



COM 3529

# SOFTWARE TESTING & ANALYSIS

Professor Phil McMinn

## 3.1 White Box Coverage Criteria based on Data Flow Analysis

# Definition

```
int x = y;
```

```
public int method (int a) {  
    // ...  
}
```



# Definition

```
int x = y;
```

```
public int method (int a) {  
    // ...  
}
```



# Use

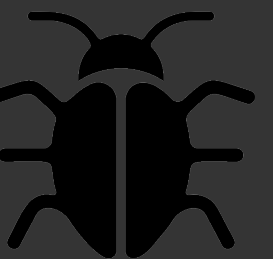
```
int x = y;
```

```
System.out.println("Hello " + name);
```

```
o.update();
```

```
if (a > b) {  
    // ...  
}
```

```
return result;
```



# Use

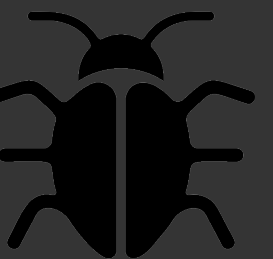
```
int x = y;
```

```
System.out.println("Hello " + name);
```

```
o.update();
```

```
if (a > b) {  
    // ...  
}
```

```
return result;
```



# Definitions *and* Uses

int **x** = **y**;

definition use

The diagram illustrates the concepts of definition and use in a code snippet. The code is 'int x = y;'. The variable 'x' is highlighted in blue, and the variable 'y' is highlighted in orange. Below 'x' is the word 'definition' in blue, and below 'y' is the word 'use' in orange. Two white curved arrows point from 'x' to 'definition' and from 'y' to 'use'.



# Definitions *and* Uses

int **x** = **y**;

definition

use

**x** = **x** + 1;

definition

use

x += 1;

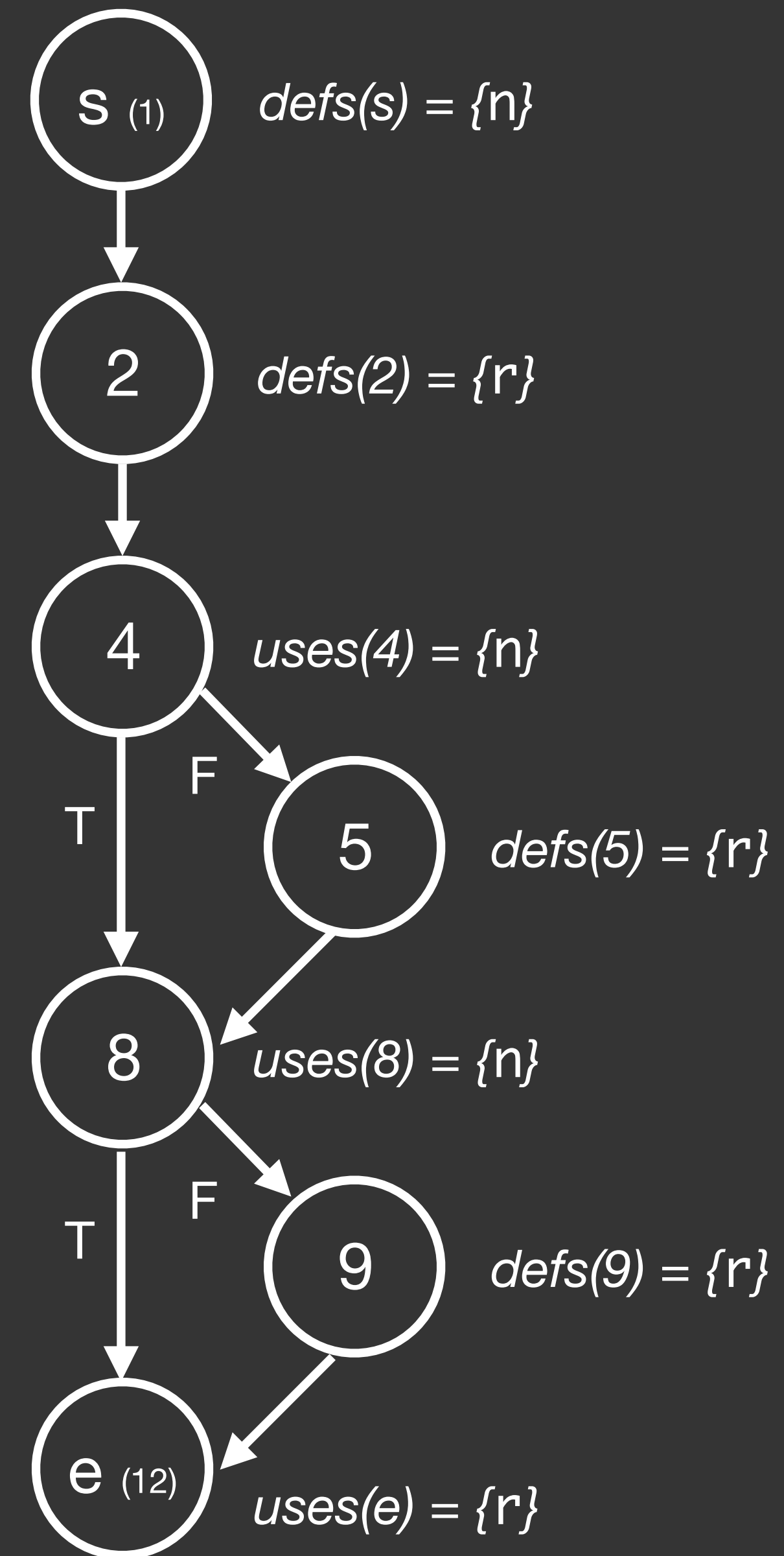
x ++;



```

1  public static int sign(int n) {
2      int r = 0;
3
4      if (n > 0) {
5          r = 1;
6      }
7
8      if (n < 0) {
9          r = -1;
10     }
11
12     return r;
13 }

```

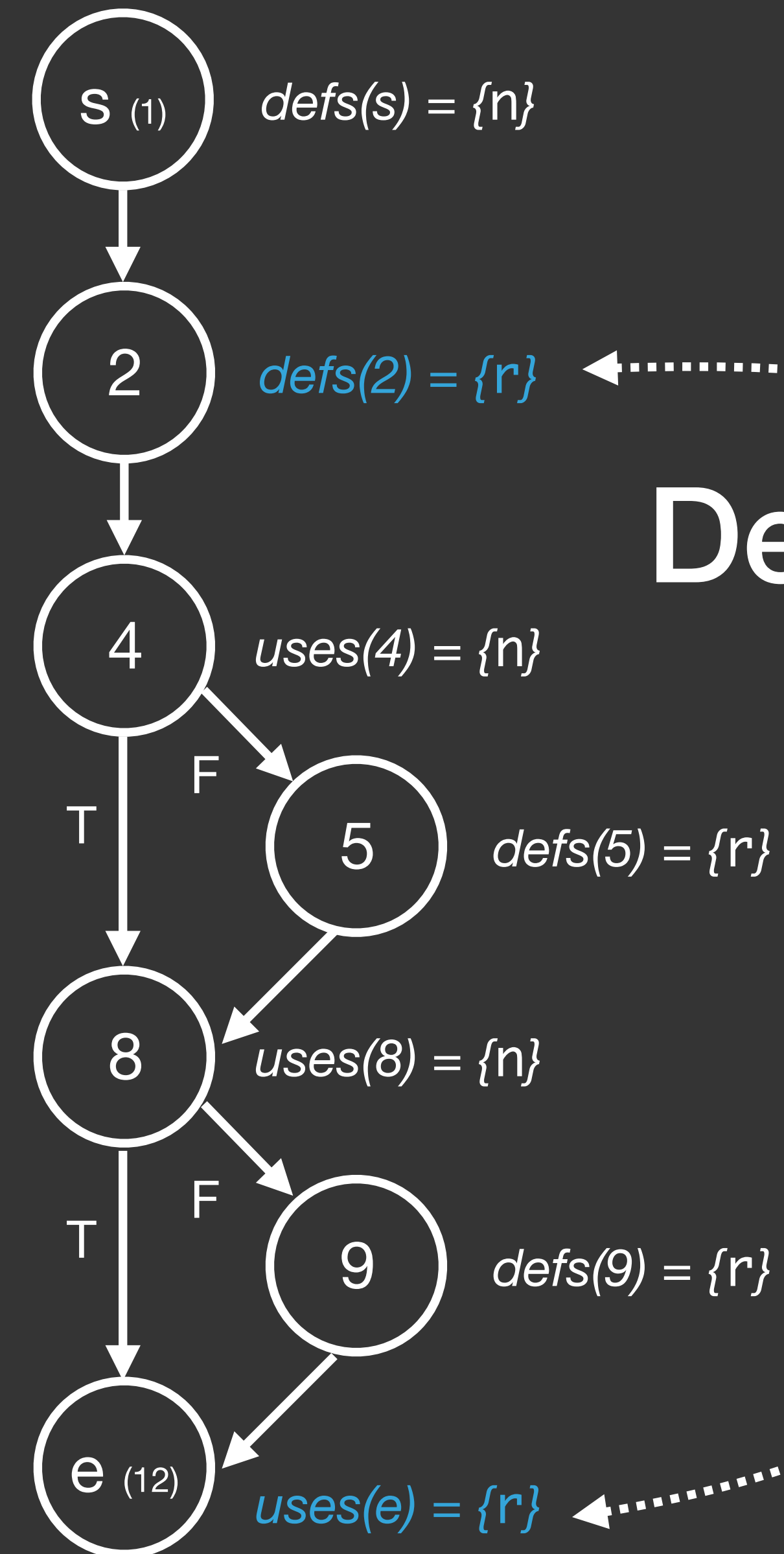




```

1  public static int sign(int n) {
2      int r = 0;
3
4      if (n > 0) {
5          r = 1;
6      }
7
8      if (n < 0) {
9          r = -1;
10     }
11
12     return r;
13 }

```



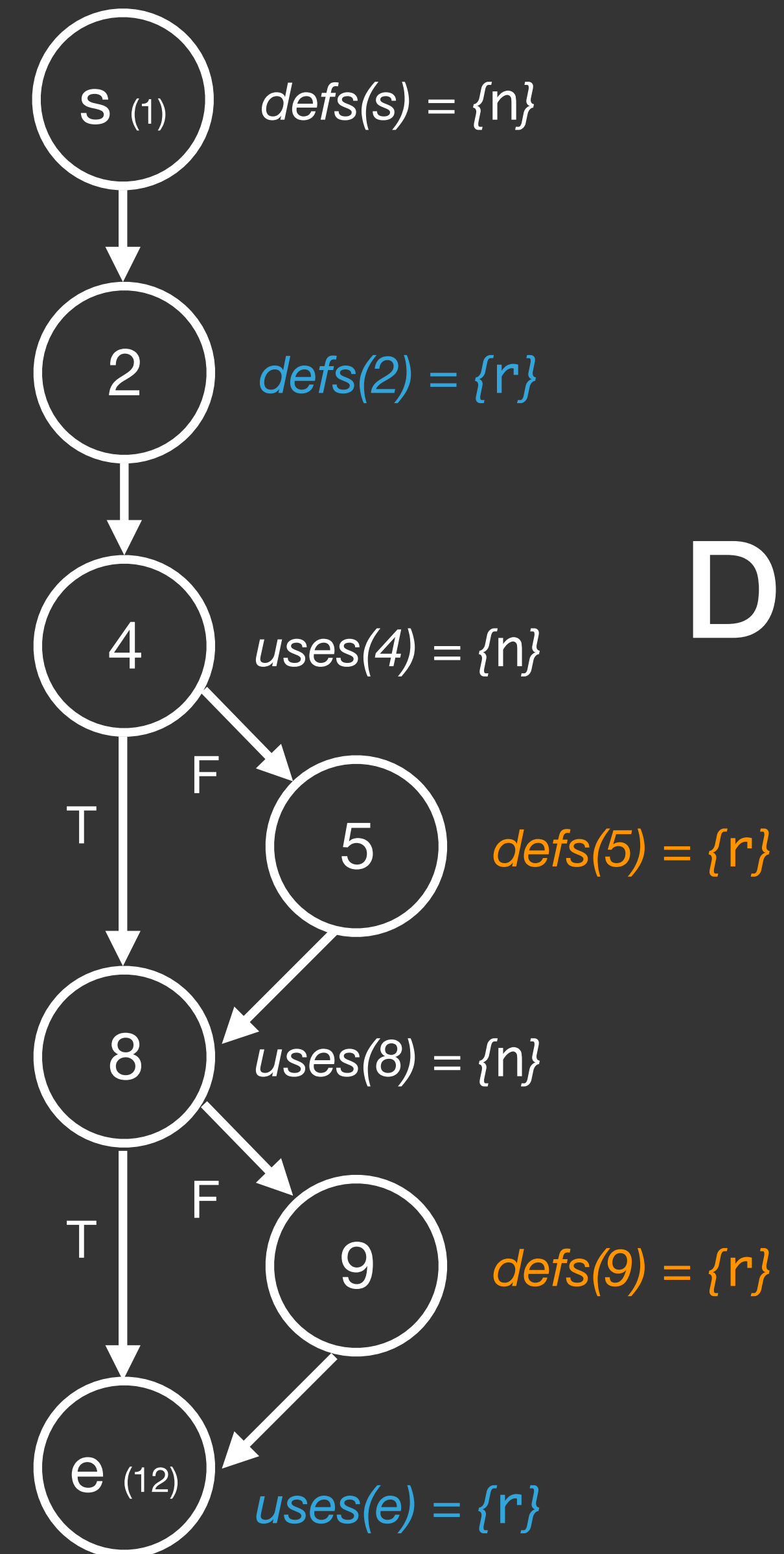
**Definition-Use  
Pair**



```

1  public static int sign(int n) {
2      int r = 0;
3
4      if (n > 0) {
5          r = 1;
6      }
7
8      if (n < 0) {
9          r = -1;
10     }
11
12     return r;
13 }

```



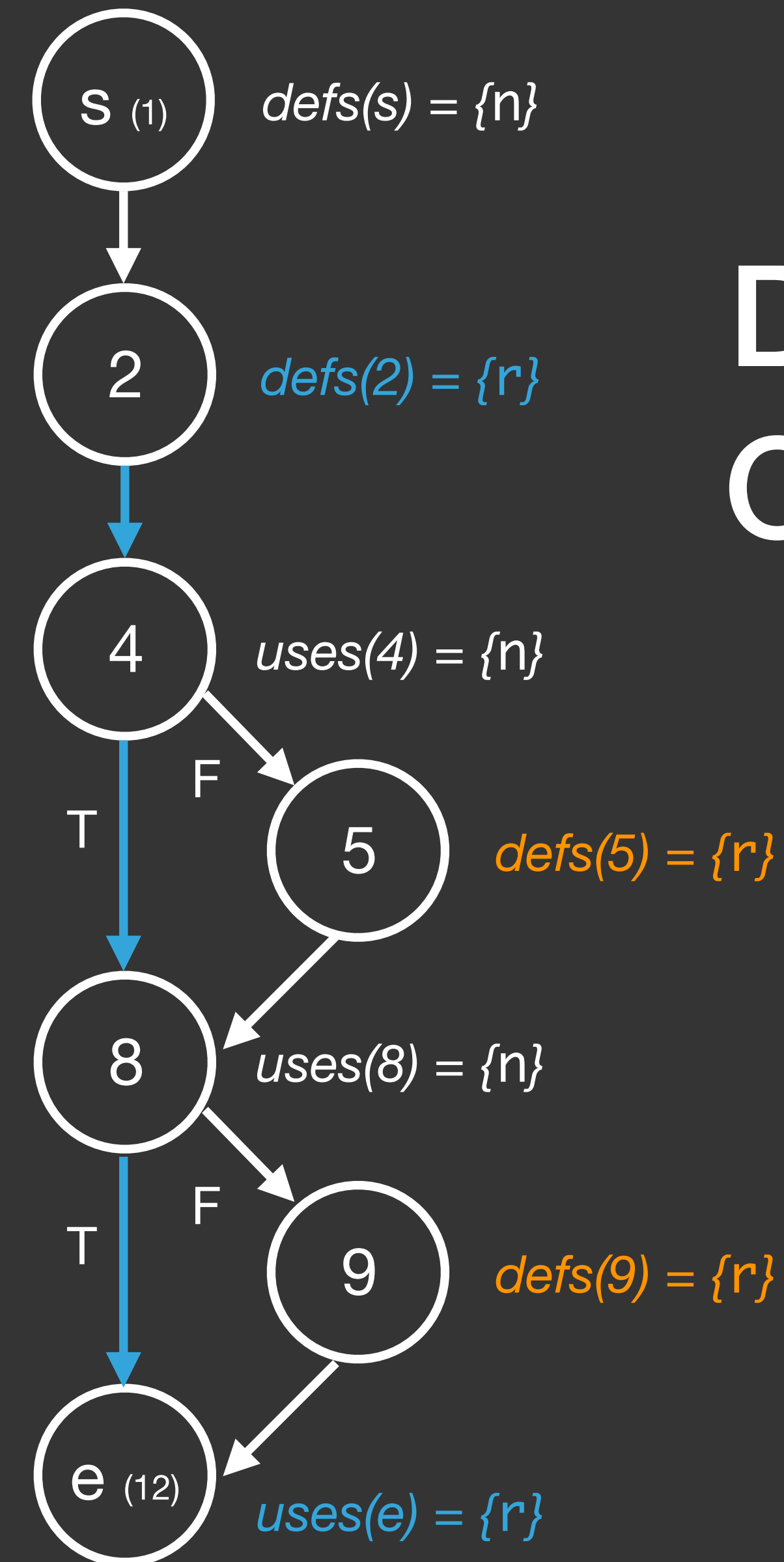
# Killing Definitions



```

1  public static int sign(int n) {
2      int r = 0;
3
4      if (n > 0) {
5          r = 1;
6      }
7
8      if (n < 0) {
9          r = -1;
10     }
11
12     return r;
13 }

```



## Definition-Clear Path

The definition *reaches* the use



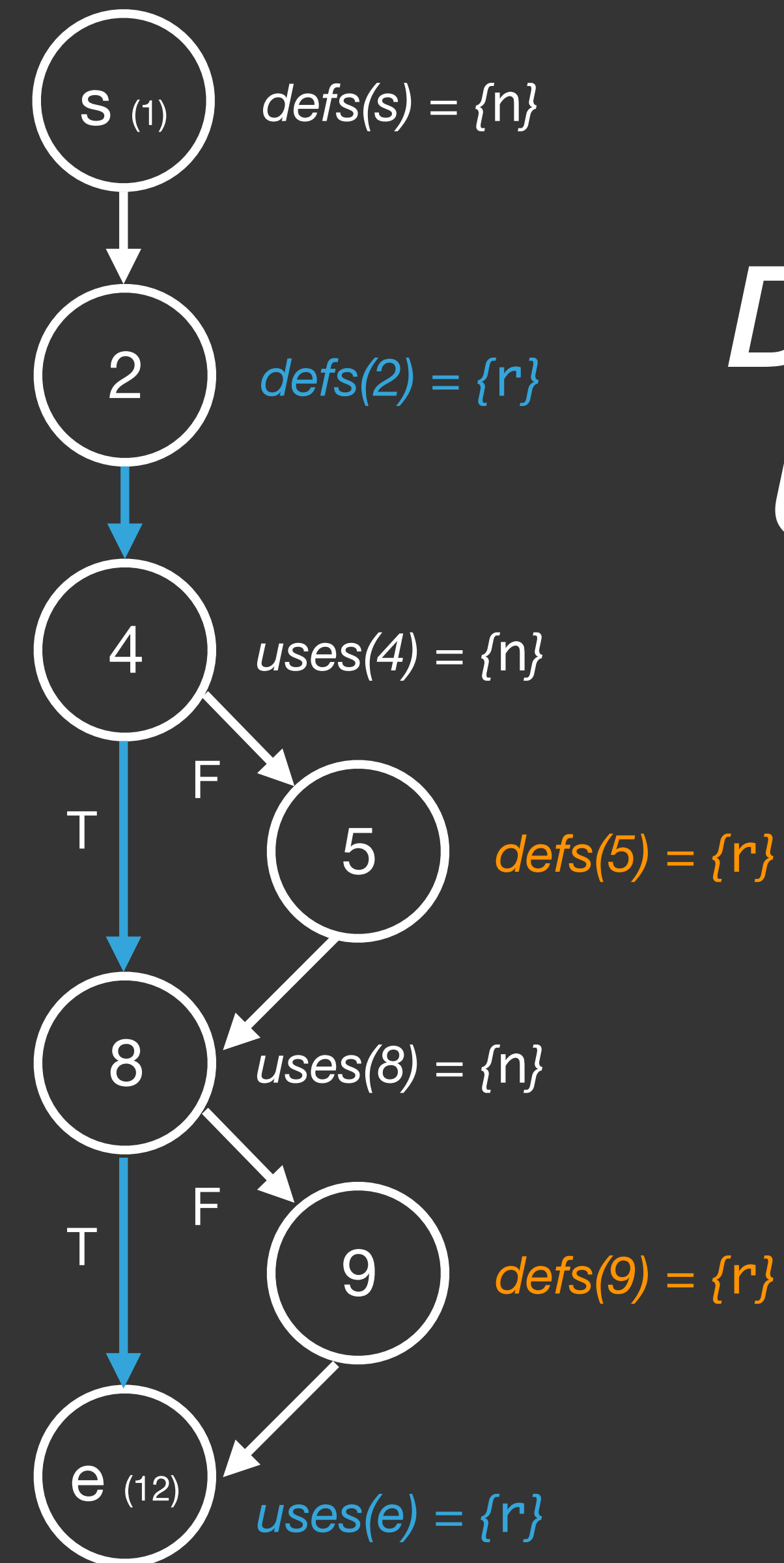
Formally, a path from  $n_i$  to  $n_j$  is definition-clear with respect to a variable  $v$  if for each node  $n_k$  on the path between  $n_i$  and  $n_j$ , (i.e.,  $n_k \neq n_i \wedge n_k \neq n_j$ ),  $v \notin \text{defs}(n_k)$ . That is, none of the nodes between  $n_i$  and  $n_j$  is a killing definition. If a definition-clear path exists from a definition of  $v$  at  $n_i$  to a use of  $v$  at  $n_j$ , the definition of  $v$  is said to **reach** the use at  $n_j$ .



```

1  public static int sign(int n) {
2      int r = 0;
3
4      if (n > 0) {
5          r = 1;
6      }
7
8      if (n < 0) {
9          r = -1;
10     }
11
12     return r;
13 }

```



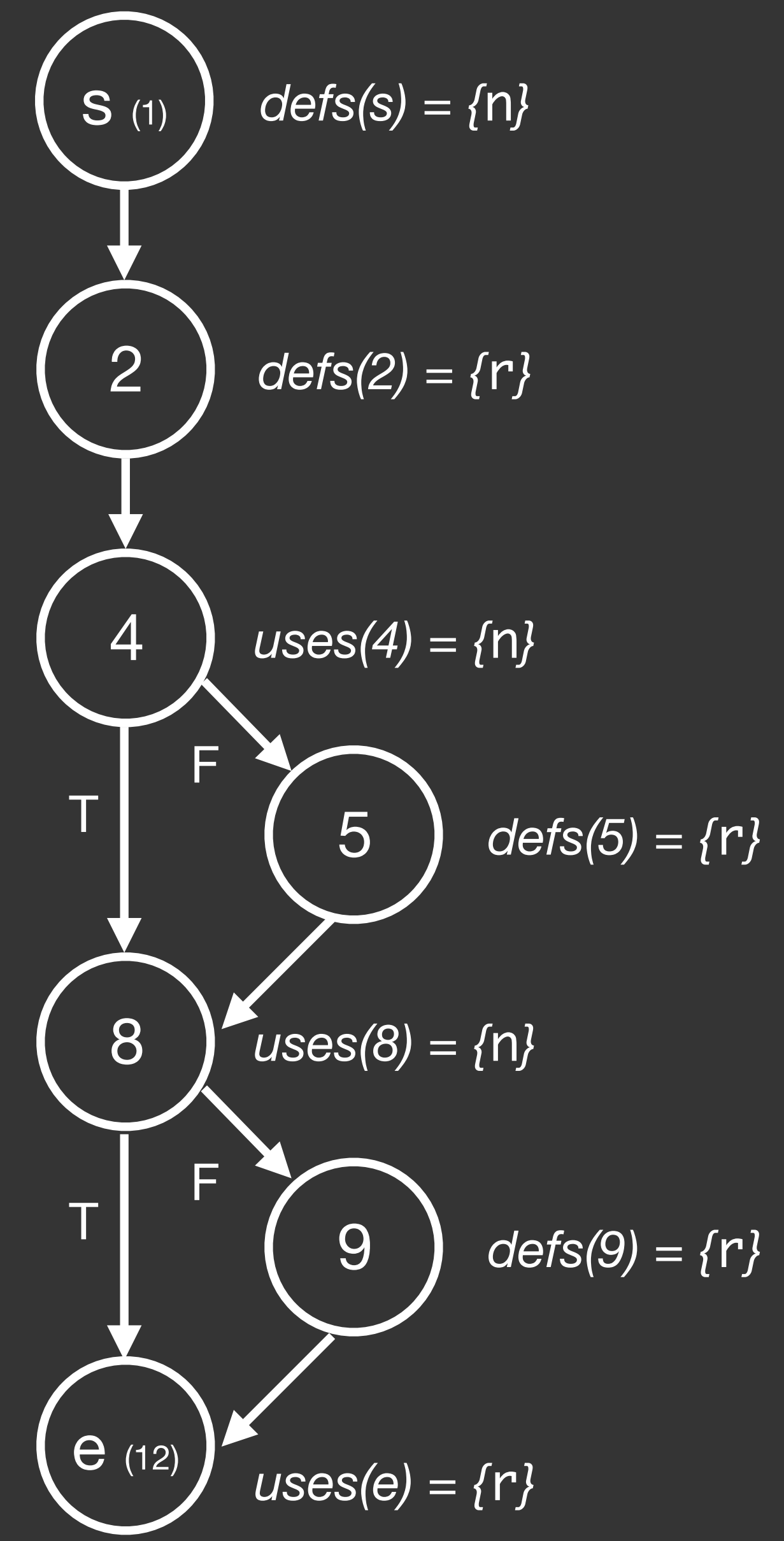
## Definition-Use Path



```
1 public static int sign(int n) {
2     int r = 0;
3
4     if (n > 0) {
5         r = 1;
6     }
7
8     if (n < 0) {
9         r = -1;
10    }
11
12    return r;
13 }
```

# The set *DU*

| No. | Variable | Definition | Use | Definition-Use Path |
|-----|----------|------------|-----|---------------------|
| 1   | r        | 2          | e   | 2 → 4 → 8 → e       |
| 2   | r        | 5          | e   | 5 → 8 → e           |
| 3   | r        | 9          | e   | 9 → e               |
| 4   | n        | s          | 4   | s → 2 → 4           |
| 5   | n        | s          | 8   | s → 2 → 4 → 8       |
| 6   | n        | s          | 8   | s → 2 → 4 → 5 → 8   |



# All Defs Coverage

Each definition reaches at least one use of the same variable

| No. | Variable | Definition | Use | Definition-Use Path   |
|-----|----------|------------|-----|---|
| 1   | r        | 2          | e   | 2 $\rightarrow$ 4 $\rightarrow$ 8 $\rightarrow$ e ✓               |
| 2   | r        | 5          | e   | 5 $\rightarrow$ 8 $\rightarrow$ e ✓                               |
| 3   | r        | 9          | e   | 9 $\rightarrow$ e ✓   |
| 4   | n        | s          | 4   | s $\rightarrow$ 2 $\rightarrow$ 4 ✓                               |
| 5   | n        | s          | 8   | s $\rightarrow$ 2 $\rightarrow$ 4 $\rightarrow$ 8                 |
| 6   | n        | s          | 8   | s $\rightarrow$ 2 $\rightarrow$ 4 $\rightarrow$ 5 $\rightarrow$ 8 |





# All Uses Coverage

Each definition reaches each use of the same variable

| No. | Variable | Definition | Use | Definition-Use Path   |
|-----|----------|------------|-----|---|
| 1   | r        | 2          | e   | 2 $\rightarrow$ 4 $\rightarrow$ 8 $\rightarrow$ e ✓               |
| 2   | r        | 5          | e   | 5 $\rightarrow$ 8 $\rightarrow$ e ✓                               |
| 3   | r        | 9          | e   | 9 $\rightarrow$ e ✓   |
| 4   | n        | s          | 4   | s $\rightarrow$ 2 $\rightarrow$ 4 ✓                               |
| 5   | n        | s          | 8   | s $\rightarrow$ 2 $\rightarrow$ 4 $\rightarrow$ 8 ✓               |
| 6   | n        | s          | 8   | s $\rightarrow$ 2 $\rightarrow$ 4 $\rightarrow$ 5 $\rightarrow$ 8 |





# All Def-Use Path Coverage

Every path in *DU* needs to be executed

| No. | Variable | Definition | Use | Definition-Use Path |
|-----|----------|------------|-----|---------------------|
| 1   | r        | 2          | e   | 2 → 4 → 8 → e ✓     |
| 2   | r        | 5          | e   | 5 → 8 → e ✓         |
| 3   | r        | 9          | e   | 9 → e ✓             |
| 4   | n        | s          | 4   | s → 2 → 4 ✓         |
| 5   | n        | s          | 8   | s → 2 → 4 → 8 ✓     |
| 6   | n        | s          | 8   | s → 2 → 4 → 5 → 8 ✓ |



# When Should You Use Data-Flow Testing?



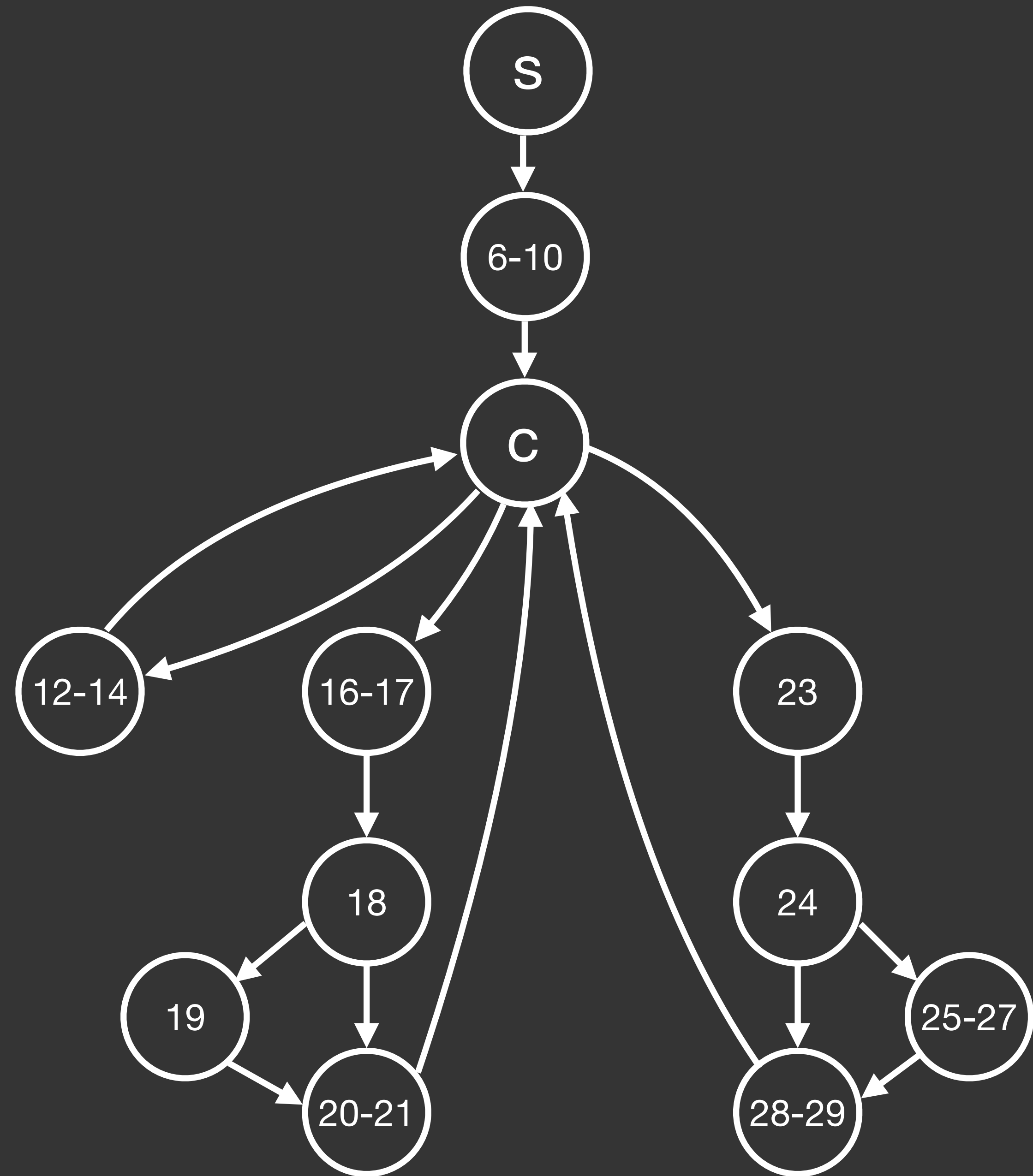
```
1 public class VendingMachine {
2
3     private int totalCoins, currentCoins;
4     private boolean allowVend;
5
6     public VendingMachine() {
7         totalCoins = 0;
8         currentCoins = 0;
9         allowVend = false;
10    }
11
12    public void returnCoins() {
13        currentCoins = 0;
14    }
15
16    public void addCoin() {
17        currentCoins ++;
18        if (currentCoins > 1) {
19            allowVend = true;
20        }
21    }
22
23    public void vend() {
24        if (allowVend) {
25            totalCoins += currentCoins;
26            currentCoins = 0;
27            allowVend = false;
28        }
29    }
30 }
```



```

1 public class VendingMachine {
2
3     private int totalCoins, currentCoins;
4     private boolean allowVend;
5
6     public VendingMachine() {
7         totalCoins = 0;
8         currentCoins = 0;
9         allowVend = false;
10    }
11
12    public void returnCoins() {
13        currentCoins = 0;
14    }
15
16    public void addCoin() {
17        currentCoins ++;
18        if (currentCoins > 1) {
19            allowVend = true;
20        }
21    }
22
23    public void vend() {
24        if (allowVend) {
25            totalCoins += currentCoins;
26            currentCoins = 0;
27            allowVend = false;
28        }
29    }
30 }

```



```

1 public class VendingMachine {
2
3     private int totalCoins, currentCoins;
4     private boolean allowVend;
5
6     public VendingMachine() {
7         totalCoins = 0;
8         currentCoins = 0;
9         allowVend = false;
10    }
11
12    public void returnCoins() {
13        currentCoins = 0;
14    }
15
16    public void addCoin() {
17        currentCoins ++;
18        if (currentCoins > 1) {
19            allowVend = true;
20        }
21    }
22
23    public void vend() {
24        if (allowVend) {
25            totalCoins += currentCoins;
26            currentCoins = 0;
27            allowVend = false;
28        }
29    }
30 }

```

