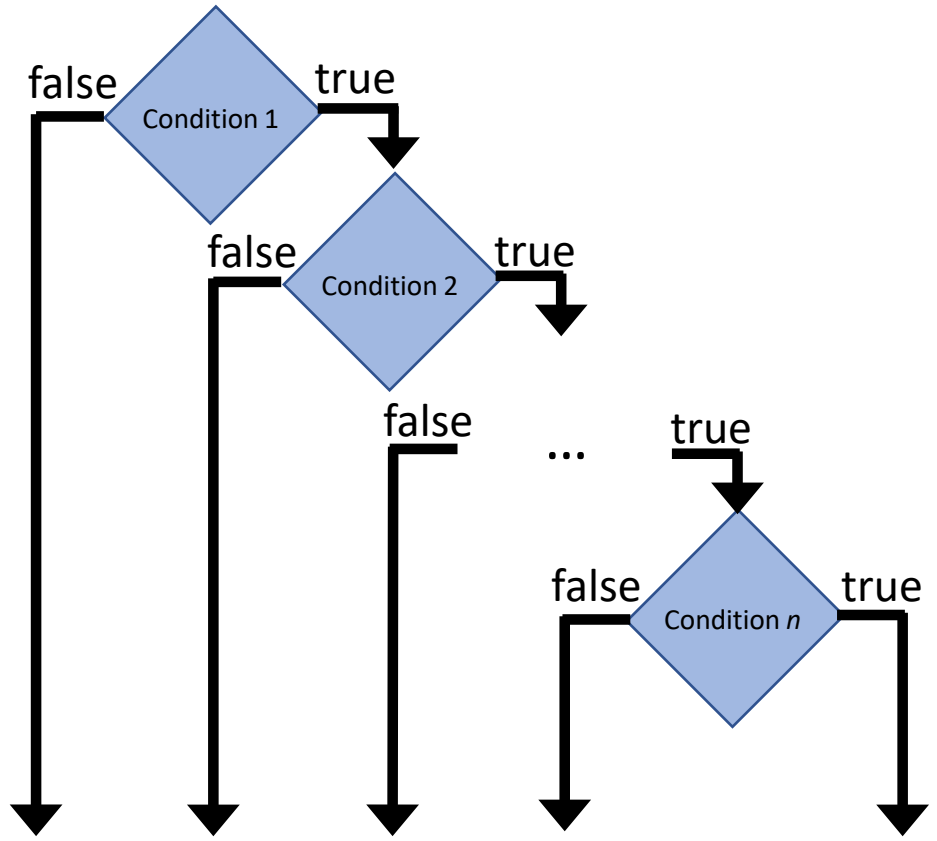
```java
public class Puzzle7 {

    public static void main(String[] args) {
        System.out.println(getResult());
    }

    private static boolean getResult() {
        try {
            try {
                return true;
            } finally {
                return true;
            }
        } finally {
            return false;
        }
    }

}
```

```java
public class Puzzle8 {

    public static void main(String[] args) {
        run();
    }

    private static void run() {
        try {
            System.out.println("Hello");
            System.exit(0);
        } finally {
            System.out.println("Goodbye");
        }
    }

}
```

Counting Program Paths

- How many paths are there for *n* nested branches?

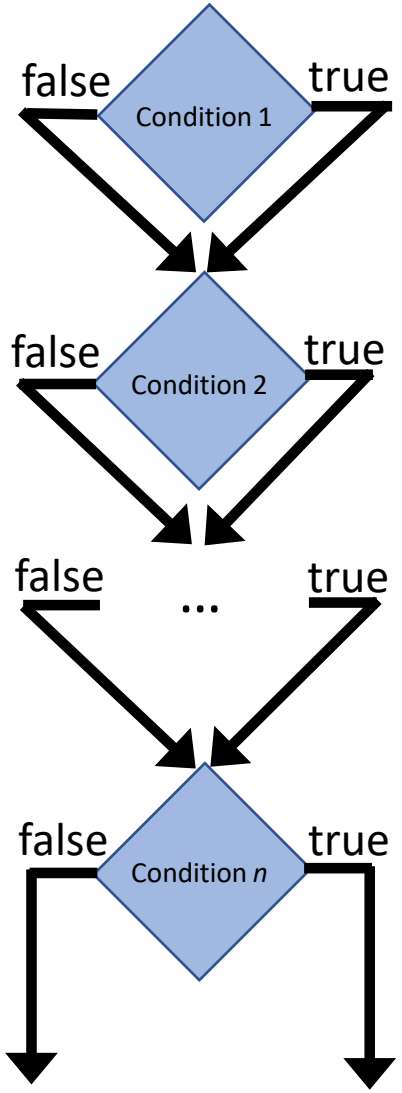
```
if(condition_1){
   if(condition_2){
      if(condition_3){
         …
         if(condition_n){
            // conditions 1 through n
            // must all be true to reach here
         }
      }
   }
}
```



Counting Program Paths

- How many paths are there for *n* non-nested branches?

```
if(condition_1){
   // code block 1
}
if(condition_2){
   // code block 2
}
if(condition_3){
   // code block 3
}
…
if(condition_n){
   // code block n
}
```

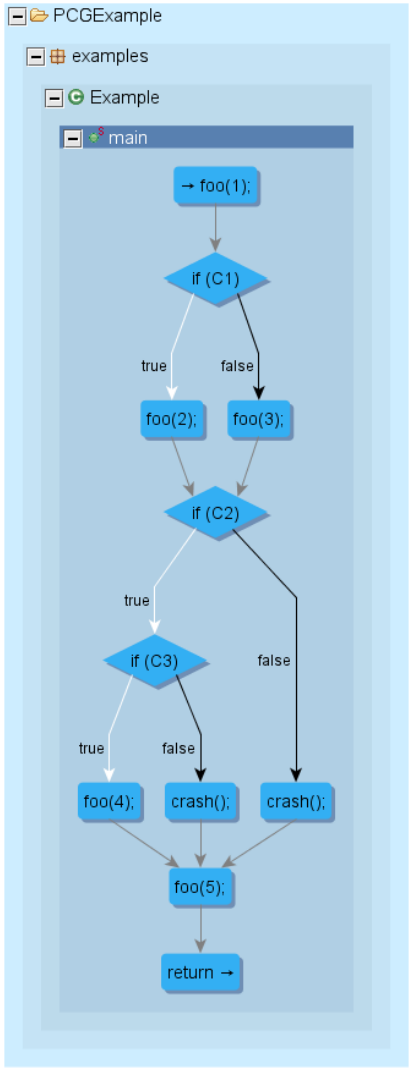


Truth Tables

- Given a truth table with *n* Boolean variables, how many rows are there in the truth table?

Truth Tables and Control Flow Graphs

```
1  public void main() {
2      foo(1);
3      if (C1) {
4          foo(2);
5      } else {
6          foo(3);
7      }
8      if (C2) {
9          if (C3) {
10             foo(4);
11         } else {
12             crash();
13         }
14     } else {
15         crash();
16     }
17     foo(5);
18     return;
19 }
```

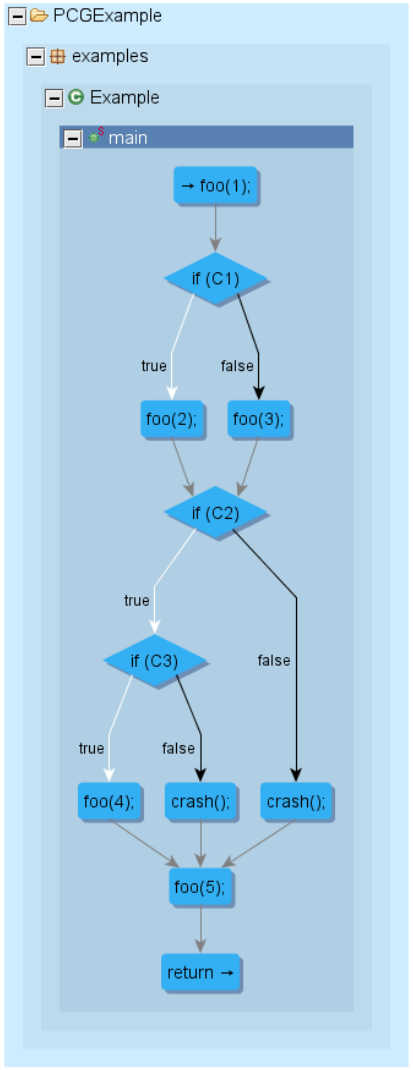


2^3=8 possible values for the tuple (C1, C2, C3)

C1	C2	C3	Behavior
False	False	False	
False	False	True	
False	True	False	
False	True	True	
True	False	False	
True	False	True	
True	True	False	
True	True	True	

Truth Tables and Control Flow Graphs

```
1  public void main() {
2      foo(1);
3      if (C1) {
4          foo(2);
5      } else {
6          foo(3);
7      }
8      if (C2) {
9          if (C3) {
10             foo(4);
11         } else {
12             crash();
13         }
14     } else {
15         crash();
16     }
17     foo(5);
18     return;
19 }
```

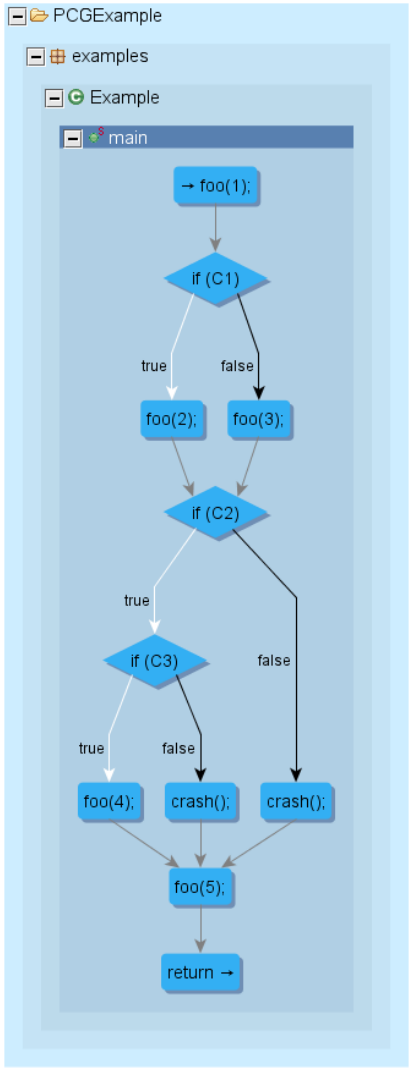


If C2 is false then C3 is not evaluated.

C1	C2	C3	Behavior
False	False	N/A	
False	False	N/A	
False	True	False	
False	True	True	
True	False	N/A	
True	False	N/A	
True	True	False	
True	True	True	

Truth Tables and Control Flow Graphs

```
1  public void main() {
2      foo(1);
3      if (C1) {
4          foo(2);
5      } else {
6          foo(3);
7      }
8      if (C2) {
9          if (C3) {
10             foo(4);
11         } else {
12             crash();
13         }
14     } else {
15         crash();
16     }
17     foo(5);
18     return;
19 }
```

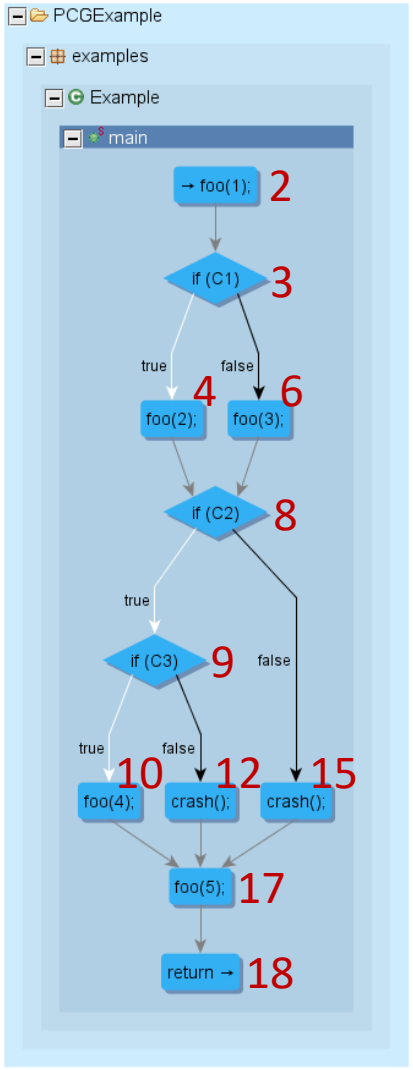


CFG has 6 paths.

C1	C2	C3	Behavior
False	False	N/A	
False	True	False	
False	True	True	
True	False	N/A	
True	True	False	
True	True	True	

Truth Tables and Control Flow Graphs

```
1  public void main() {
2      foo(1);
3      if (C1) {
4          foo(2);
5      } else {
6          foo(3);
7      }
8      if (C2) {
9          if (C3) {
10             foo(4);
11         } else {
12             crash();
13         }
14     } else {
15         crash();
16     }
17     foo(5);
18     return;
19 }
```

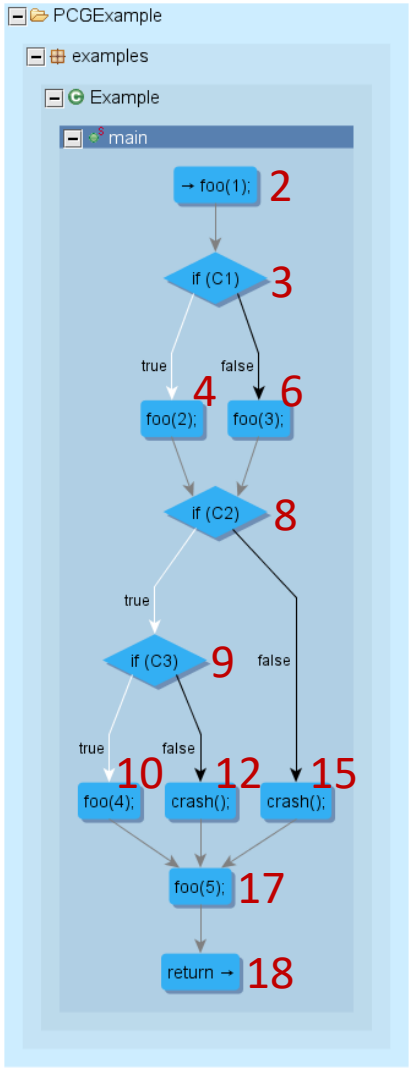


What paths include a “crash” event?

C1	C2	C3	Behavior
False	False	N/A	2,3,6,8,15,17,18
False	True	False	2,3,6,8,9,12,17,18
False	True	True	2,3,6,8,9,10,17,18
True	False	N/A	2,3,4,8,15,17,18
True	True	False	2,3,4,8,9,12,17,18
True	True	True	2,3,4,8,9,10,17,18

Truth Tables and Control Flow Graphs

```
1  public void main() {
2      foo(1);
3      if (C1) {
4          foo(2);
5      } else {
6          foo(3);
7      }
8      if (C2) {
9          if (C3) {
10             foo(4);
11         } else {
12             crash();
13         }
14     } else {
15         crash();
16     }
17     foo(5);
18     return;
19 }
```



4 of 6 behaviors have “crash” events.
2 of 6 behaviors do not have “crash” events.

C1	C2	C3	Behavior
False	False	N/A	2,3,6,8,15,17,18
False	True	False	2,3,6,8,9,12,17,18
~~False~~	~~True~~	~~True~~	~~2,3,6,8,9,10,17,18~~
True	False	N/A	2,3,4,8,15,17,18
True	True	False	2,3,4,8,9,12,17,18
~~True~~	~~True~~	~~True~~	~~2,3,4,8,9,10,17,18~~

Path Feasibility

- A control flow graph (CFG) is a graph representation that captures the paths that ***might*** be traversed through a program during its execution, (i.e. the orderings that the program's statements may be executed in at runtime).
- A feasible path is a path that *could be possible* to traverse during a program execution.
- An infeasible path is a path that is *not possible* to traverse during a program execution.
- A CFG does not distinguish between feasible and infeasible paths.

Path Feasibility

- Determining if a path is feasible requires computing variable inputs that satisfy the branch conditions that would allow for a given path, which is the Boolean satisfiability problem.
  - SAT was the first known NP-complete problem, as proved by Stephen Cook at the University of Toronto in 1971 and independently by Leonid Levin at the National Academy of Sciences in 1973. Until that time, the concept of an NP-complete problem did not even exist. [Wikipedia]

Dead Code Detection

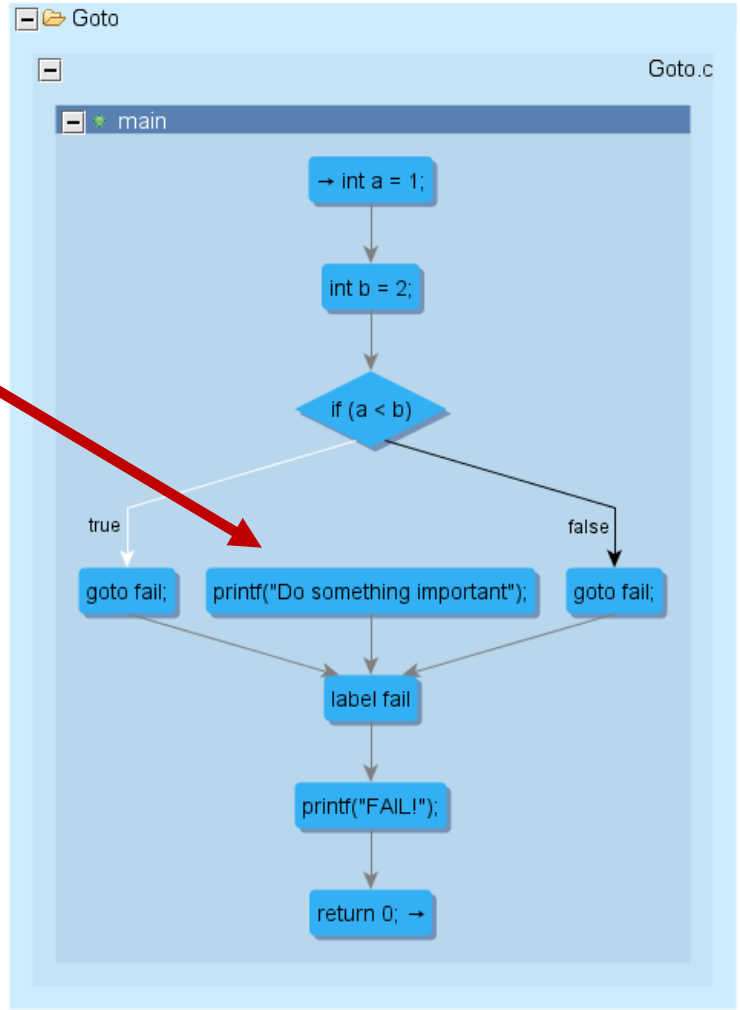
- A CFG can be used however to detect obviously dead code (not all dead code).
  - If the CFG has a structure graph root (no incoming edges) that is not the control flow root, then the statement corresponding to the root is dead code.
  - Q cfg = CommonQueries.*cfg(Common.functions("foo"));*
  - Q deadCode = cfg.roots().difference(cfg.nodes(XCSG.***controlFlowRoot));***

```
int main () {
   int a = 1;
   int b = 2;
   if (a<b)
    goto fail;
    goto fail;

   printf("Do something important");

   fail:
   printf("FAIL!");

   return 0;
}
```

Dead Code Detection

```
int main () {
    int a = 1;
    int b = 2;
    if (a<b)
     goto fail;
     goto fail;

    printf("Do something important");

    fail:
    printf("FAIL!");

    return 0;
}
```

Dead Code!



Path Counting

- Strategy 1: Create truth table and collapse after exploring all 2^n paths, always 2^n
- Strategy 2: DFS counting paths, worst case still 2^n

Why count paths?

- The number of paths is a measure of how complex a piece of code is
  - Cyclomatic Complexity is a cheap estimation of complexity
  - Cyclomatic Complexity = (|CFG Edges| – |CFG Nodes|) + 2
- Bugs tend to be located around more complex code

Activity: Does this program contain a vulnerability?

```
#include <stdio.h>
int main(int argc, char *argv) {
    char buf[64];
    strcpy(buf, argv[1]);
    return 0;
}
```

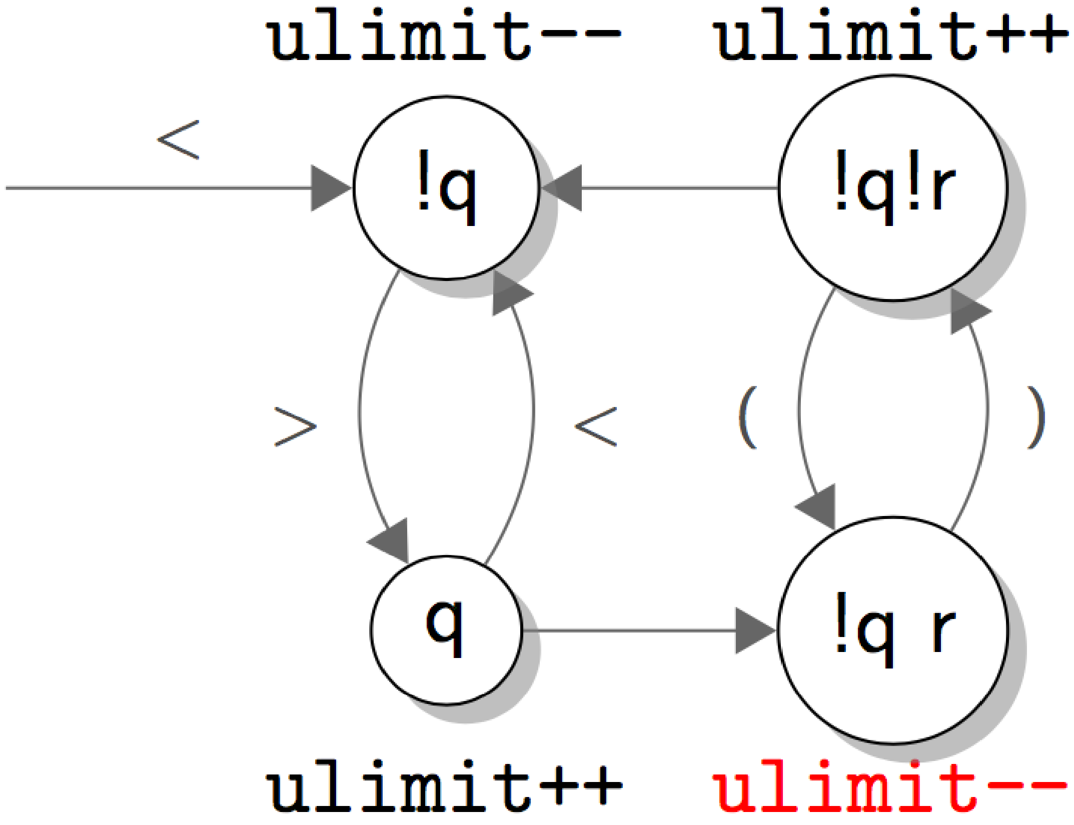
Activity: Does this program contain a vulnerability?

```
#define BUFFERSIZE 200
int copy_it (char* input , unsigned int length){
   char c, localbuf[BUFFERSIZE];
   unsigned int upperlimit = BUFFERSIZE - 10;
   unsigned int quotation = roundquote = FALSE;
   unsigned int input_index = output_index = 0;
   while (input_index < length){ c = input[input_index++];
      if((c == '<') && (!quotation)){ quotation = true; upperlimit--; }
      if((c == '>') && (quotation)){ quotation = false; upperlimit++; }
      if((c == '(') && (!quotation) && (!roundquote)){ roundquote = true; }
      if((c == ')') && (!quotation) && (roundquote)){ roundquote = false; upperlimit++; }
      // if there is sufficient space in the buffer, write the character
      if(output_index < upperlimit){ localbuf[output_index] = c; output_index++; }
   }
   if(roundquote){ localbuf[output_index] = ')'; output_index++; }
   if(quotation){ localbuf[output_index] = '>'; output_index++; }
   return output_index;
}
```

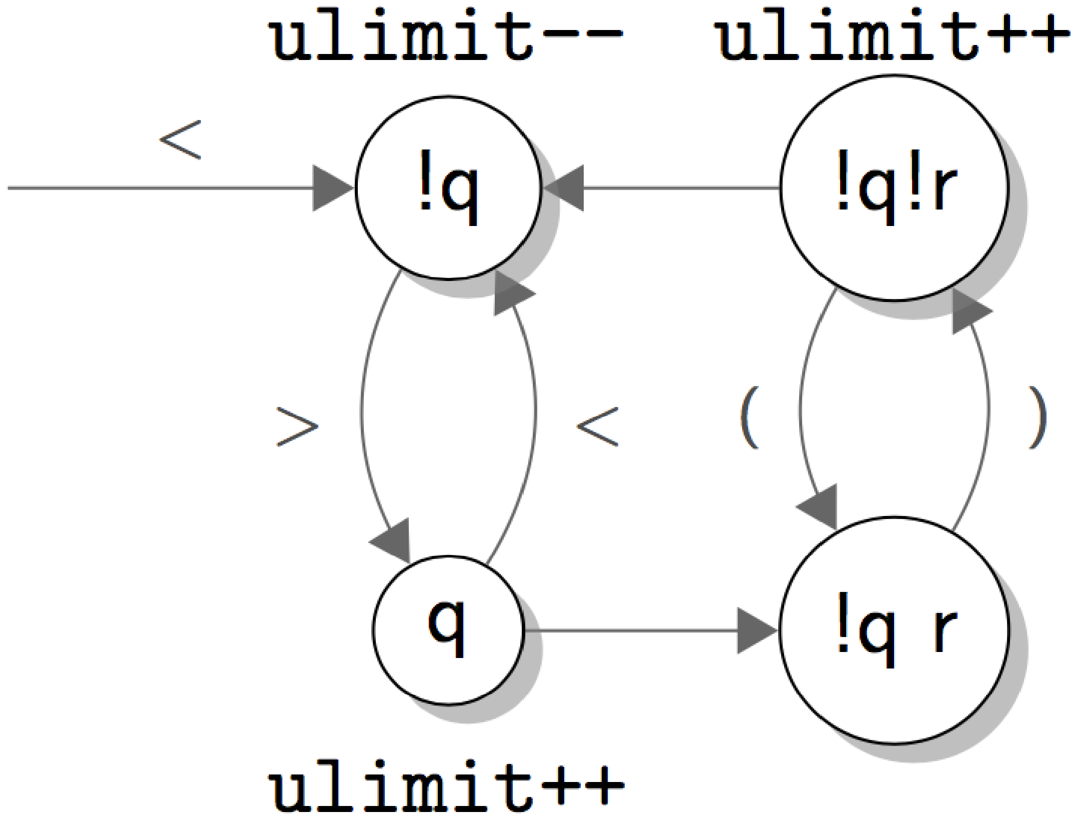
Activity: Does this program contain a vulnerability?

```
#define BUFFERSIZE 200
int copy_it (char* input , unsigned int length){
   char c, localbuf[BUFFERSIZE];
   unsigned int upperlimit = BUFFERSIZE - 10;
   unsigned int quotation = roundquote = FALSE;
   unsigned int input_index = output_index = 0;
   while (input_index < length){ c = input[input_index++];
      if((c == '<') && (!quotation)){ quotation = true; upperlimit--; }
      if((c == '>') && (quotation)){ quotation = false; upperlimit++; }
      if((c == '(') && (!quotation) && (!roundquote)){ roundquote = true; /* (missing) upperlimit--; */ }
      if((c == ')') && (!quotation) && (roundquote)){ roundquote = false; upperlimit++; }
      // if there is sufficient space in the buffer, write the character
      if(output_index < upperlimit){ localbuf[output_index] = c; output_index++; }
   }
   if(roundquote){ localbuf[output_index] = ')'; output_index++; }
   if(quotation){ localbuf[output_index] = '>'; output_index++; }
   return output_index;
}
```

Good:



Bad:



input = "Name Lastname < name@mail.org >
()
()"

Path Counting + Human Reasoning

- Human intuition: This code is complex and it has an array write → I bet someone messed this up and there is a vulnerability here.
  - They were correct: Sendmail Crackaddr CVE-2002-1337 (discovered by Mark Dowd)
  - Buffer overflow in an email address parsing function of Sendmail. Consists of a parsing loop using a state machine. ~500 LOC
  - Bounty for Static Analyzers since 2011 by Halvar Flake Halvar extracted a smaller version of the bug as an example of a hard problem for static analyzers. ~50 LOC

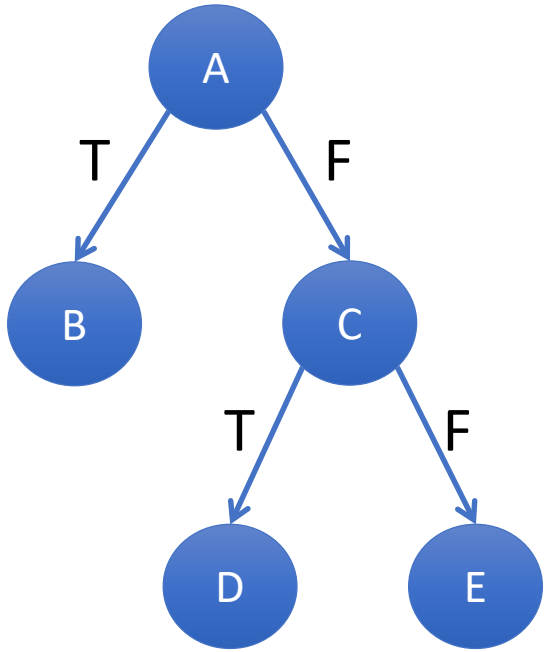
DFS Enumeration Strategy (Top Down)

Push A (root) on stack.

Stack: [A]

History:

Paths:



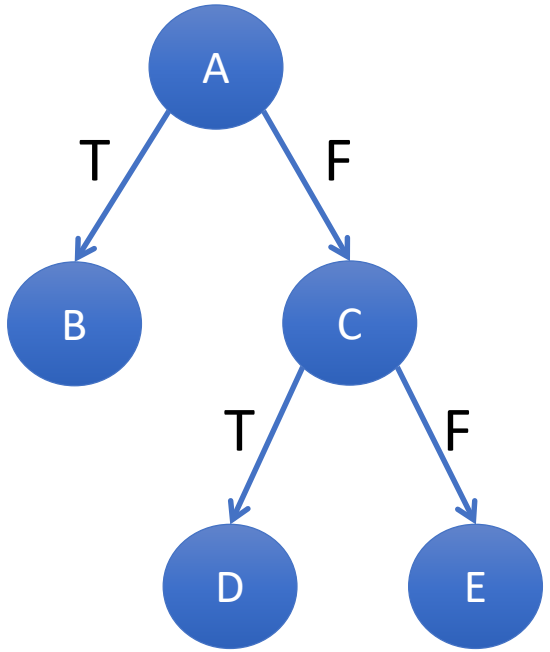
DFS Enumeration Strategy (Top Down)

Pop A onto history. A is not a leaf so push C,B.

Stack: [C, B]

History: [A]

Paths:



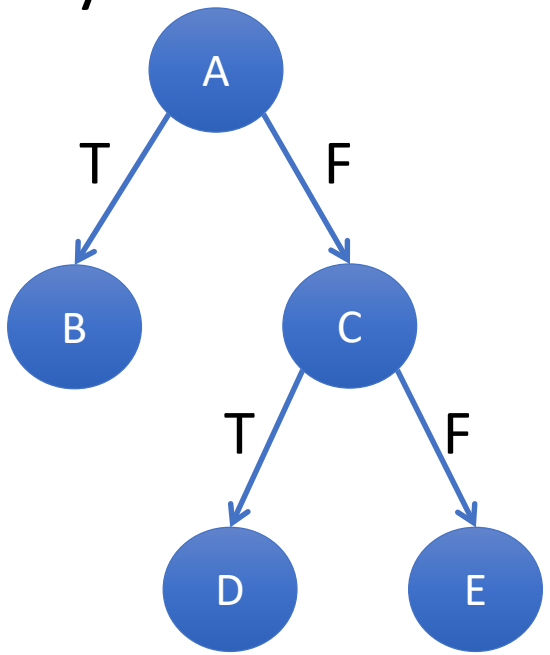
DFS Enumeration Strategy (Top Down)

Pop B onto history. B is a leaf so save path and trim history.

Stack: [C]

History: [A, ~~B~~]

Paths: [A, B]



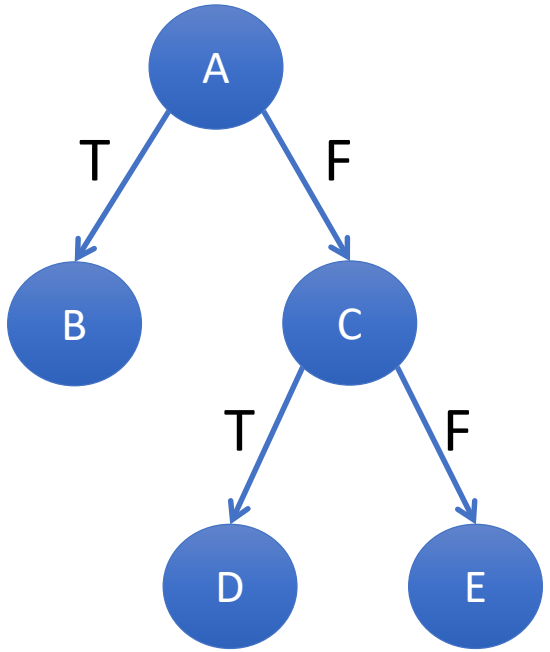
DFS Enumeration Strategy (Top Down)

Pop C onto history. C is not a leaf so push E,D.

Stack: [E, D]

History: [A, C]

Paths: [A, B]



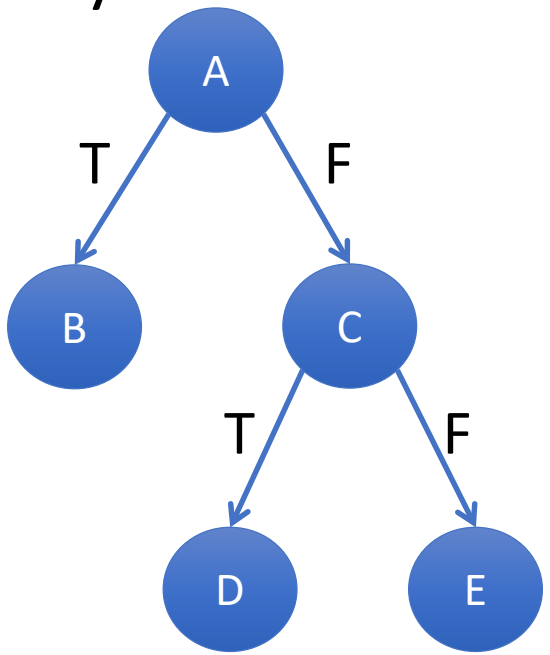
DFS Enumeration Strategy (Top Down)

Pop D onto history. D is a leaf so save path and trim history.

Stack: [E]

History: [A, C, ~~D~~]

Paths: [A, B], [A, C, D]



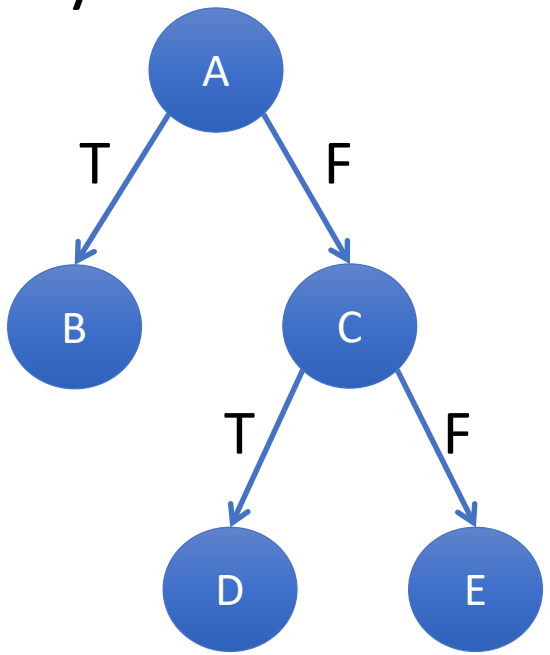
DFS Enumeration Strategy (Top Down)

Pop E onto history. E is a leaf so save path and trim history.

Stack: []

History: [A, C, ~~E~~]

Paths: [A, B], [A, C, D], [A, C, E]



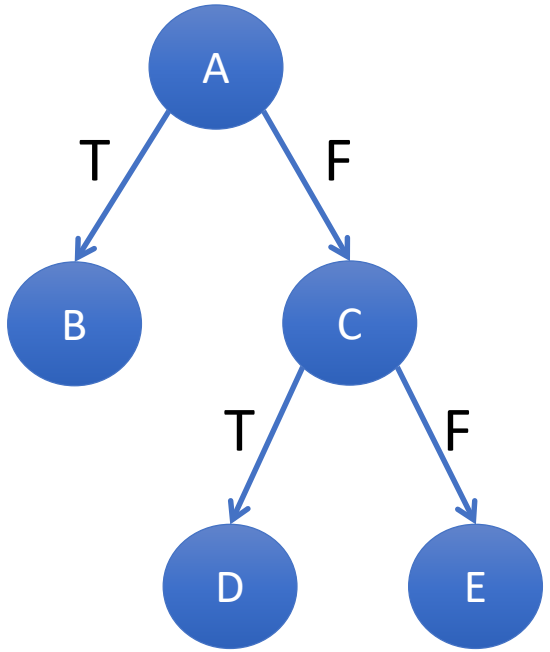
DFS Enumeration Strategy (Top Down)

Stack is empty. Paths are enumerated.

Stack: []

History: [A, C]

Paths: [A, B], [A, C, D], [A, C, E]



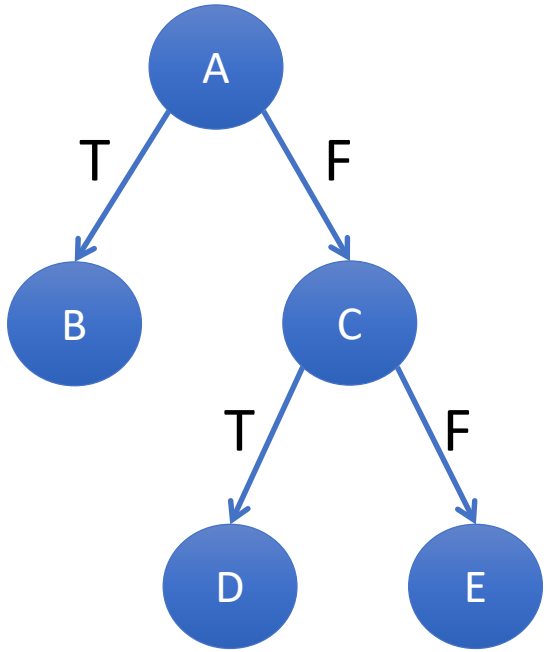
DFS Enumeration Strategy (Bottom Up)

Push B,D,E (leaves) on stack.

Stack: [B, D, E]

History:

Paths:



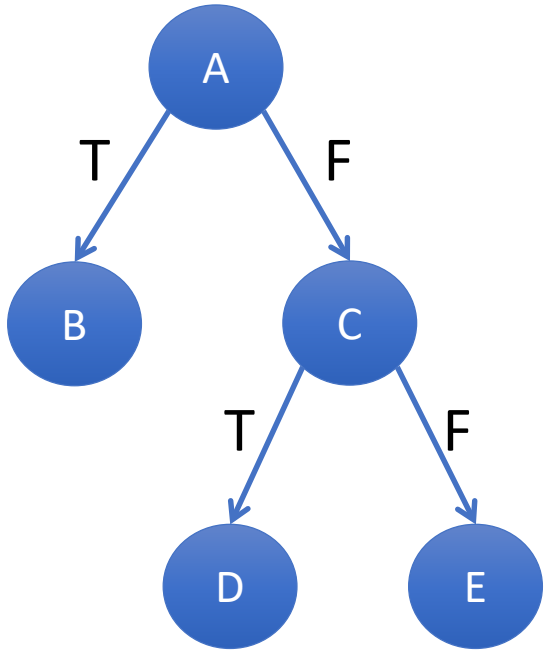
DFS Enumeration Strategy (Bottom Up)

Pop E onto history. E is not a root so push C.

Stack: [B, D, C]

History: [E]

Paths:



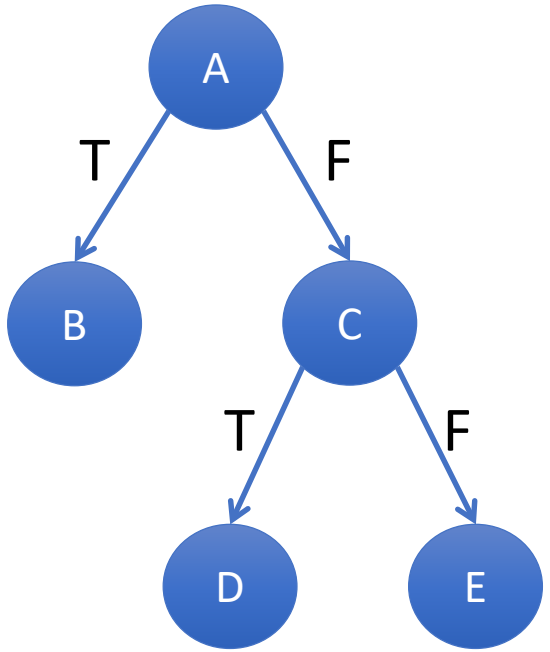
DFS Enumeration Strategy (Bottom Up)

Pop C onto history. C is not a root so push A.

Stack: [B, D, A]

History: [E, C]

Paths:



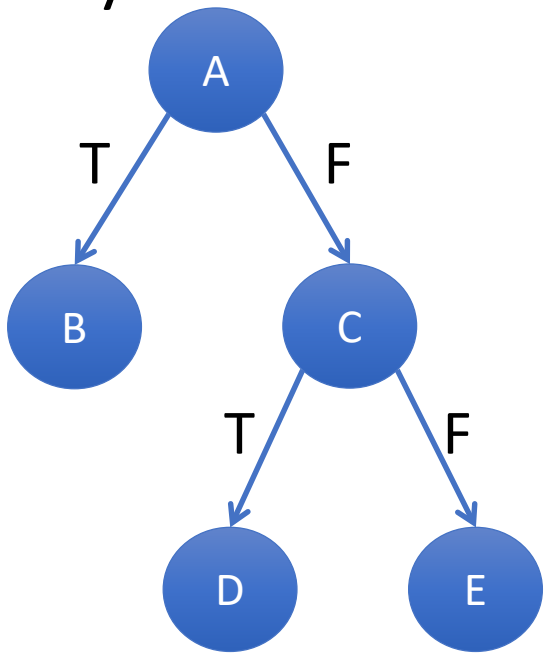
DFS Enumeration Strategy (Bottom Up)

Pop A onto history. A is a root so save path and trim history.

Stack: [B, D]

History: [~~E, C, A~~]

Paths: [E, C, A]



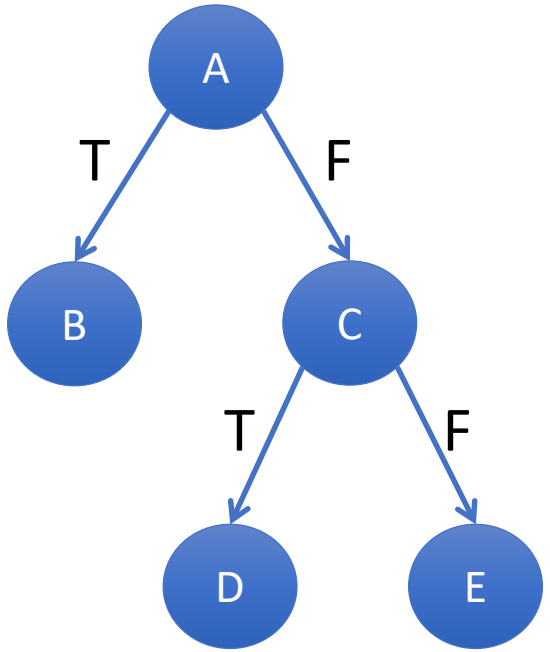
DFS Enumeration Strategy (Bottom Up)

Pop D onto history. D is not a root so push C.

Stack: [B, C]

History: [D]

Paths: [E, C, A]



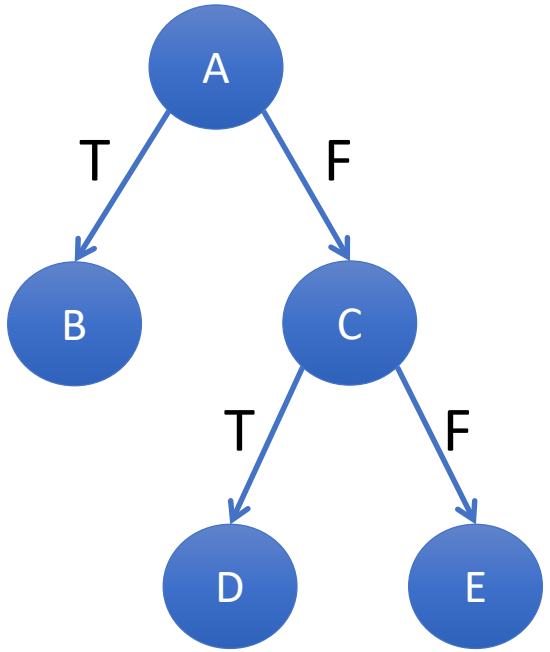
DFS Enumeration Strategy (Bottom Up)

Pop C onto history. C is not a root so push A.

Stack: [B, A]

History: [D, C]

Paths: [E, C, A]



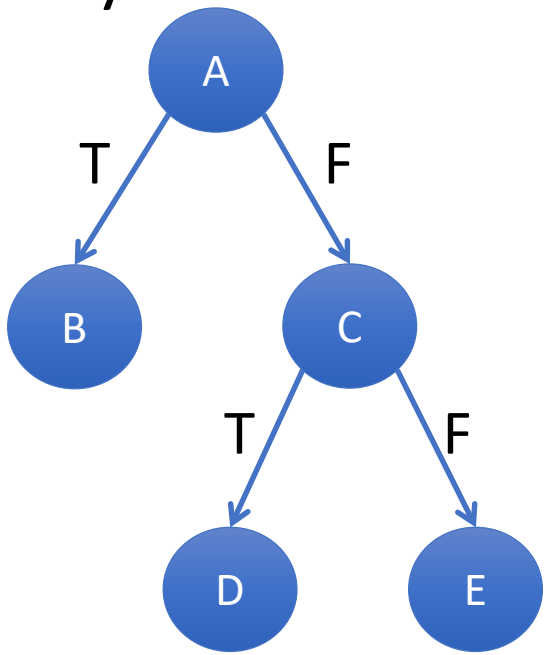
DFS Enumeration Strategy (Bottom Up)

Pop A onto history. A is a root so save path and trim history.

Stack: [B]

History: [~~D, C, A~~]

Paths: [E, C, A], [D, C, A]



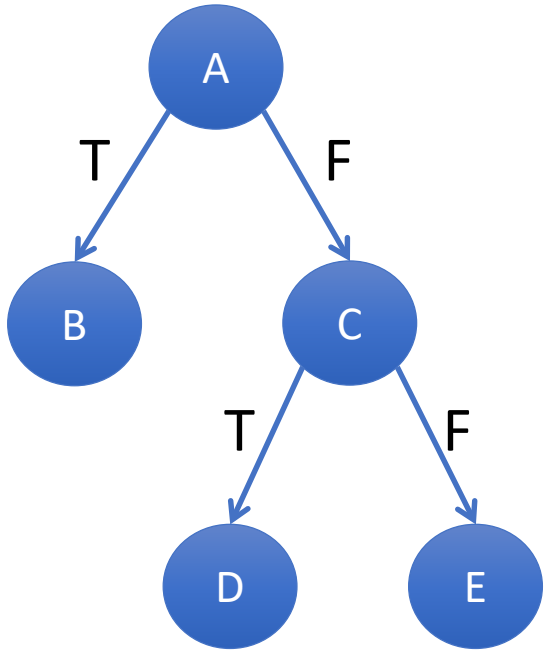
DFS Enumeration Strategy (Bottom Up)

Pop B onto history. B is not a root so push A.

Stack: [A]

History: [B]

Paths: [E, C, A], [D, C, A]



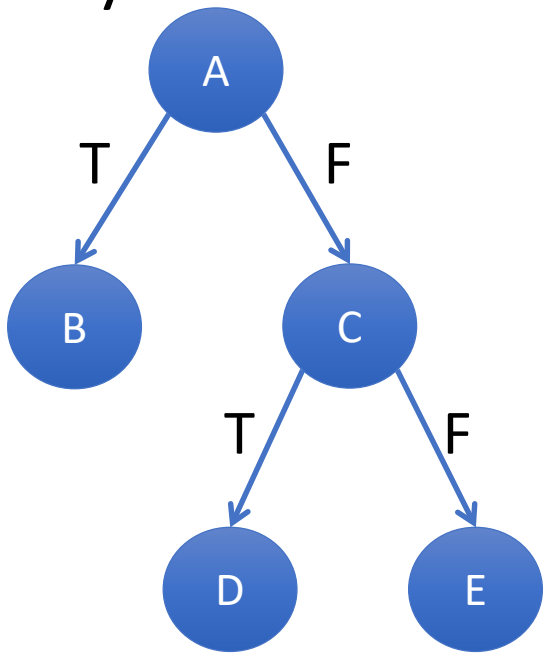
DFS Enumeration Strategy (Bottom Up)

Pop A onto history. A is a root so save path and trim history.

Stack: []

History: [~~B, A~~]

Paths: [E, C, A], [D, C, A], [B, A]



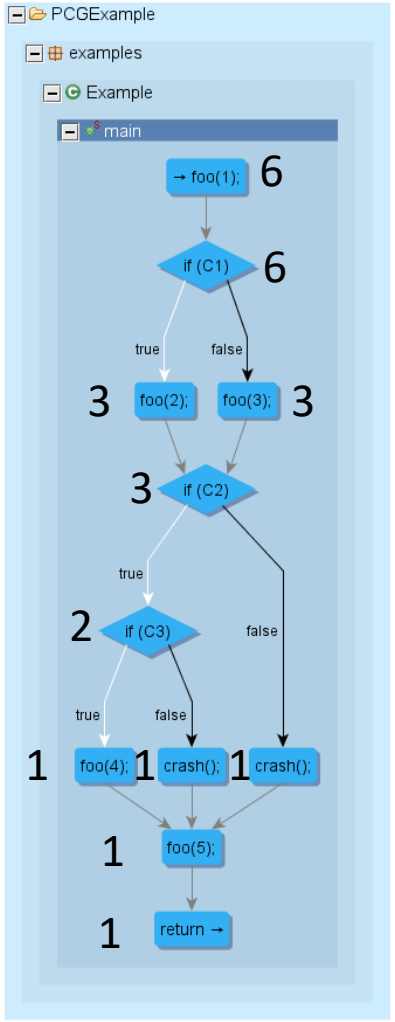
Efficient Path Counting

- Strategy 1: Create truth table and collapse after exploring all 2^n paths, always 2^n
- Strategy 2: DFS counting paths, worst case still 2^n
- Strategy 3: ?

Efficient Path Counting

- Strategy 1: Create truth table and collapse after exploring all 2^n paths, always 2^n
- Strategy 2: DFS counting paths, worst case still 2^n
- Strategy 3: Path Multiplicities O(n)

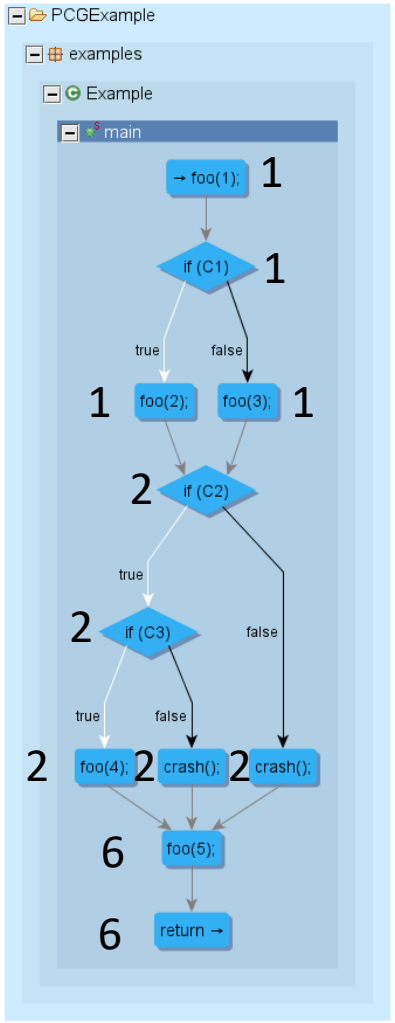
Leaf Multiplicities



1. Let each leaf node represent a value of 1 and all other nodes have default value 0
2. Propagate the node multiplicities upward by summing the value of the parent node and child node and assigning the result to the parent
   - Note that a node value cannot be propagated until all of its incoming child values have been propagated
   - Note if a function call is made then the number of paths is multiplied by the number of paths in the function (not added)
3. Repeat until no values are left to propagate and then take the value on the root for the final count of paths

*In this example we assume foo() has 1 path

Root Multiplicities



1. Let each root node represent a value of 1 and all other nodes have default value 0
2. Propagate the node multiplicities downward by summing the value of the child node and parent node and assigning the result to the child
   - Note that a node value cannot be propagated until all of its incoming parent values have been propagated
   - Note if a function call is made then the number of paths is multiplied by the number of paths in the function (not added)
3. Repeat until no values are left to propagate and then sum the values across all leaves for the final count of paths

*In this example we assume foo() has 1 path