Program Slicing

Original image from YouTuber Craftingeek EN

Ok, let's get a look at this AST thing

```python
sum_acc = 0
prod_acc = 1
i = 1
while (i < 11):
    sum_acc = sum_acc + i
    prod_acc = prod_acc * i
    i = i + 1
print("sum", sum_acc)
print("prod", prod_acc)
```

Our Python program (the one we're analyzing, not the one we're running)

```python
code = open(filename).read()
tree = ast.parse(code)
astpretty.pprint(tree)
```

Here's the one we're running…

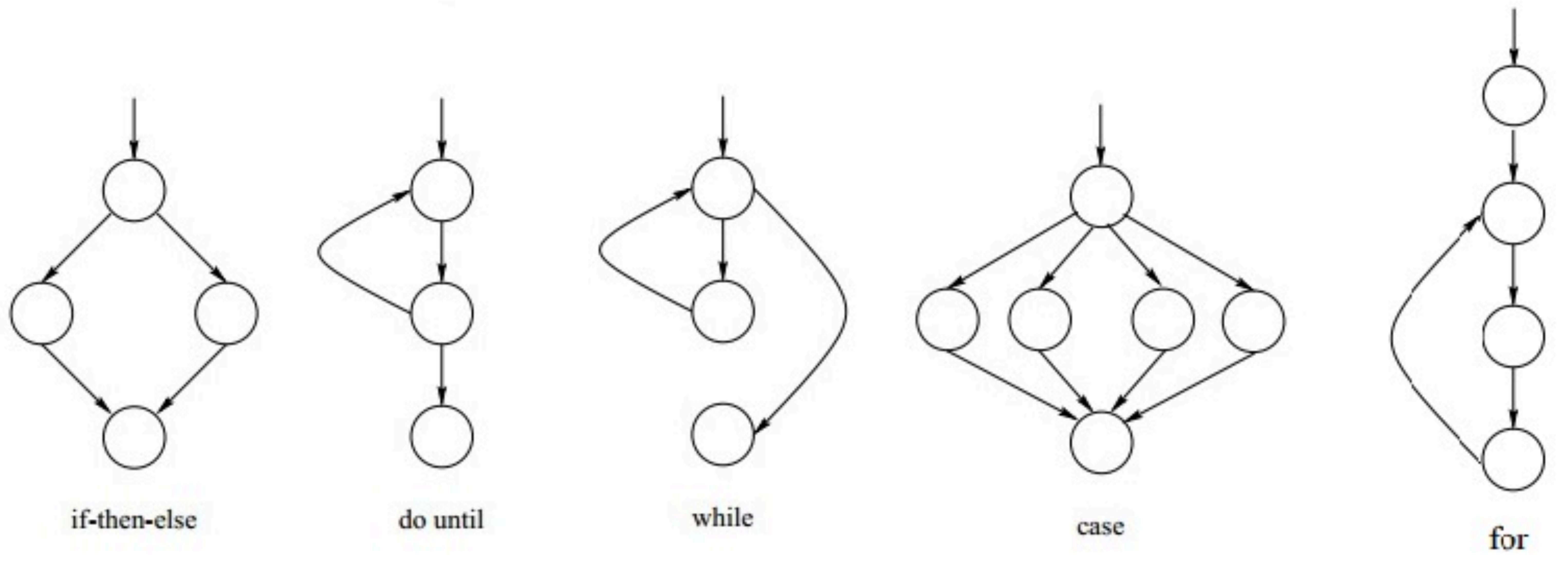
Look at that beautiful AST!

```
Module(
    body=[
        Assign(
            lineno=1,
            col_offset=0,
            end_lineno=1,
            end_col_offset=11,
            targets=[Name(lineno=1, col_offset=0, end_lineno=1, end_col_offset=7, id='sum_acc', ctx=Store()
            value=Constant(lineno=1, col_offset=10, end_lineno=1, end_col_offset=11, value=0, kind=None),
            type_comment=None,
        ),
        Assign(
            lineno=2,
            col_offset=0,
            end_lineno=2,
            end_col_offset=12,
            targets=[Name(lineno=2, col_offset=0, end_lineno=2, end_col_offset=8, id='prod_acc', ctx=Store(
            value=Constant(lineno=2, col_offset=11, end_lineno=2, end_col_offset=12, value=1, kind=None),
            type_comment=None,
        ),
        Assign(
            lineno=3,
            col_offset=0,
            end_lineno=3,
            end_col_offset=5,
            targets=[Name(lineno=3, col_offset=0, end_lineno=3, end_col_offset=1, id='i', ctx=Store())],
            value=Constant(lineno=3, col_offset=4, end_lineno=3, end_col_offset=5, value=1, kind=None),
            type_comment=None,
        ),
        While(
            lineno=4,
            col_offset=0,
            end_lineno=7,
            end_col_offset=10,
            test=Compare(
                lineno=4,
                col_offset=7,
```

Next, we need to know how control flows through the program

Enter…the control flow graph (CFG)

We'll build up a graph representing all the paths we could take through the program during execution

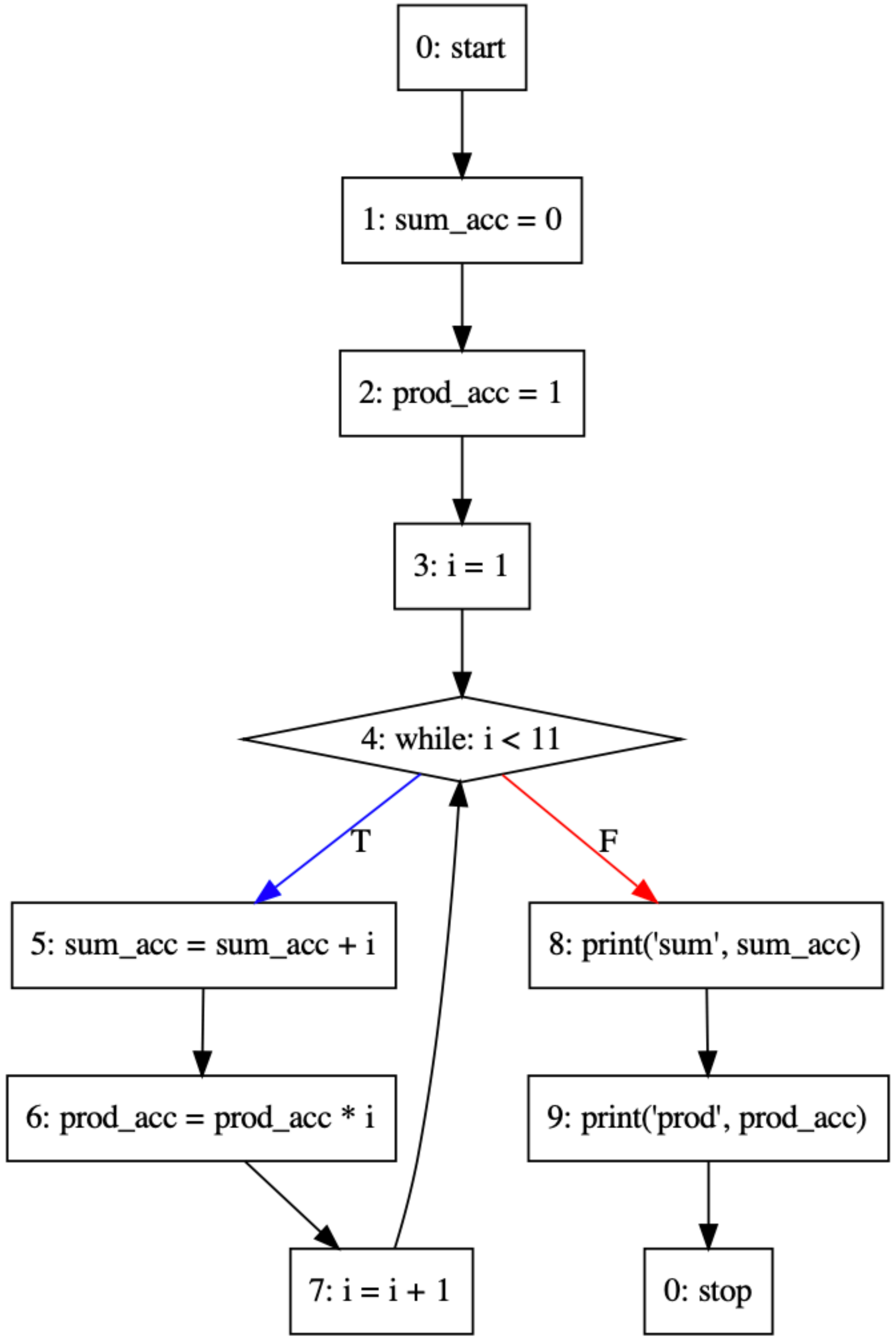


Another entry in our theme of 'there are so many ways to represent programs'!

CFGs

```
sum_acc = 0
prod_acc = 1
i = 1
while (i < 11):
    sum_acc = sum_acc + i
    prod_acc = prod_acc * i
    i = i + 1
print("sum", sum_acc)
print("prod", prod_acc)
```

Program to analyze



The CFG!

But how do we get the slice from this thing?

Starting from a CFG, we'll compute data flow information about the set of relevant variables at each node

n	Statement	ref(n)	def(n)	relevant(n)

We'll use this ^ and this ^ to figure out this ^

Referenced at node n

Defined at node n

Program Slicing: Straight-Line Code

Slice for node ***n*** and variables ***V***

1. Initialize the relevant sets of all nodes to the empty set.
2. Insert all variables of ***V*** into **relevant(*n*)**.
3. For ***n***'s immediate predecessor ***m***, compute **relevant(*m*)** by:

    // first exclude all variables defined at ***m*** (because we're overwriting it)

    relevant(*m*) := **relevant(*n*)** - **def(*m*)**

    // if **m** defines a variable that's relevant at **n**

    if **def(*m*)** in **relevant(*n*)** then

        // include the variables that are referenced at **m**

        relevant(*m*) := **relevant(m)** ∪ **ref(*m*)**

        include ***m*** in the slice

    end

4. Repeat (3) for ***m***'s immediate predecessors, and work backwards in the CFG until we reach the start node or the relevant set is empty

Bolded n are included in the slice

n	Statement	ref(n)	def(n)	relevant(n)
1	b = 1		b	
2	c = 2		c	b
3	d = 3		d	b, c
4	a = d	d	a	b, c
5	d = b + d	b, d	d	b, c
6	b = b + 1	b	b	b, c
7	a = b + c	b, c	a	b, c
8	print a	a		a

slice for <8, {a}>

Step 2: relevant(8) = {a}

Step 3: relevant(7) = relevant(8) - def(7) = {a} - {a} = {}

relevant(7) = relevant(7) ∪ ref(7) = {} ∪ {b, c} = {b, c}

Since node 7 defines a variable relevant at node 8, it is included into the slice.

Step 3: relevant(6) = relevant(7) - def(6) = {b, c} - {b} = {c}

relevant(6) = relevant(6) ∪ ref(6) = {c} ∪ {b} = {b, c}

Since node 6 defines a variable relevant at node 7, it is included into the slice.

Step 3: relevant(5) = relevant(6) - def(5) = {b, c} - {d} = {b, c}

Step 3: relevant(4) = relevant(5) - def(4) = {b, c} - {a} = {b, c}

Step 3: relevant(3) = relevant(4) - def(3) = {b, c} - {d} = {b, c}

Step 3: relevant(2) = relevant(3) - def(2) = {b, c} - {c} = {b}

relevant(2) = relevant(2) ∪ ref(2) = {b} ∪ {} = {b}

Since node 2 defines a variable relevant at node 3, it is included into the slice.

Step 3: relevant(1) = relevant(2) - def(1) = {b} - {b} = {}

relevant(1) = relevant(1) ∪ ref(1) = {} ∪ {} = {}

Since node 1 defines a variable relevant at node 2, it is included into the slice.

What will happen if we add an if statement into our program?

- Any guesses?

Moving towards handling control flow…

- We have to extend our earlier approach to:
  - If we add a node ***m*** to our slice:
    - also add the *control set* of ***m*** to our slice
      - (the control set is the set of predicates that directly control its execution)
      - for each node ***c*** included based on being in the control set:
        - make a new slice! Starting at node ***c*** for variables **ref(*c*)**. The original slice (for <***n***, ***V***>) will now also include all nodes in the slice for <***c***, **ref(*c*)**>
  - Union the relevant sets (e.g., **relevant(m_1)** and **relevant(m_2)**) for cases where we have multiple descendants with a shared predecessor
    - (Remember that once we have control flow, we can have multiple descendants!)

n	Statement	ref(n)	def(n)	control(n)	relevant(n)
1	b = 1		b		
2	c = 2		c		b
3	d = 3		d		b, c
4	a = d	d	a		b, c, d
5	if a then	a			b, c, d
6	d = b + d	b, d	d	5	b, d
7	c = b + d	b, d	c	5	b, d
	else				
8	b = b + 1	b	b	5	b, c
9	d = b + 1	b	d	5	b, c
	endif				b, c
10	a = b + c	b, c	a		b, c
11	print a	a			a

slice for <11, {a}>

Step 2: relevant(11) = {a}
Step 3: relevant(10) = relevant(11) - def(10) = {a} - {a} = {}
relevant(10) = relevant(10) ∪ ref(10) = {} ∪ {b, c} = {b, c}
Since node 10 defines a variable relevant at node 11, it is included into the slice.
Step 3: relevant(9) = relevant(10) - def(9) = {b, c} - {d} = {b, c}
Step 3: relevant(8) = relevant(9) - def(8) = {b, c} - {b} = {c}
relevant(8) = relevant(8) ∪ ref(8) = {c} ∪ {b} = {b, c}
Since node 8 defines a variable relevant at node 9, it is included into the slice.
Since control(8) = 5, node 5 is included into the slice.
The slice for node 5 with respect to ref(5) is computed below.
Step 3: relevant(7) = relevant(10) - def(7) = {b, c} - {c} = {b}
relevant(7) = relevant(7) ∪ ref(7) = {b} ∪ {b, d} = {b, d}
Since node 7 defines a variable relevant at node 10, it is included into the slice.
Since control(7) = 5, node 5 is included into the slice.
The slice for node 5 with respect to ref(5) is computed below.
Step 3: relevant(6) = relevant(7) - def(6) = {b, d} - {d} = {b}
relevant(6) = relevant(6) ∪ ref(6) = {b} ∪ {b, d} = {b, d}
Since node 6 defines a variable relevant at node 7, it is included into the slice.
Step 3: relevant(5) = relevant(6) ∪ relevant(8) = {b, d} ∪ {b, c} = {b, c, d}
Step 3: relevant(4) = relevant(5) - def(4) = {b, c, d} - {a} = {b, c, d}
Step 3: relevant(3) = relevant(4) - def(3) = {b, c, d} - {d} = {b, c}
relevant(3) = relevant(3) ∪ ref(3) = {b, c} ∪ {} = {b, c}
Since node 3 defines a variable relevant at node 4, it is included into the slice.
Step 3: relevant(2) = relevant(3) - def(2) = {b, c} - {c} = {b}
relevant(2) = relevant(2) ∪ ref(2) = {b} ∪ {} = {b}
Since node 2 defines a variable relevant at node 3, it is included into the slice.
Step 3: relevant(1) = relevant(2) - def(1) = {b} - {b} = {}
relevant(1) = relevant(1) ∪ ref(1) = {} ∪ {} = {}
Since node 1 defines a variable relevant at node 2, it is included into the slice.

slice currently contains: 10, 8, 7, 6, 5, 3, 2, 1

We're not done yet! Remember slice for node 5 w.r.t. ref(5)!

Let's take care of that subslice

n	Statement	ref(n)	def(n)	control(n)	relevant(n)
1	b = 1		b		
2	c = 2		c		
3	d = 3		d		{}
4	a = d	d	a		{d}
5	if a then	a			{a}
6	d = b + d	b, d	d	5	
7	c = b + d	b, d	c	5	
	else				
8	b = b + 1	b	b	5	
9	d = b + 1	b	d	5	
	endif				
10	a = b + c	b, c	a		
11	print a	a			

slice for <5, {a}>

Step 2: relevant(5) = {a}
Step 3: relevant(4) = relevant(5) - def(4) = {a} - {a} = {}
Since node 4 defines a variable relevant at node 5, it is included into the slice.
relevant(4) = relevant(4) ∪ ref(4) = {} ∪ {d} = {d}
Step 3: relevant(3) = relevant(4) - def(3) = {d} - {d} = {}
Since node 3 defines a variable relevant at node 4, it is included into the slice.
relevant(3) = relevant(3) ∪ ref(3) = {} ∪ {} = {}
Since the relevant set is empty, no more nodes will be included into the slice.

final slice contains: 10, 8, 7, 6, 5, 4, 3, 2, 1

More reading

- The nice worked examples in these slides come from:
  - Program Slicing for Object-Oriented Programming Languages, Christoph Steindl (dissertation)
- If you want to dig in on these specific worked examples, take a look at Chapter 3 of the dissertation:
  - http://www.ssw.uni-linz.ac.at/General/Staff/CS/Research/Publications/Ste99a.html
- A more comprehensive resource:
  - Cooper and Torczon's Engineering a Compiler textbook
  - http://www.r-5.org/files/books/computers/compilers/writing/Keith_Cooper_Linda_Torczon-Engineering_a_Compiler-EN.pdf

What about loops?

- If we have loops, we have to keep iterating over the CFG until our slice and our relevant sets stabilize
- You won’t be required to handle loops for your homework, but it’s pretty fun if you’re interested :)

Let's do this!

- Fire up your 

- This is going to be our last programming assignment of the semester, so get ready to do some language hacking :)