Program Analysis and Verification

0368-4479

Noam Rinetzky

Lecture 2b: Symbolic Execution

Slides credit: Oded Navon, Ofek Amir, Cristian Cadar, Darko Marinov

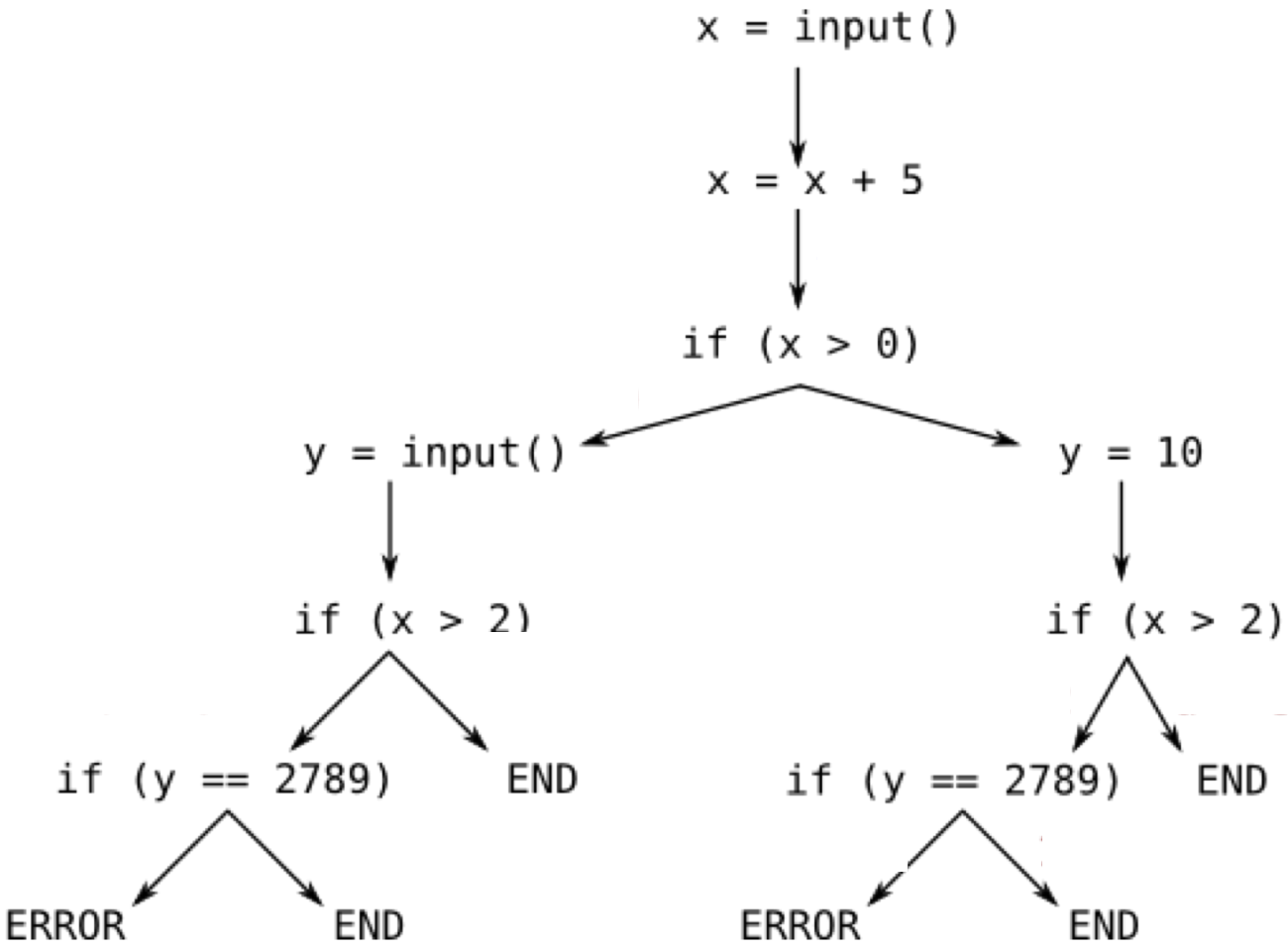
Dynamic Symbolic Execution

- Dynamic symbolic execution is a technique for ***automatically exploring paths*** through a program
  - Determines the feasibility of each explored path using a ***constraint solver***
  - Checks if there are ***any*** values that can cause an error on each explored path
  - For each path, can generate a ***concrete input triggering the path***

Example

```
x:=input();
x:=x+5;
if (x>0)
   y := input();
   if (x > 2)
      if (y == 2789) ERROR
else
   y := 10
   if (x>2)
      if (y == 2789) ERROR
```

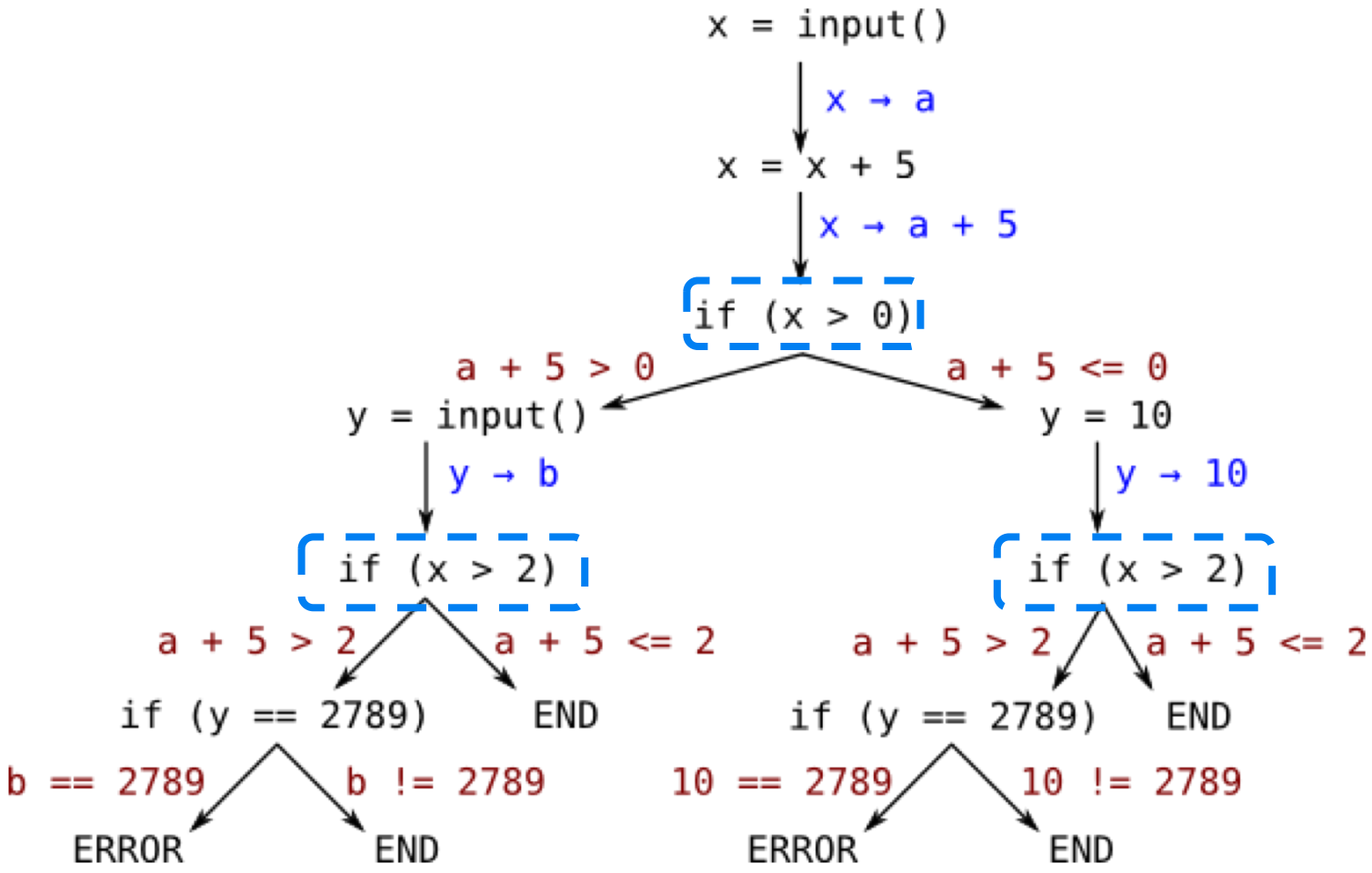
SOS Executions



Symbolic Values

- Symbolic execution would uses symbolic values instead of concrete values

SOS Execution Tree



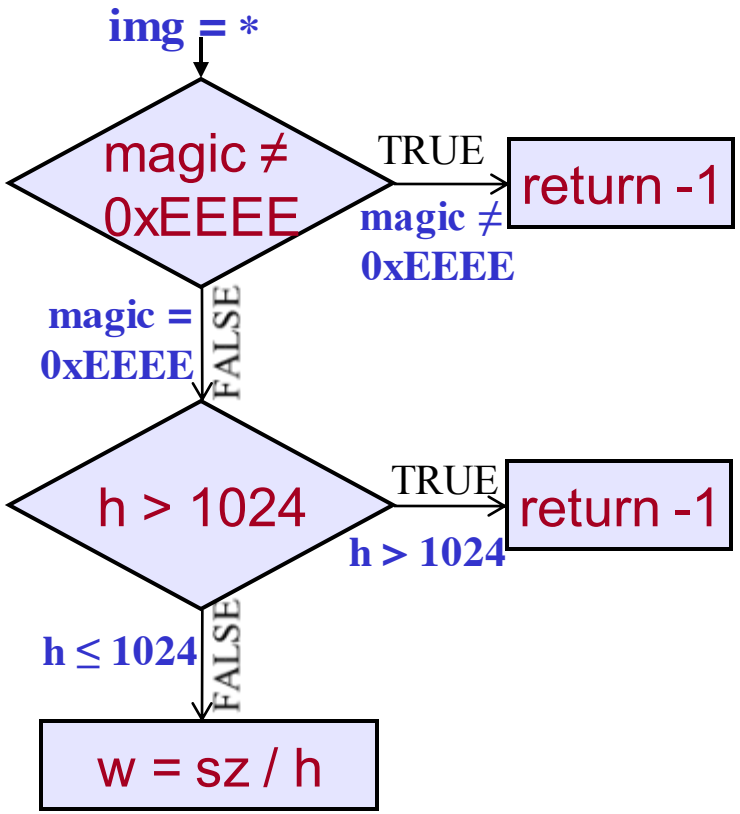
Symbolic Execution Tree

- We can generate an execution tree for all possible execution paths
- Each statement (code line) is a node
- Each IF statement generates two nodes, for True and False
- Each node is associated with the current variables values, PC and statement counter

Toy Example

```
struct image_t {
    unsigned short magic;
    unsigned short h, sz;
    ...
```

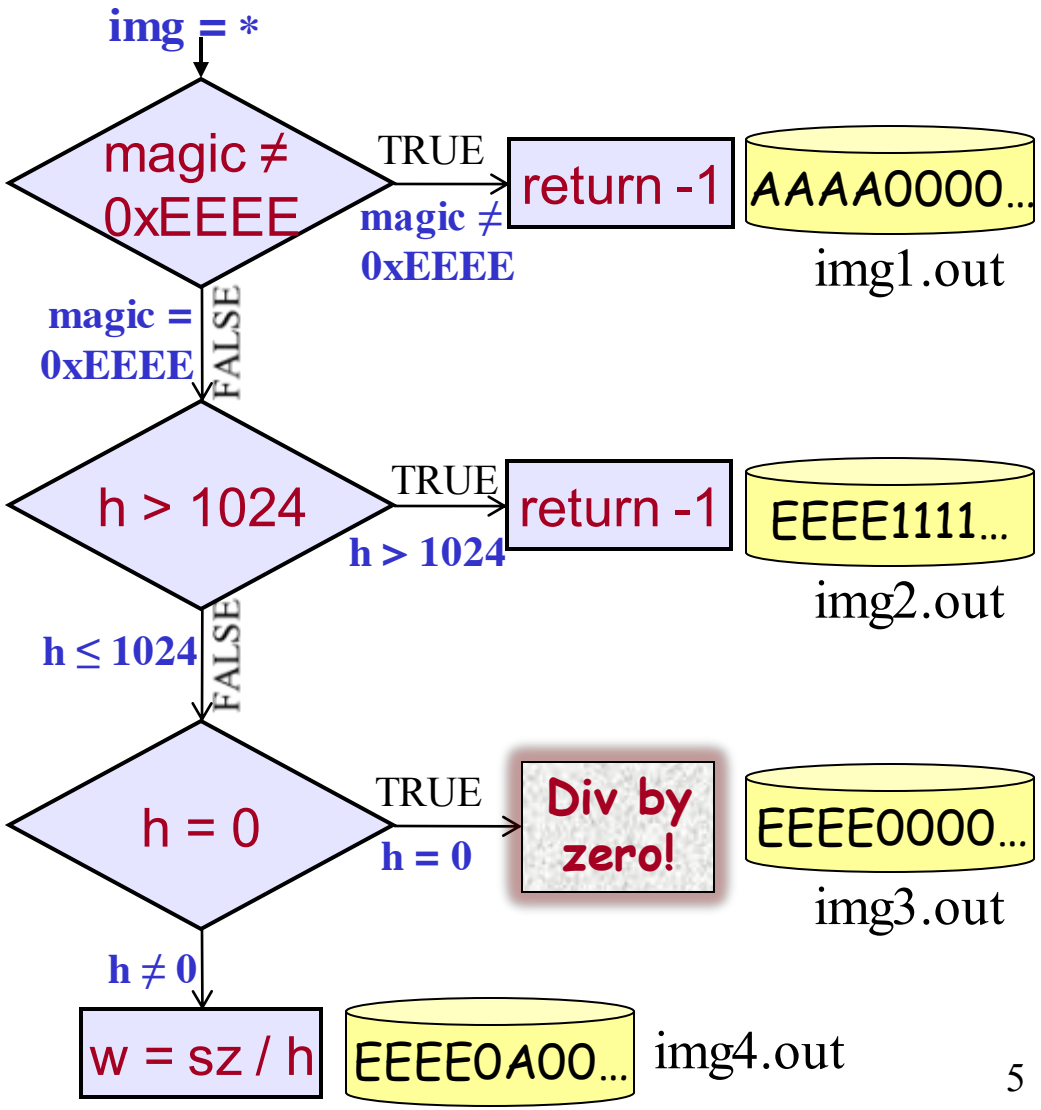
```
int main(int argc, char** argv) {
  ...
  image_t img = read_img(file);
  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  w = img.sz / img.h;
  ...
}
```



Toy Example

```
struct image_t {
     unsigned short magic;
     unsigned short h, sz;
     ...
```

```
int main(int argc, char** argv) {
  ...
  image_t img = read_img(file);
  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  w = img.sz / img.h;
  ...
}
```



All-Value Checks

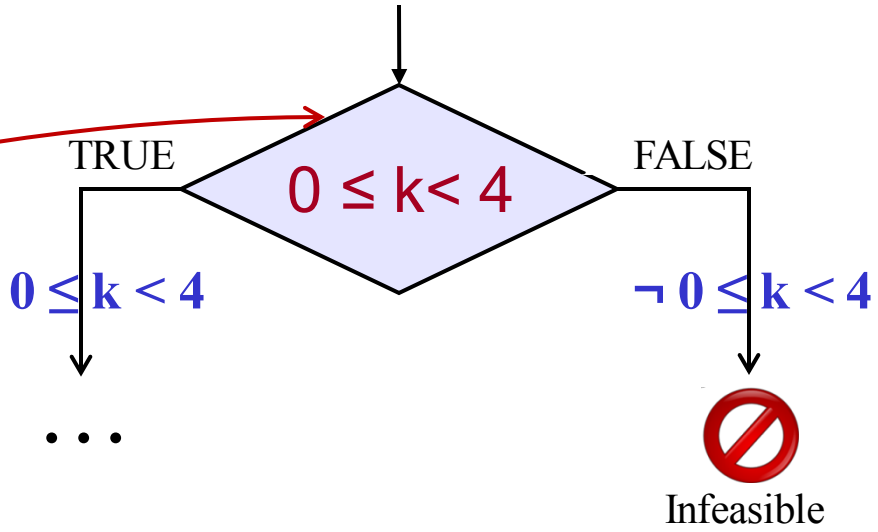
Implicit checks before each dangerous operation

- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

All-value checks!

- Errors are found if **any** buggy values exist on that path!

```
int foo(unsigned k) {
  int a[4] = {3, 1, 0, 4};
  k = k % 4;
  return a[a[k]];
}
```



All-Value Checks

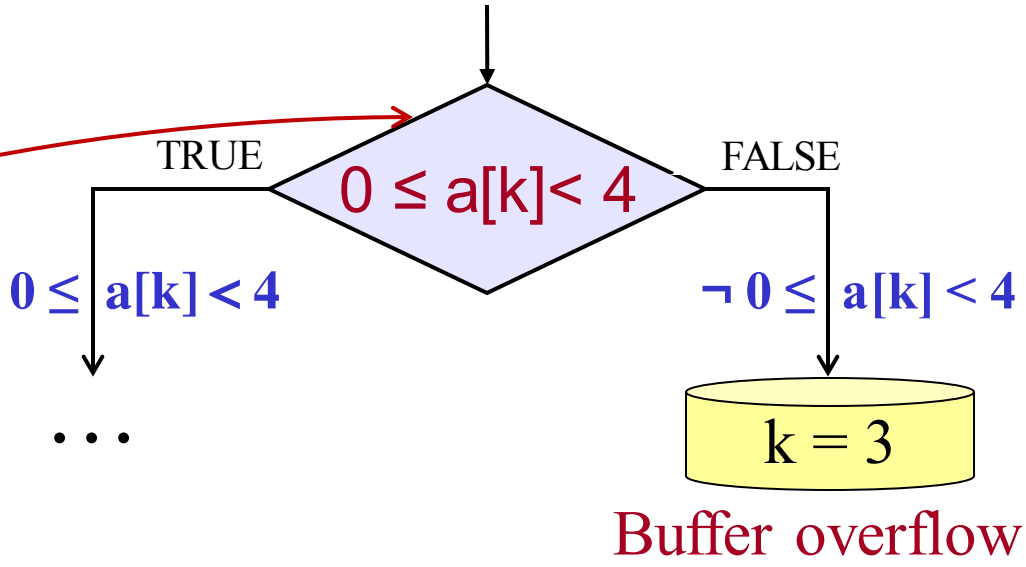
Implicit checks before each dangerous operation

- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

All-value checks!

- Errors are found if **any** buggy values exist on that path!

```
int foo(unsigned k) {
  int a[4] = {3, 1, 0, 4};
  k = k % 4;
  return a[a[k]];
}
```



Symbolic execution [King, 1976]

- A practical approach between the two extremes – Formal proving and manually testing
- Executing program with symbolic variables instead of values
  - $\boldsymbol{foo(\alpha_1, \alpha_2, \alpha_3) \rightarrow \{\alpha_1 \geq 0 \wedge (\alpha_1 + 2 \cdot \alpha_2) \geq 0 \wedge (\alpha_3 \geq 0)\}}$
- Checks for feasibility of each constraint:

$$\pi\colon (10 + \alpha) \leq 3 \ \wedge \ \alpha \geq 4 \ ?$$

- Each symbolic execution is equivalent to many “normal tests”

Another example program

Fig. 4. Procedure POWER.

```
1  POWER: PROCEDURE(X, Y);
2              Z ← 1;
3              J ← 1;
4  LAB:        IF Y ≥ J THEN
5              DO; Z ← Z * X;
6                  J ← J + 1;
7                  GO TO LAB; END;
8              RETURN (Z);
9  END;
```

Fig. 5. Symbolic execution of POWER (α_1, α_2).

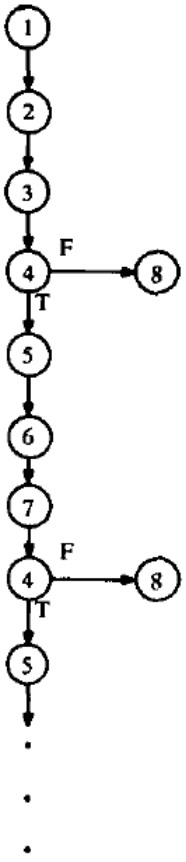
After statement	J	X	Y	Z	*pc*
	Values would be				
1	?	α_1	α_2	?	*true*
2	—	—	—	1	—
3	1	—	—	—	—
4 execution in detail: (a) evaluate $Y \geq J$ getting $\alpha_2 \geq 1$. (b) use path condition and check: (i) *true* $\supset \alpha_2 \geq 1$ (ii) *true* $\supset \neg(\alpha_2 \geq 1$ (c) neither are theorems, so fork.					
Case $\neg(\alpha_2 \geq 1)$:					
4	1	α_1	α_2	1	$\neg(\alpha_2 \geq 1)$
8 this case completed. (returns 1 when $\alpha_2 < 1$.)					
Case $\alpha_2 \geq 1$:					
4	1	α_1	α_2	1	$\alpha_2 \geq 1$
5	—	—	—	α_1	—
6	2	—	—	—	—
7	—	—	—	—	—
4 execution in detail: (a) evaluate $Y \geq J$, getting $\alpha_2 \geq 2$ (b) use *pc*: (i) $\alpha_2 \geq 1 \supset \alpha_2 \geq 2$ (ii) $\alpha_2 \geq 1 \supset \neg(\alpha_2 \geq 2)$ (c) neither *true*, so fork.					
Case $\neg(\alpha_2 \geq 2)$:					
4	2	α_1	α_2	α_1	$\alpha_2 \geq 1 \wedge \neg(\alpha_2 \geq 2)$ (or simply $\alpha_2 = 1$)
8 this case completed. (returns α_1 when $\alpha_2 = 1$.)					
Case $\alpha_2 \geq 2$:					
4	2	α_1	α_2	α_1	$\alpha_2 \geq 1 \wedge \alpha_2 \geq 2$ (or simply $\alpha_2 \geq 2$)
⋮	⋮				
In this example the symbolic execution will continue indefinitely.					

Symbolic Execution Tree

Fig. 4. Procedure POWER.

```
1  POWER: PROCEDURE(X, Y);
2              Z ← 1;
3              J ← 1;
4  LAB:        IF Y ≥ J THEN
5              DO; Z ← Z * X;
6                  J ← J + 1;
7                  GO TO LAB; END;
8              RETURN (Z);
9  END;
```

Fig. 6. Execution tree for POWER(α_1 , α_2).

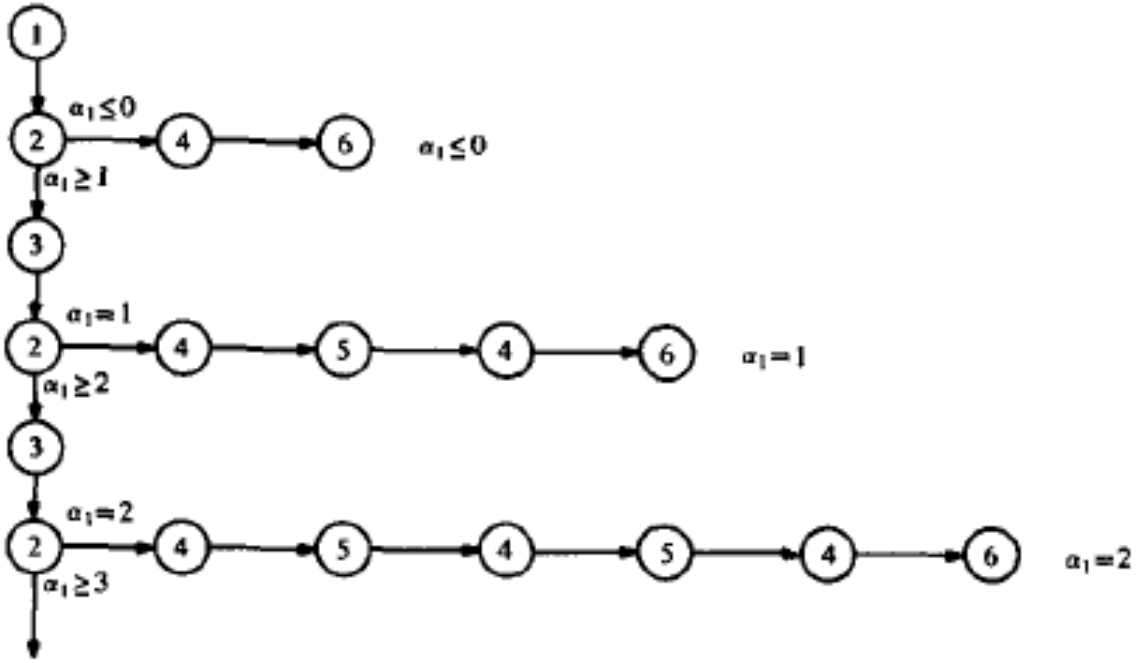


Case *pc* is ($\alpha_2 < 1$):
RETURNS 1

Case *pc* is ($\alpha_2 = 1$):
RETURNS α_1

{RETURNS α_1^n when *pc* is ($\alpha_2 = n$)}

Symbolic Execution Tree




```
1     TWOLOOPS: PROCEDURE (N);
2                 DO J=1 TO N;
3                   (body of statements) END;
4                 DO K=1 TO N;
5                   (body of statements) END;
6                 END;
```

Commutativity

- We can interchange the symbolic values with their corresponding specific integers
- The execution with each value is the same
- Switching a symbolic value with a real one is called **instantiation**