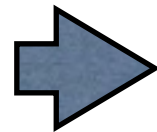


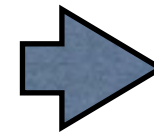
Concolic Testing

Generación Automática de Casos de Test - 2018

Random Testing



**Programa
bajo Test**

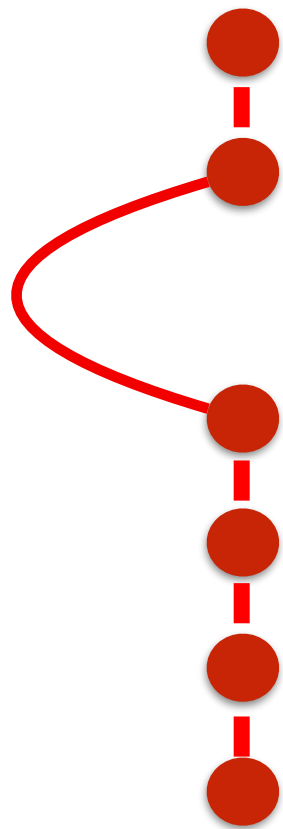


Oráculo



```
1. def testme(x, y):  
2.     if (y<0):  
3.         return -x  
4.     else:  
5.         z = x - y  
6.         if (y+z=10000):  
7.             raise Exception("error")  
8.         else:  
9.             return z
```

```
1. def testme(x, y):
2.     if (y<0):
3.         return -x
4.     else:
5.         z = x - y
6.         if (y+z=10000):
7.             raise Exception("error")
8.         else:
9.             return z
```



```
1. def testme(x, y):  
2.     if (y<0):  
3.         return -x  
4.     else:  
5.         z = x - y  
6.         if (y+z=10000):  
7.             raise Exception("error")  
8.         else:  
9.             return z
```

$y \geq 0$
 $x = 10000$



$$2^{31} / 2^{32} = 50\%$$
$$1 / 2^{32} \sim 0\%$$

Ejecución Simbólica

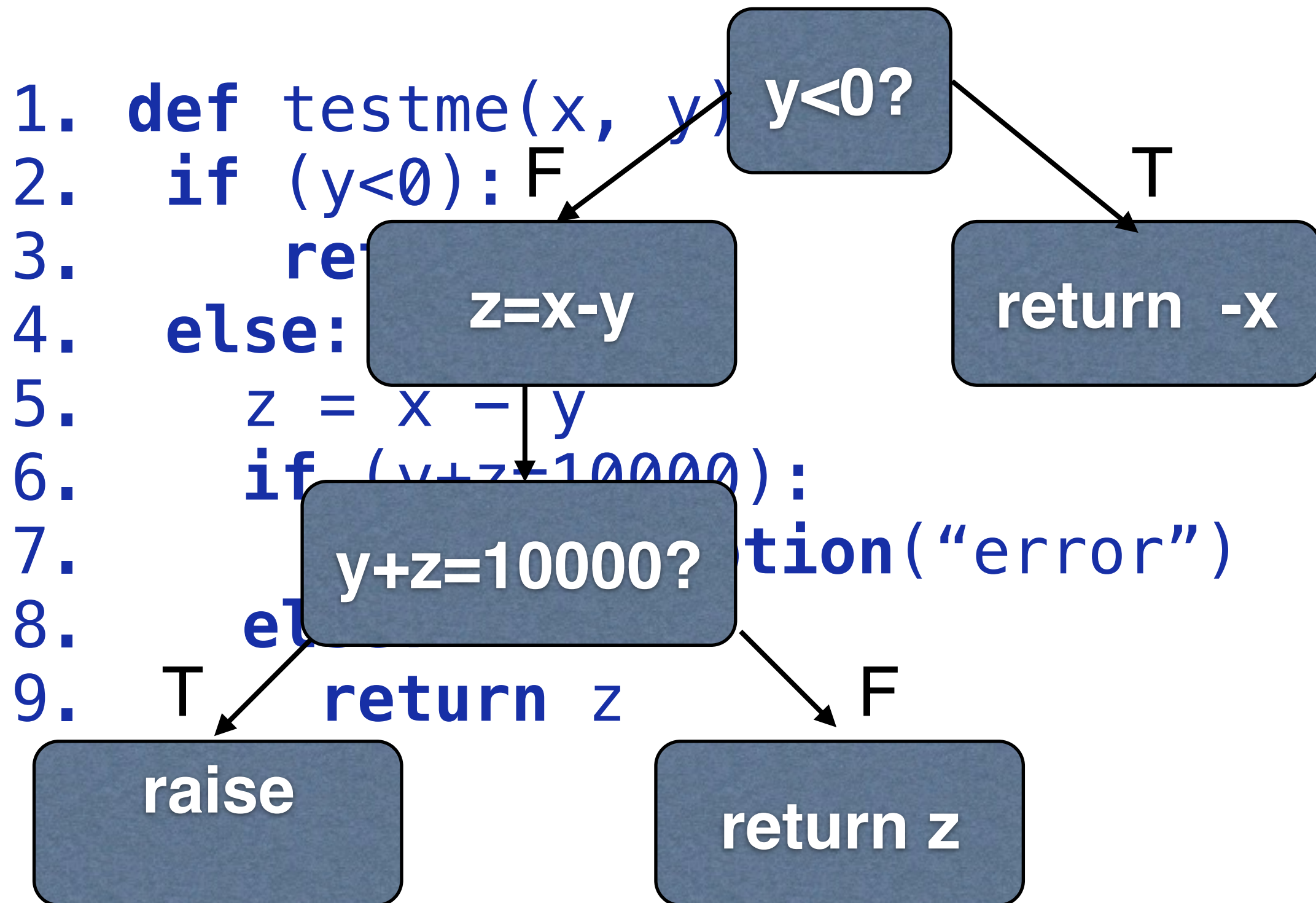
Al Rescate!

Necesitamos...

- Recolectar restricciones del código del programa
- Resolver restricciones para crear nuevos inputs



Ejecución Simbólica

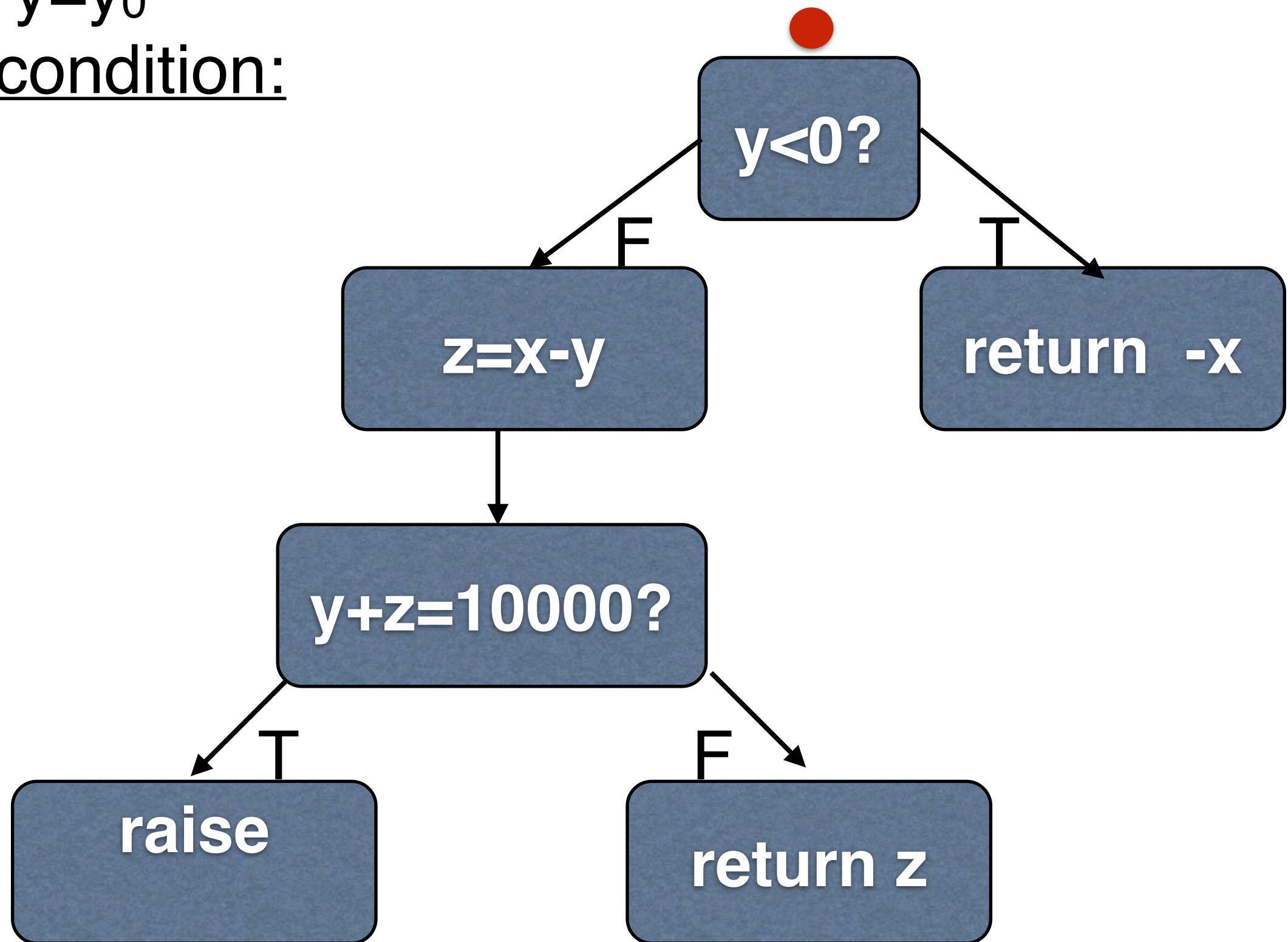


symbolic state:

$x=x_0, y=y_0$

path condition:

true

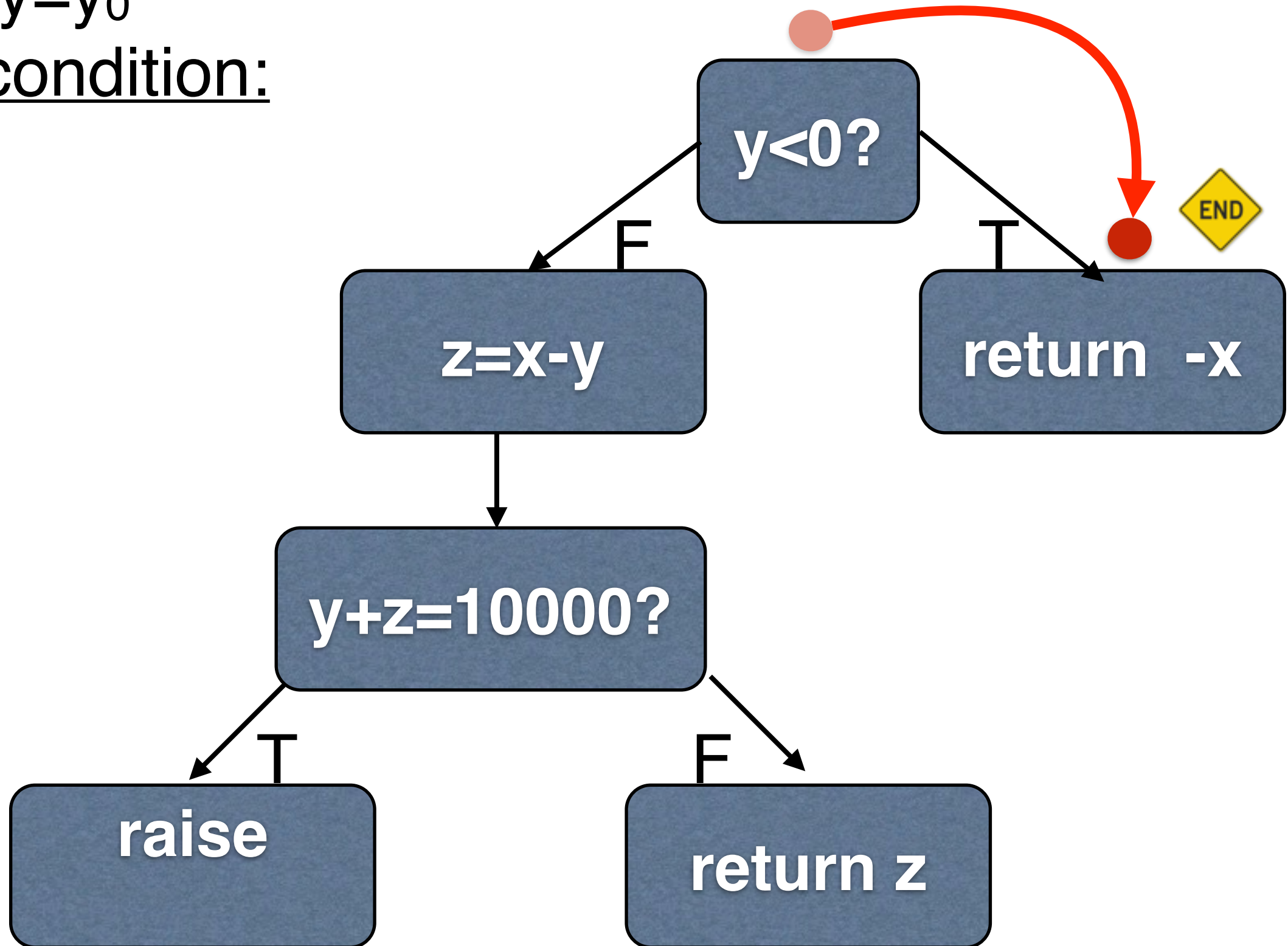


symbolic state:

$x=x_0, y=y_0$

path condition:

$y_0 < 0$

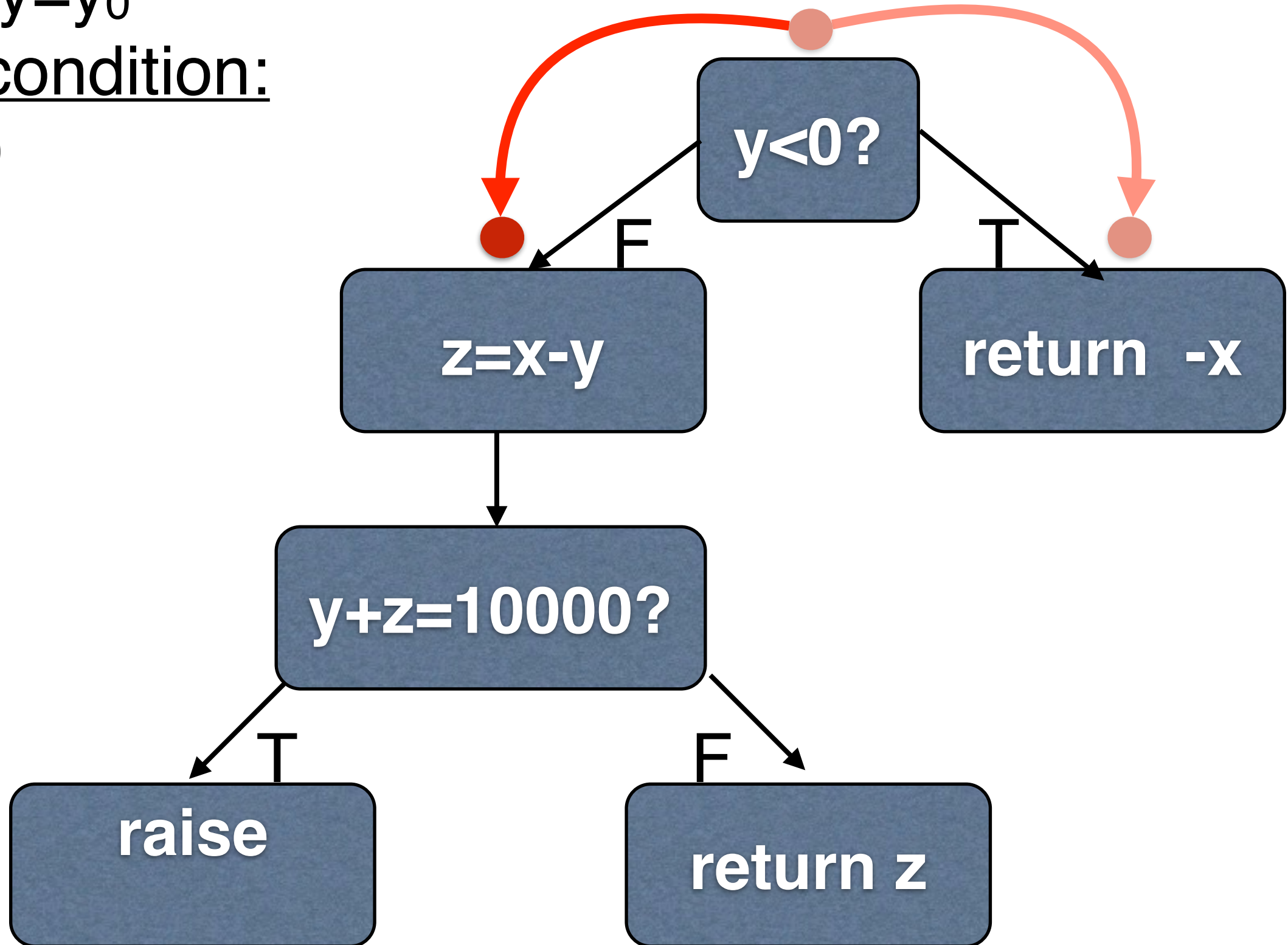


symbolic state:

$x=x_0, y=y_0$

path condition:

$y_0 \geq 0$

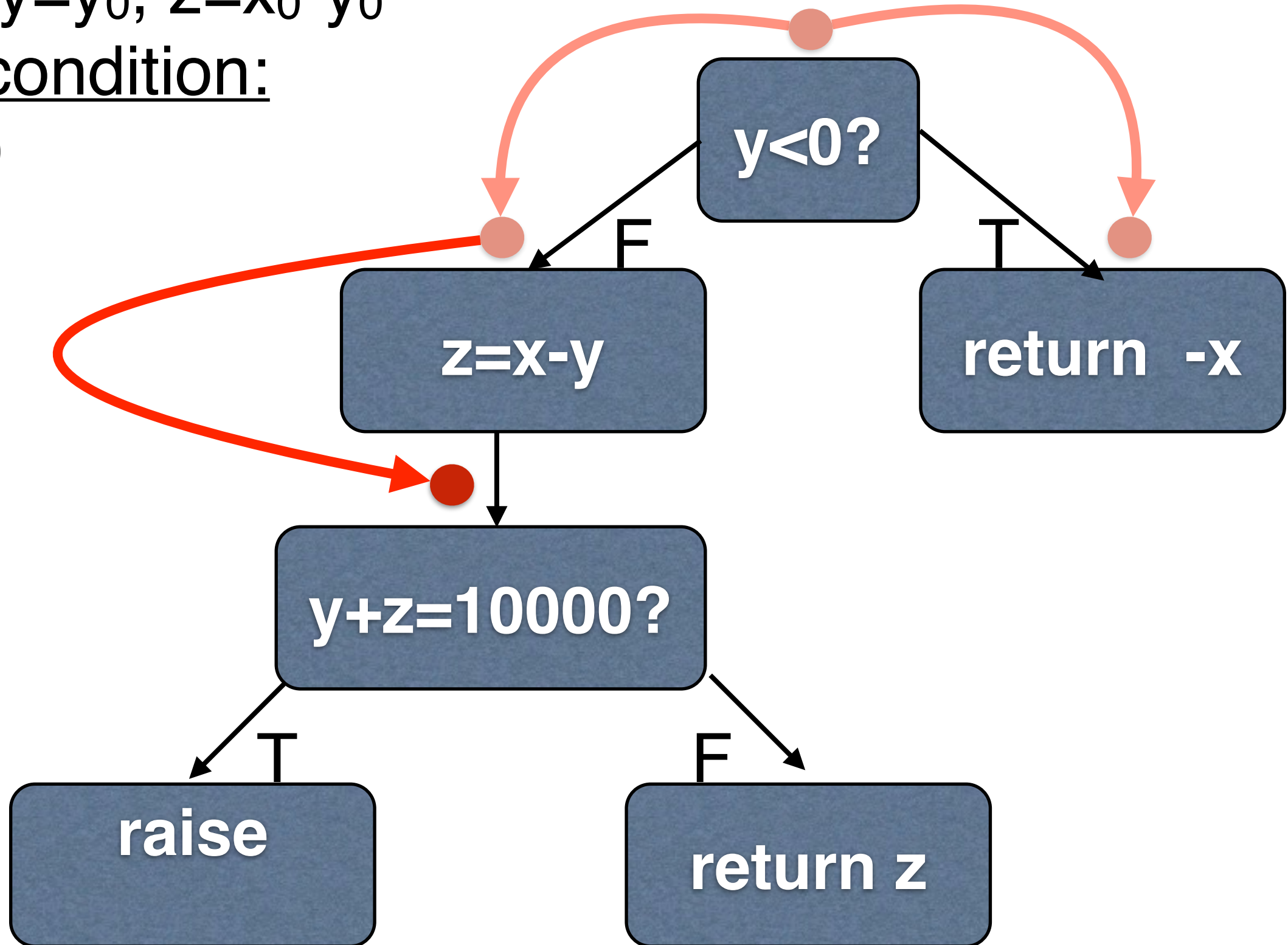


symbolic state:

$x=x_0, y=y_0, z=x_0-y_0$

path condition:

$y_0 \geq 0$



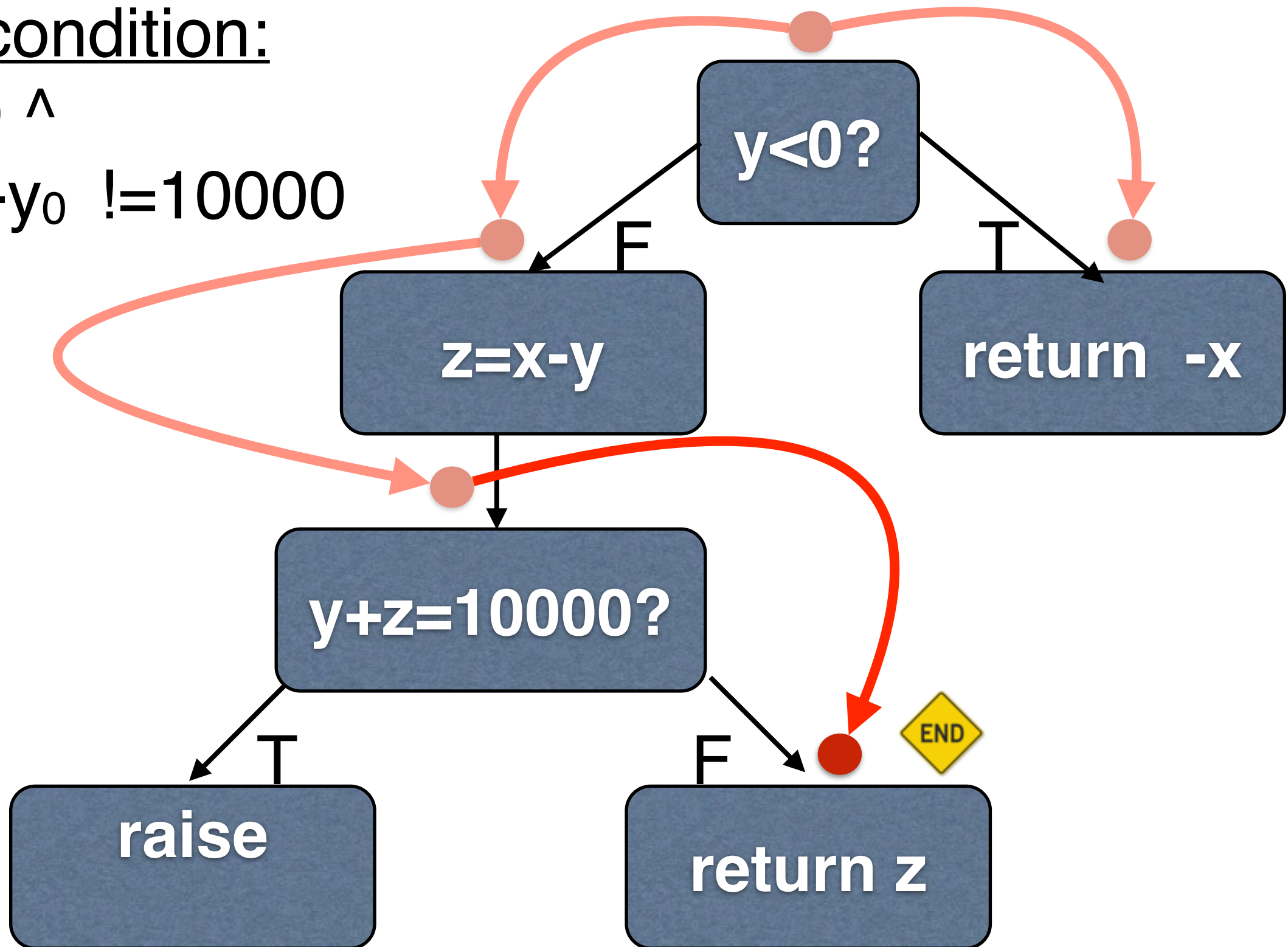
symbolic state:

$x=x_0, y=y_0, z=x_0-y_0$

path condition:

$y_0 \geq 0 \wedge$

$y_0 + x_0 - y_0 \neq 10000$



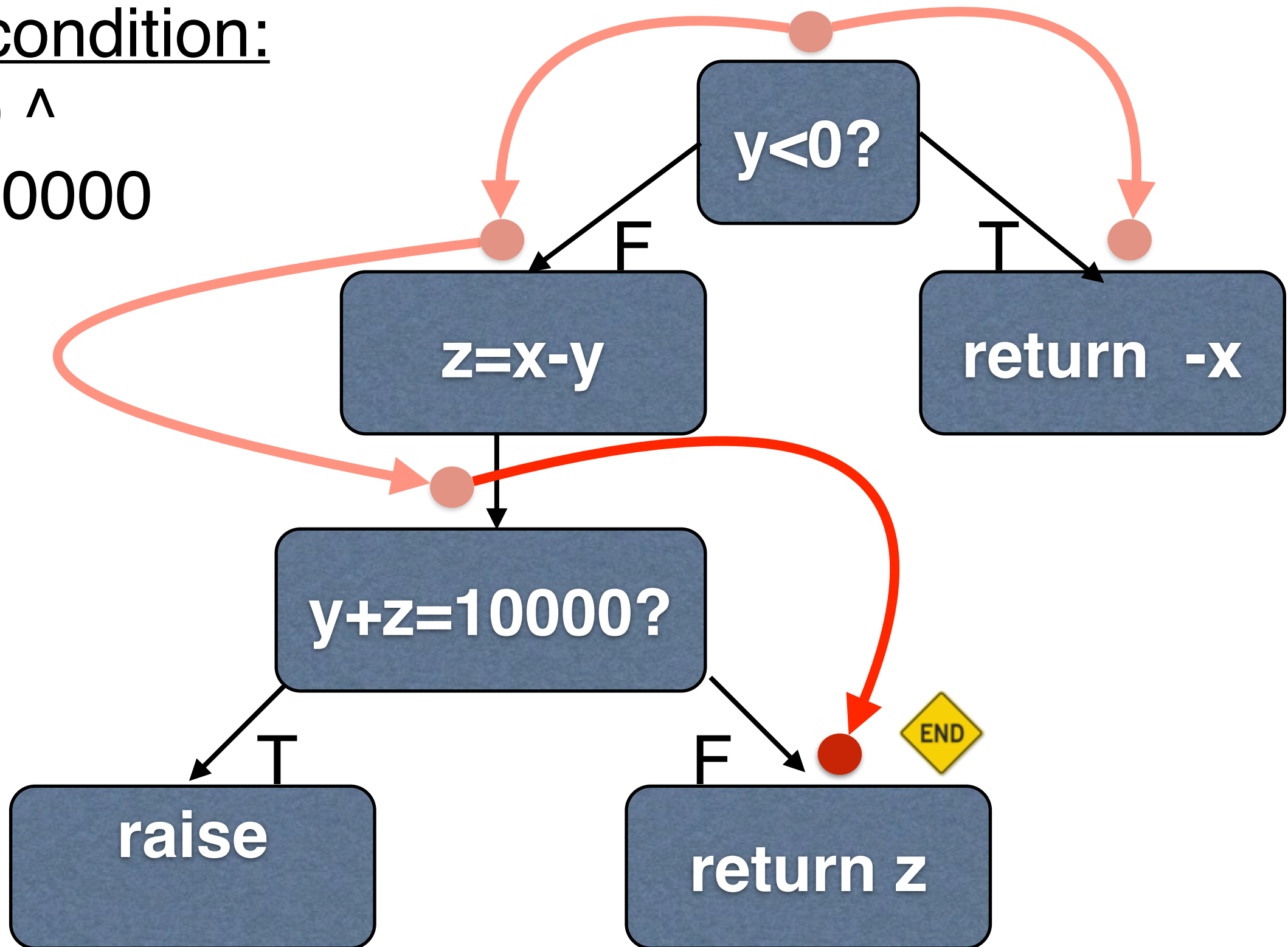
symbolic state:

$x=x_0, y=y_0, z=x_0-y_0$

path condition:

$y_0 \geq 0 \wedge$

$x_0 \neq 10000$



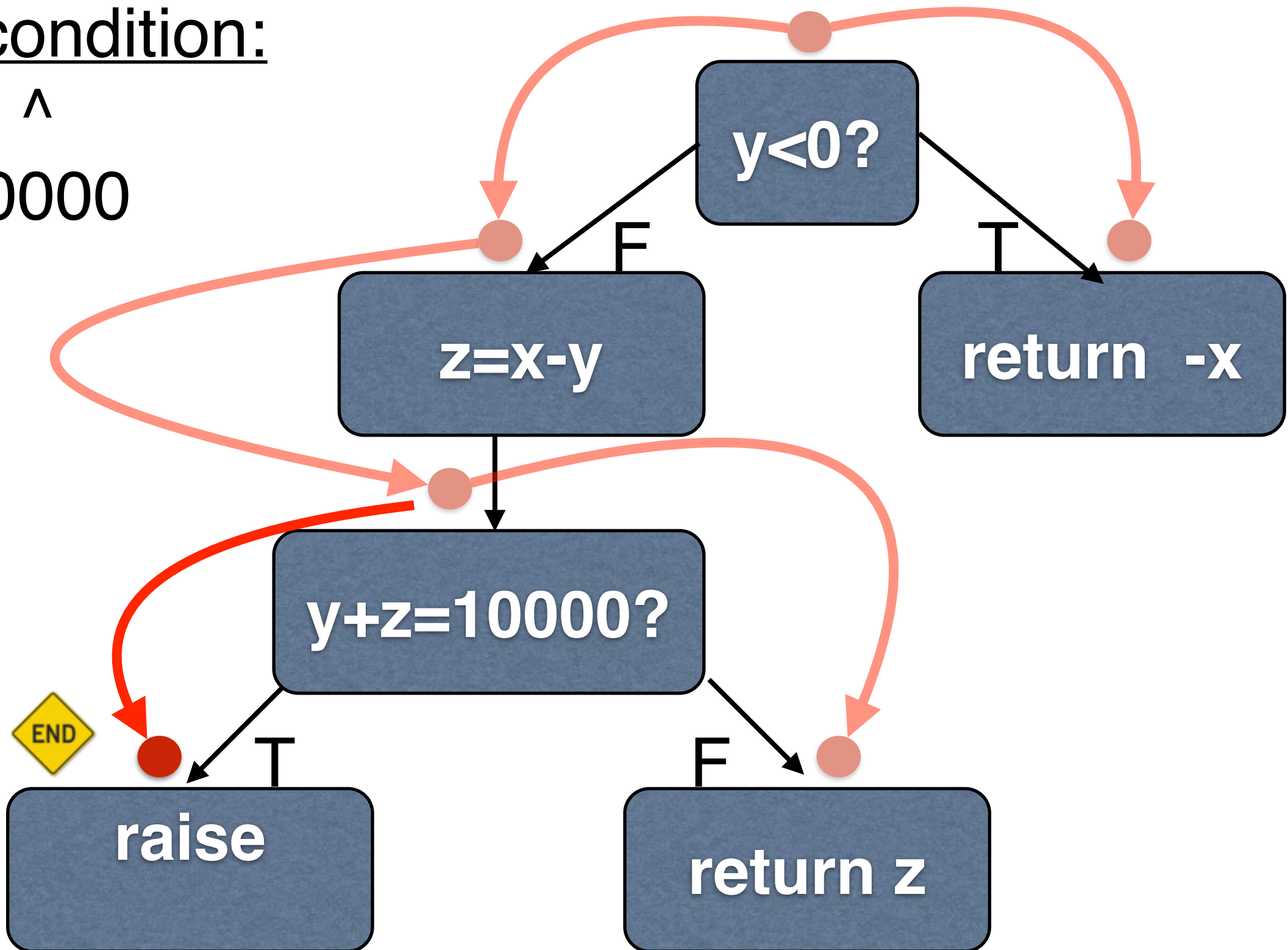
symbolic state:

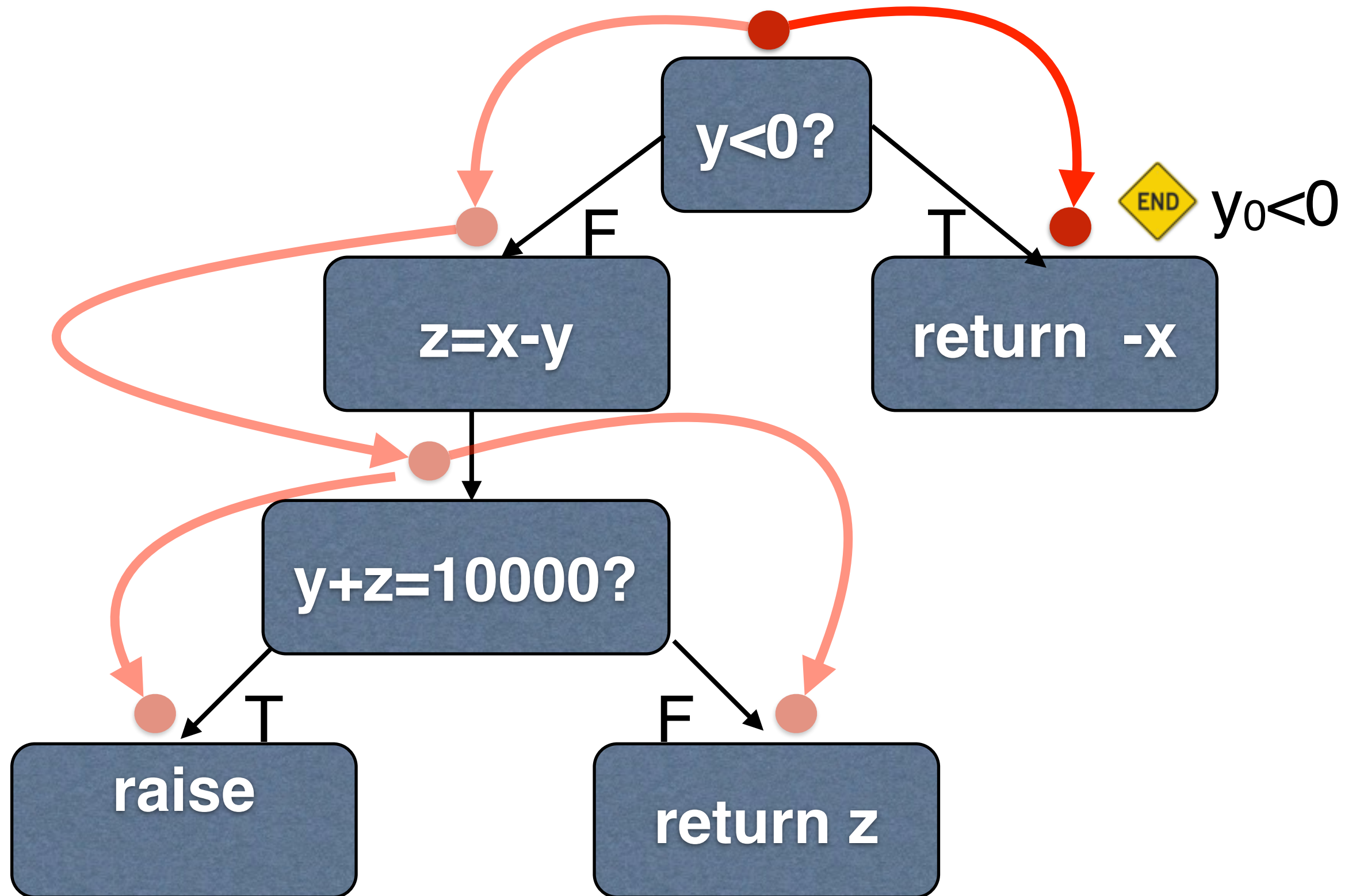
$x=x_0, y=y_0, z=x_0-y_0$

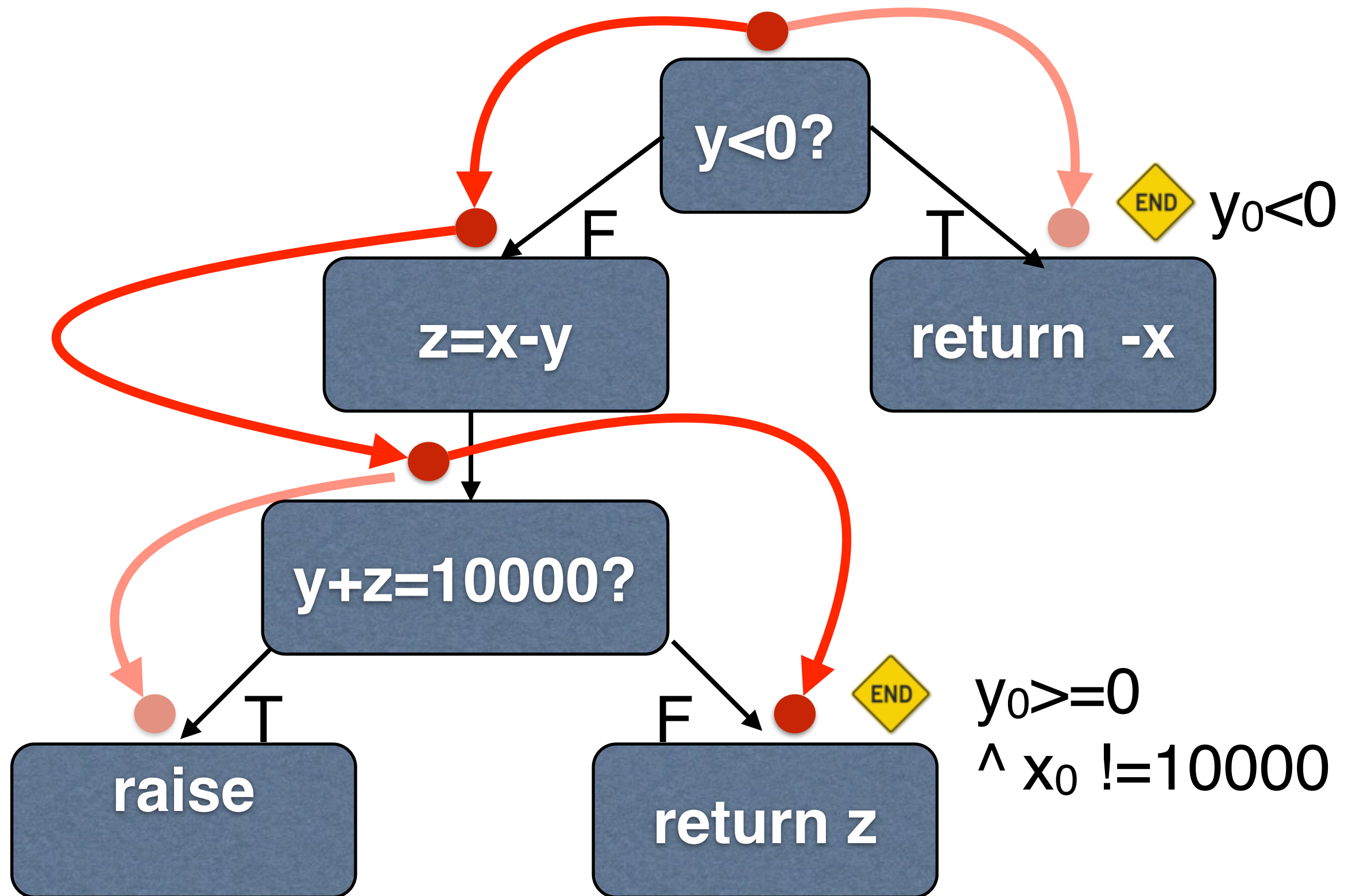
path condition:

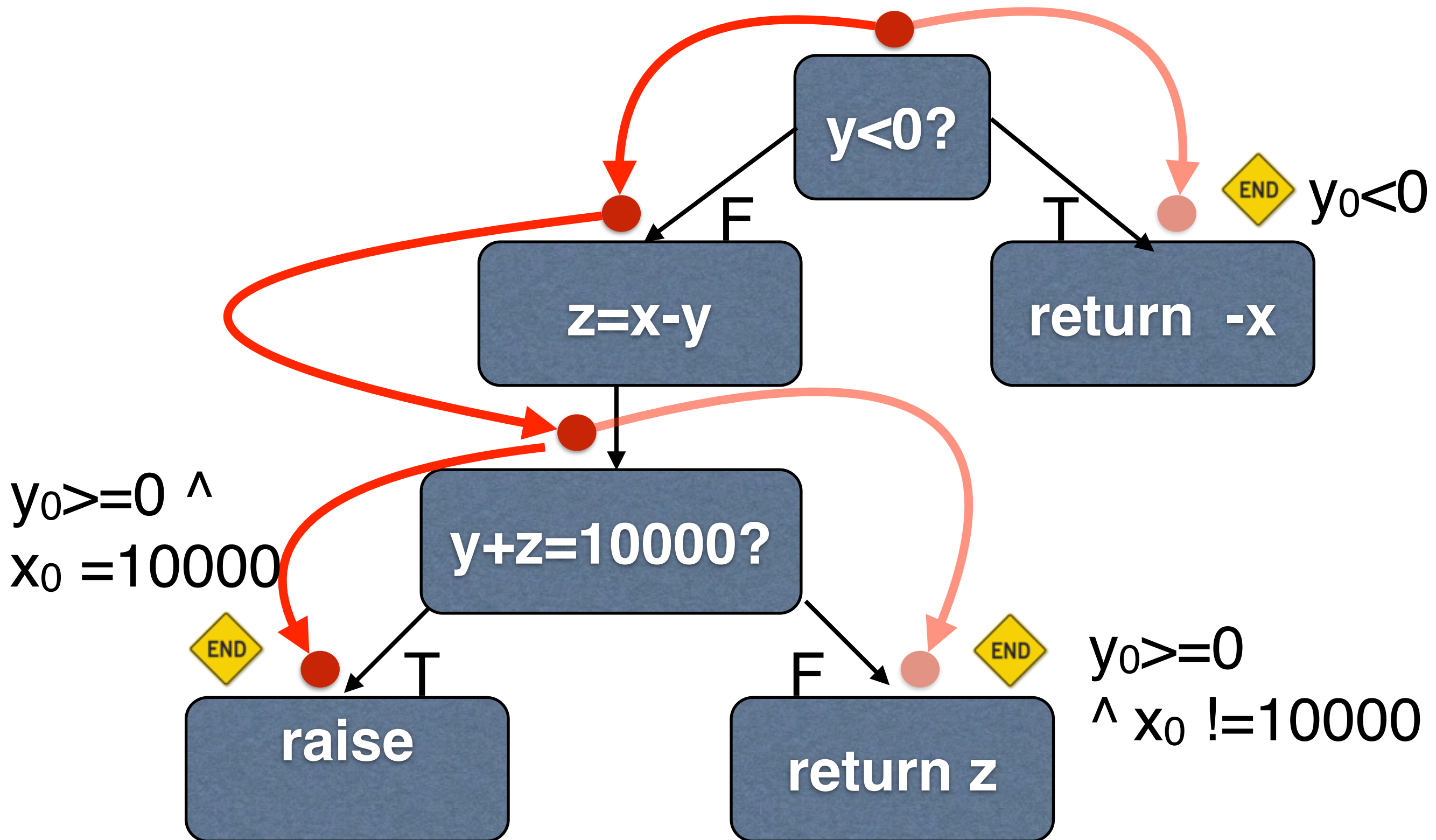
$y_0 \geq 0 \wedge$

$x_0 = 10000$







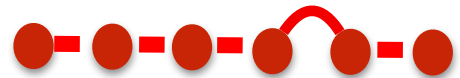


Constraint Solving

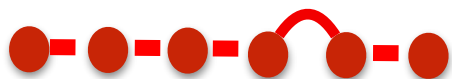


- Es un programa que resuelve fórmulas lógicas descritas en un lenguaje
- SAT
- UNSAT
- UNKNOWN/TIMEOUT
- Si es SAT, da un valor para cada variable

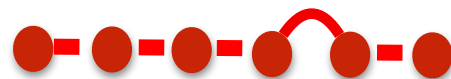
Constraint Solving



$$y_0 < 0$$



$$y_0 \geq 0 \wedge x_0 \neq 10000$$



$$y_0 \geq 0 \wedge x_0 = 10000$$



$$y_0 = 0 \wedge x_0 = 10000$$

testme(0,-1)

testme(0,0)

testme(10000,0)

```
1. def testme(x, y):  
2.     if (y<0):  
3.         return -x  
4.     else:  
5.         z = x - y  
6.         if (y+z=10000):  
7.             raise Exception("error")  
8.         else:  
9.             return z
```

Ejecución Simbólica

```
// Input: entry function  
// Output: set of test cases  
def generate_se(entry_f):  
  
    tests = {}  
  
    // Iterator has a unroll limit to finitize it  
    path_it = execution_paths(cfg(entry_f), limit).iterator()  
while hasTime() and path_it.hasNext():  
    cfg_path = path_it.next()  
    // perform the symbolic execution of that CFG path  
    path_condition = exec_symbolic(entry_f, cfg_path)
```

```
// Input: entry function
// Output: set of test cases
def generate_se(entry_f):

    tests = {}

    // Iterator has a unroll limit to finitize it
    path_it = execution_paths(cfg(entry_f), limit).iterator()
    while hasTime() and path_it.hasNext():
        cfg_path = path_it.next()
        // perform the symbolic execution of that CFG path
        path_condition = exec_symbolic(entry_f, cfg_path)

        // invoke the constraint solver
        solution = solve(path_condition, solver_timeout)

        // we care only if the constraint solver returns SAT
        if solution.is_SAT():
            new_test = create_test(entry_f, solution)
            tests.add(new_test)

    return tests
```

Parameterized Unit Tests

- Es el **entry point** de la generación de tests con ejecución simbólica
- Todos los argumentos tienen que ser soportados por la constraint solver (ej: enteros, reales, strings)
- La generación de casos de tests **no crea secuencias de invocaciones a métodos** como lo hace Random Testing Orientado a Objetos

Parameterized Unit Tests

```
public static void p_test1(int i0, int i1, int i2) {  
    RBTREE root = new RBTREE();  
    root.insert(i0);  
    root.insert(i1);  
    root.insert(i2);  
    assertTrue(root.contains(i0));  
    assertTrue(root.contains(i1));  
    assertTrue(root.contains(i2));  
}
```

Constraint Language

Expresiones Enteras

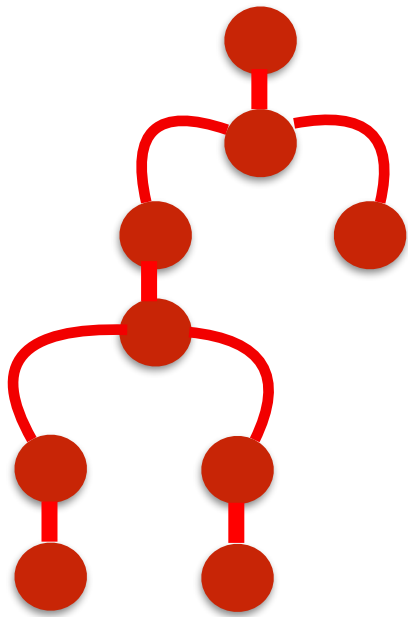
- Expresiones Enteras:
 - $0, 1, 2, \dots$
 - $\text{var0}, \text{var1}, \text{var}$
 - $E + E, E - E, E * E, E / E, E \% E$
- Constraints:
 - $E == E, E != E, E > E, E >= E, E < E, E <= E$

Constraint Language

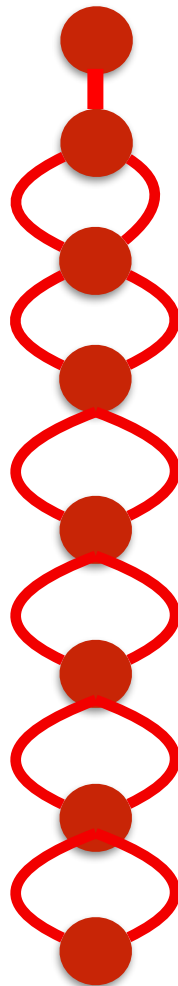
Cadenas de Strings

- Expresiones sobre Strings:
 - `""`, `"A"`, `"B"`, ...
 - `var0`, `var1`, `var`
 - `E+E`, `E.substring(int,int)`, etc.
- Constraints:
 - `E.equals(E)`, `!E.equals(E)`, `E.contains(E)`, etc

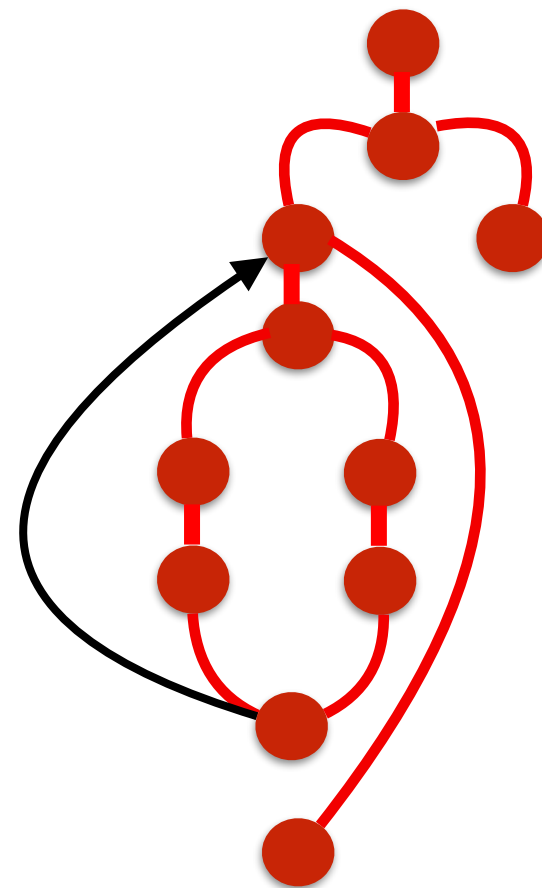
Ejecución Simbólica: Limitaciones



3 caminos



2^n
caminos



¡Infinitos
caminos!

Ejecución Simbólica:

Limitaciones

```
1. def nonlinear(x, y, z):  
2.     if (x%y*z=42):  
3.         raise Exception("error")  
4.     else:  
5.         return hash_str
```

Requiere aritmética
No-lineal

```
1. def hashing(x, y):  
2.     m = hashlib.md5()  
3.     m.update(x)  
4.     hash_str = m.hexdigest()  
5.     if (y==hash_str):  
6.         raise Exception("error")  
7.     else:  
8.         return hash_str
```

Limitaciones intrínsecas
de constraint solving

Ejecución Simbólica:

Limitaciones

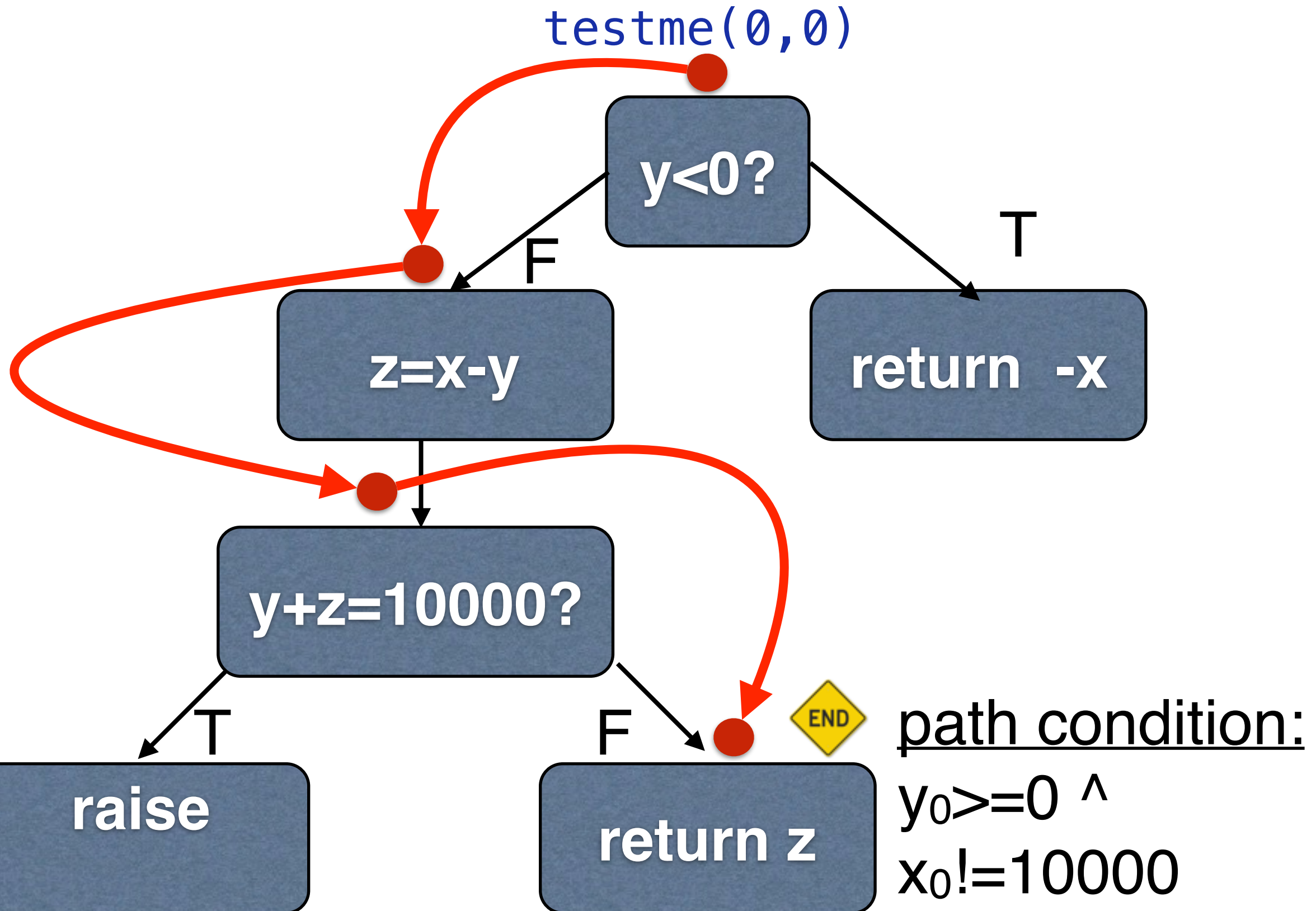
- Explosión combinatoria (o incluso infinita) de caminos en el CFG para evaluar
- Limitaciones del Constraint Solving
 - String handling, aritmética no-lineal, aritmética de punto flotante, etc.
 - Problemas NP-completos
- Información en Runtime (files, sockets, native code, etc.)

Ejecución Simbólica: Limitaciones

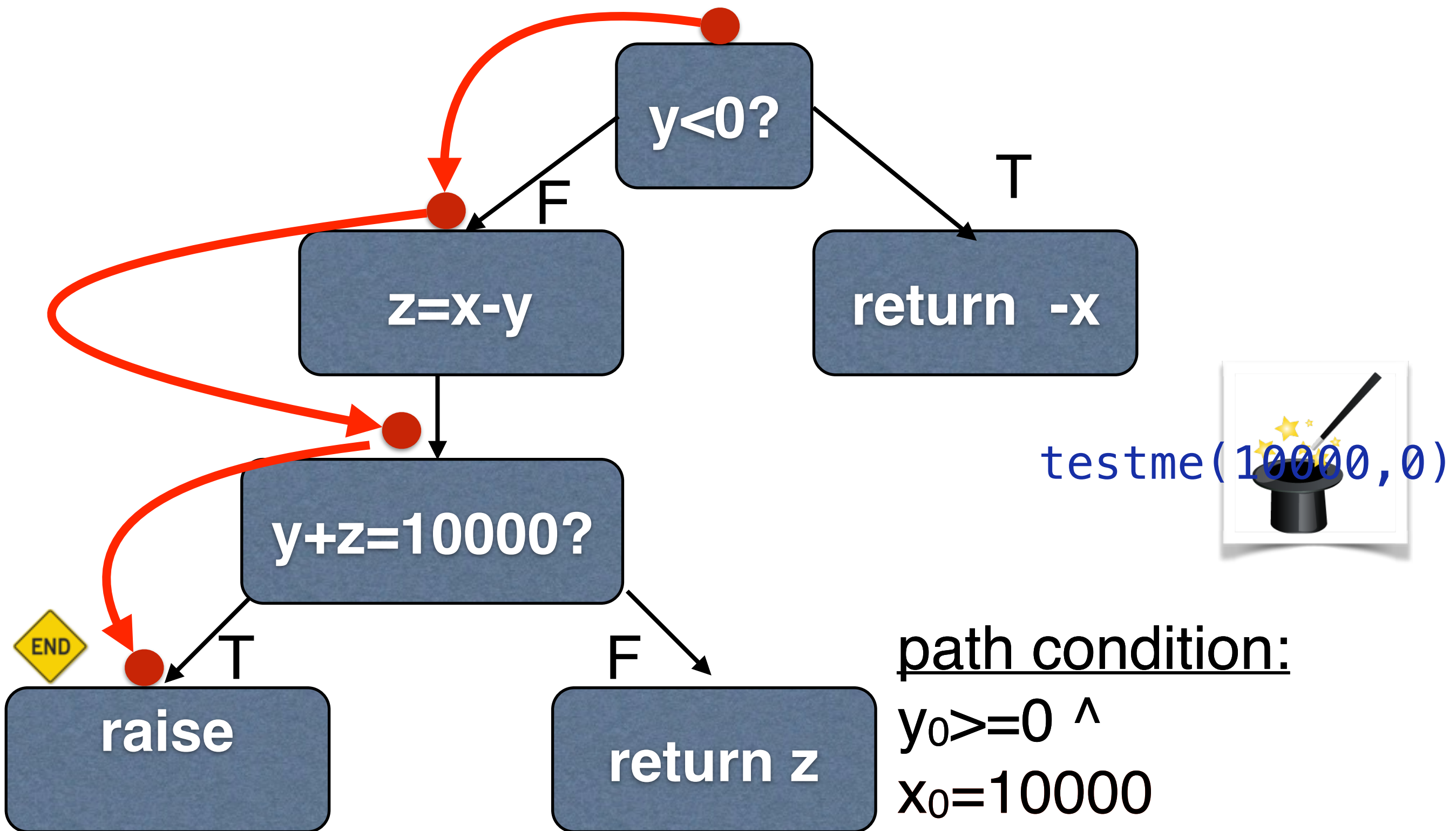
- Hay una cantidad combinatoria de caminos en el CFG
- Muchos de esos caminos pueden ser irrealizables
- ¿Hay algún modo de guiar la selección de caminos para realizar la ejecución simbólica?

Ejecución Simbólica Dinámica

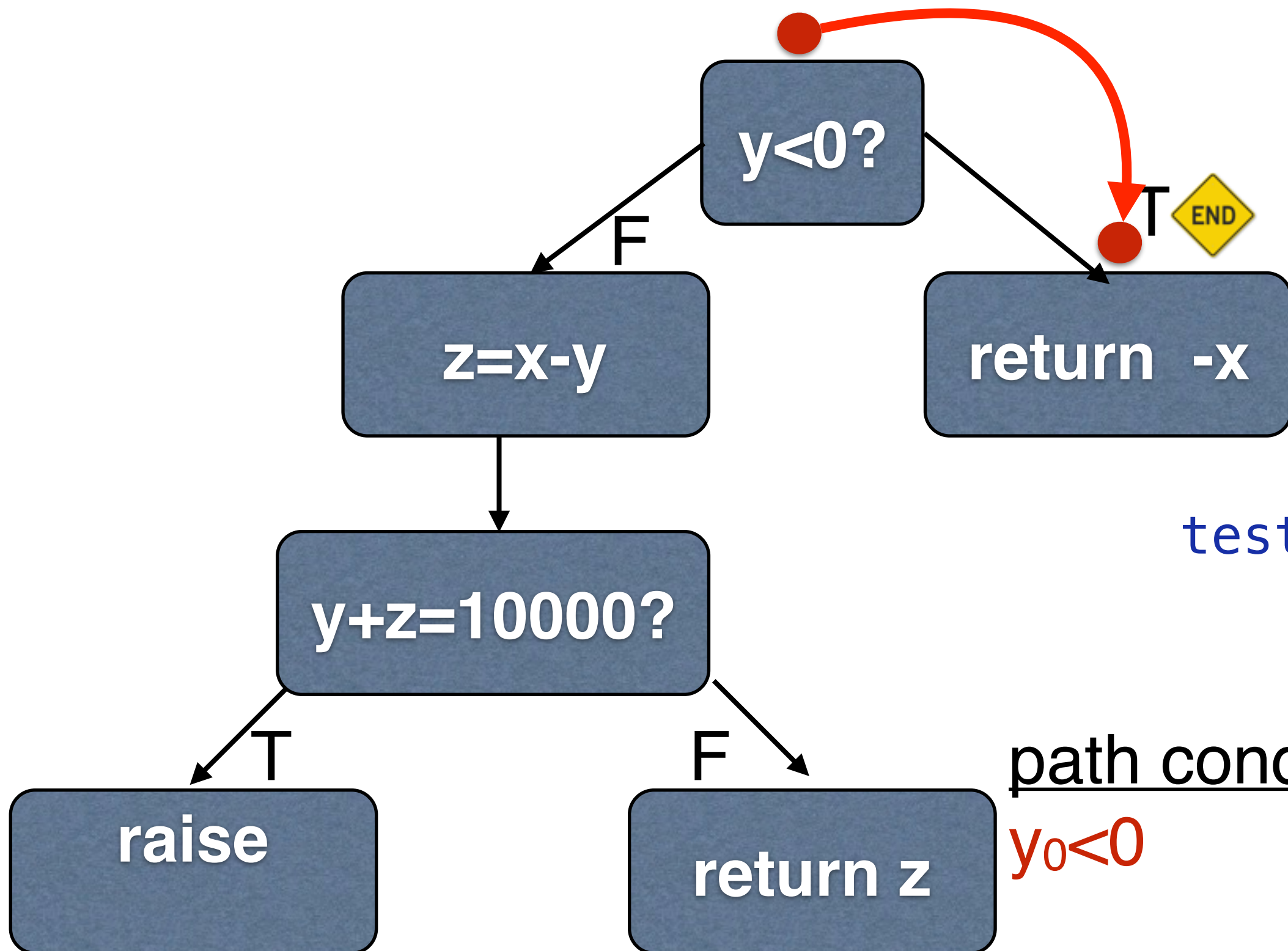
testme(0,0)



Ejecución Simbólica Dinámica



Ejecución Simbólica Dinámica



testme(10000, -1



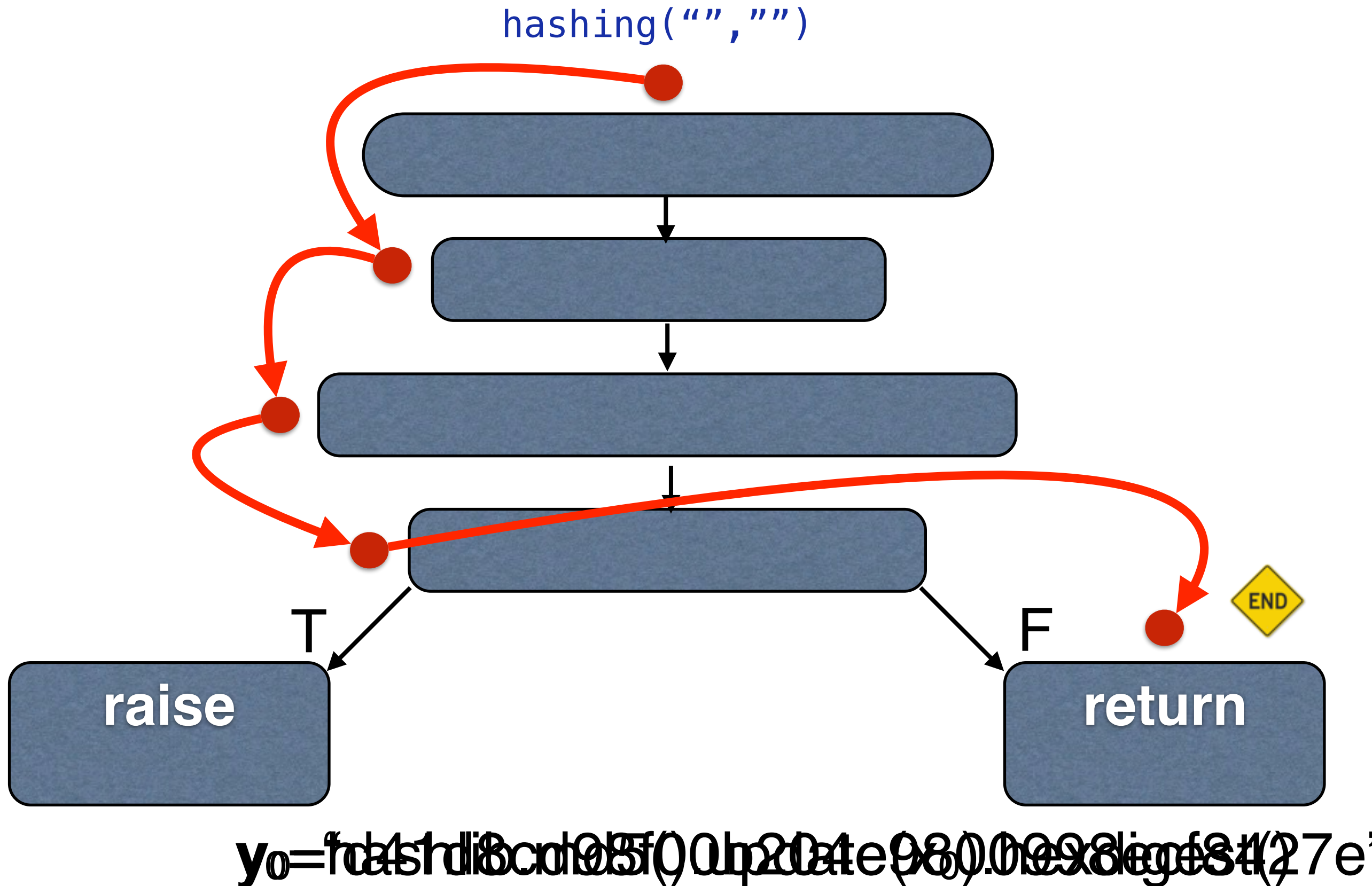
path condition:

$y_0 < 0$

Ejecución Simbólica Dinámica

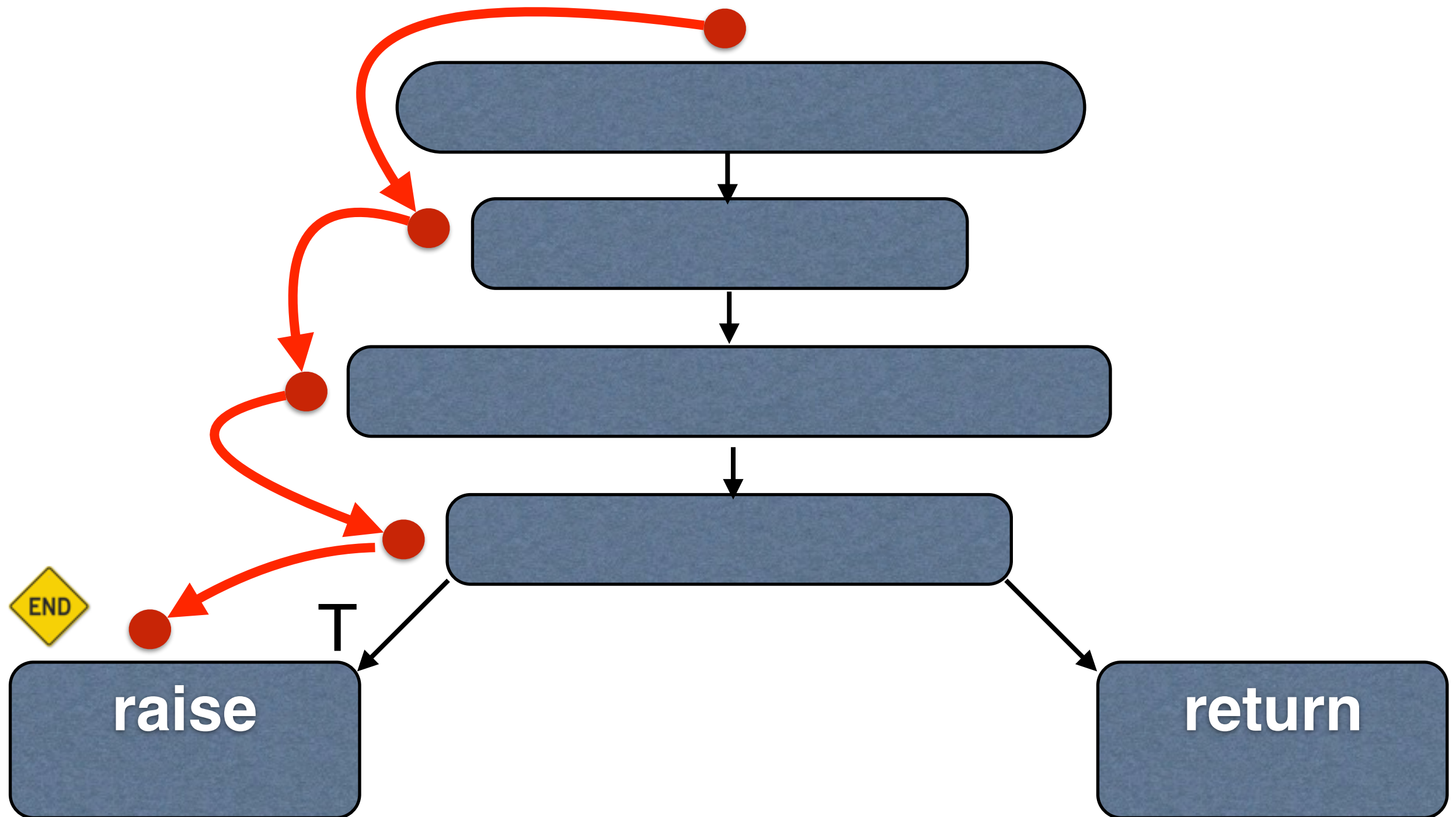
```
1. def hashing(x, y):  
2.     m = hashlib.md5()  
3.     m.update(x)  
4.     hash_str = m.hexdigest()  
5.     if (y==hash_str):  
6.         raise Exception("error")  
7.     else:  
8.         return hash_str
```


Ejecución Simbólica Dinámica



Ejecución Simbólica Dinámica

hashing("", "d41d8cd98f00b204e9800998ecf8427e")



Ejecución Simbólica Dinámica

```
// Input: entry function  
// Output: set of test cases  
def generate_dse(entry_f):  
  
    t = create_test(entry_f, None)  
    tests = { t }
```

Ejecución Simbólica Dinámica

```
// Input: entry function  
// Output: set of test cases  
def generate_dse(entry_f):  
  
    t = create_test(entry_f, None)  
    tests = [ t ]  
  
    while True:  
        // obtener nueva path condition  
        path_condition = exec_concolic(t)  
        for i in range(len(path_condition)):  
            // terminar si no hay mas budget  
            if not hasTime():  
                return tests
```

Ejecución Simbólica Dinámica

```
// Input: entry function  
// Output: set of test cases  
def generate_dse(entry_f):  
  
    t = create_test(entry_f, None)  
    tests = [ t ]  
  
    while True:  
        // obtener nueva path condition  
        path_condition = exec_concolic(t)  
        for i in range(len(path_condition)):  
            // terminar si no hay mas budget  
            if not hasTime():  
                return tests  
            // obtener un branch sin cubrir  
            branch = path_condition[i]  
            if branch.isCoveredBothWays():  
                continue
```

```
// Input: entry function
// Output: set of test cases
def generate_dse(entry_f):

    t = create_test(entry_f, None)
    tests = [ t ]

    while True:
        // obtener nueva path condition
        path_condition = exec_concolic(t)
        for i in range(len(path_condition)):
            // terminar si no hay mas budget
            if not hasTime():
                return tests
            // obtener un branch sin cubrir
            branch = path_condition[i]
            if branch.isCoveredBothWays():
                continue
            else:
                // negar el branch cubierto
                new_path = path_condition[0:i-1] + not(branch)
                solution = solve(new_path, solver_timeout)
```

```
t = create_test(entry_f, None)
tests = [ t ]
```

```
while True:
```

```
    // obtener nueva path condition
```

```
    path_condition = exec_concolic(t)
```

```
    for i in range(len(path_condition)):
```

```
        // terminar si no hay mas budget
```

```
        if not hasTime():
```

```
            return tests
```

```
        // obtener un branch sin cubrir
```

```
        branch = path_condition[i]
```

```
        if branch.isCoveredBothWays():
```

```
            continue
```

```
        else:
```

```
            // negar el branch cubierto
```

```
            new_path = path_condition[0:i-1] + not(branch)
```

```
            solution = solve(new_path, solver_timeout)
```

```
            // chequear si existe solucion
```

```
            if solution.isSAT():
```

```
                t = create_test(entry_f, solution)
```

```
                tests.append(t)
```

```
                break
```

```
return tests
```


Ejecución Simbólica Dinámica

- La exploración (i.e. nuevas path conditions a resolver) es guiada por ejecuciones concretas
- Concolic Execution: Concrete+Symbolic
- Algunas limitations de la ejecución simbólica clásica pueden ser superadas aproximando valores simbólicos usando **valores concretos**
- Expresiones complejas
- Información de runtime (files,sockets,native,etc)

¿Por qué Ejecución Concólica?

- Problema: si ejecutamos el input y solo obtenemos el camino recorrido en el CFG, entonces no sabemos cuales fueron los valores concretos en cada estado intermedio
- Por eso, si necesitamos “fallar” a un valor concreto, en ese momento no lo tenemos
- `exec_concolic(t)` hace ambas cosas a la vez!

Divergencia entre Ejecución Simbólica vs. Concreta

- Cuando aproximamos un valor simbólico usando un valor concreto corremos el riesgo de introducir **imprecisión**:
 - La solución retornada no recorre el camino esperado
 - Tenemos que considerar este caso y adaptar nuestro algoritmo de ejecución simbólica **en caso de introducir imprecisiones**

Cambios en la Generación si hay Divergencia

- Conservamos una lista de todas las path conditions que generamos con ejecución simbólica
- Cada vez que creamos una nueva path condition para resolver, chequeamos que sea un prefijo de la path condition obtenida de ejecutar el nuevo test case
- Si no lo es, entonces comparamos si ya obtuvimos con anterioridad esa path condition
 - En caso que ya la hayamos explorado, decidimos si vale la pena continuar con el algoritmo de generación

Heap Constraints

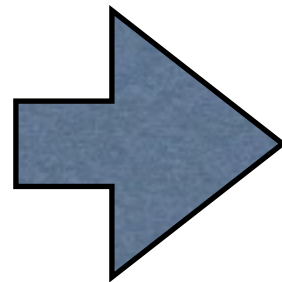
- Muchas veces tenemos constraints que predicen sobre el grafo de memoria (heap)
 - Ejemplo: `list0.header!=null`
- Extendemos el lenguaje de constraints para poder expresar expresiones sobre direcciones de memoria (ie referencias)

Constraint Language

Referencias a Objetos

- Expresiones sobre Referencias:
 - null
 - var0, var1, var
 - E.f
- Constraints:
 - $E == E$, $E != E$

Heap Constraints



```
var0=0x001  
var0.header=0x002  
var0.header.next=0x003  
var1=0x002
```

Como *header* y *next* son campos privados, no sabemos como crear esta configuración de la memoria!

Heap Constraints

- Muchas veces tenemos constraints que predicen sobre el grafo de memoria (heap)
 - Ejemplo: `list0.header!=null`
- La solución nos da un grafo posible, pero desconocemos la secuencia de métodos para generarlo si hay métodos privados
 - Podemos reemplazarlo (en parte) si el lenguaje es **reflexivo**

Lenguajes reflexivos

- Nos permiten introspectivamente analizar el mismo programa (ej: Python)
- En Java, podemos:
 - Crear objetos (aunque no haya constructor por defecto)
 - Asignar un valor a campos privados

```
var0=List$0  
var0.header=Node$0  
var0.header.next=Node$1  
var1=Node$1
```



```
Object list_0 = createObject(List.class);  
Object node_0 = createObject(Node.class);  
Object node_1 = createObject(Node.class);  
setPrivateField(list_0,"header",node_0);  
setPrivateField(node_0,"next",node_1);
```

Otros Desafíos Abiertos para ejecución simbólica

- Constraint Solvers:
 - Riqueza del lenguaje para expresar constraints
 - Longitud y complejidad de las path conditions
- Herramienta de Generación:
 - Código no instrumentado, dependencias externas (no accesibles)
 - Complejidad de las entradas (XML, etc)

Ejecución Simbólica Al Rescate!

Necesitamos...

- Recolectar restricciones del código del programa
- Resolver restricciones para crear nuevos inputs



Constraint Solver

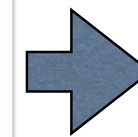


- Es un programa que resuelve fórmulas lógicas descritas en un lenguaje
 - SAT
 - UNSAT
 - UNKNOWN/TIMEOUT
- Si es SAT, da un valor para cada variable

Ejecución Simbólica Dinámica

- La exploración (i.e. nuevas path conditions a resolver) es guiada por ejecuciones concretas
 - Concolic Execution: Concrete+Symbolic
- Algunas limitations de la ejecución simbólica clásica pueden ser superadas aproximando valores simbólicos usando **valores concretos**
 - Expresiones complejas
 - Información de runtime (files, sockets, native, etc)

Heap Constraints



```
var0=List$0  
var0.header=Node$0  
var0.header.next=Node$1  
var1=Node$1
```

Como *header* y *next* son campos privados, no sabemos como crear esta configuración de la memoria!