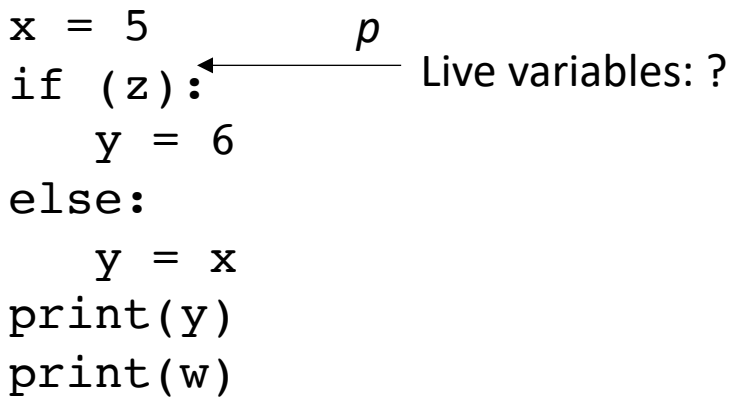


Live variables: ?

Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:



Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

```
x = 5
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

p ← Live variables: ?

Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

p

```
//start
x = 5
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

Live variables: ?

What are live variables at the start of the program?

Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

p

```
//start
x = 5
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

Live variables: ?

What are live variables at the start of the program?

a potential use of an uninitialized variable!

Example

- See code in godbolt

```
int foo(int num) {
  int i;
  int j;
  if (num > 0) {
    i = 5;
    j = 4;
  }
  else {
    i = 6;
  }
  return i + j;
}
```

Example

- See code in godbolt

```
int foo(int num) {
  int i;
  int j;
  if (num > 0) {
    i = 5;
    j = 4;
  }
  else {
    i = 6;
  }
  return i + j;
}
```

Code gives detailed warning in Clang

No warning in gcc

Example

- See code in godbolt

```
int foo(int num) {
  int i;
  int j;
  i = 6;
  return i + j;
}
```

Example

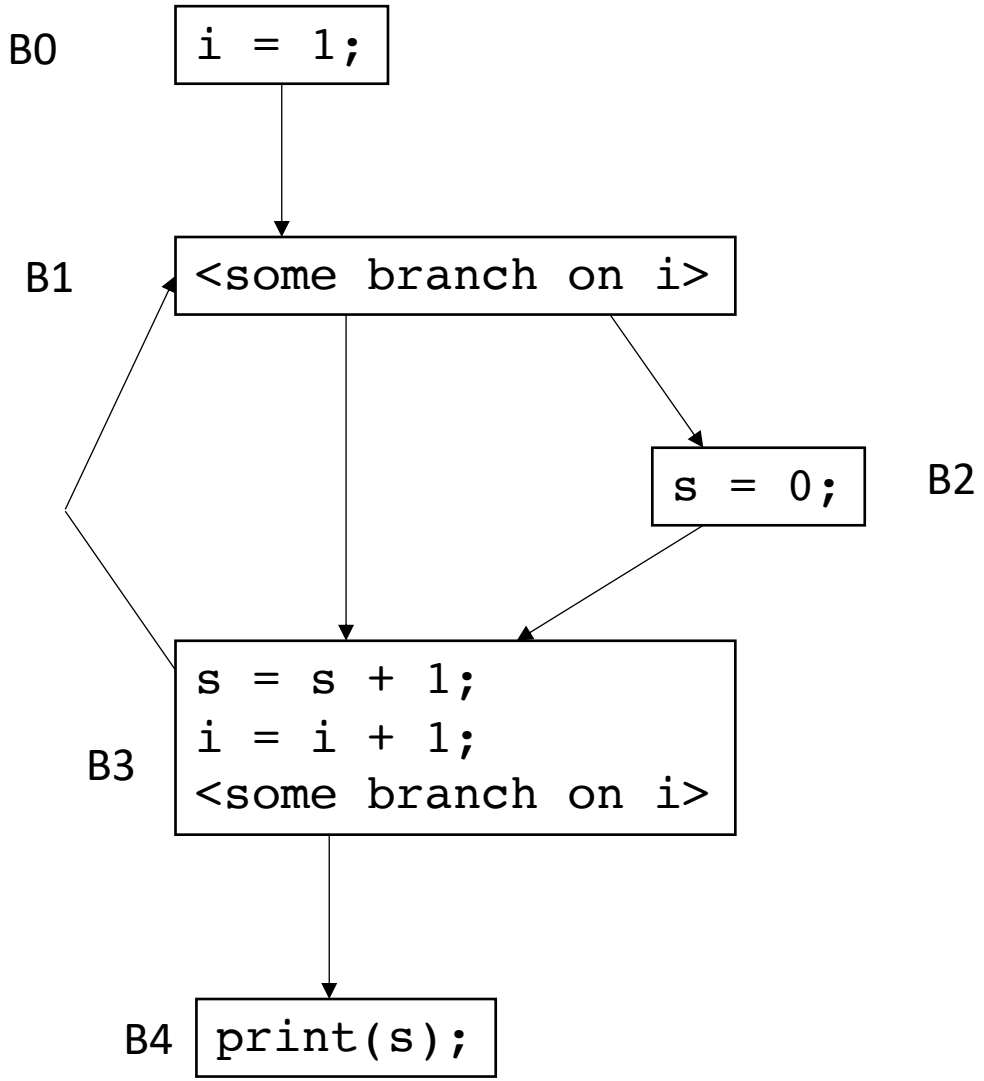
- See code in godbolt

```
int foo(int num) {
  int i;
  int j;
  i = 6;
  return i + j;
}
```

Now code gives warning in gcc

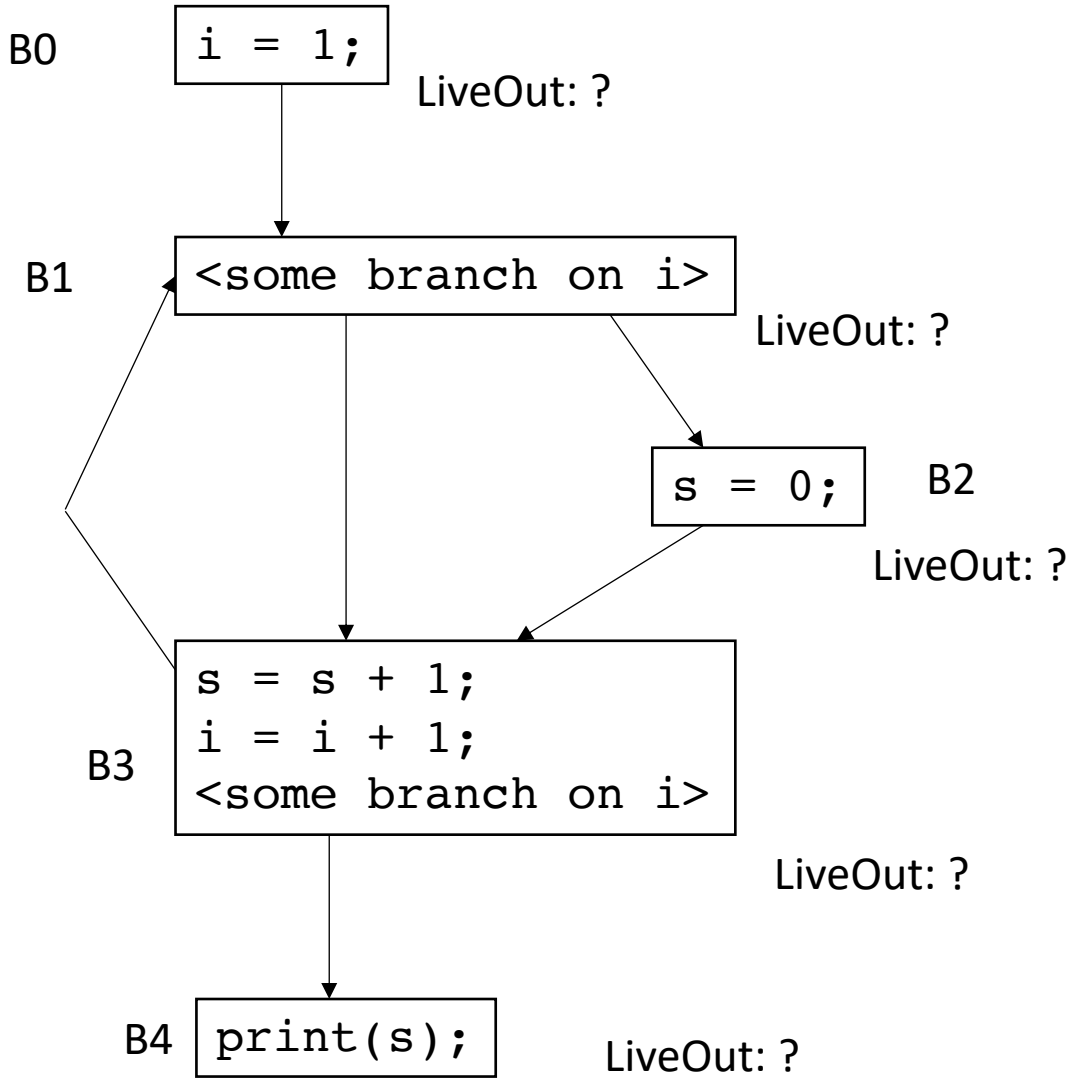
So gcc must only implement their live variable analysis as a local analysis!

Live variable analysis in the CFG:

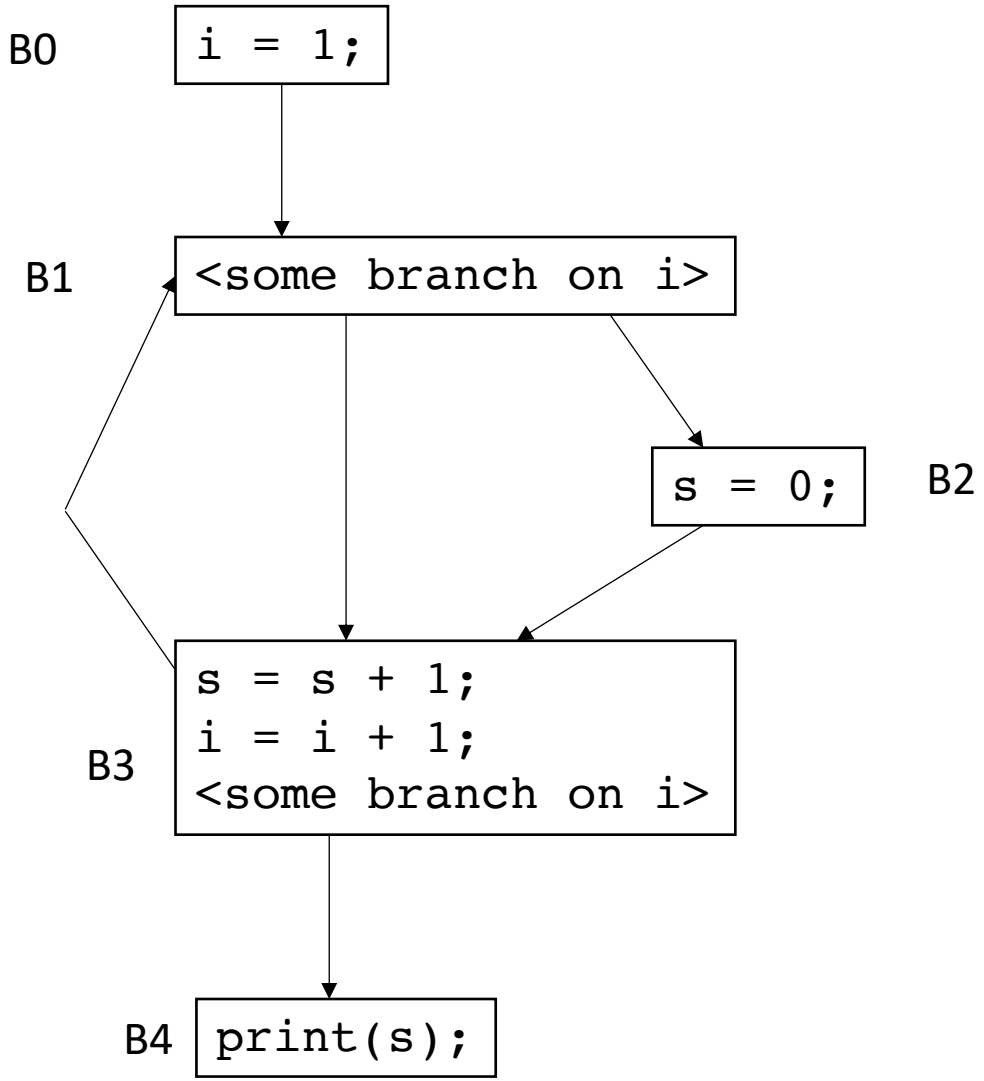


For each block B_x : we want to compute LiveOut:
The set of variables that are live at the end of B_x

Live variable analysis in the CFG:



Live variable analysis in the CFG:



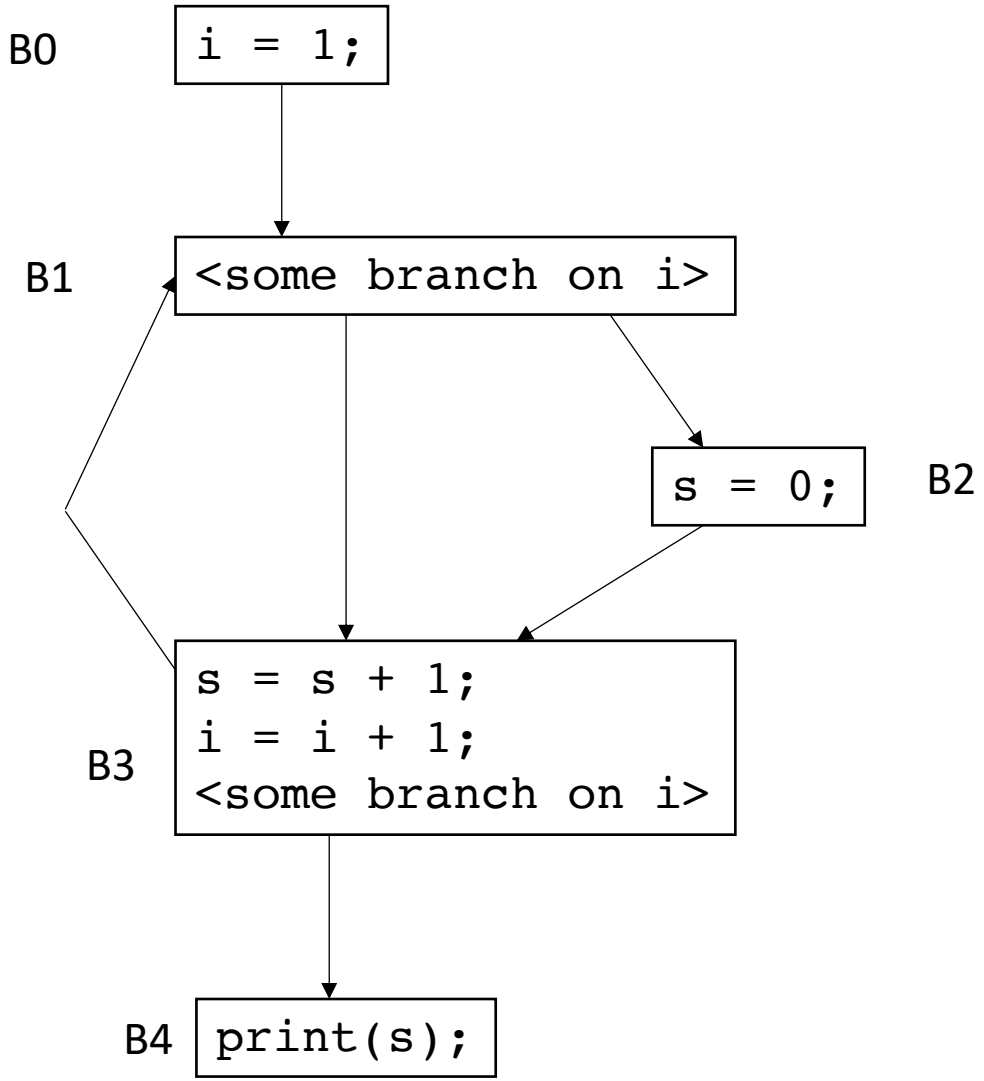
To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten

Block	VarKill	UEVar
B0		
B1		
B2		
B3		
B4		

Live variable analysis in the CFG:



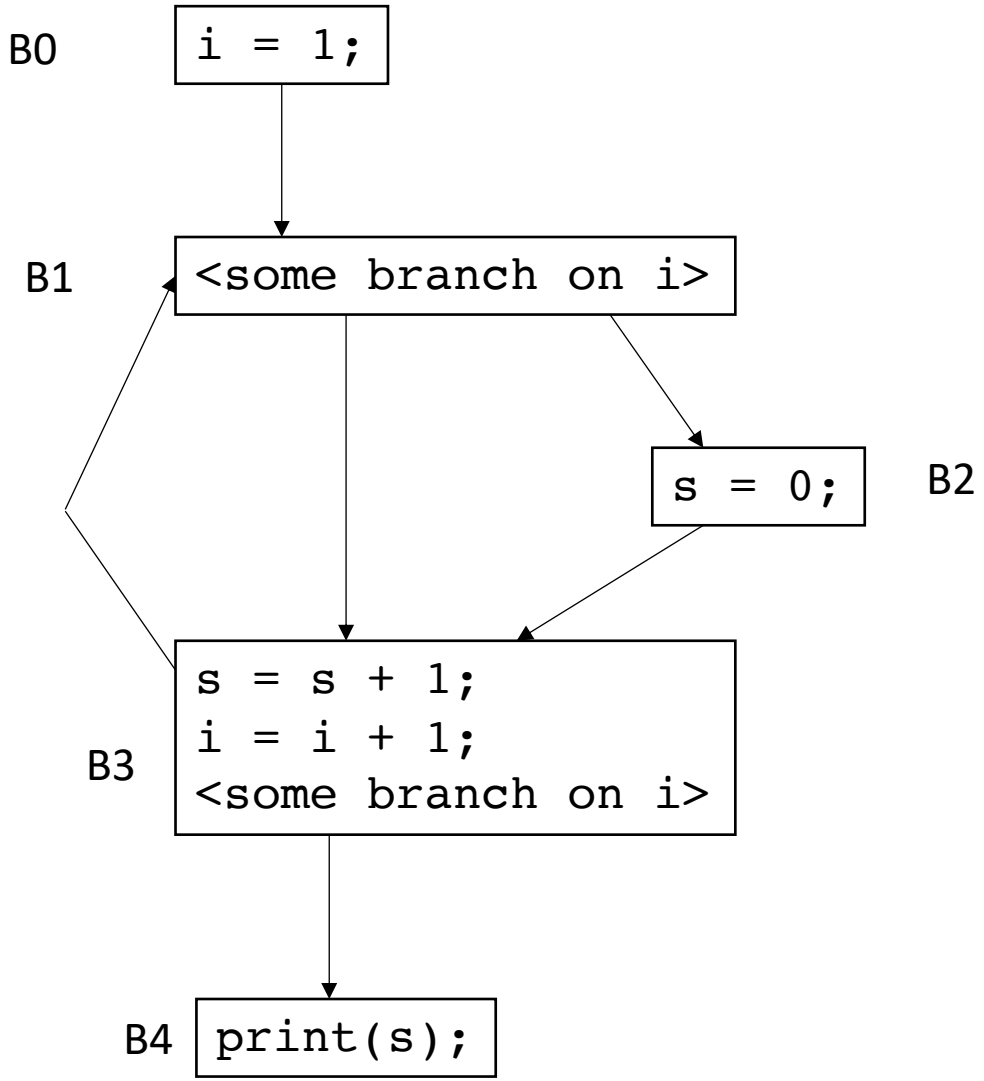
To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten

Block	VarKill	UEVar
B0	i	
B1	{}	
B2	s	
B3	s,i	
B4	{}	

Live variable analysis in the CFG:



To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten

Block	VarKill	UEVar
B0	i	{}
B1	{}	i
B2	s	{}
B3	s,i	s,i
B4	{}	s

Live variable analysis in the CFG:

- Initial condition: LiveOut(n) = {} for all nodes
  - Ground truth, no variables are live at the exit of the program, i.e. end node n_{end} has LiveOut(n_{end})= {}

Live variable analysis in the CFG:

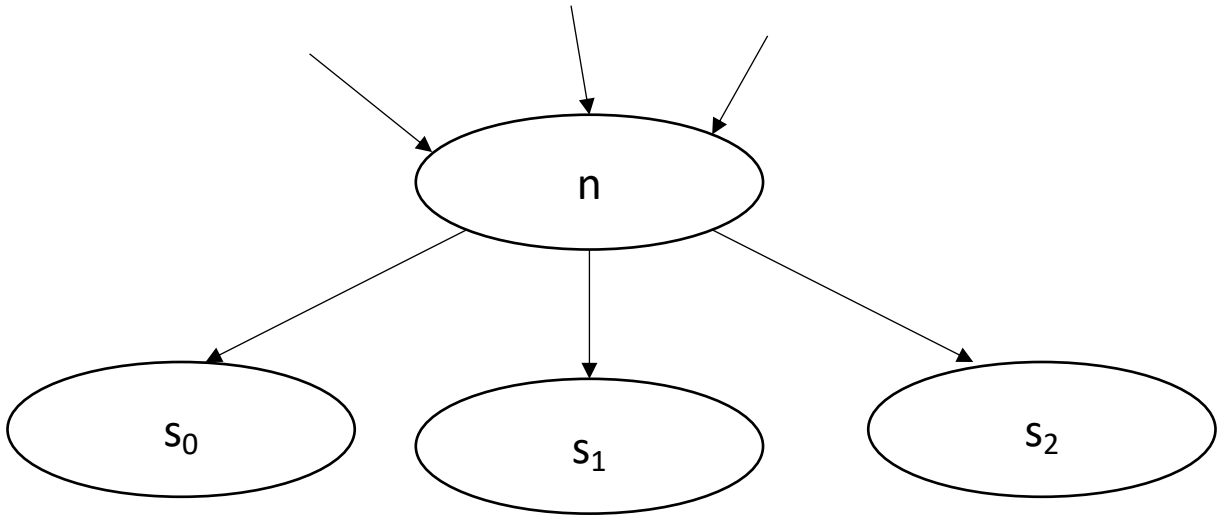
- Initial condition: LiveOut(n) = {} for all nodes
  - Ground truth, no variables are live at the exit of the program, i.e. end node n_{end} has LiveOut(n_{end})= {}

Now we can perform the iterative fixed point computation:

$$\textit{LiveOut(n)} = \cup_{s \text{ in } succ(n)} (\textit{UEVar(s)} \cup (\text{LiveOut(s)} \cap \overline{\textit{VarKill(s)}}))$$

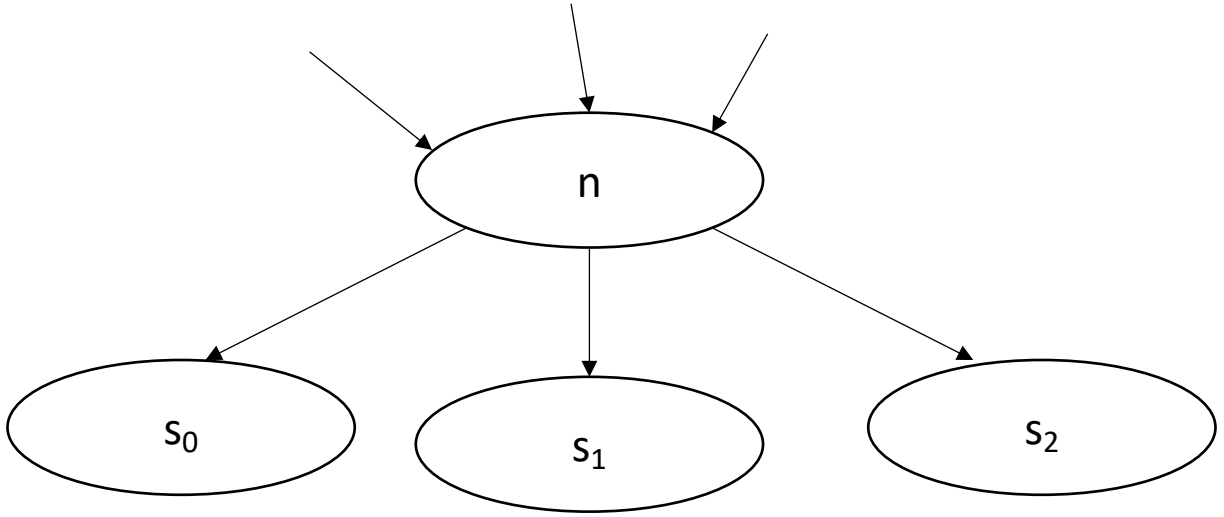
Live variable analysis in the CFG:

$$LiveOut(n) = \cup_{s \text{ in } succ(n)} (UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}))$$



Live variable analysis in the CFG:

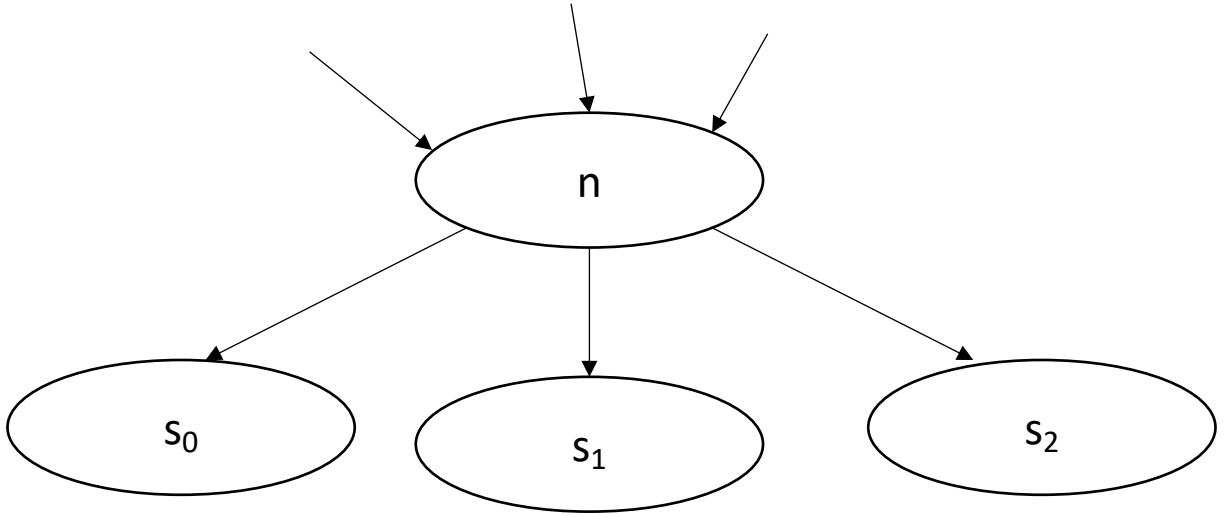
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} (UEVar(s) \cup (LiveOut(s) \cap \overline{VarKill(s)}))$$



any variable in UEVar(s) is live at n

Live variable analysis in the CFG:

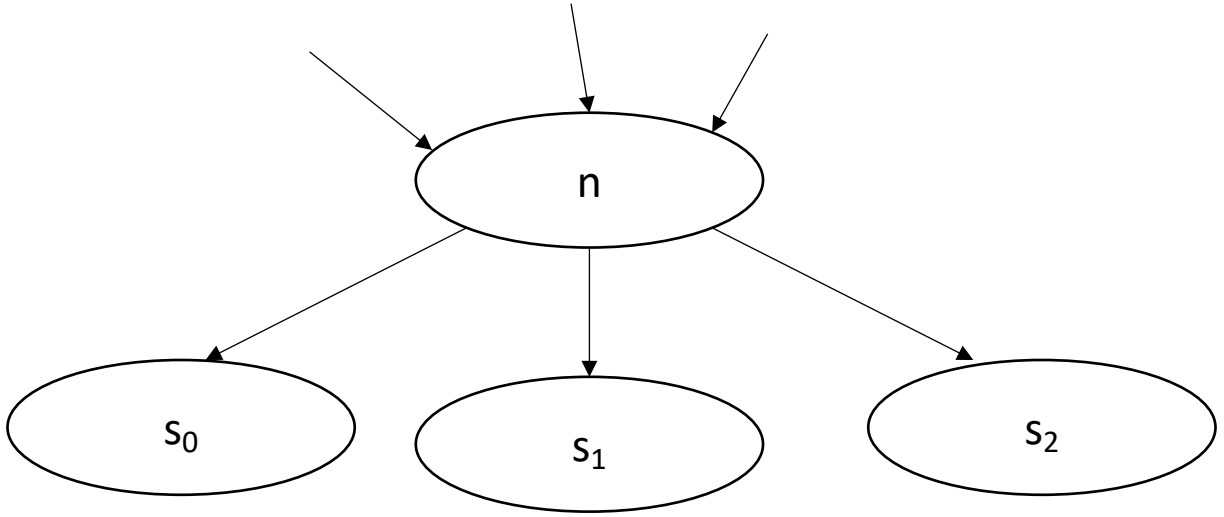
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} (UEVar(s) \cup (LiveOut(s) \cap \overline{VarKill(s)}))$$



variables that are not overwritten in s

Live variable analysis in the CFG:

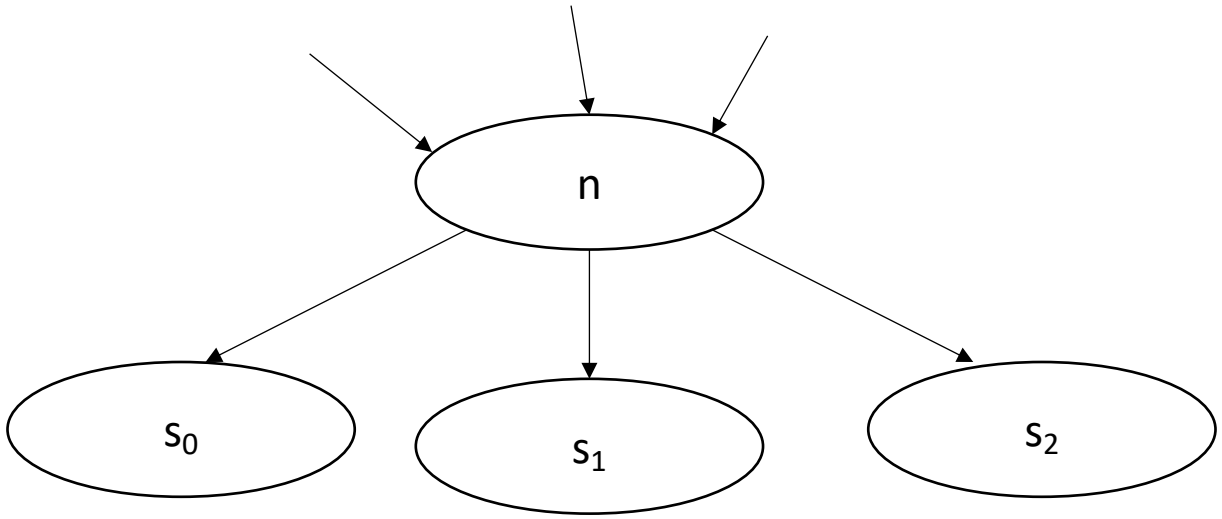
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} (UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}))$$



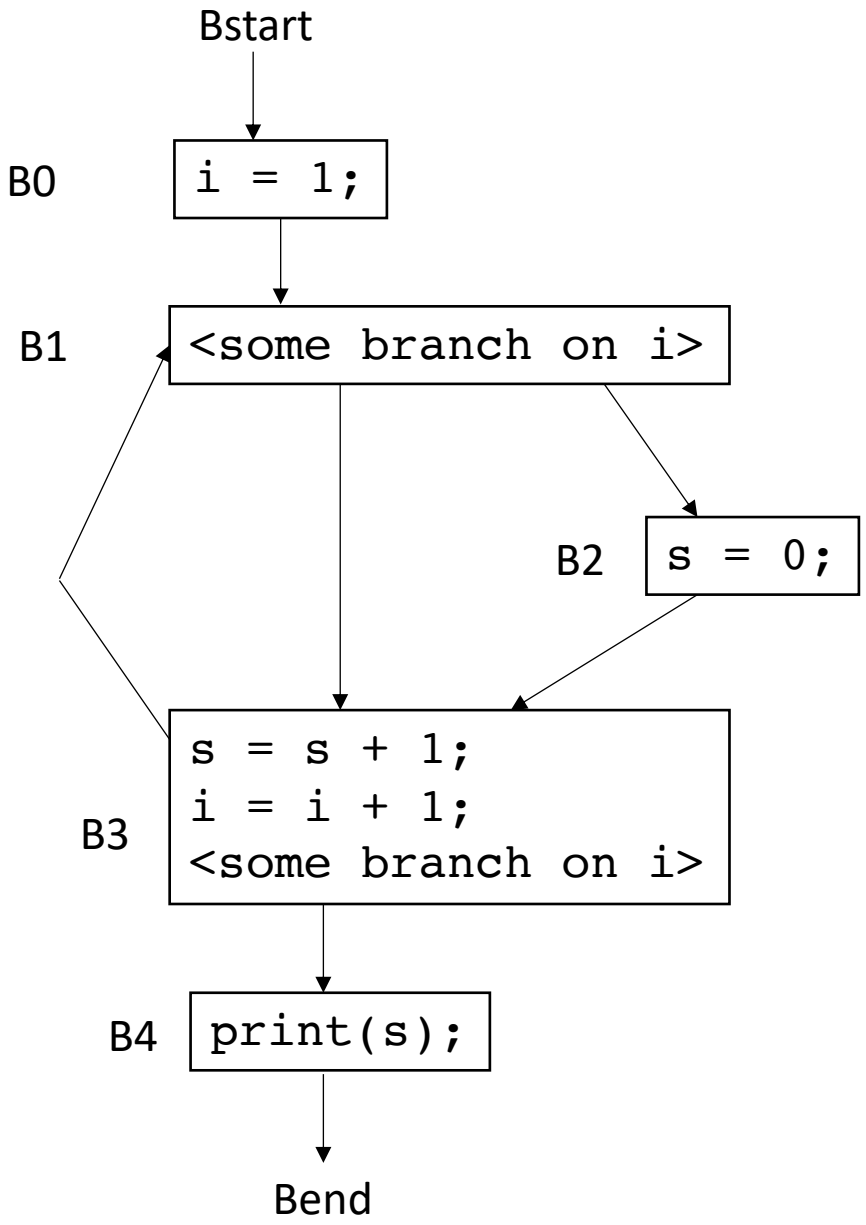
variables that are live at the end of s

Live variable analysis in the CFG:

$$LiveOut(n) = \cup_{s \text{ in } succ(n)} (UEVar(s) \cup (LiveOut(s) \cap \overline{VarKill(s)}))$$



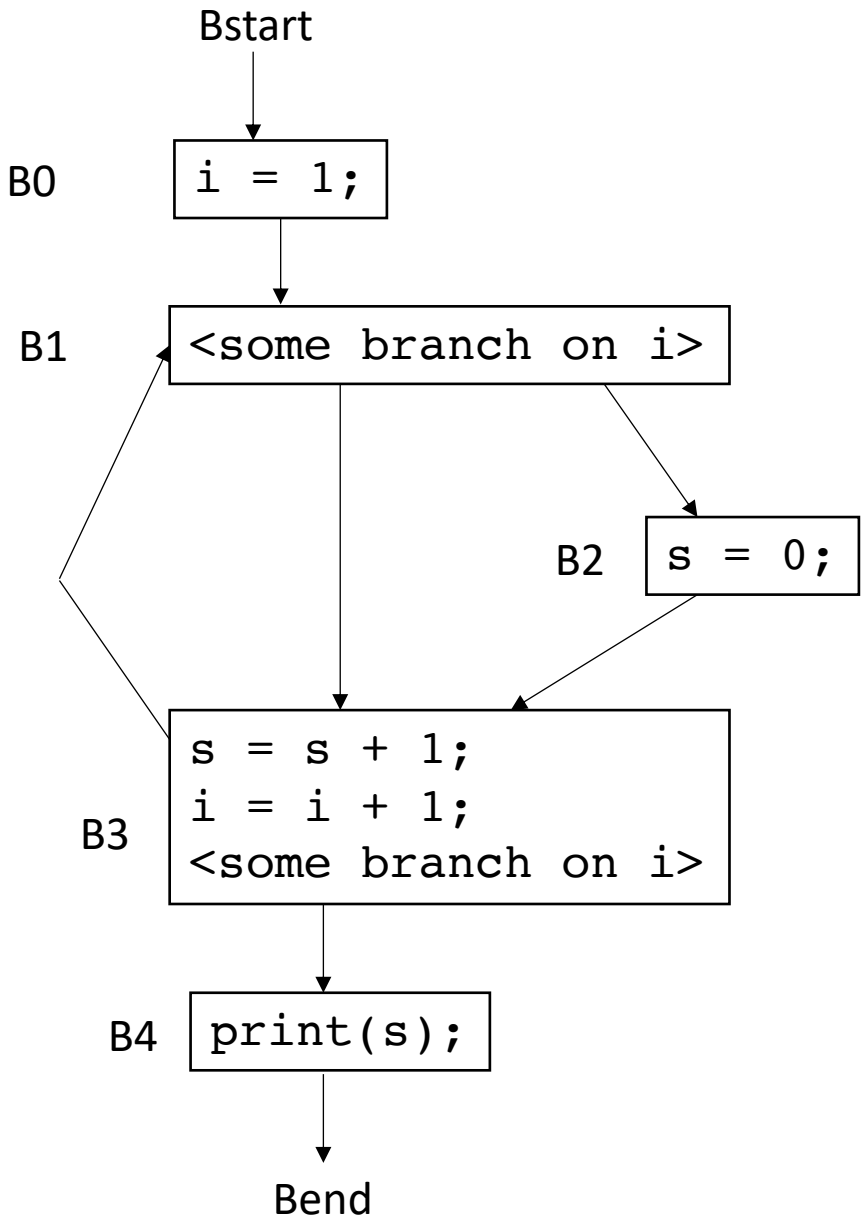
variables that are live at the end of s, and not overwritten by s



Now we can perform the iterative fixed point computation:

$$LiveOut(n) = \cup_{s \text{ in succ}(n)} (UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}))$$

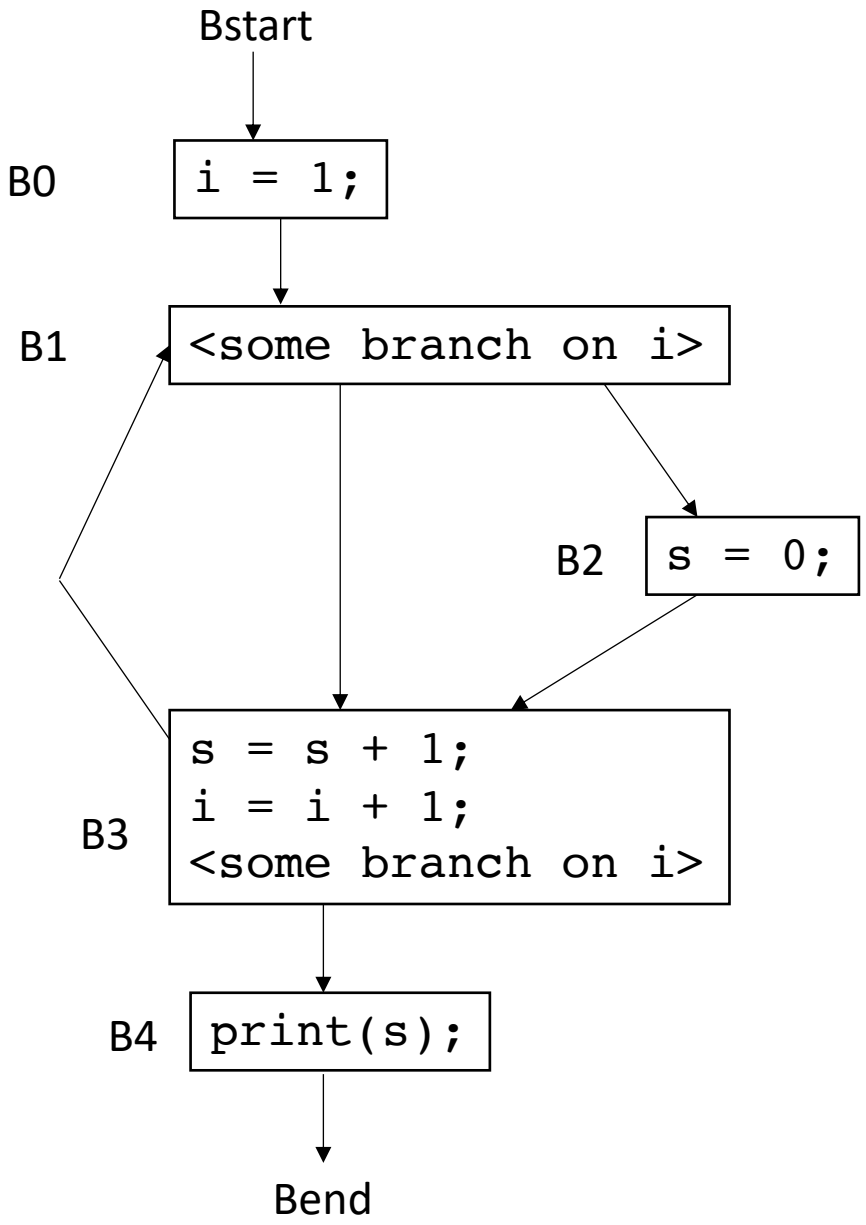
Block	VarKill	UEVar	~VarKill	LiveOut I_0
Bstart	{}	{}	i,s	{}
B0	i	{}	s	{}
B1	{}	i	i,s	{}
B2	s	{}	i	{}
B3	i,s	i,s	{}	{}
B4	{}	s	i,s	{}
Bend	{}	{}	i,s	{}



Now we can perform the iterative fixed point computation:

$$LiveOut(n) = \cup_{s \text{ in } succ(n)} (UEVar(s) \cup (LiveOut(s) \cap \overline{VarKill(s)}))$$

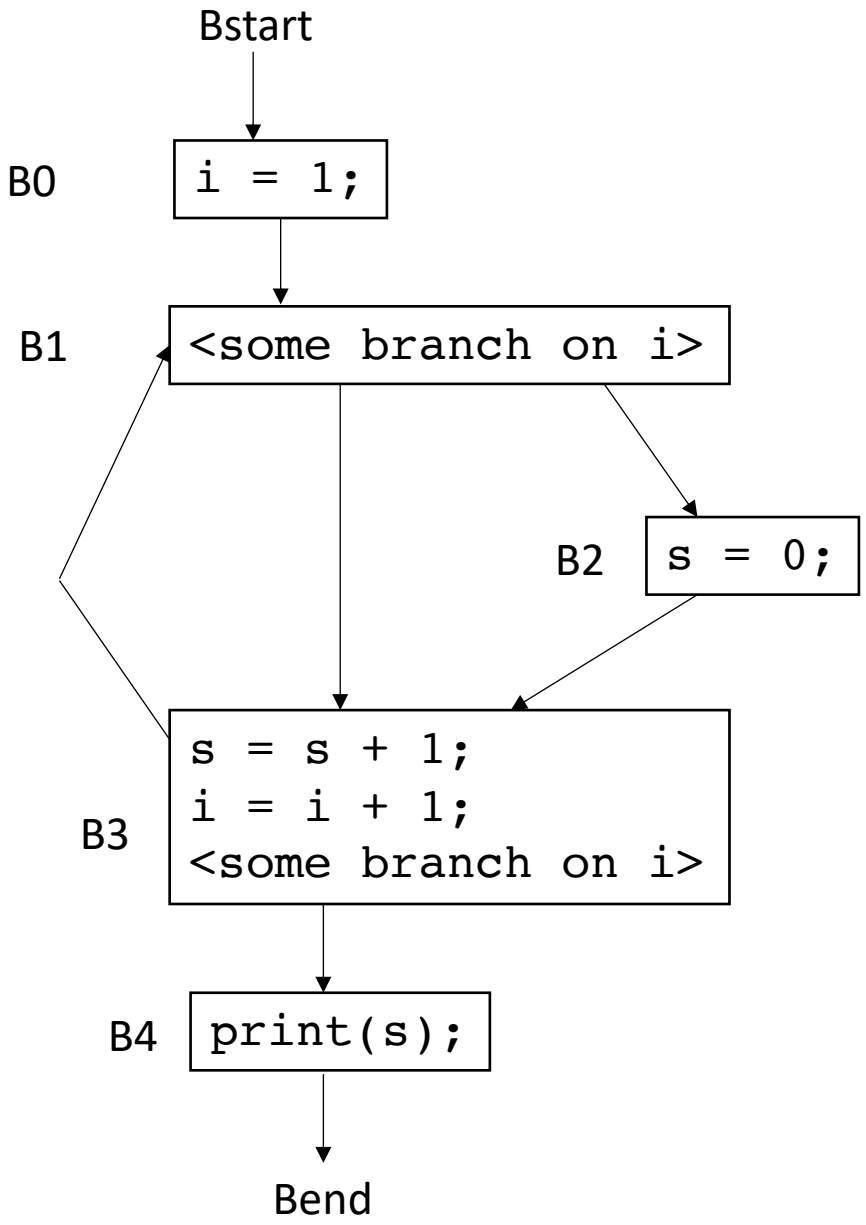
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1
Bstart	{}	{}	i,s	{}	
B0	i	{}	s	{}	
B1	{}	i	i,s	{}	
B2	s	{}	i	{}	
B3	i,s	i,s	{}	{}	
B4	{}	s	i,s	{}	
Bend	{}	{}	i,s	{}	



Now we can perform the iterative fixed point computation:

$$LiveOut(n) = \cup_{s \text{ in } succ(n)} (UEVar(s) \cup (LiveOut(s) \cap \overline{VarKill(s)}))$$

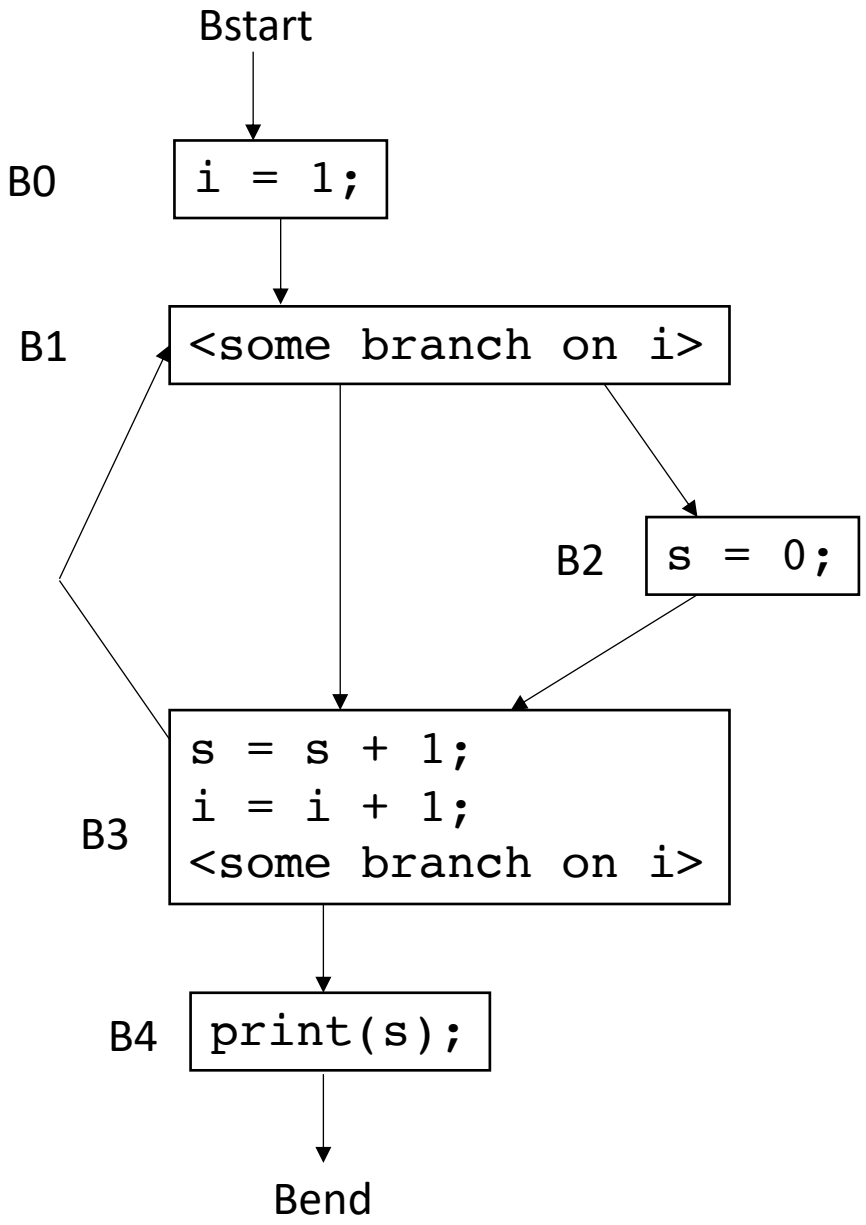
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1
Bstart	{}	{}	i,s	{}	{}
B0	i	{}	s	{}	i
B1	{}	i	i,s	{}	i,s
B2	s	{}	i	{}	i,s
B3	i,s	i,s	{}	{}	i,s
B4	{}	s	i,s	{}	{}
Bend	{}	{}	i,s	{}	{}



Now we can perform the iterative fixed point computation:

$$LiveOut(n) = \cup_{s \text{ in succ}(n)} (UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}))$$

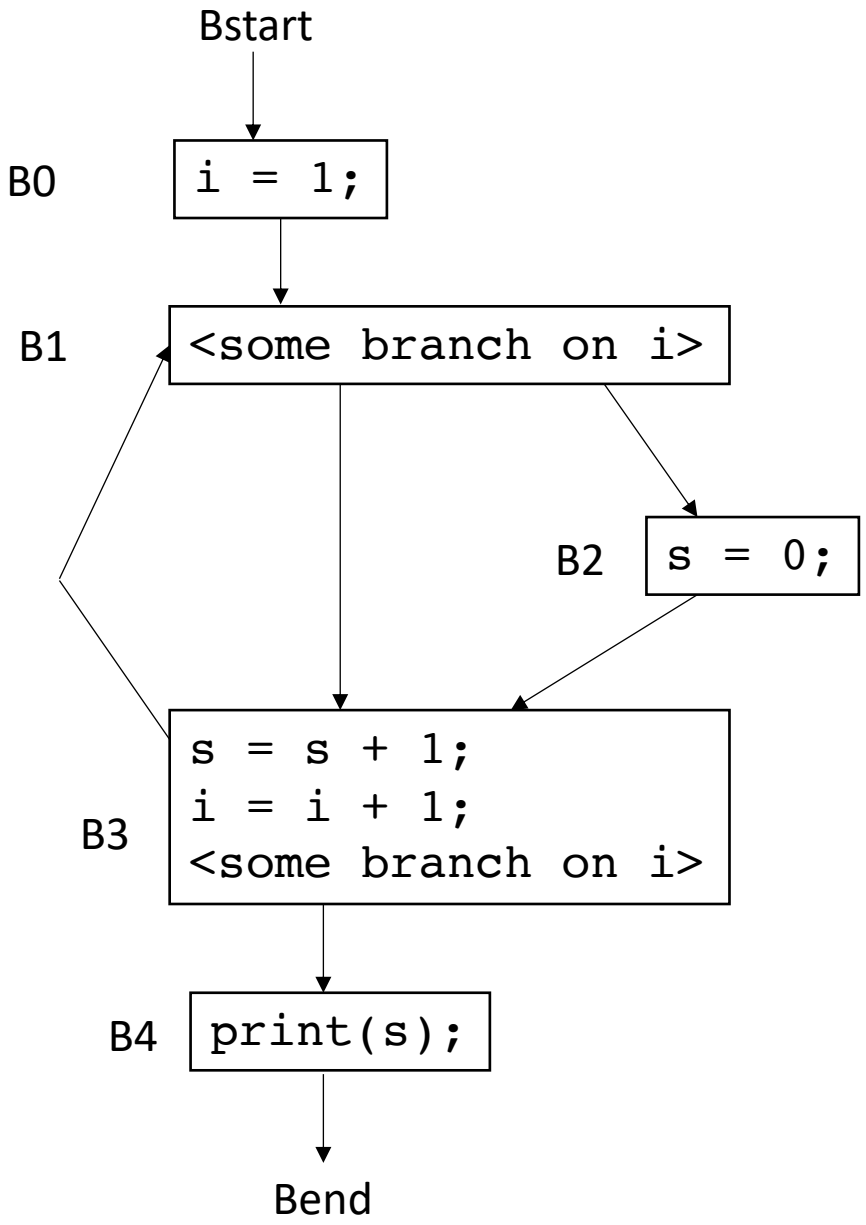
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2
Bstart	{}	{}	i,s	{}	{}	
B0	i	{}	s	{}	i	
B1	{}	i	i,s	{}	i,s	
B2	s	{}	i	{}	i,s	
B3	i,s	i,s	{}	{}	i,s	
B4	{}	s	i,s	{}	{}	
Bend	{}	{}	i,s	{}	{}	



Now we can perform the iterative fixed point computation:

$$LiveOut(n) = \cup_{s \text{ in succ}(n)} (UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}))$$

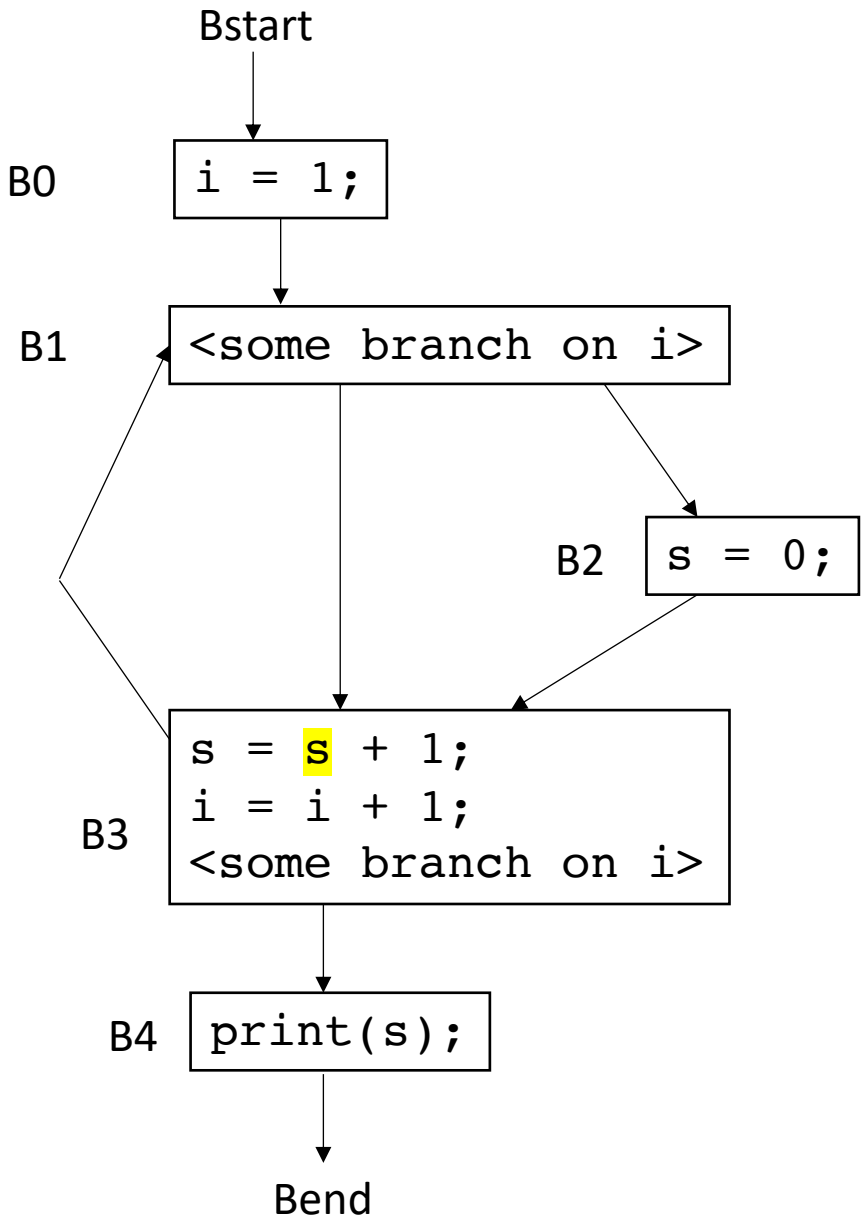
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2
Bstart	{}	{}	i,s	{}	{}	{}
B0	i	{}	s	{}	i	i,s
B1	{}	i	i,s	{}	i,s	i,s
B2	s	{}	i	{}	i,s	i,s
B3	i,s	i,s	{}	{}	i,s	i,s
B4	{}	s	i,s	{}	{}	{}
Bend	{}	{}	i,s	{}	{}	{}



Now we can perform the iterative fixed point computation:

$$LiveOut(n) = \cup_{s \text{ in succ}(n)} (UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}))$$

Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2	.. I_3
Bstart	{}	{}	i,s	{}	{}	{}	
B0	i	{}	s	{}	i	i,s	
B1	{}	i	i,s	{}	i,s	i,s	
B2	s	{}	i	{}	i,s	i,s	
B3	i,s	i,s	{}	{}	i,s	i,s	
B4	{}	s	i,s	{}	{}	{}	
Bend	{}	{}	i,s	{}	{}	{}	

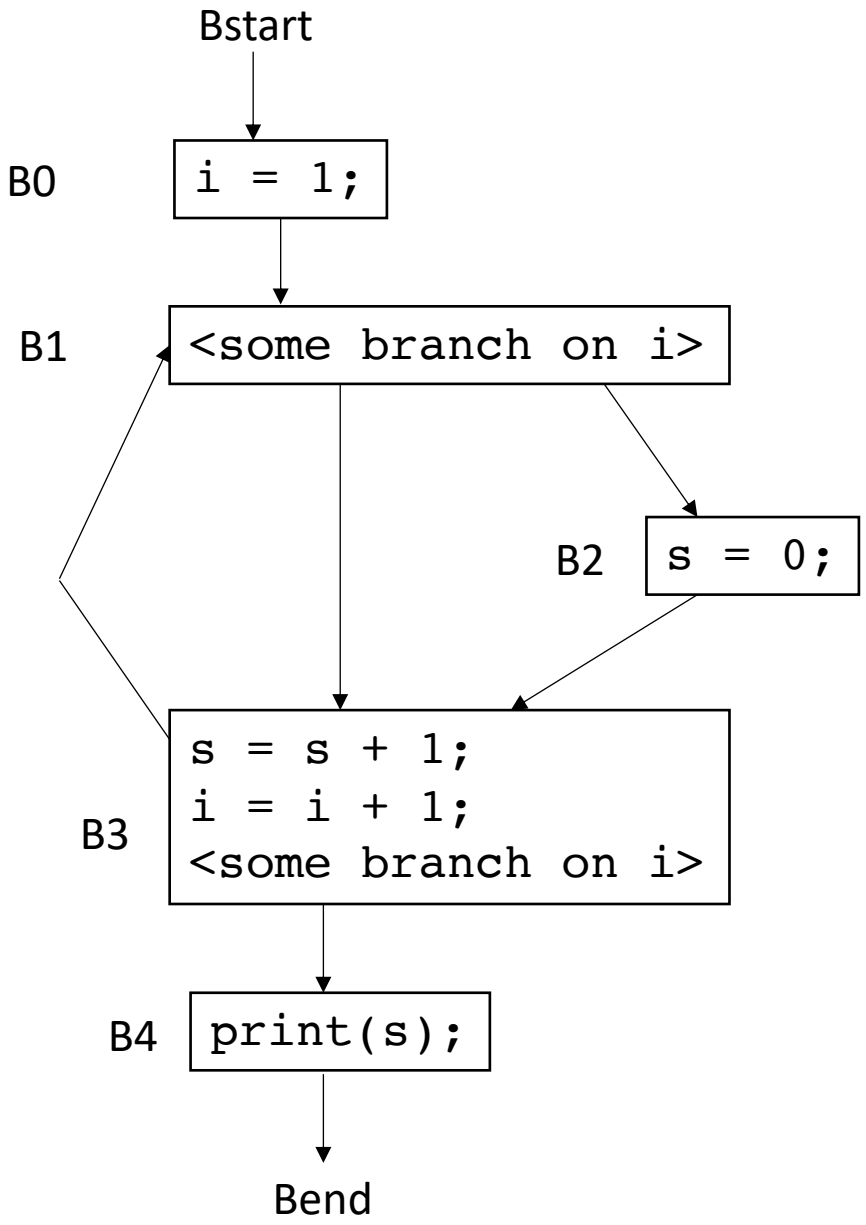


Now we can perform the iterative fixed point computation:

$$LiveOut(n) = \cup_{s \text{ in succ}(n)} (UEVar(s) \cup (LiveOut(s) \cap \overline{VarKill(s)}))$$

Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2	.. I_3
Bstart	{}	{}	i,s	{}	{}	{}	s
B0	i	{}	s	{}	i	i,s	i,s
B1	{}	i	i,s	{}	i,s	i,s	i,s
B2	s	{}	i	{}	i,s	i,s	i,s
B3	i,s	i,s	{}	{}	i,s	i,s	i,s
B4	{}	s	i,s	{}	{}	{}	{}
Bend	{}	{}	i,s	{}	{}	{}	{}

What if we traversed the CFG in a different order?



Now we can perform the iterative fixed point computation:

$$LiveOut(n) = \cup_{s \text{ in succ}(n)} (UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}))$$

Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1
Bstart	{}	{}	i,s	{}	
B0	i	{}	s	{}	
B1	{}	i	i,s	{}	
B2	s	{}	i	{}	
B3	i,s	i,s	{}	{}	
B4	{}	s	i,s	{}	
Bend	{}	{}	i,s	{}	

Lets do it backwards this time

Traversal order in data flow algorithms

- If your analysis flows backwards (get information from your children)
  - You want a post-order traversal
  - visit as many children as possible before visiting the parents
  - live variable analysis is a backwards flow analysis

- If you flow forward, then you want a reverse post order traversal
  - Visit as many parents as possible
  - Global constant propagation is an example

Live variable limitations

To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten.

Consider:

```
s = a[x] + 1;
```

Live variable limitations

To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten.

Consider:

```
s = a[x] + 1;
```

UEVar *needs to assume a[x] is any memory location that it cannot prove non-aliasing*

$$LiveOut(n) = \cup_{s \text{ in } succ(n)} (UEVar(s) \cup (LiveOut(s) \cap \overline{VarKill(s)}))$$

Live variable limitations

To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten.

Consider:

```
a[x] = s + 1;
```

$$LiveOut(n) = \cup_{s \text{ in } succ(n)} (UEVar(s) \cup (LiveOut(s) \cap \overline{VarKill(s)}))$$

Live variable limitations

To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten.

Consider:

```
a[x] = s + 1;
```

VarKill *also needs to know about aliasing*

$$LiveOut(n) = \cup_{s \text{ in } succ(n)} (UEVar(s) \cup (LiveOut(s) \cap \overline{VarKill(s)}))$$

Demo

- Godbolt demo

```
int foo(int num, int x, int y) {
  int i[4];
  int j[4];

  return i[x] + j[y];
}
```

Demo

- Godbolt demo

```
int foo(int num, int x, int y) {
  int i[4];
  int j[4];

  return i[x] + j[y];
}
```

no warning in clang...

warning in gcc

Demo

- Godbolt demo

```
int foo(int num, int x, int y) {
  int i[4];
  int j[4];

  j[0] = 0;
  i[0] = 0;

  return i[x] + j[y];
}
```

Demo

- Godbolt demo

```
int foo(int num, int x, int y) {
  int i[4];
  int j[4];

  j[0] = 0;
  i[0] = 0;

  return i[x] + j[y];
}
```

No more warning.

Thus analysis must not be very precise

Live variable limitations

Imprecision can come from CFG construction:

consider:

```
br 1 < 0, dead_branch, alive_branch
```

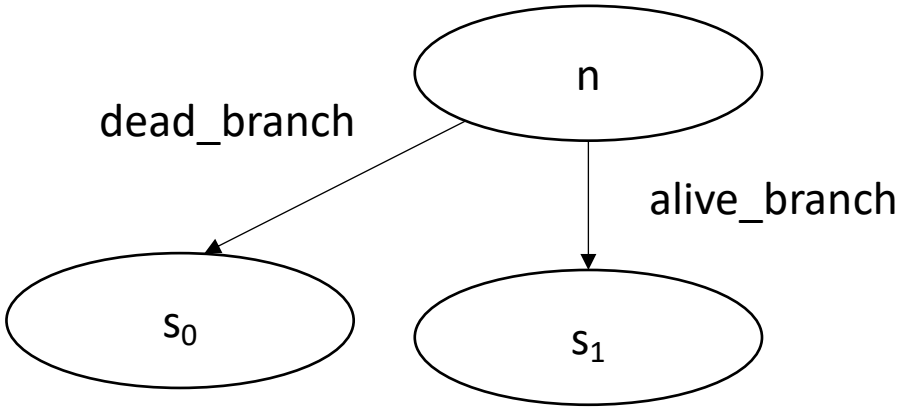
Live variable limitations

Imprecision can come from CFG construction:

consider:

```
br 1 < 0, dead_branch, alive_branch
```

could come from arguments, etc.



Live variable limitations

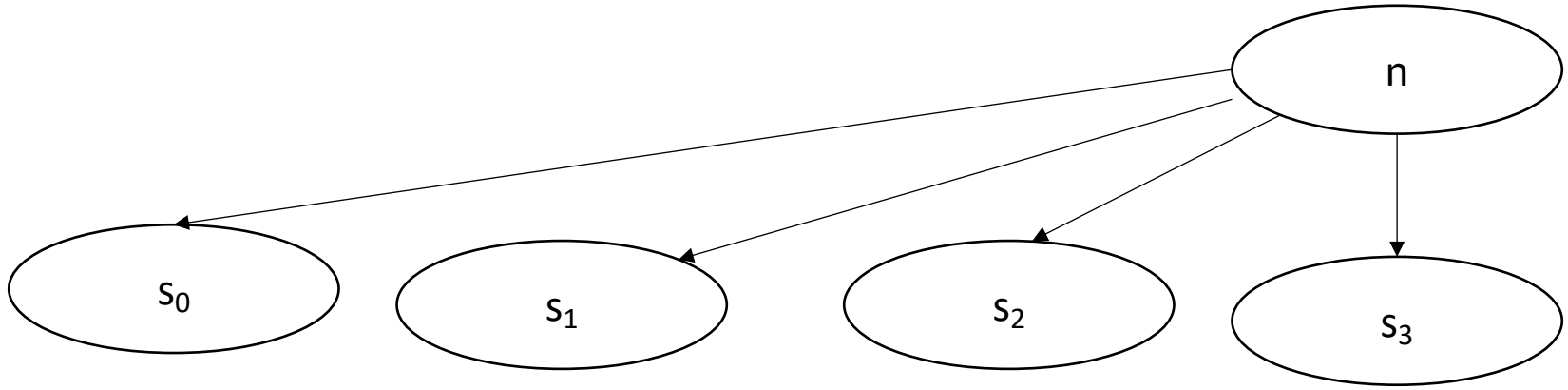
Imprecision can come from CFG construction:

consider first class labels (or functions):

```
br label_reg
```

where label_reg is a register that contains a register

need to branch to all possible basic blocks!



Summary

- Global analysis is difficult and often very imprecise
- Algorithms operate over CFGs and model how information can flow through the CFG
- Live variable analysis can be used to catch potential uses of initialized variables

See everyone on Friday

- Last day of class!