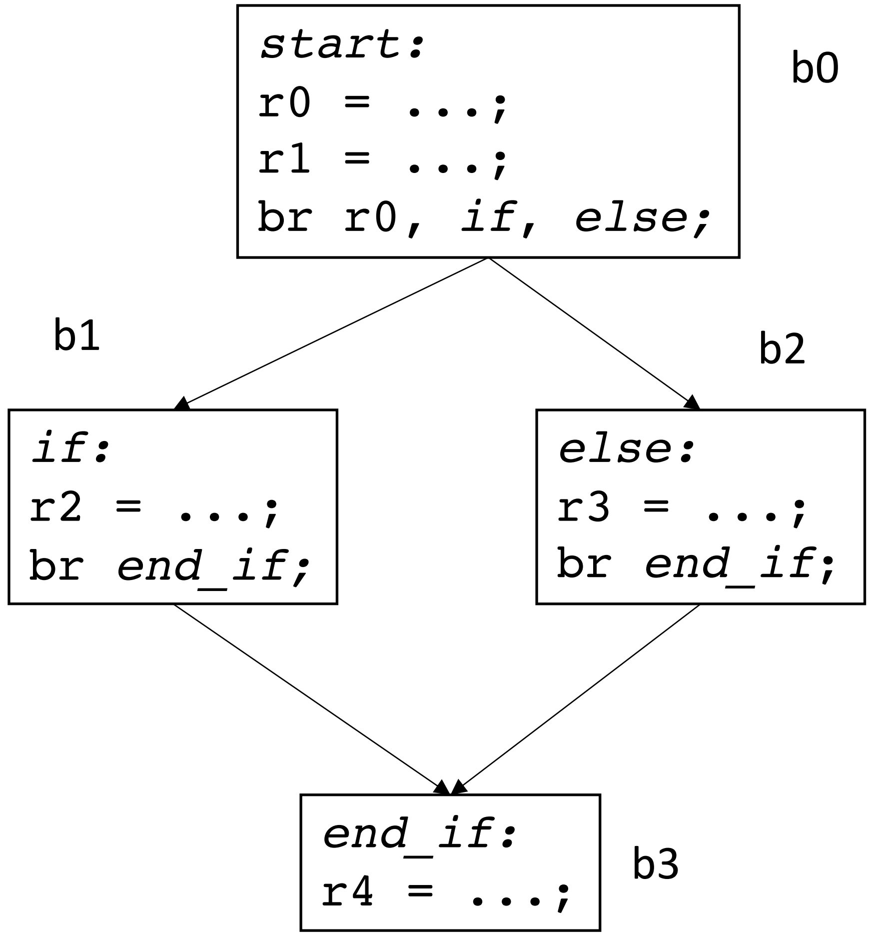
CSE110A: Compilers

June 3, 2022

Topics:

- *Live variable analysis*
- *Class conclusion*



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

```
if:
r2 = ...;
br end_if;
```

```
else:
r3 = ...;
br end_if;
```

```
end_if:
r4 = ...;
```

Announcements

- Homework 3 grades are out!
  - Let us know if there are issues ASAP

- Homework 4 is out
  - 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)

No quiz from last time

Review

Control flow graphs

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

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

```
if:
r2 = ...;
br end_if;
```

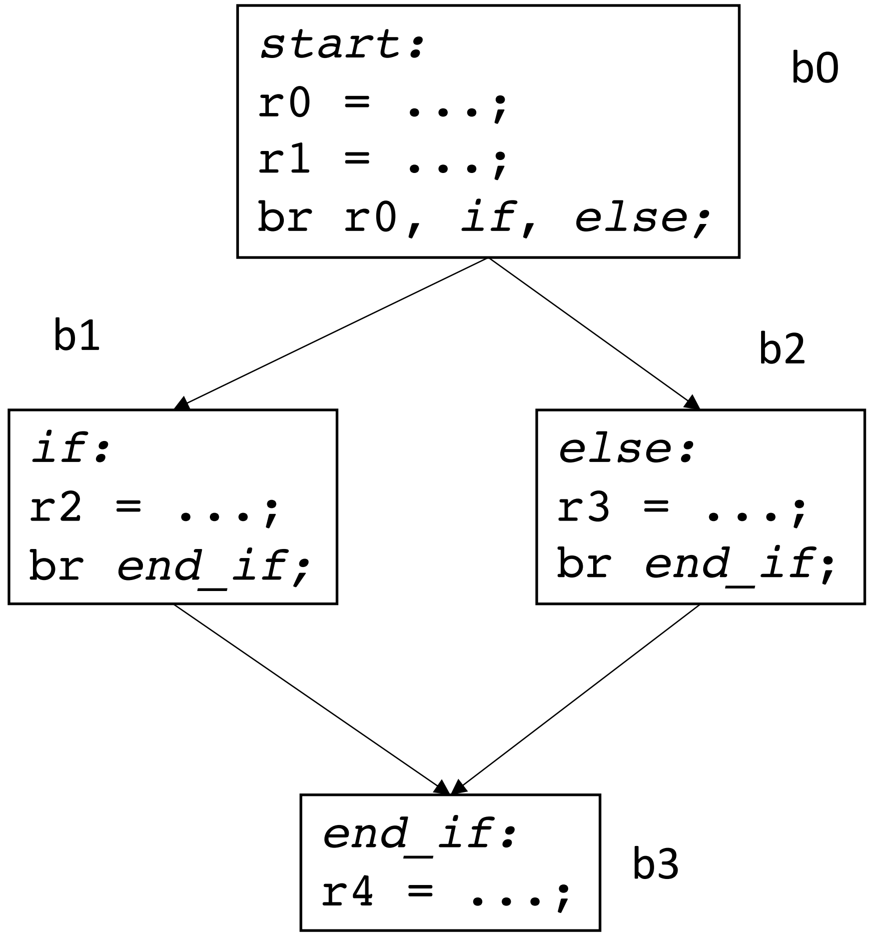
```
else:
r3 = ...;
br end_if;
```

```
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



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

```
if:
r2 = ...;
br end_if;
```

```
else:
r3 = ...;
br end_if;
```

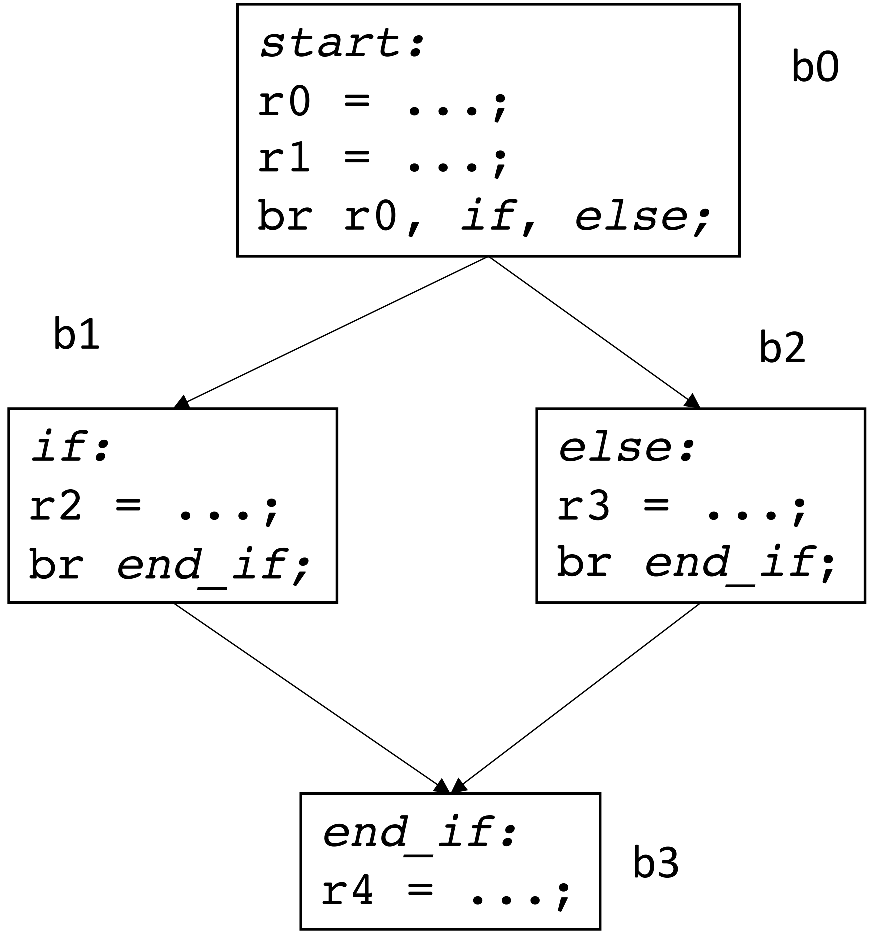
```
end_if:
r4 = ...;
```

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



b0

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

b1

```
if:
r2 = ...;
br end_if;
```

b2

```
else:
r3 = ...;
br end_if;
```

b3

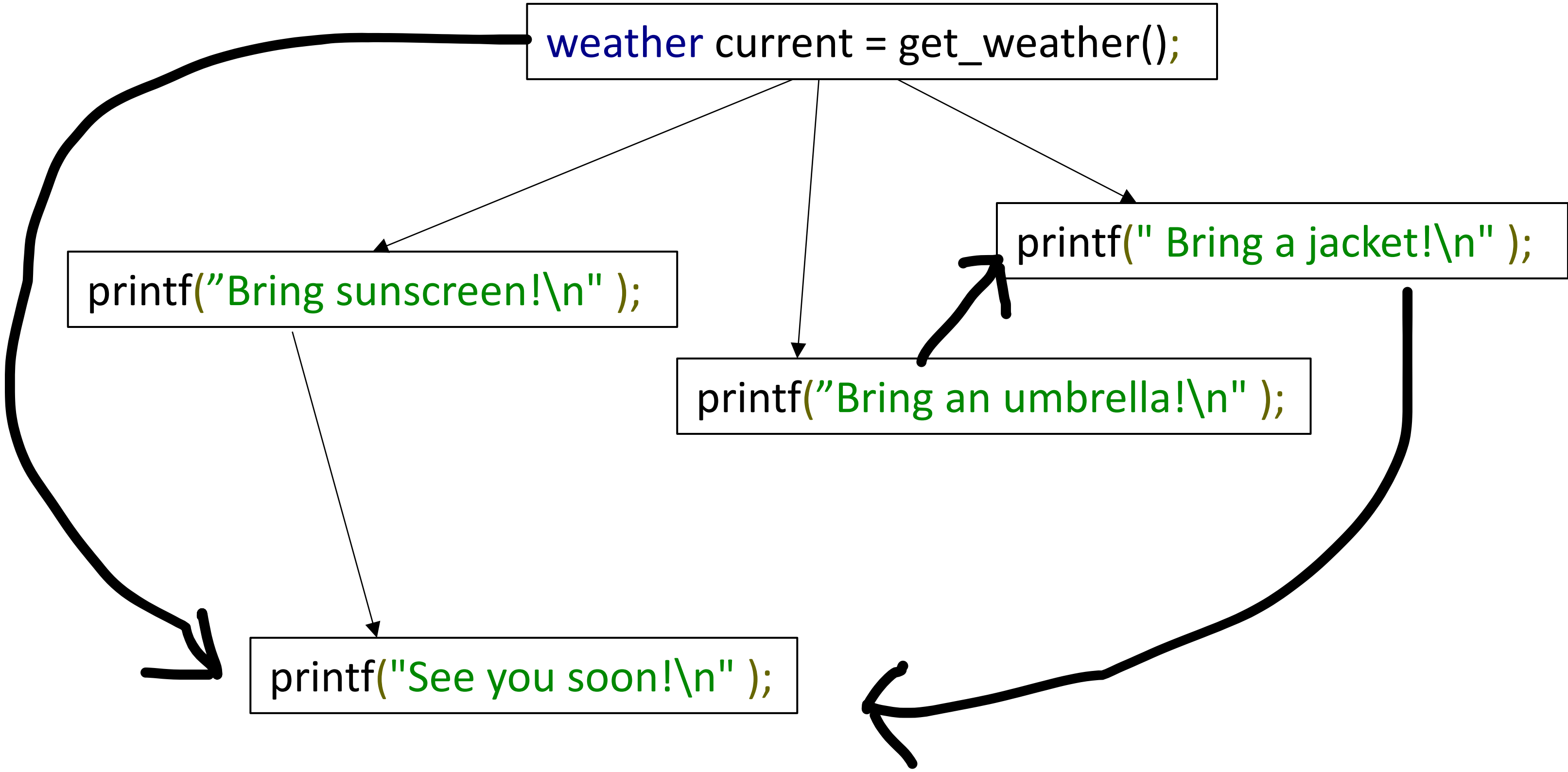
```
end_if:
r4 = ...;
```

if/else pattern

Interesting CFGs

Case statements

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

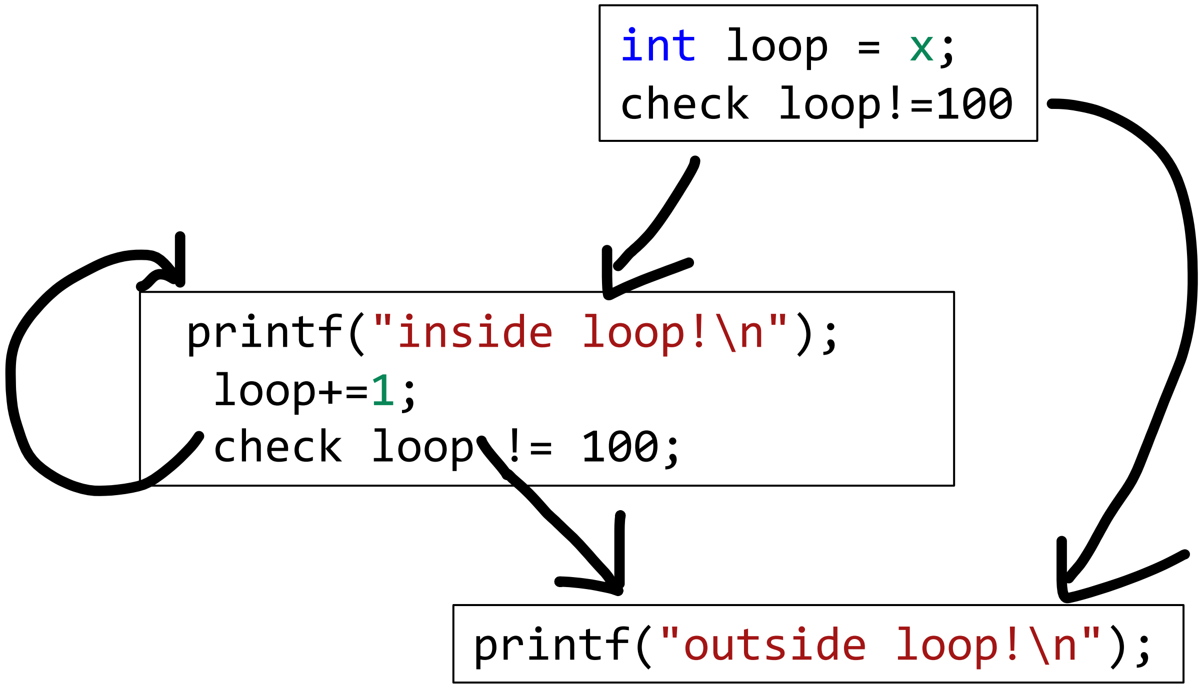


```
weather current = get_weather();
printf(”Bring sunscreen!\n" );
printf(”Bring an umbrella!\n" );
printf(" Bring a jacket!\n" );
printf("See you soon!\n" );
```

Interesting CFGs

Loops

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



```
int loop = x;
check loop!=100
```

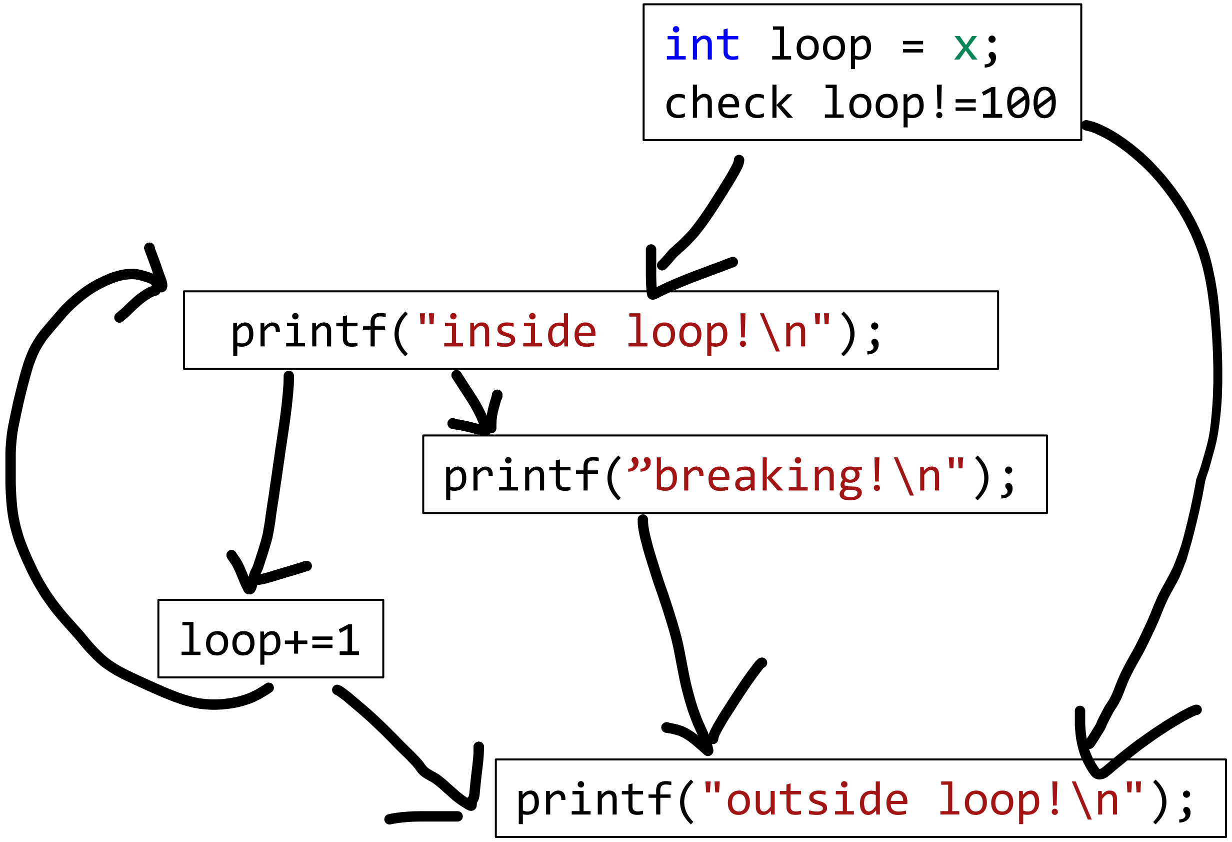
```
printf("inside loop!\n");
 loop+=1;
 check loop != 100;
```

```
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");
```



int loop = x;
check loop!=100

printf("inside loop!\n");

printf(”breaking!\n");

loop+=1

printf("outside loop!\n");

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:

p

Live variables: ?

```
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:

```
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:

```
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

Live variables: z,w

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

Potentially using 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

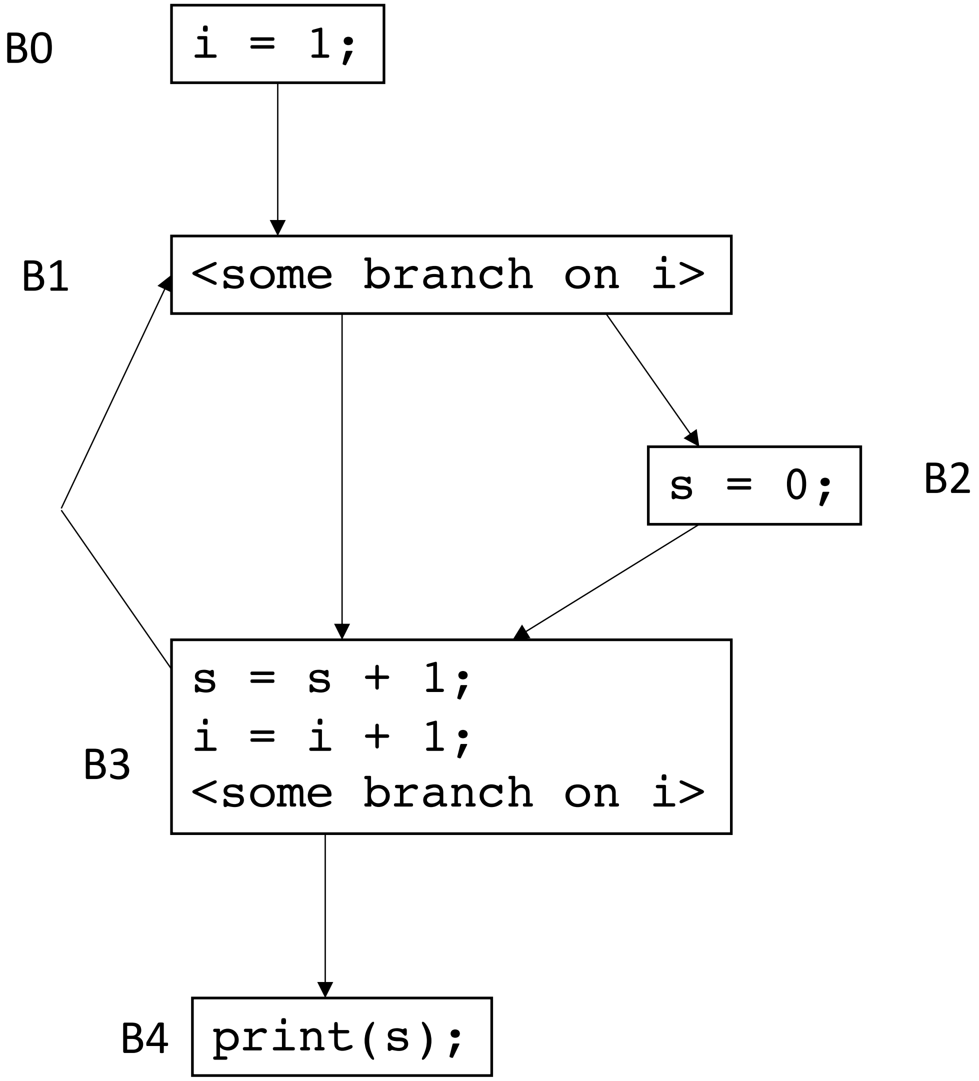
- See code in godbolt

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

Now code gives warning in gcc

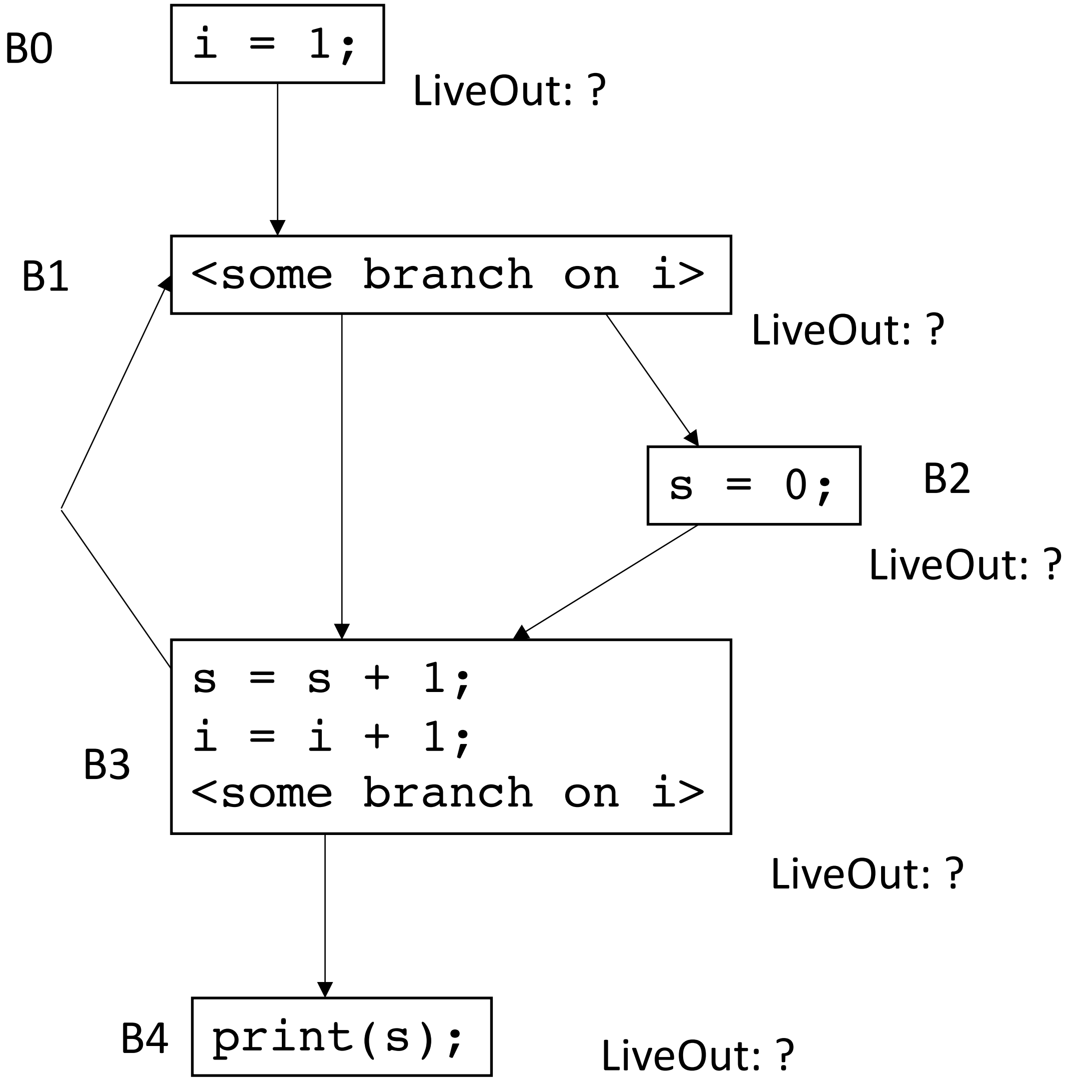
So gcc must only implement their live variable analysis as a local analysis!

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:



B0

```
i = 1;
```

LiveOut: ?

B1

```
<some branch on i>
```

LiveOut: ?

B2

```
s = 0;
```

LiveOut: ?

B3

```
s = s + 1;
i = i + 1;
<some branch on i>
```

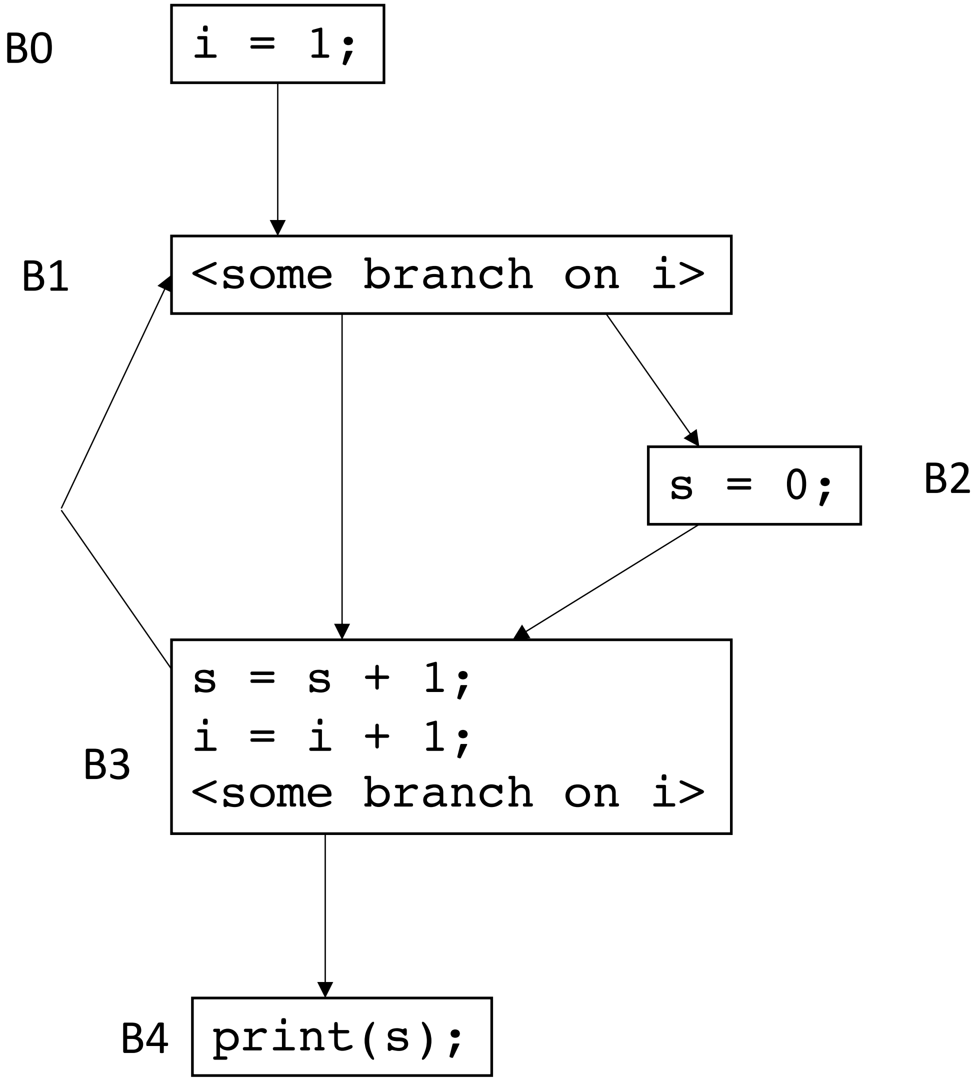
LiveOut: ?

B4

```
print(s);
```

LiveOut: ?

Live variable analysis in the CFG:



B0
```
i = 1;
```

B1
```
<some branch on i>
```

B2
```
s = 0;
```

B3
```
s = s + 1;
i = i + 1;
<some branch on i>
```

B4
```
print(s);
```

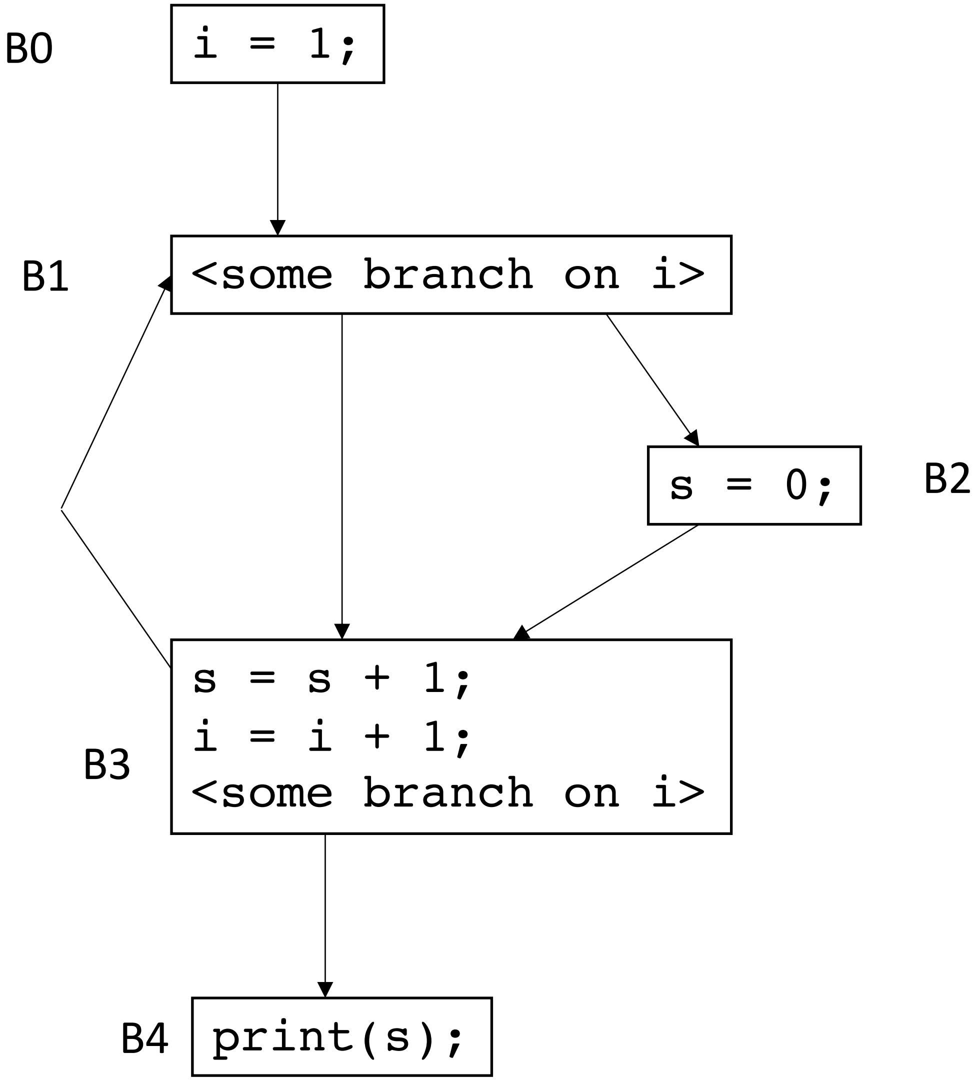
To compute the LiveOut sets, we need two initial sets:

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:



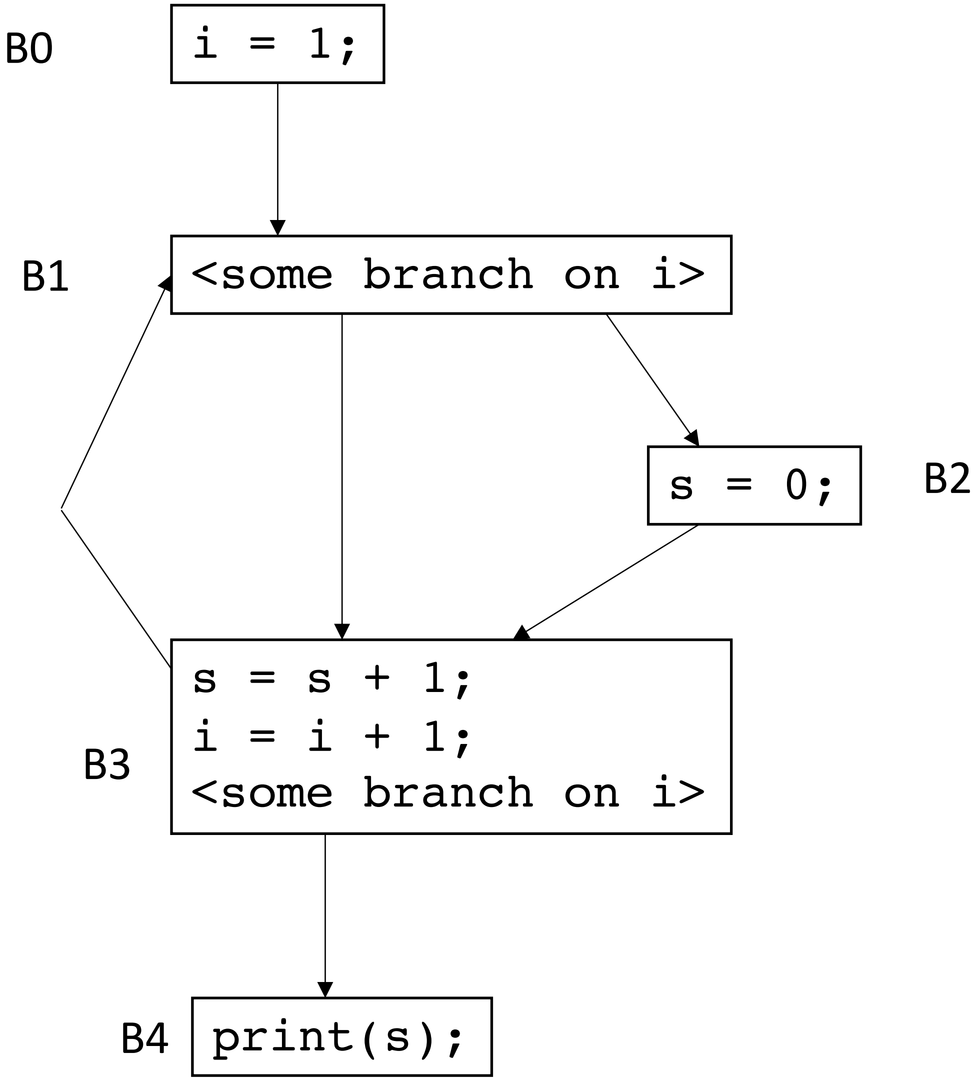
To compute the LiveOut sets, we need two initial sets:

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:

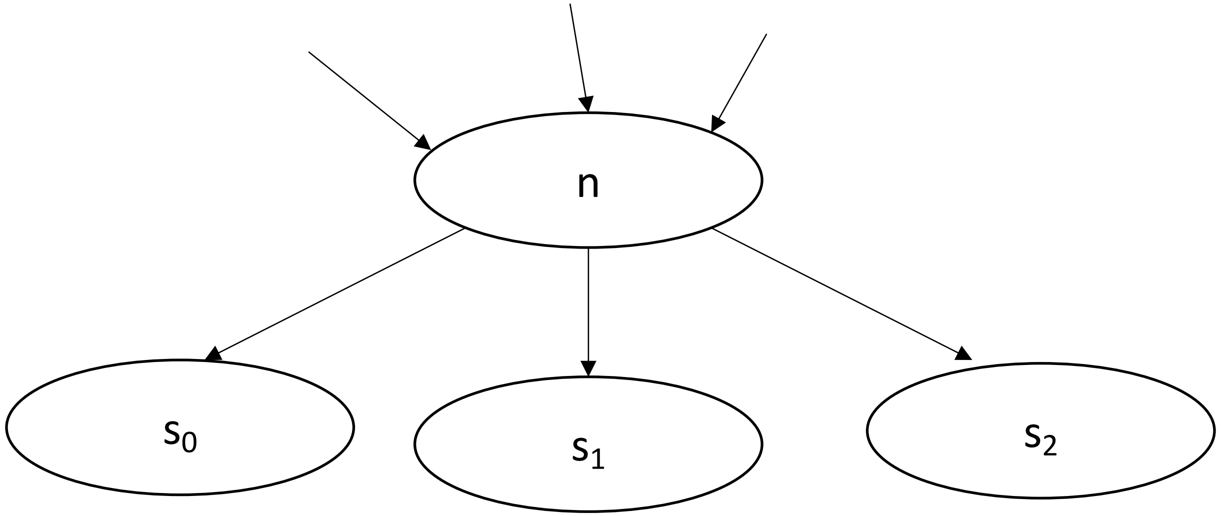
- Initial condition: LiveOut(n) = {} for all nodes
  - Ground truth, no variables are live at the exit of the program, i.e. end node n_{end} has LiveOut(n_{end})= {}

Now we can perform the iterative fixed point computation:

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

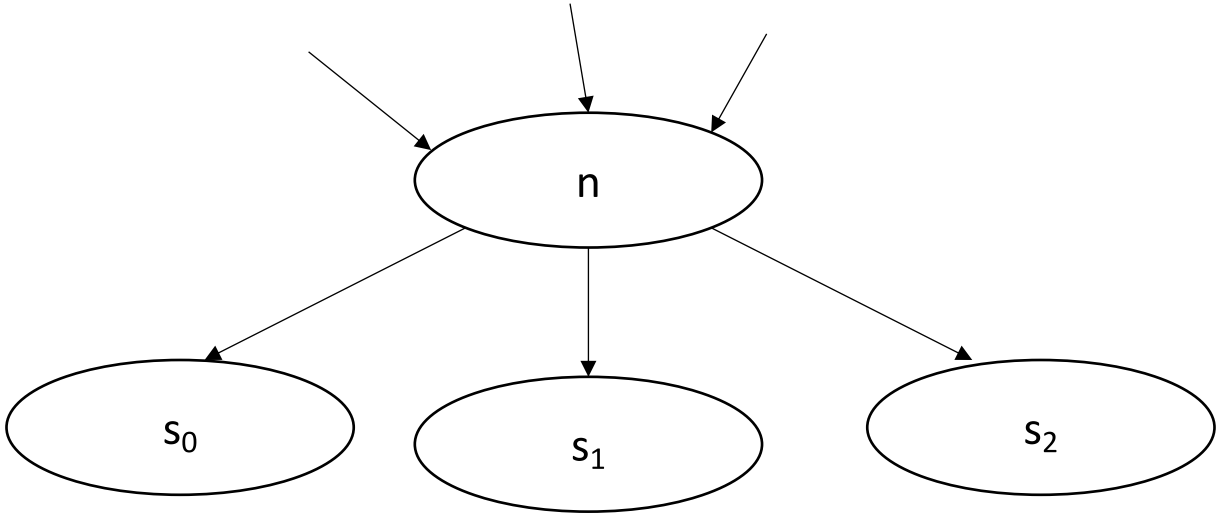
Live variable analysis in the CFG:

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



Live variable analysis in the CFG:

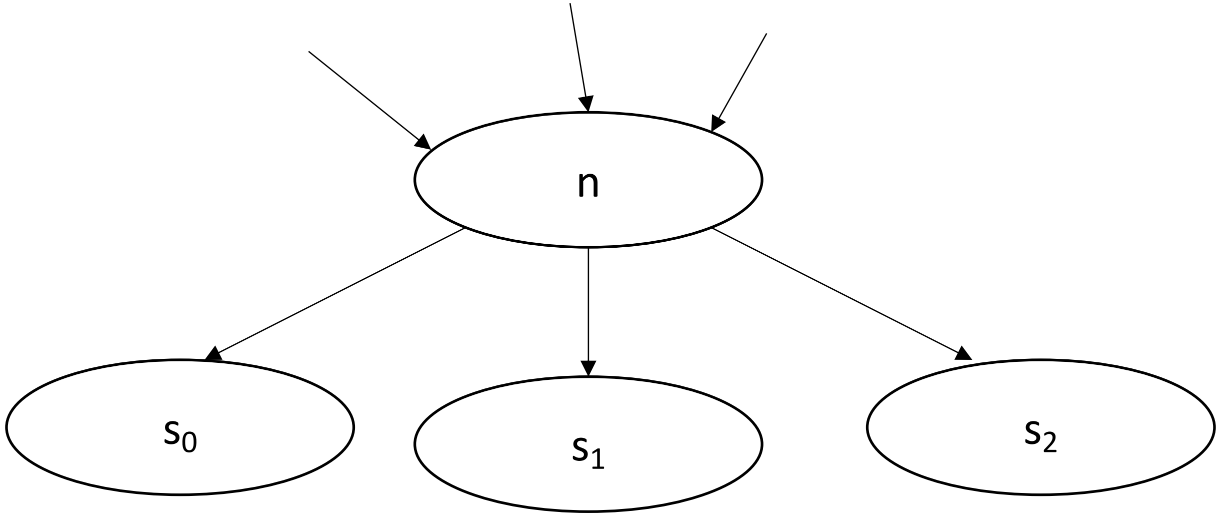
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\mathrm{LiveOut}(s) \cap \overline{VarKill(s)}\,) \right)$$



any variable in UEVar(s)
is live at n

Live variable analysis in the CFG:

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



if s_0 is just this

x = a + b

Then

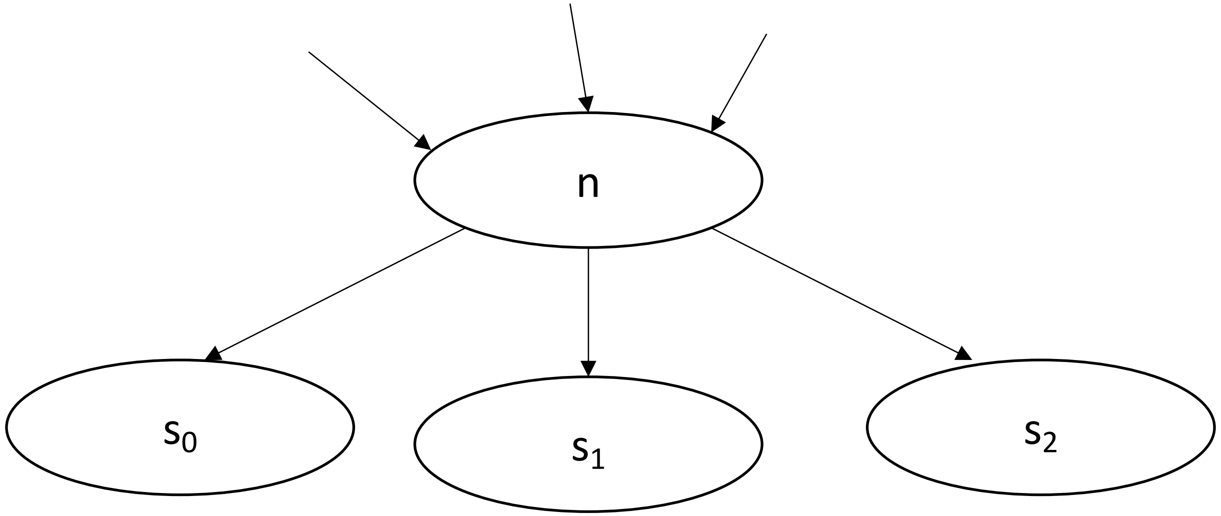
$UEVar(s_0) = \{a,b\}$

These are live at the end of n!

any variable in UEVar(s)
is live at n

Live variable analysis in the CFG:

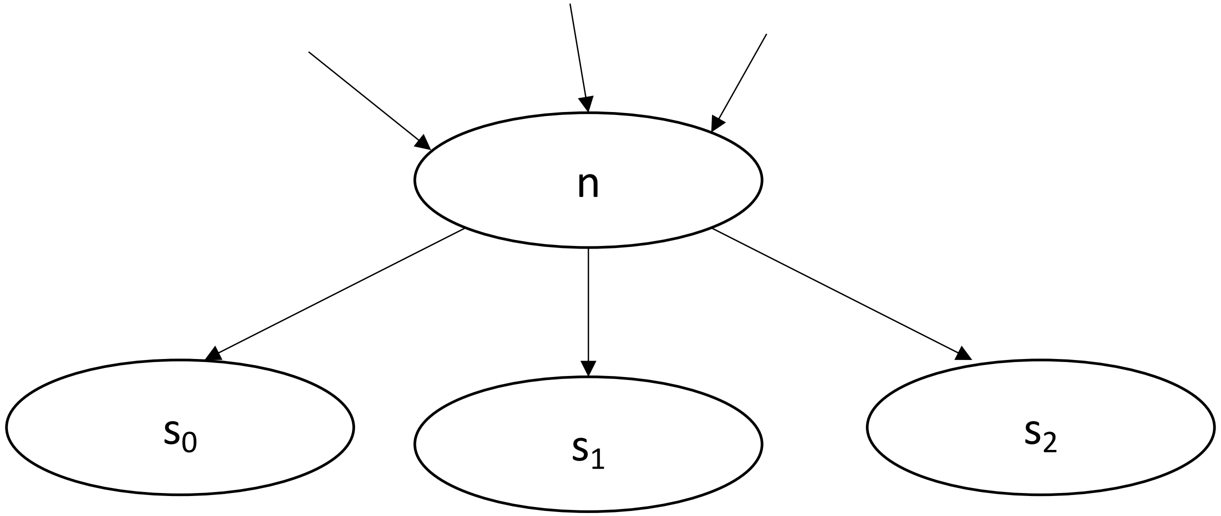
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(LiveOut(s) \cap \overline{VarKill(s)}\right)\right)$$



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

Live variable analysis in the CFG:

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



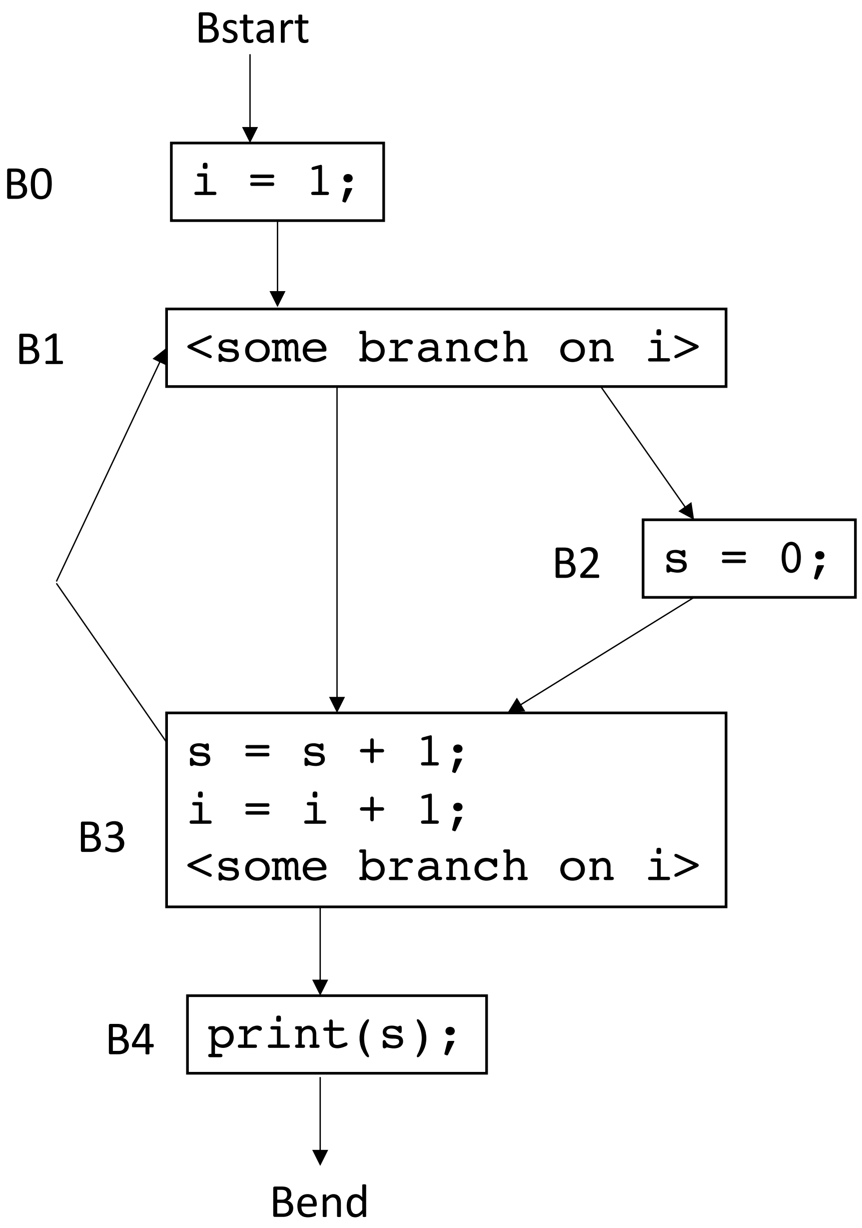
if s_0 is just this

$x = a + b$

Lets say:

Liveout(s_0) = x,c

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



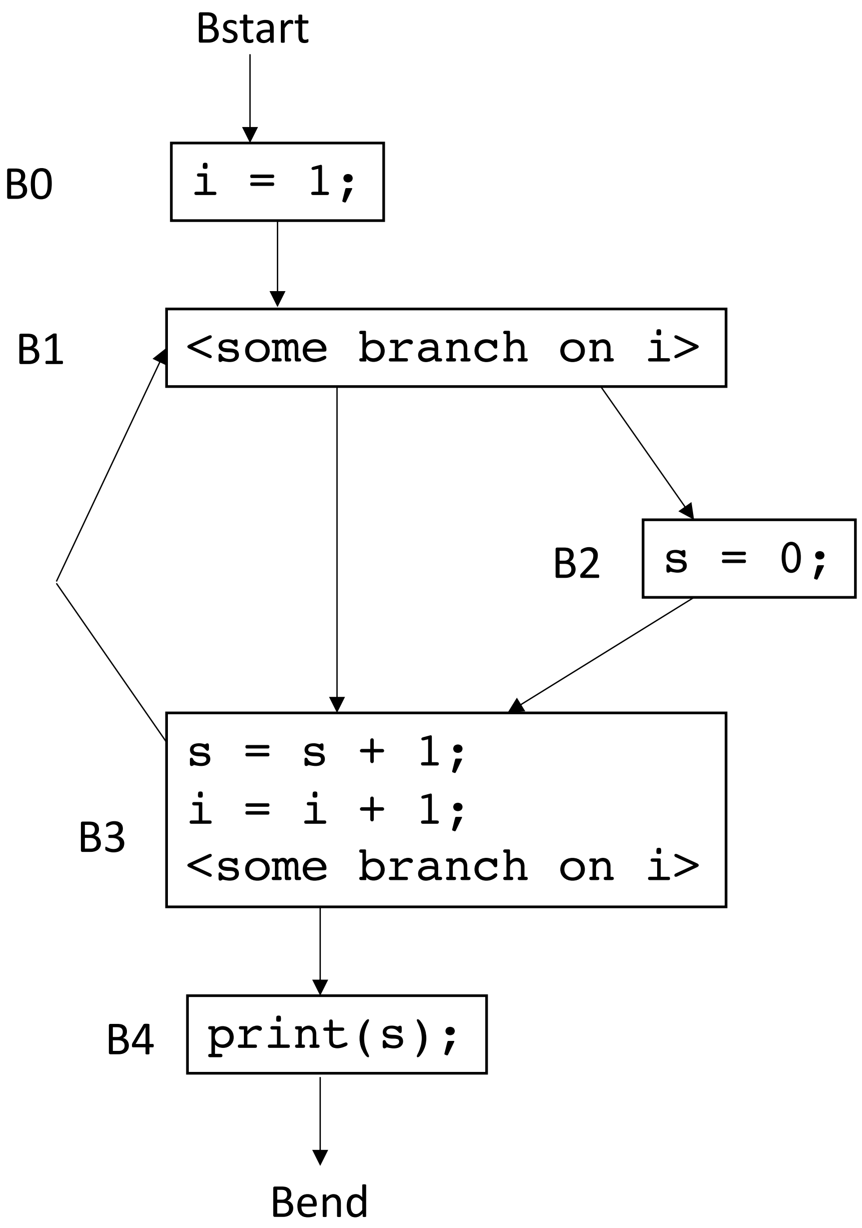
Now we can perform the iterative fixed point computation:

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

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:

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



```
i = 1;
```

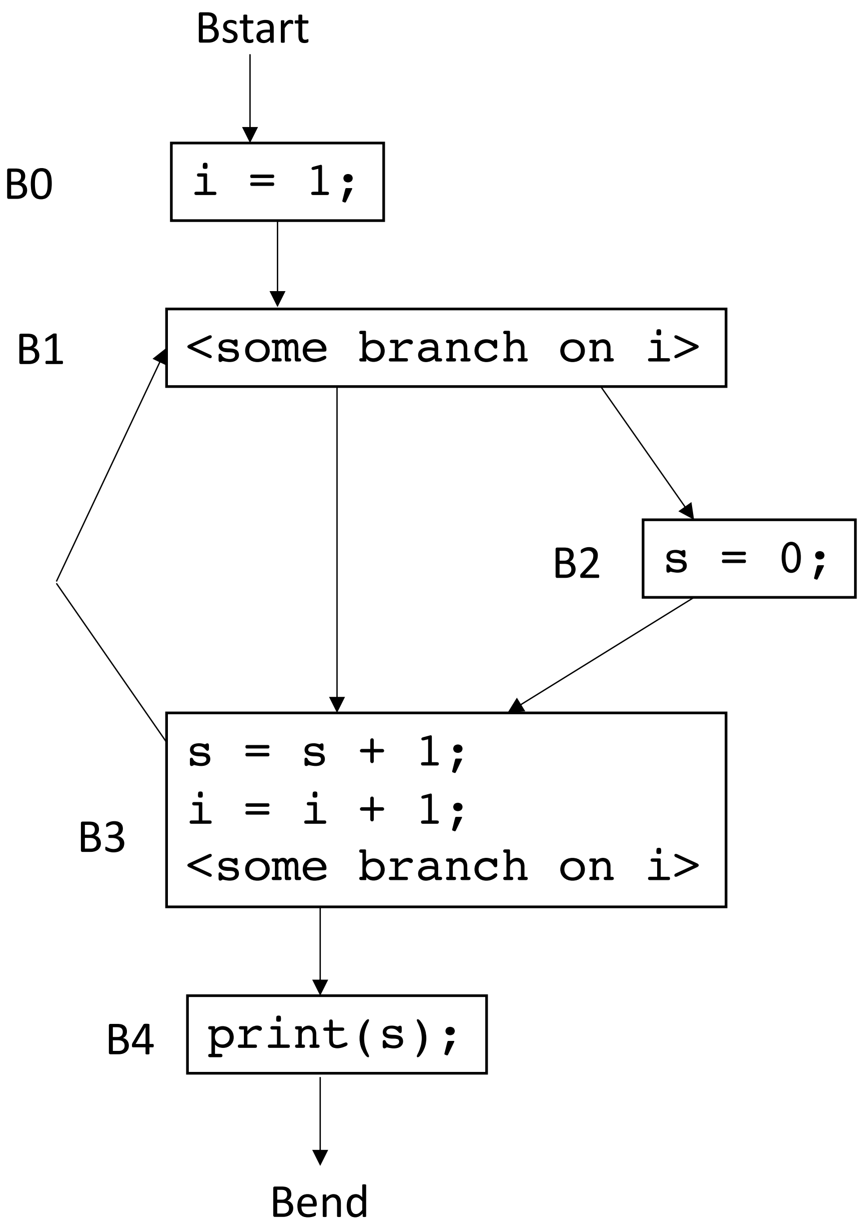
```
<some branch on i>
```

```
s = 0;
```

```
s = s + 1;
i = i + 1;
<some branch on i>
```

```
print(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	{}	



Bstart

B0
`i = 1;`

B1
`<some branch on i>`

B2
`s = 0;`

B3
```
s = s + 1;
i = i + 1;
<some branch on i>
```

B4
`print(s);`

Bend

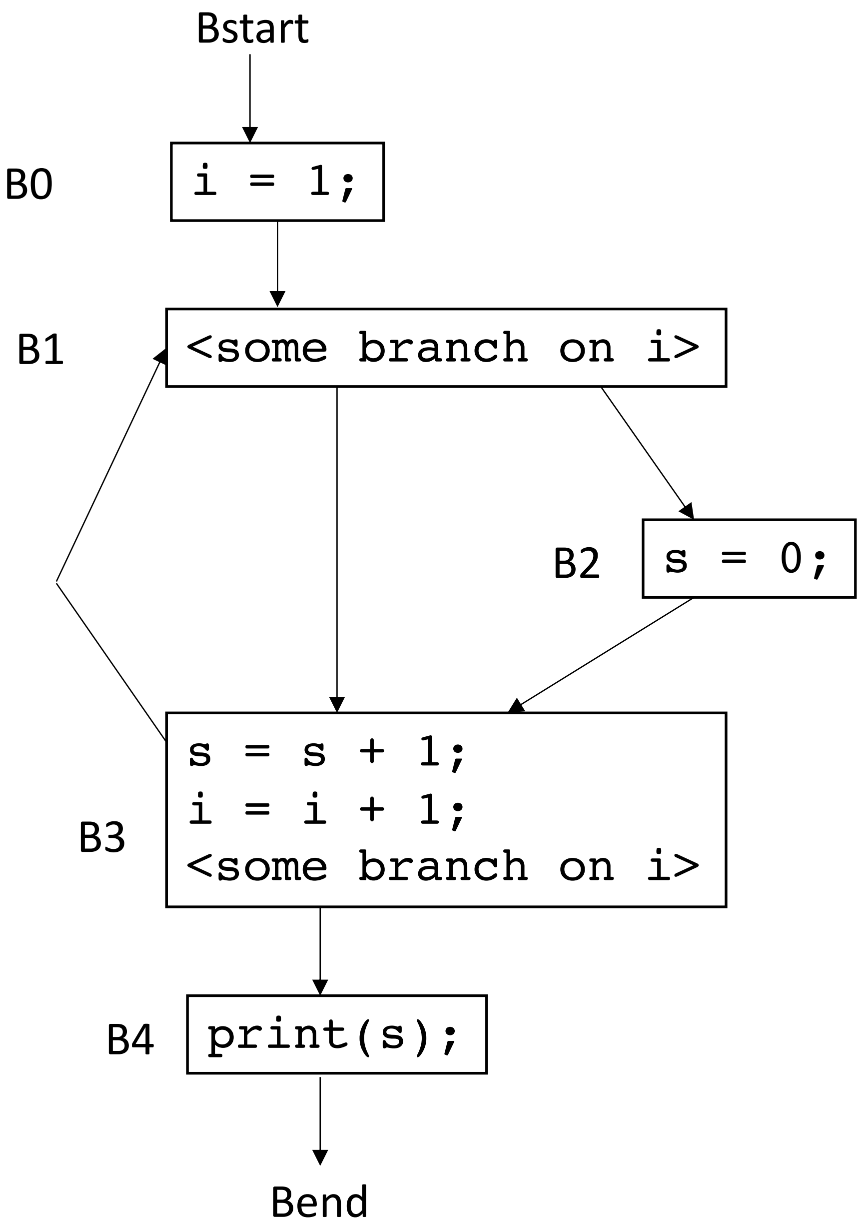
Now we can perform the iterative fixed point computation:

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

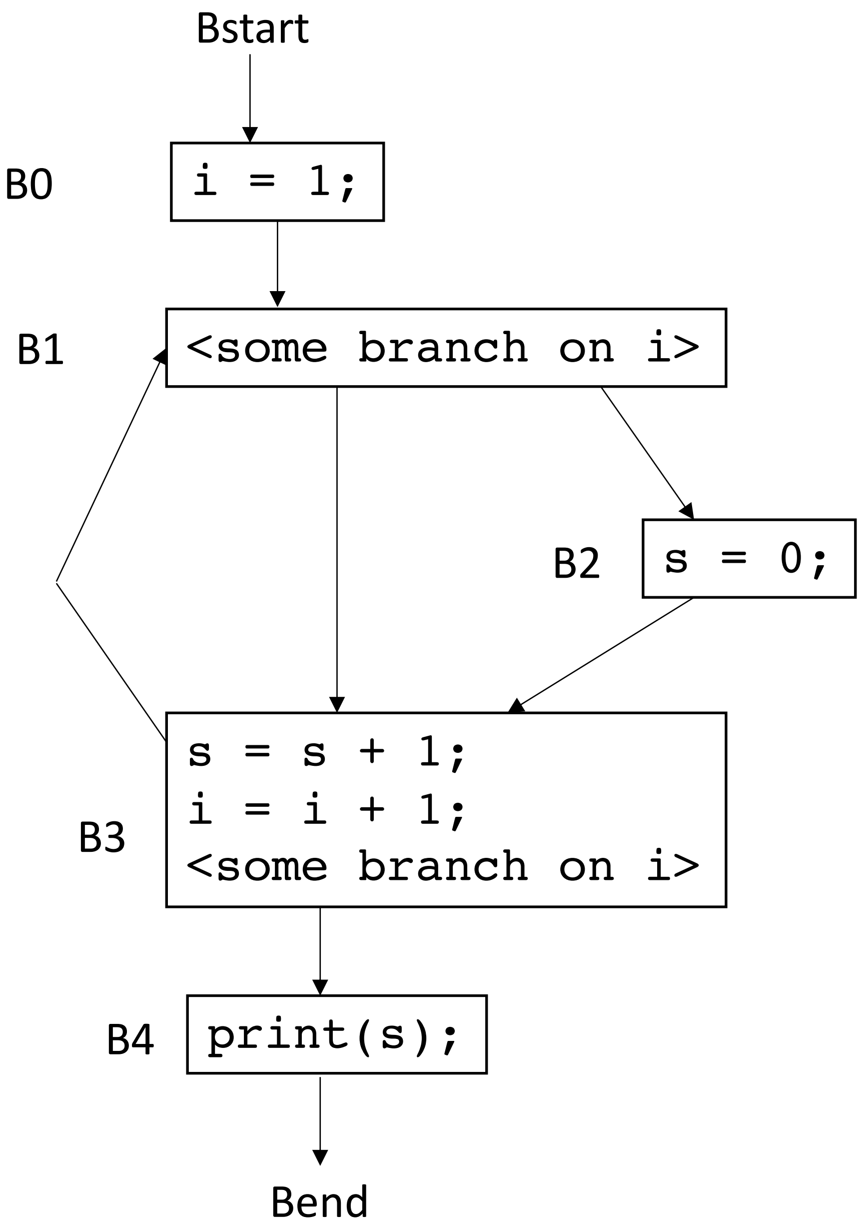
Block	**VarKill**	**UEVar**	**~VarKill**	**LiveOut I_0**	**LiveOut I_1**
Bstart	{}	{}	i,s	{}	{}
B0	i	{}	s	{}	i
B1	{}	i	i,s	{}	i,s
B2	s	{}	i	{}	i,s
B3	i,s	i,s	{}	{}	i,s
B4	{}	s	i,s	{}	{}
Bend	{}	{}	i,s	{}	{}

Now we can perform the iterative fixed point computation:

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



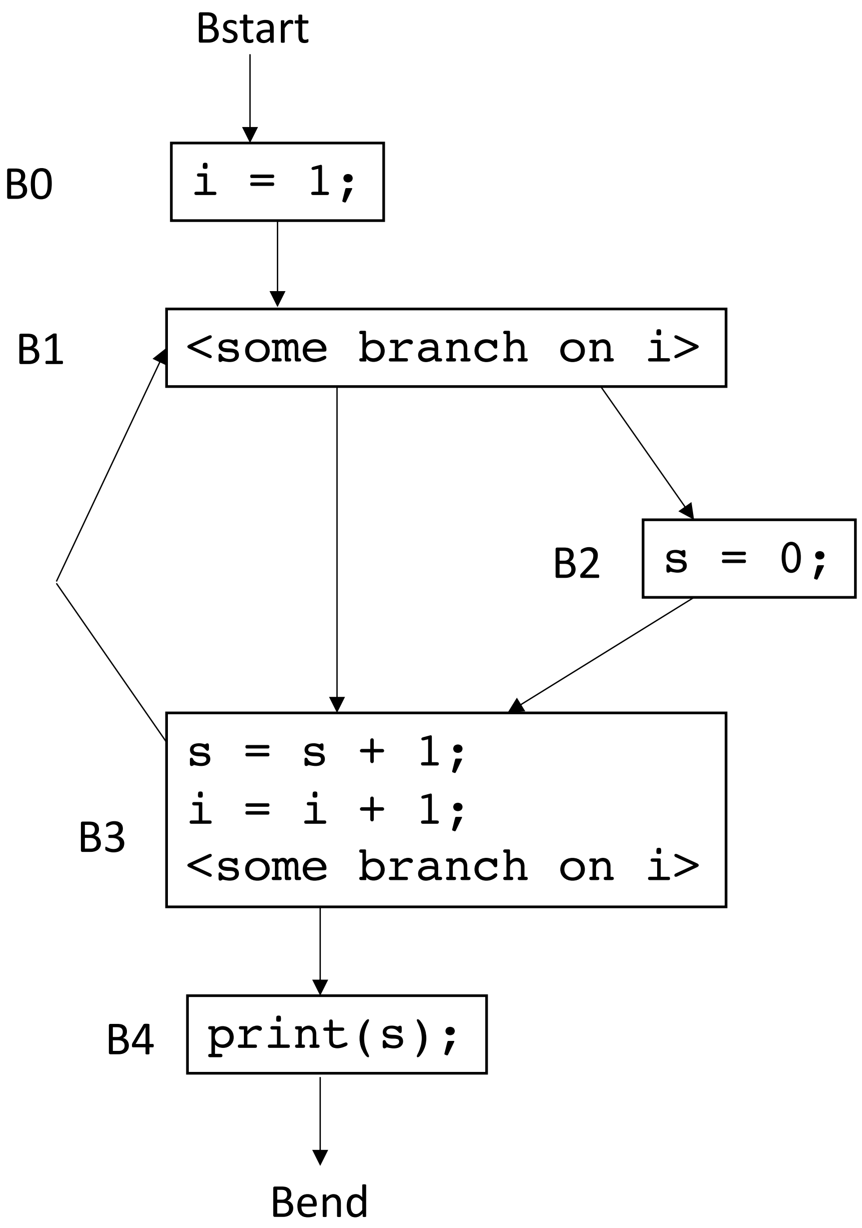
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2
Bstart	{}	{}	i,s	{}	{}	
B0	i	{}	s	{}	i	
B1	{}	i	i,s	{}	i,s	
B2	s	{}	i	{}	i,s	
B3	i,s	i,s	{}	{}	i,s	
B4	{}	s	i,s	{}	{}	
Bend	{}	{}	i,s	{}	{}	



Now we can perform the iterative fixed point computation:

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

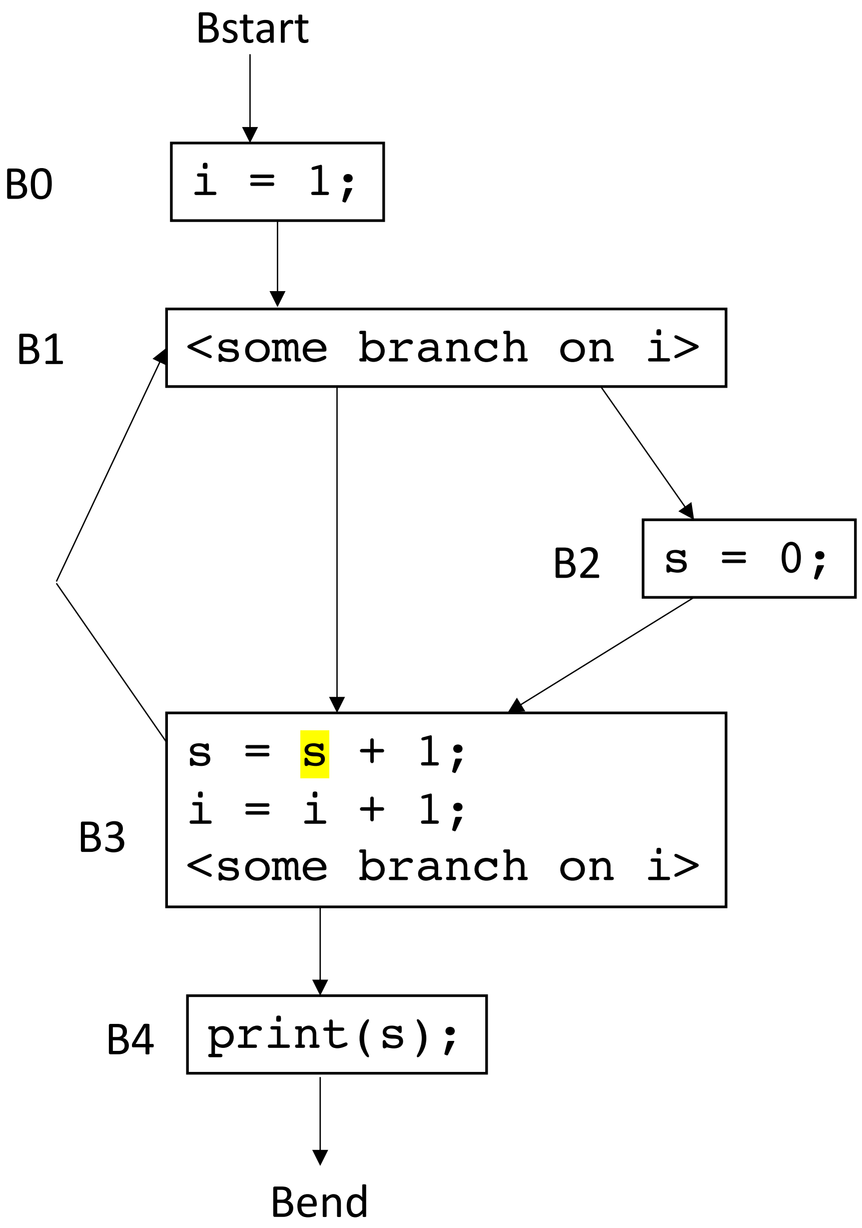
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2
Bstart	{}	{}	i,s	{}	{}	{}
B0	i	{}	s	{}	i	i,s
B1	{}	i	i,s	{}	i,s	i,s
B2	s	{}	i	{}	i,s	i,s
B3	i,s	i,s	{}	{}	i,s	i,s
B4	{}	s	i,s	{}	{}	{}
Bend	{}	{}	i,s	{}	{}	{}



Now we can perform the iterative fixed point computation:

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

Block	**VarKill**	**UEVar**	**~VarKill**	**LiveOut I_0**	**LiveOut I_1**	**LiveOut I_2**	**.. I_3**
Bstart	{}	{}	i,s	{}	{}	{}	
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	{}	{}	{}	

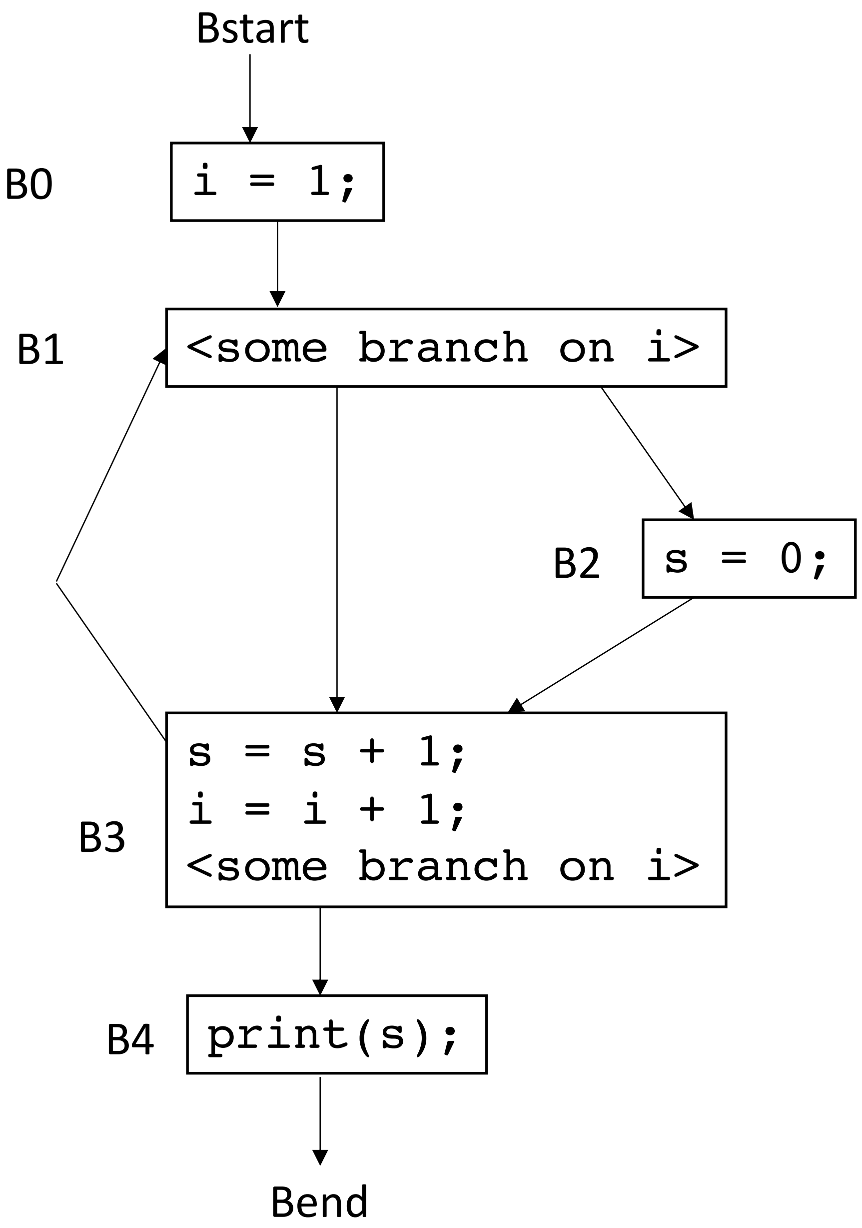


Now we can perform the iterative fixed point computation:

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

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?



```
B0: i = 1;
B1: <some branch on i>
B2: s = 0;
B3: s = s + 1;
    i = i + 1;
    <some branch on i>
B4: print(s);
```

Now we can perform the iterative fixed point computation:

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

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*

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

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;
```

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

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*

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

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
```

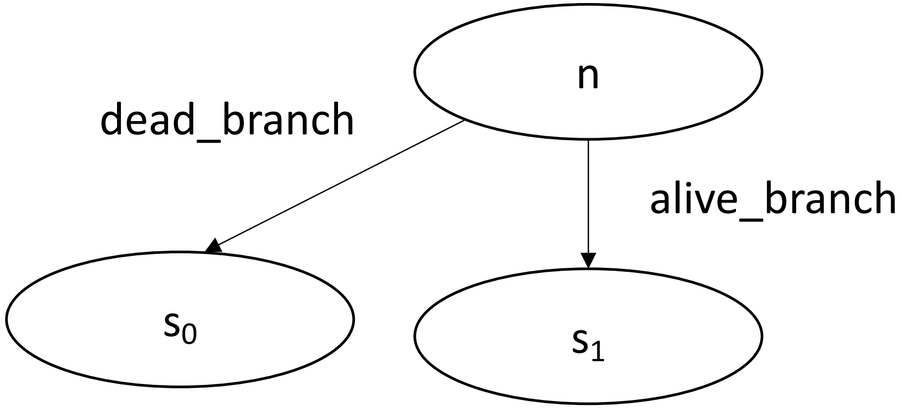
Live variable limitations

Imprecision can come from CFG construction:

consider:

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

could come from arguments, etc.



Live variable limitations

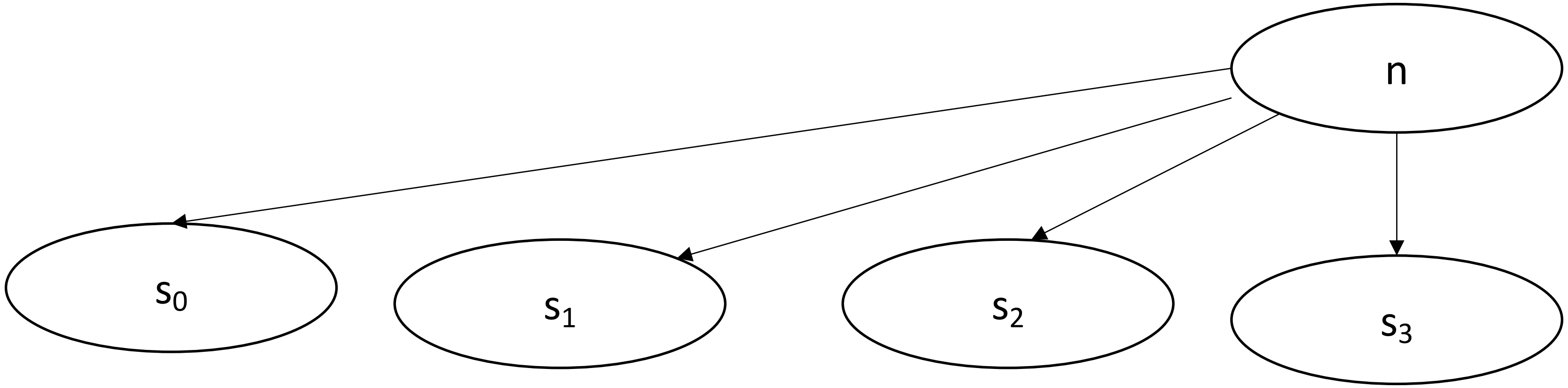
Imprecision can come from CFG construction:

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
- Other data flow instantiations can be used to do global constant propagation, global copy folding, etc.

Done with lectures!

Recap

- Module 1 - Scanners: using regular expressions to break down programs into tokens
- Module 2 - Parsing: using context free grammars to turn program strings into trees
- Module 3 - Intermediate representation: explicitly constructing ASTs, performing type checking and generating 3 address code
- Module 4 - Optimization and analysis: local, regional, and global analysis/operations. We can speed up some code significantly and make code safer

Recap

- Combined, your homeworks compile a non-trivial subset of C into an (optimized) IR that is very close to an ISA
- Even though Clang and GCC and millions of lines of code long, I hope this class made them slightly less magical to you!
- My hope is that this class made you think hard about programming languages, architectures, and how to negotiate between them
- Thank you for your patience as we designed the class!

If you want to work more on your compiler

- Chapter 11: instruction selection
  - Different strategies depending on RISC or CISC
  - Currently changing landscape in modern computing (ARM, Apple M1, RISC-V)
- Chapter 12: instruction scheduling
- Chapter 13: register allocation

If you are interested in this material

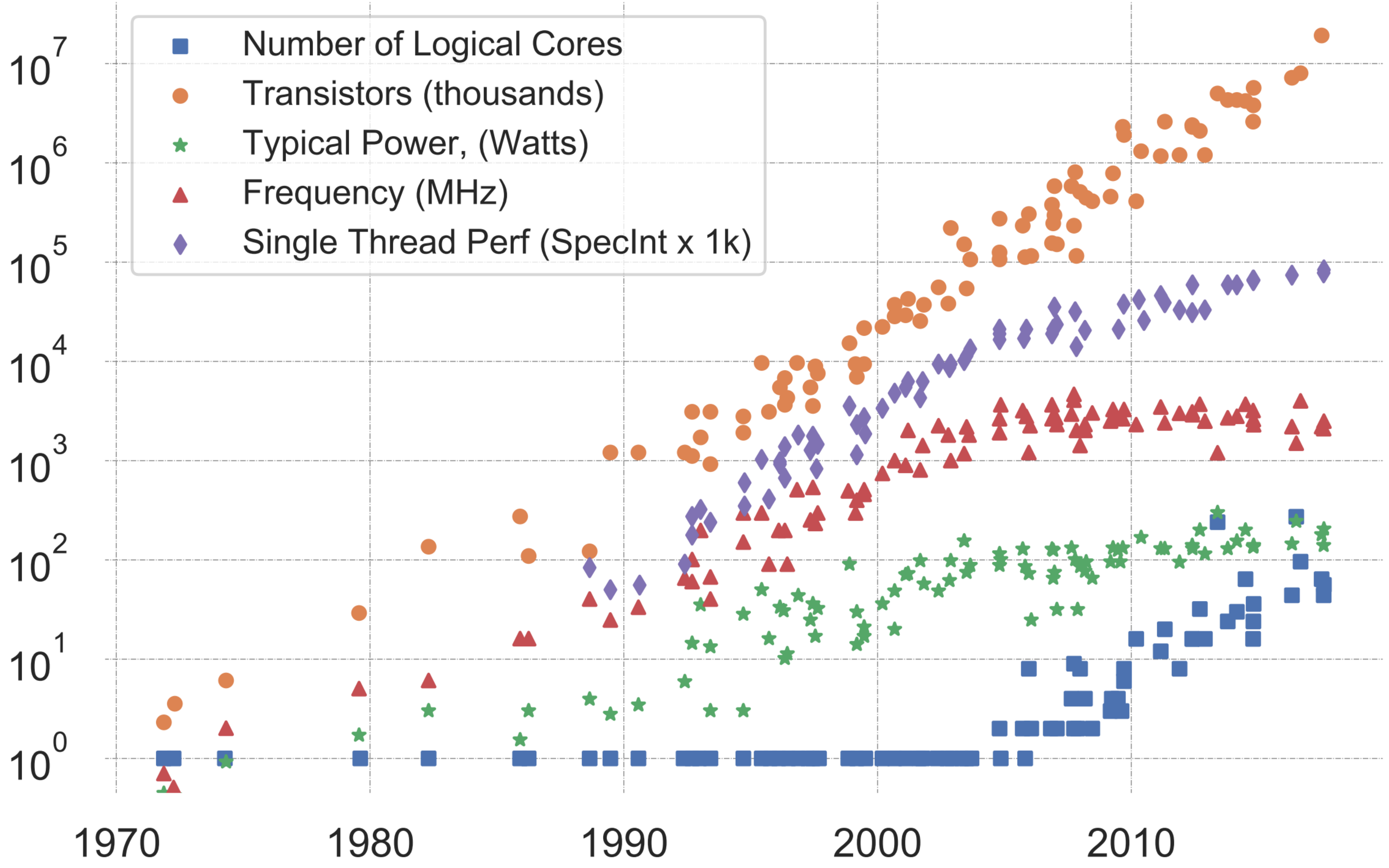
- Grad compilers: Discusses more about data flow analysis, SSA intermediate representation, and domain specific languages
- Programming languages: Discusses properties of programming languages, their structure, and their semantics
- Formal methods: Discusses how we can use the source code to prove more in depth properties about the program (e.g. that there are no bugs)
- Architecture: Discusses how the processor works; however, in order to be useful, architecture features must be available somehow to the programmer (usually through a programming language and a compiler)

Tons of opportunities

- Grad school: there is tons of research going on in all of these areas
- Industry:
    - Nearly every major tech company has (several) compiler teams now
    - Apple: LLVM
    - Microsoft: VSCode
    - Microsoft: Github
    - Nvidia: nvcc
    - Intel: icc
    - Game dev
    - ...

https://mgaudet.github.io/CompilerJobs/

Compilers are going to be increasingly important in the next era of computing



computer
raw speed
has leveled off

Number of cores
is starting to level
off

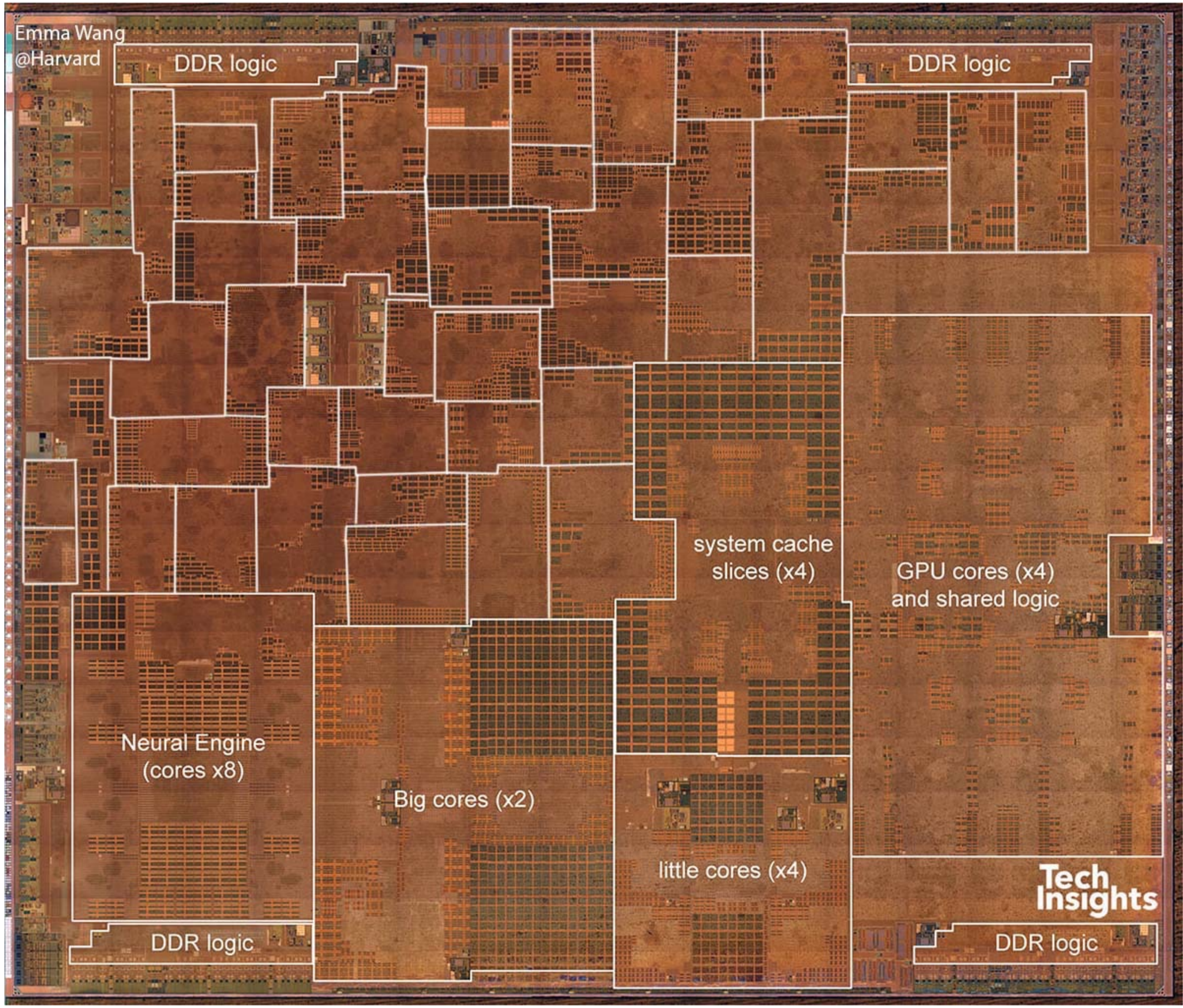
K. Rupp, “40 Years of Mircroprocessor Trend Data,” https://www. karlrupp.net/2015/06/40-years-of-microprocessor-trend-data, 2015.

Modern SoC

- From David Brooks lab at Harvard:

  http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/

- Compilers will need to be able to map software efficiently to a range of different accelerators



Thanks everyone!!

- For those of you who are graduating: congrats!
    - A CS degree is an incredible accomplishment!

- For those of you who are not:
    - I hope to see you around next year!

- Don’t be a stranger! We love hearing from you!

- If you have any feedback about the class, please let me know!

- Good luck on the final and enjoy your summer!