Program Slicing
let's diy!

```
prod = 1
i = 1
while (i < 11)
    prod = prod * i
    i = i + 1
```
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...
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’!
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

<table>
<thead>
<tr>
<th>n</th>
<th>Statement</th>
<th>ref(n)</th>
<th>def(n)</th>
<th>relevant(n)</th>
</tr>
</thead>
</table>

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 $\text{relevant}(n)$.
3. For $n$'s immediate predecessor $m$, compute $\text{relevant}(m)$ by:
   // first exclude all variables defined at $m$ (because we’re overwriting it)
   $\text{relevant}(m) := \text{relevant}(n) - \text{def}(m)$
   // if $m$ defines a variable that’s relevant at $n$
   if $\text{def}(m)$ in $\text{relevant}(n)$ then
     // include the variables that are referenced at $m$
     $\text{relevant}(m) := \text{relevant}(m) \cup \text{ref}(m)$
     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
<table>
<thead>
<tr>
<th>n</th>
<th>Statement</th>
<th>ref(n)</th>
<th>def(n)</th>
<th>relevant(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b = 1</td>
<td></td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c = 2</td>
<td></td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>d = 3</td>
<td></td>
<td>d</td>
<td>b, c</td>
</tr>
<tr>
<td>4</td>
<td>a = d</td>
<td></td>
<td>a</td>
<td>b, c</td>
</tr>
<tr>
<td>5</td>
<td>d = b + d</td>
<td>b, d</td>
<td>d</td>
<td>b, c</td>
</tr>
<tr>
<td>6</td>
<td>b = b + 1</td>
<td></td>
<td>b</td>
<td>b, c</td>
</tr>
<tr>
<td>7</td>
<td>a = b + c</td>
<td>b, c</td>
<td>a</td>
<td>b, c</td>
</tr>
<tr>
<td>8</td>
<td>print a</td>
<td></td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

**Bolded n are included in the slice**

**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 $\text{ref}(c)$. The original slice (for $<n, V>$) will now also include all nodes in the slice for $<c, \text{ref}(c)>$
  - Union the relevant sets (e.g., $\text{relevant}(m_1)$ and $\text{relevant}(m_2)$) for cases where we have multiple descendants with a shared predecessor
    - (Remember that once we have control flow, we can have multiple descendants!)
We're not done yet! Remember slice for node 5 w.r.t. ref(5)!

slice currently contains: 10, 8, 7, 6, 5, 3, 2, 1
Let's take care of that subslice

<table>
<thead>
<tr>
<th>n</th>
<th>Statement</th>
<th>ref(n)</th>
<th>def(n)</th>
<th>control(n)</th>
<th>relevant(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b = 1</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c = 2</td>
<td></td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>d = 3</td>
<td></td>
<td>d</td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>4</td>
<td>a = d</td>
<td>d</td>
<td>a</td>
<td></td>
<td>{d}</td>
</tr>
<tr>
<td>5</td>
<td>if a then</td>
<td>a</td>
<td></td>
<td></td>
<td>{a}</td>
</tr>
<tr>
<td>6</td>
<td>d = b + d</td>
<td>b, d</td>
<td>d</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>c = b + d</td>
<td>b, d</td>
<td>c</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

else

<table>
<thead>
<tr>
<th>n</th>
<th>Statement</th>
<th>ref(n)</th>
<th>def(n)</th>
<th>control(n)</th>
<th>relevant(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>b = b + 1</td>
<td>b</td>
<td>b</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>d = b + 1</td>
<td>b</td>
<td>d</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

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) \cup ref(4) = {} \cup \{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) \cup ref(3) = {} \cup {} = {}  
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 :)