Liveness Analysis

propagating liveness

- Suppose we have a set of variables that are live at a particular point in the program
- What does it mean to “execute” a statement backwards?

{???}

```
A = B * C
```

{A, D}

propagating liveness

- Suppose we have a set of variables that are live at a particular point in the program
- What does it mean to “execute” a statement backwards?

{B, C, D}

```
A = B * C
```

{A, D}

propagating liveness

- Suppose we have a set of variables that are live at a particular point in the program
- What does it mean to "execute" a statement backwards?

{B, C, D} L_{in}

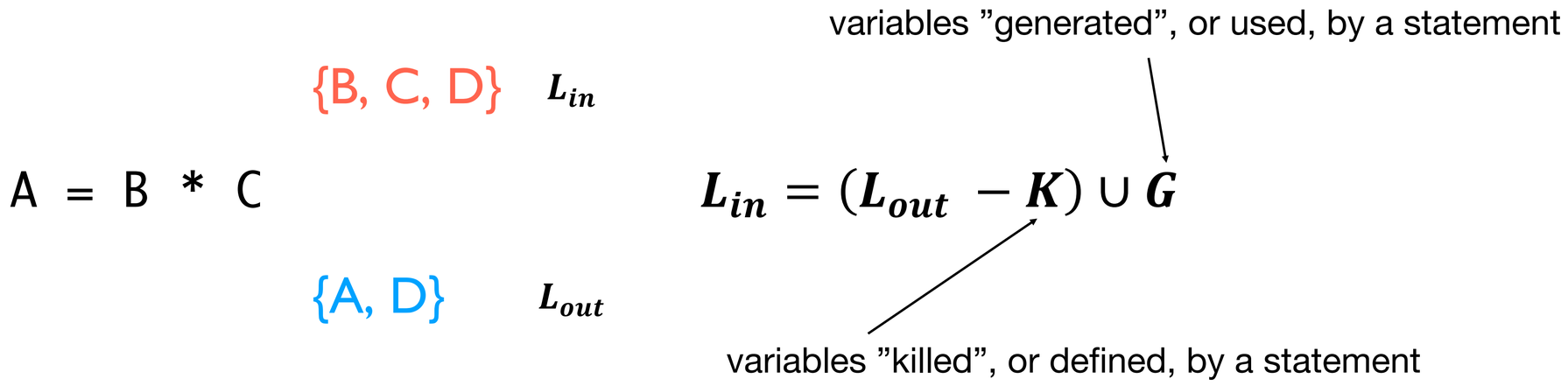
```
A = B * C
```

$$L_{in} = (L_{out} - K) \cup G$$

{A, D} L_{out}

propagating liveness

- Suppose we have a set of variables that are live at a particular point in the program
- What does it mean to “execute” a statement backwards?



liveness example

What is live in this code?

```
1:    A = B + C
2:    C = A + B
3:    T1 = B + C
4:    T2 = T1 + C
5:    D = T2
6:    E = A + B
7:    B = E + D
8:    A = C + D
9:    T3 = A + B
10:   WRITE(T3)
```

{B, C}
{A, B}
{A, B, C}
{A, B, C, T1}
{A, B, C, T2}
{A, B, C, D}
{C, D, E}
{B, C, D}
{A, B}
{T3}
{}

what about aliasing?

- Aliasing, as usual is a problem
- Reminder: compilers must be conservative
- Liveness is a *may* property → OK to say something is live when it isn't
  - This *may* be used in the future (even if it really won't be)

- Deal with aliasing by being conservative:
  - A variable stops being live when it is written to
  - Only *kill* variables that are *definitely* written to

next: finding dead code