Symbolic Execution

Ejecución Simbólica Dinámica

Ingeniería del Software 2

Juan P. Galeotti

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

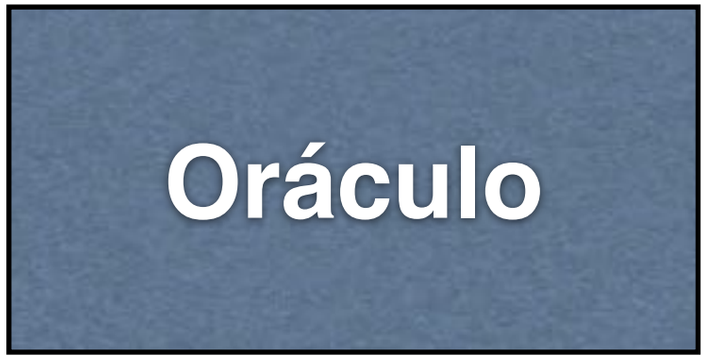
ICC Instituto de Ciencias de la Computación

Random Testing

Programa bajo Test

Oráculo

Random Testing

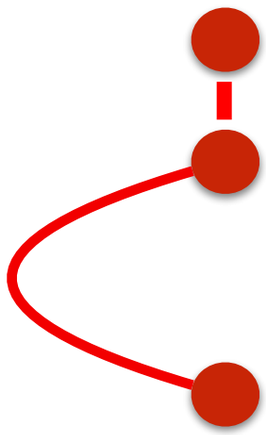


```
1. def testme(x, y):
2.   if (y<0):
3.       return -x
4.   else:
5.       z = x - y
6.       if (y+z=10000):
7.         raise Exception(“error”)
8.       else:
9.         return z
```

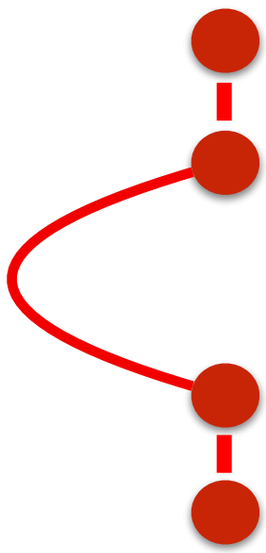
```
1. def testme(x, y):
2.   if (y<0):
3.       return -x
4.   else:
5.       z = x - y
6.       if (y+z=10000):
7.           raise Exception(“error”)
8.       else:
9.           return z
```

```
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
```

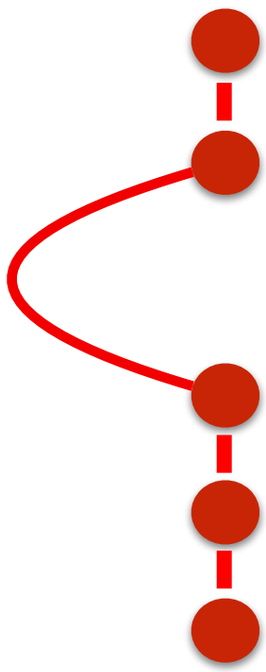
```
1. def testme(x, y):
2.   if (y<0):
3.       return -x
4.   else:
5.       z = x - y
6.       if (y+z=10000):
7.           raise Exception(“error”)
8.       else:
9.           return z
```



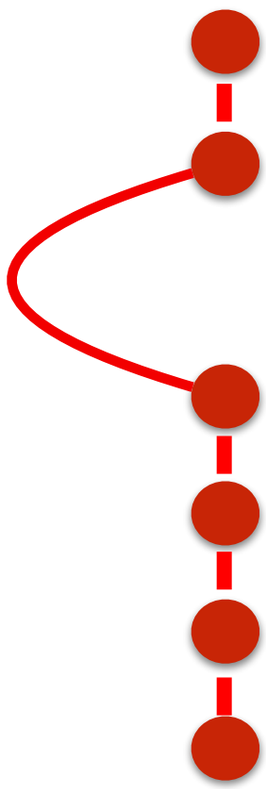
```
1. def testme(x, y):
2.   if (y<0):
3.       return -x
4.   else:
5.       z = x - y
6.       if (y+z=10000):
7.           raise Exception(“error”)
8.       else:
9.           return z
```



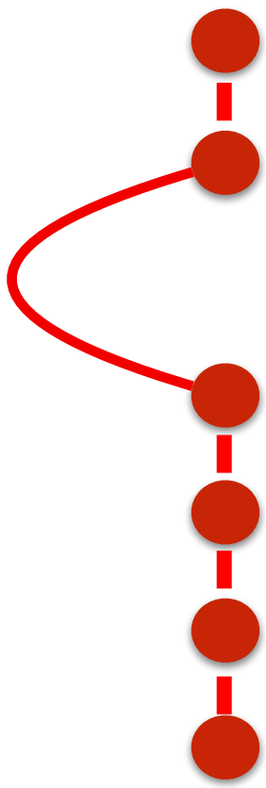
```
1. def testme(x, y):
2.   if (y<0):
3.       return -x
4.   else:
5.       z = x - y
6.       if (y+z=10000):
7.           raise Exception(“error”)
8.       else:
9.           return z
```



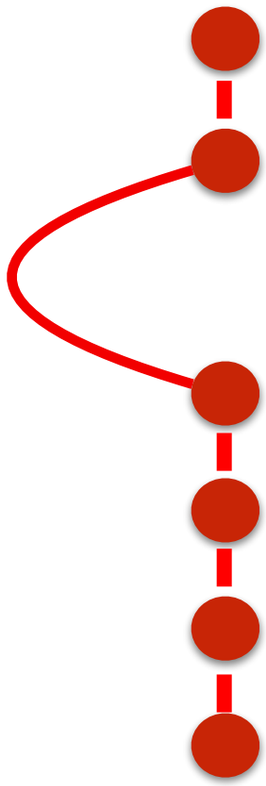
```
1. def testme(x, y):
2.   if (y<0):
3.       return -x
4.   else:
5.       z = x - y
6.       if (y+z=10000):
7.           raise Exception(“error”)
8.       else:
9.           return z
```



```
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
```

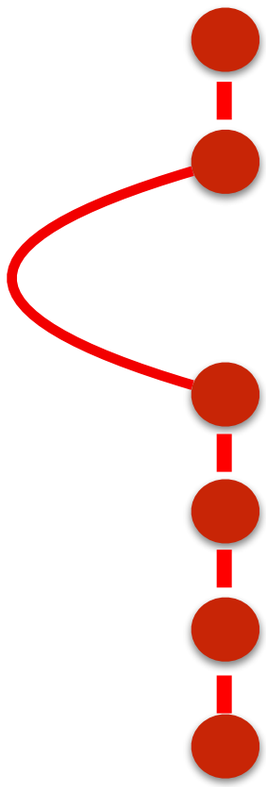


```
1. def testme(x, y):
2.   if (y<0):
3.       return -x
4.   else:
5.       z = x - y
6.       if (y+z=10000):
7.           raise Exception(“error”)
8.       else:
9.           return z
```



```
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
```

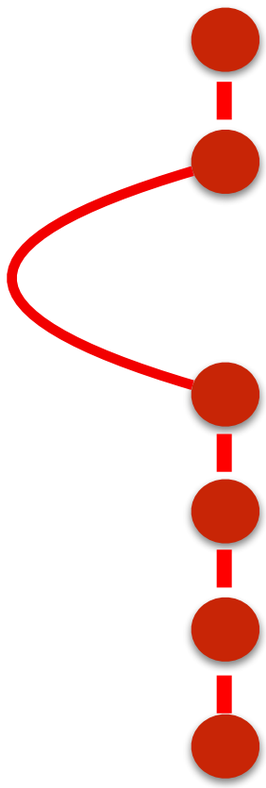
y>=0
x=10000



```
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>=0
x=10000

2^31 / 2^32 = 50%



```
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>=0

x=10000

2^31 / 2^32 = 50%

1 / 2^32 ~ 0%

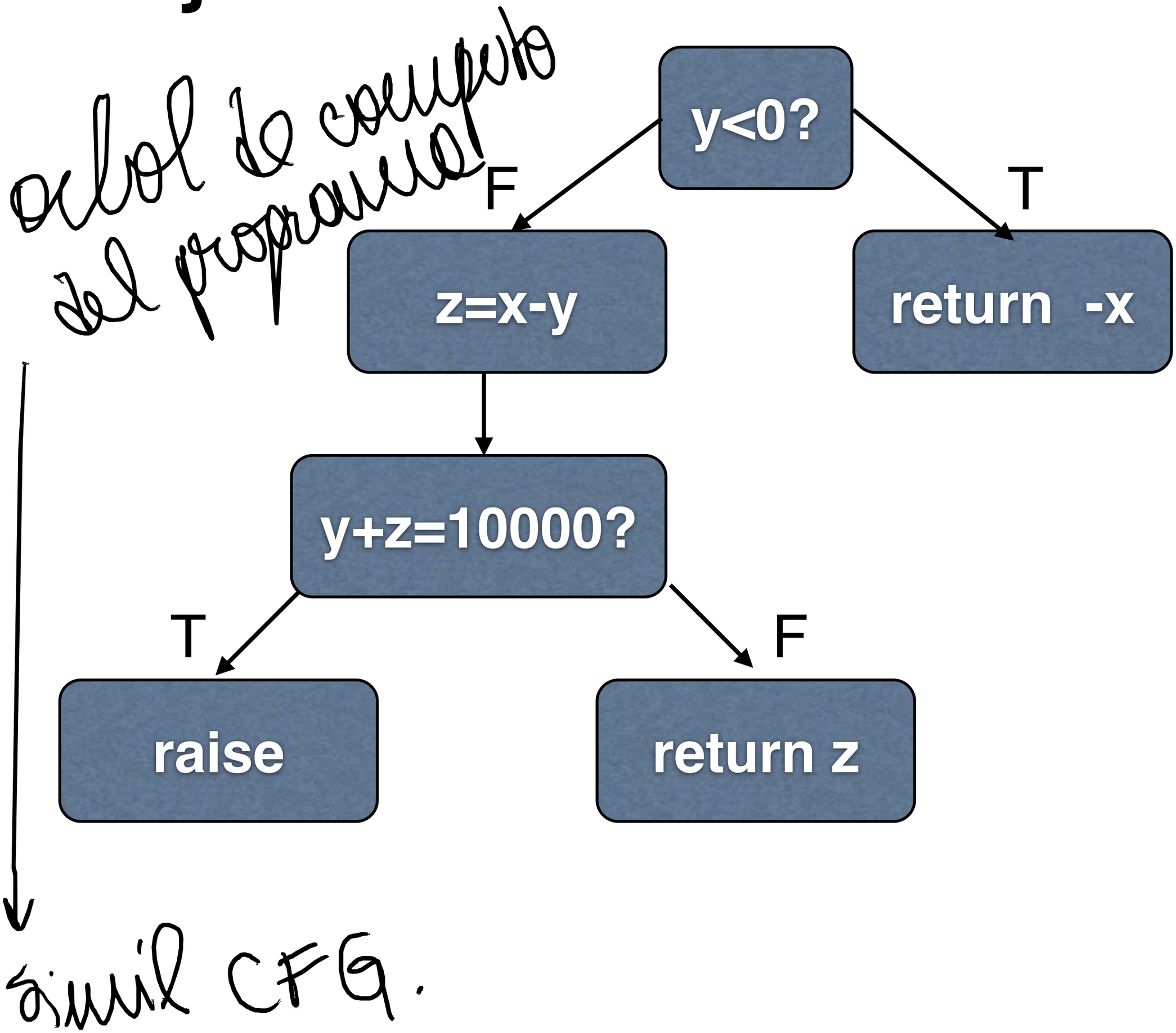
↓

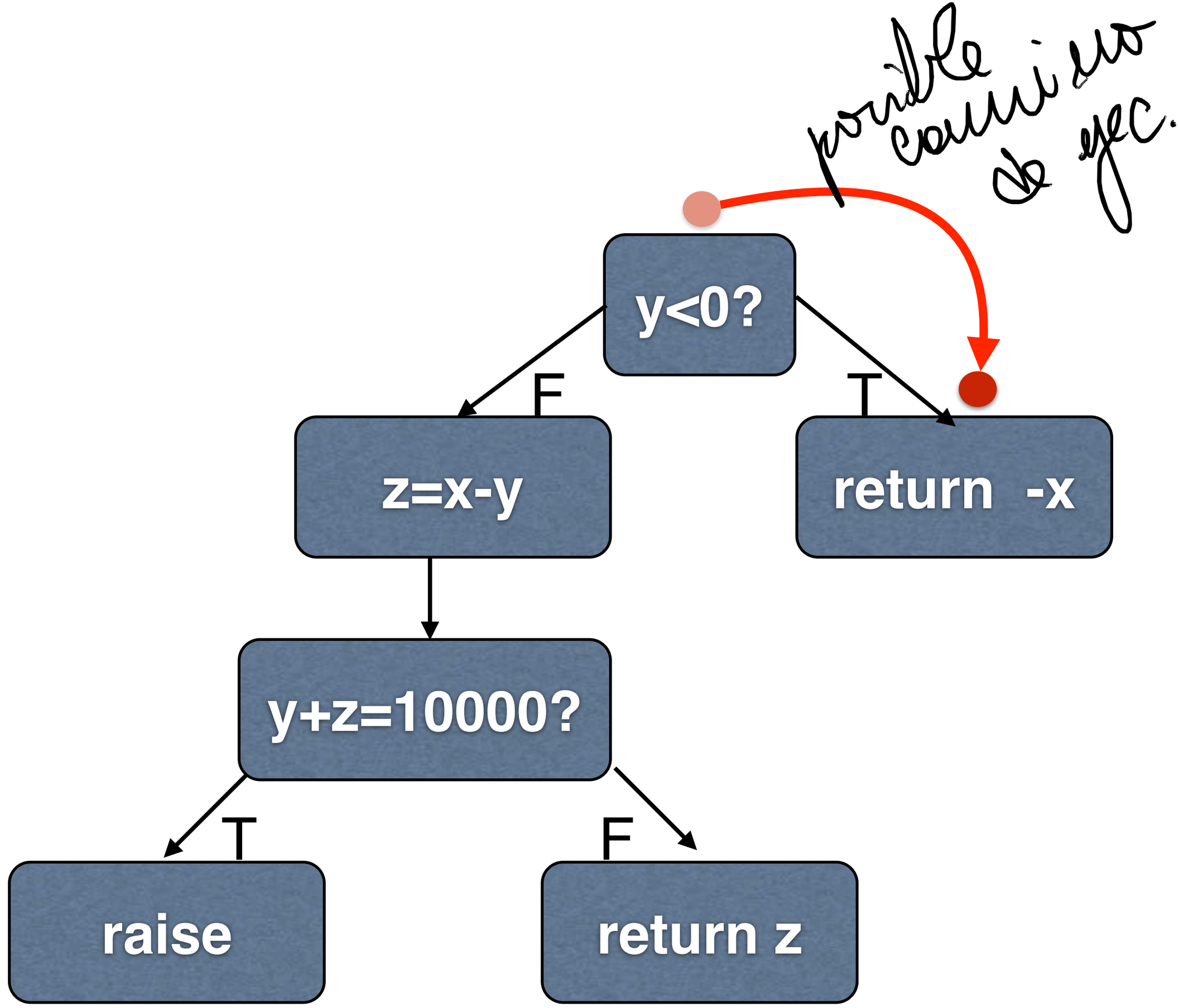
extremadamente improbable que random choice pueda el test llegue 7

Ejecución Simbólica

```
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




possible execution
y<0?
F
T
z=x-y
return -x
y+z=10000?
T
F
raise
return z

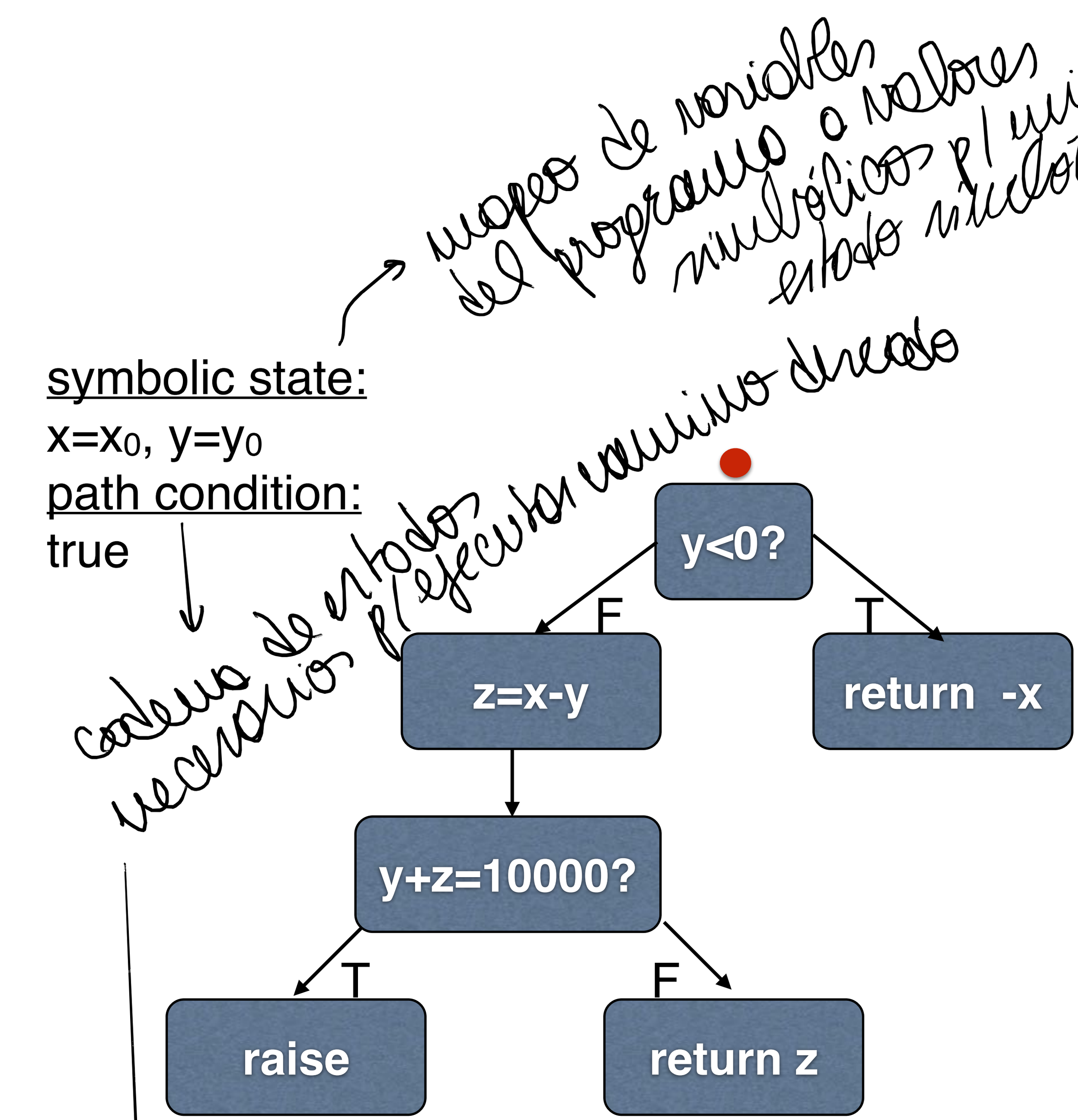
mapeo de variables del programa a valores simbólicos p/ un estado alcanzado

<u>symbolic state:</u>

$x=x_0$, $y=y_0$

<u>path condition:</u>

true



cadena de estados alcanzados durante la ejecución

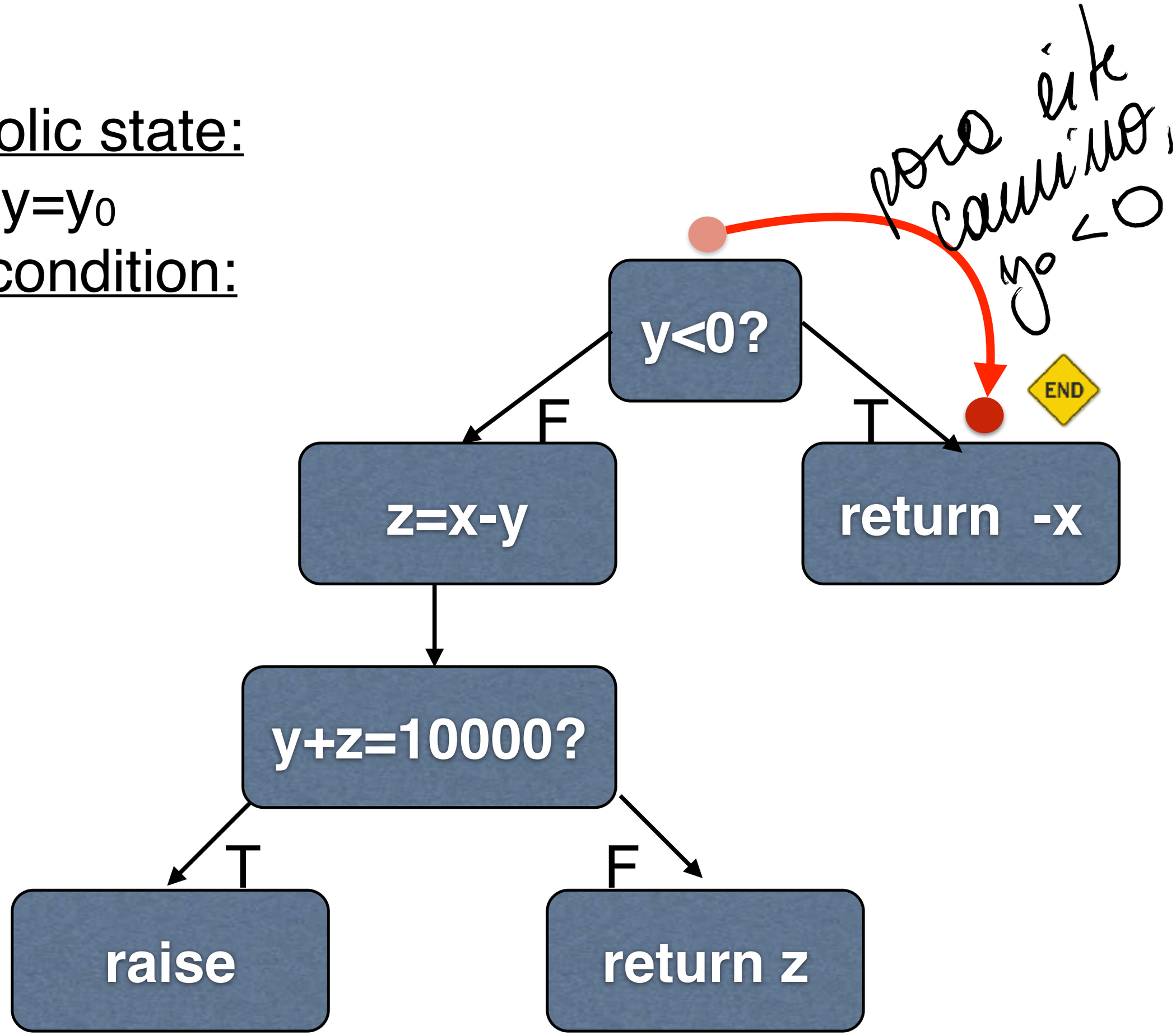
conjunto condiciones p/ que todo camino sea correcto

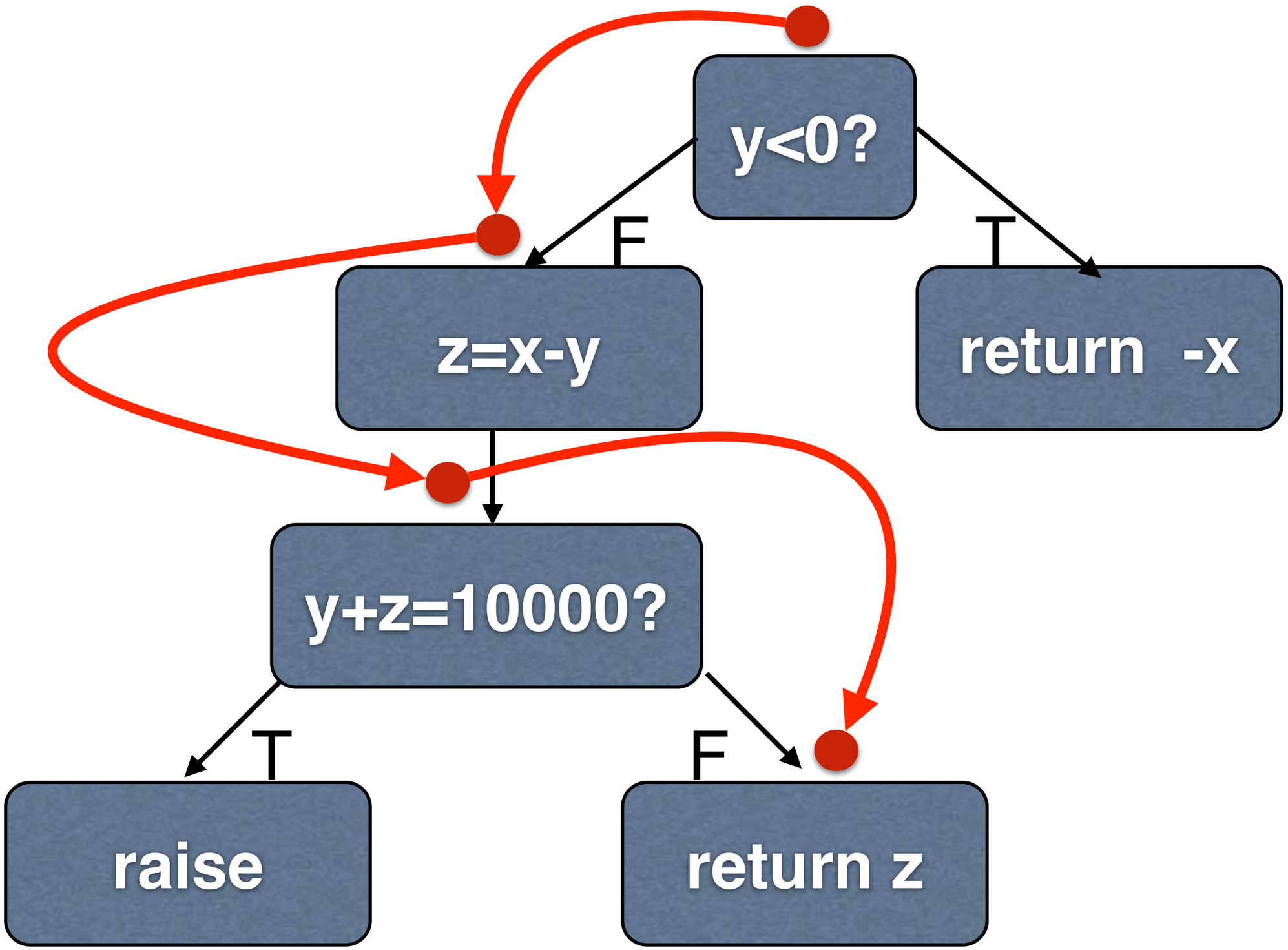
<u>symbolic state:</u>

$x=x_0$, $y=y_0$

<u>path condition:</u>

$y_0<0$



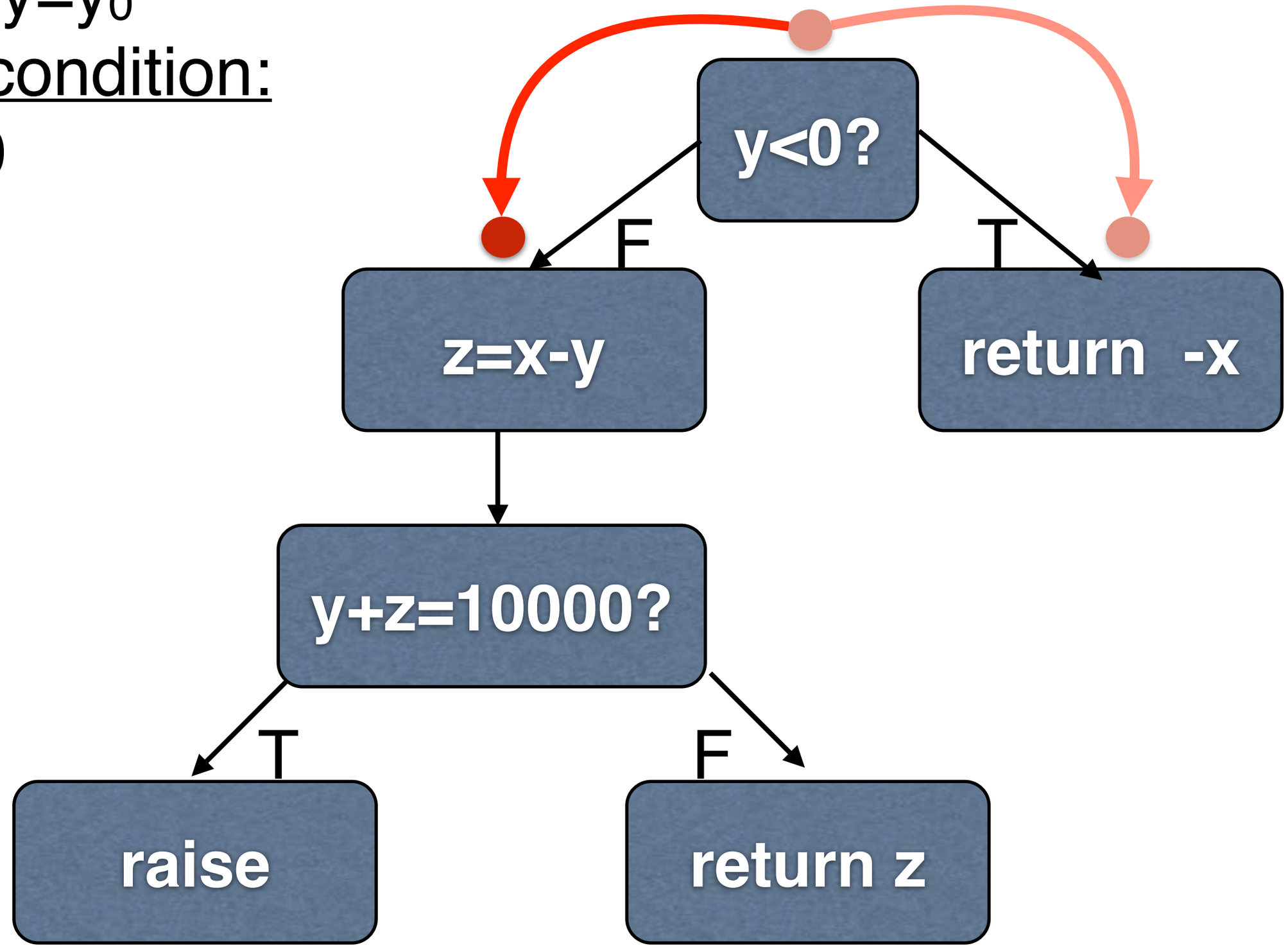

y<0?
F
T
z=x-y
return -x
y+z=10000?
T
F
raise
return z

<u>symbolic state:</u>

$x=x_0$, $y=y_0$

<u>path condition:</u>

$y_0>=0$

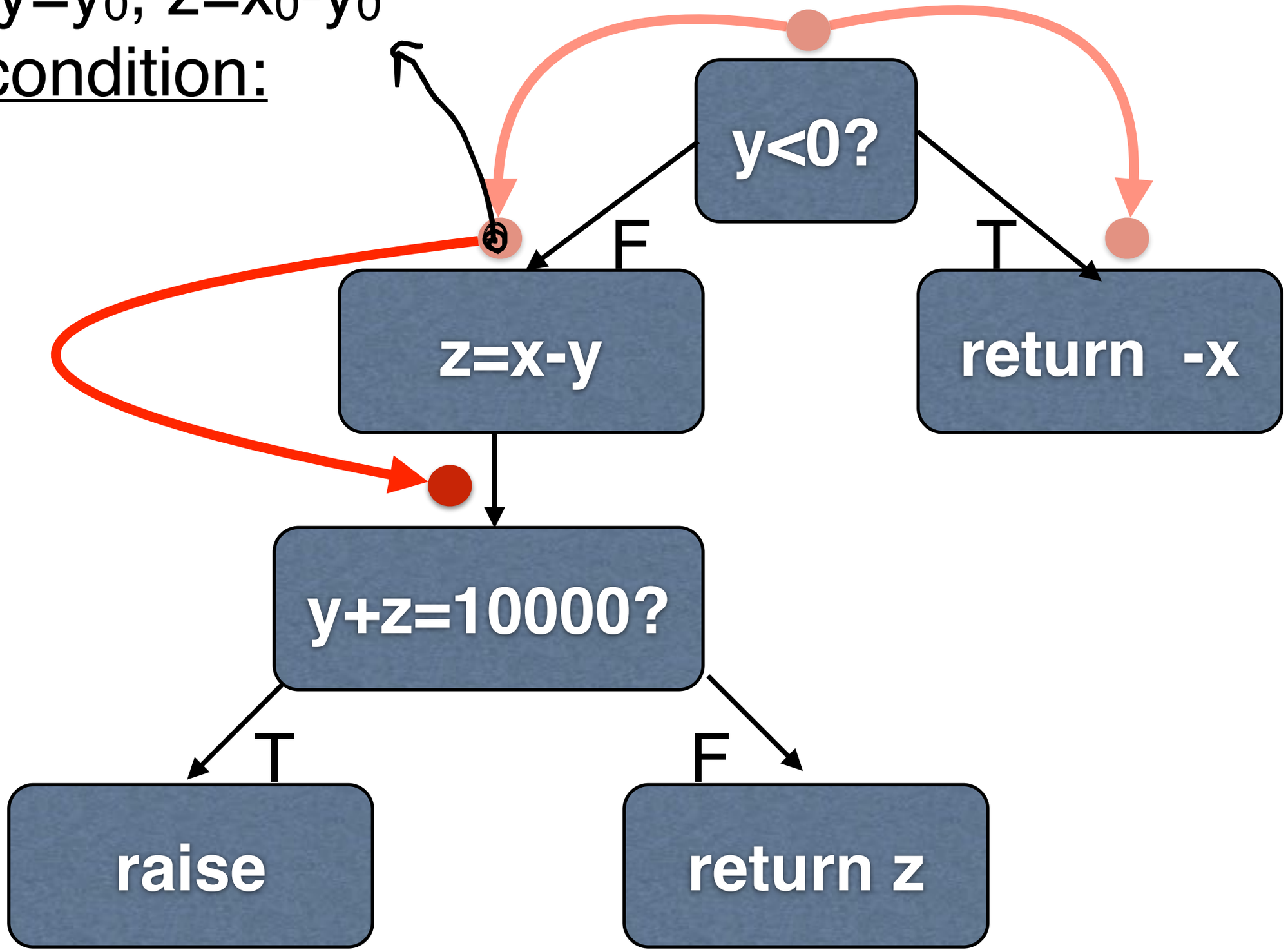


<u>symbolic state:</u>

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

<u>path condition:</u>

$y_0>=0$



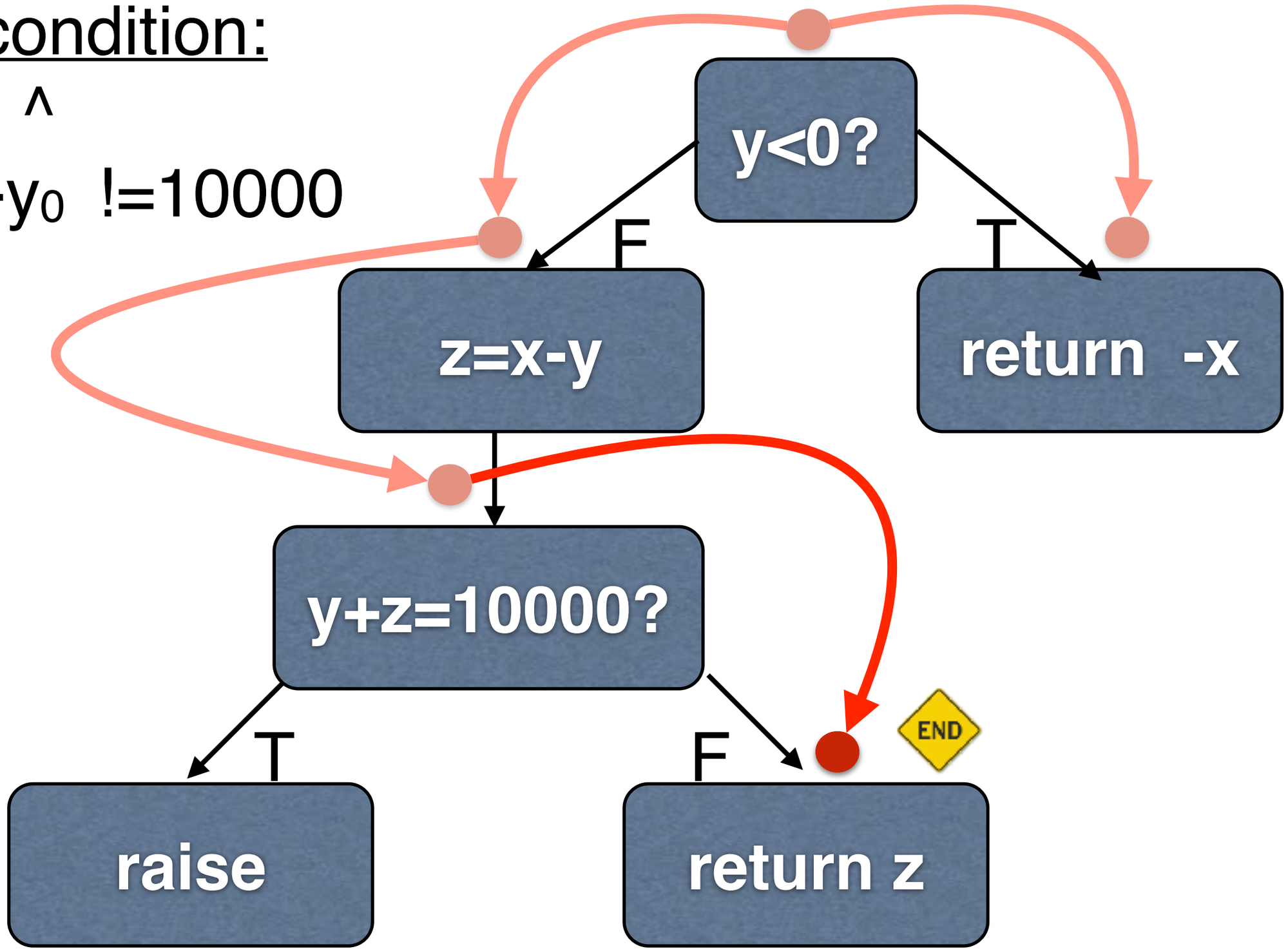
<u>symbolic state:</u>

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

<u>path condition:</u>

$y_0>=0$ ^

$y_0+x_0-y_0$!=10000



conjunción de condiciones

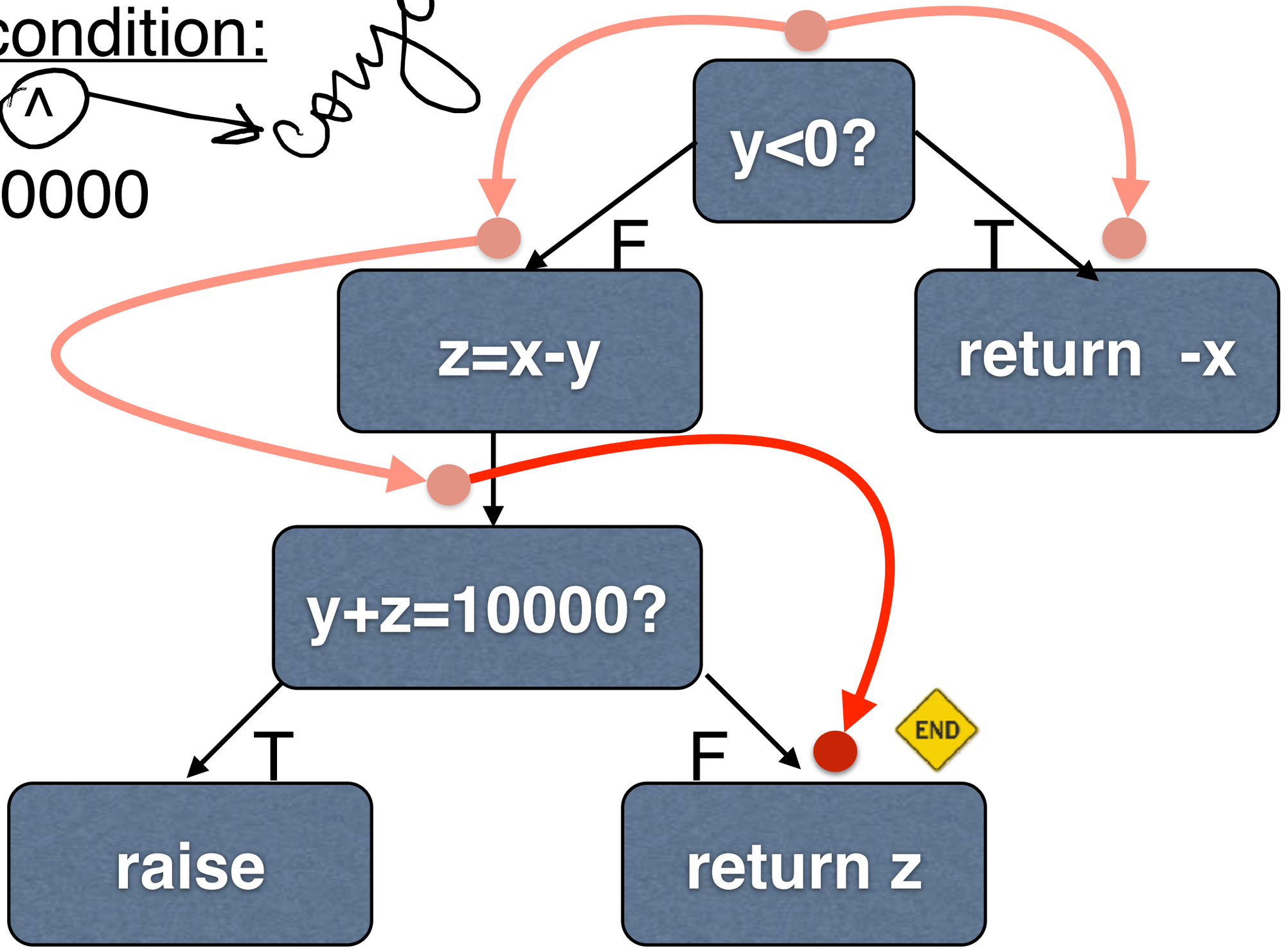
<u>symbolic state:</u>

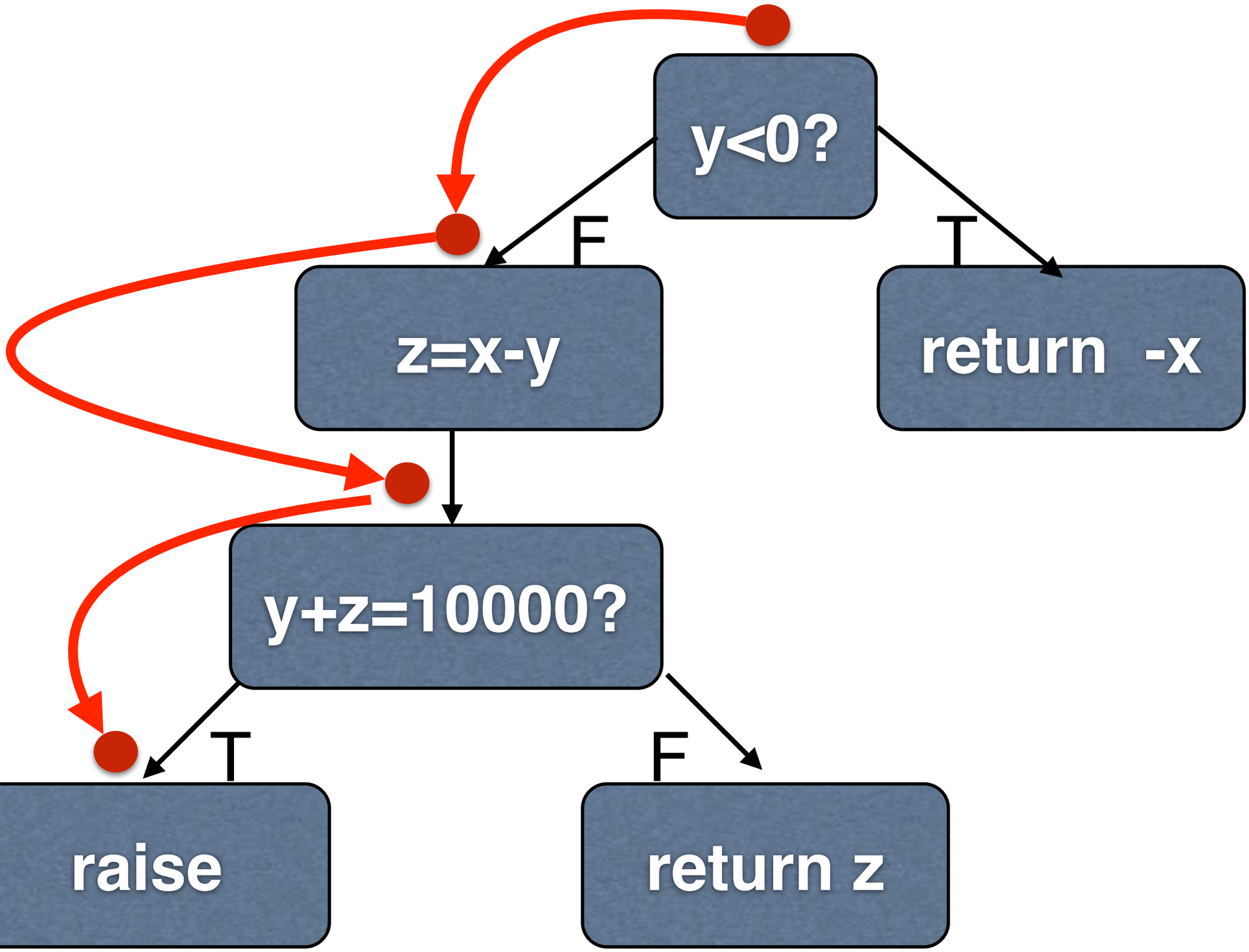
$x=x_0$, $y=y_0$, $z=x_0-y_0$

<u>path condition:</u>

$y_0>=0$ ^

x_0 !=10000



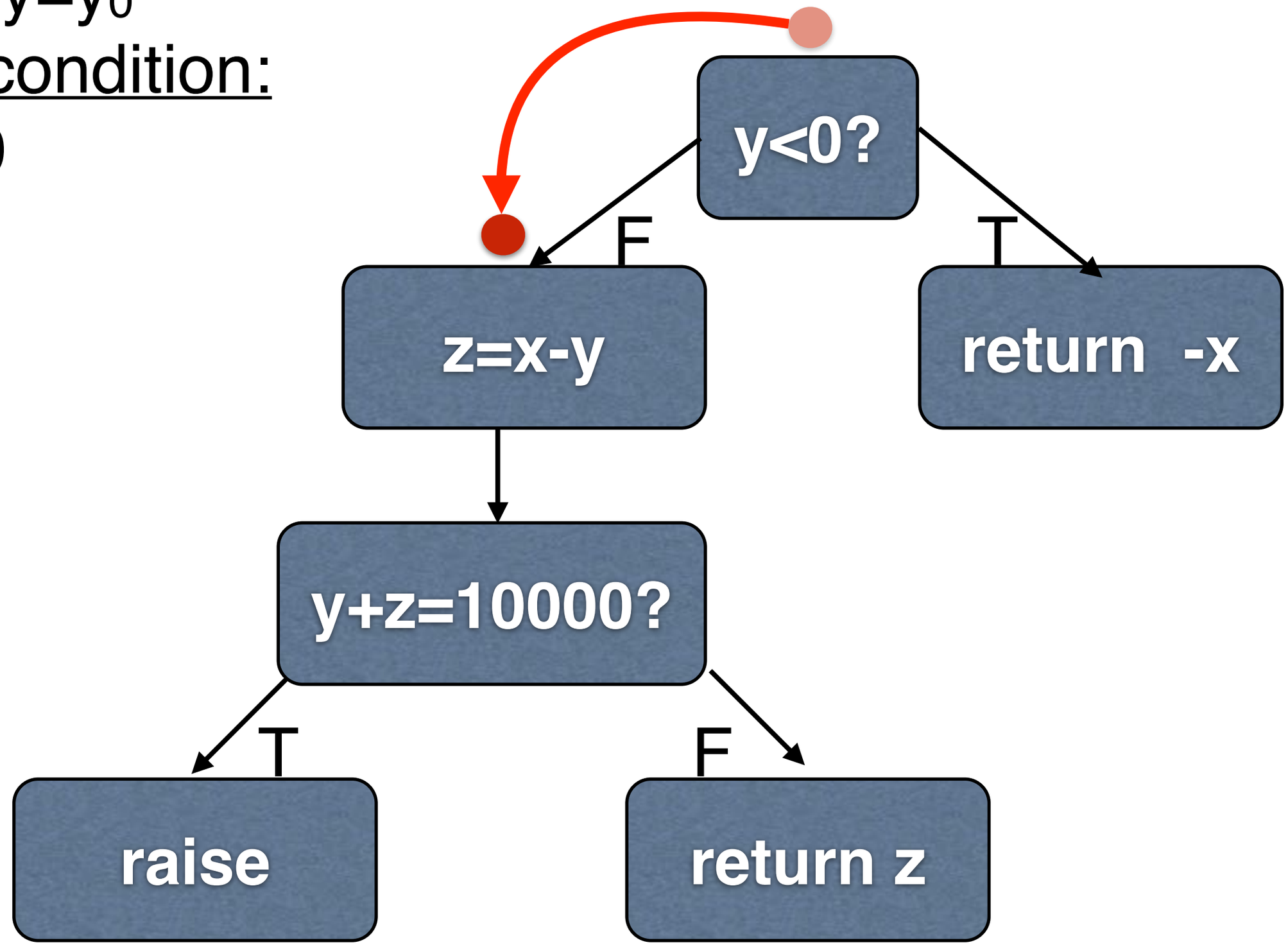

y<0?
F
T
z=x-y
return -x
y+z=10000?
T
F
raise
return z

<u>symbolic state:</u>

$x=x_0$, $y=y_0$

<u>path condition:</u>

$y_0>=0$

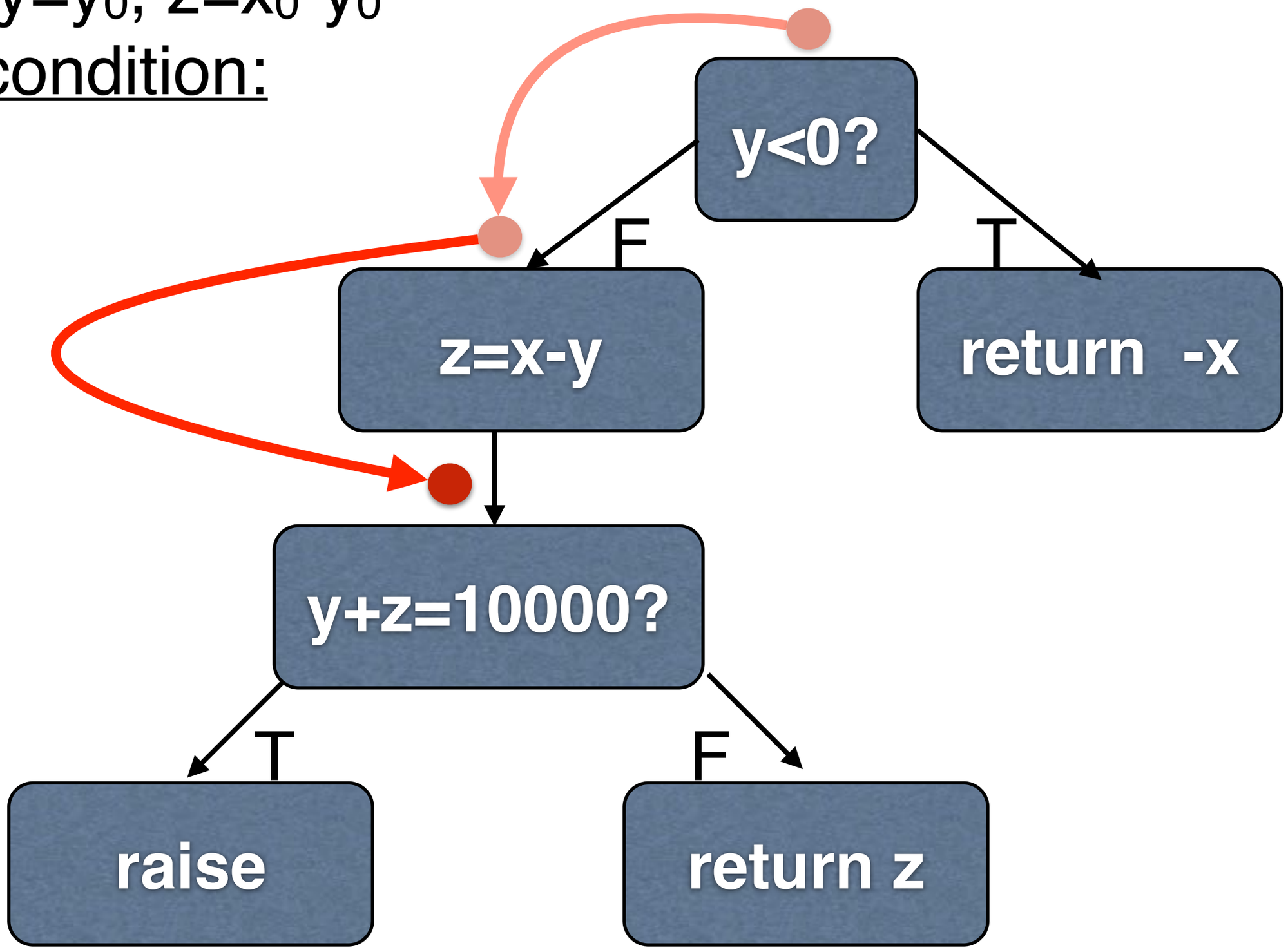


<u>symbolic state:</u>

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

<u>path condition:</u>

$y_0>=0$



<u>symbolic state:</u>

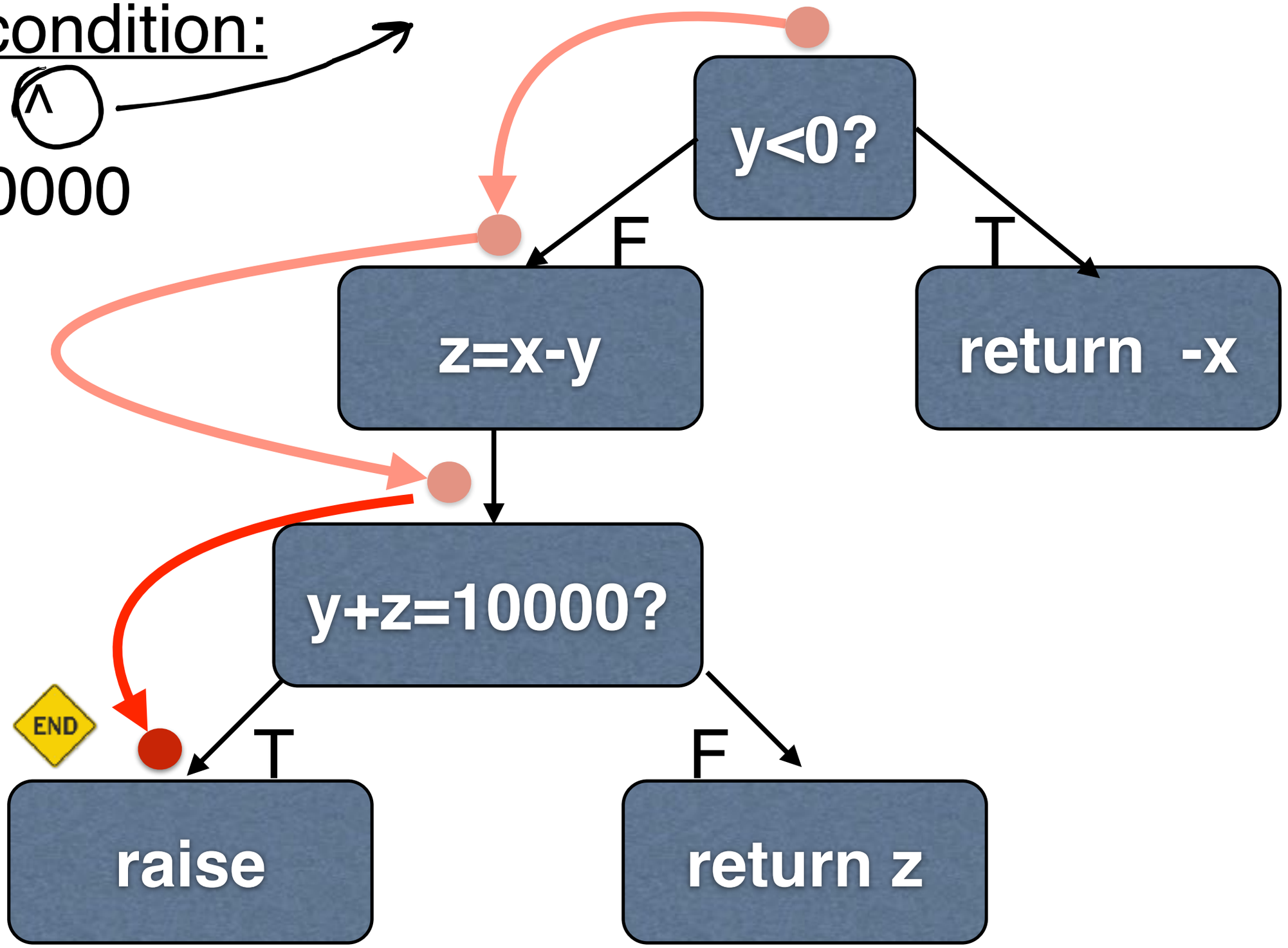
$x=x_0$, $y=y_0$, $z=x_0-y_0$

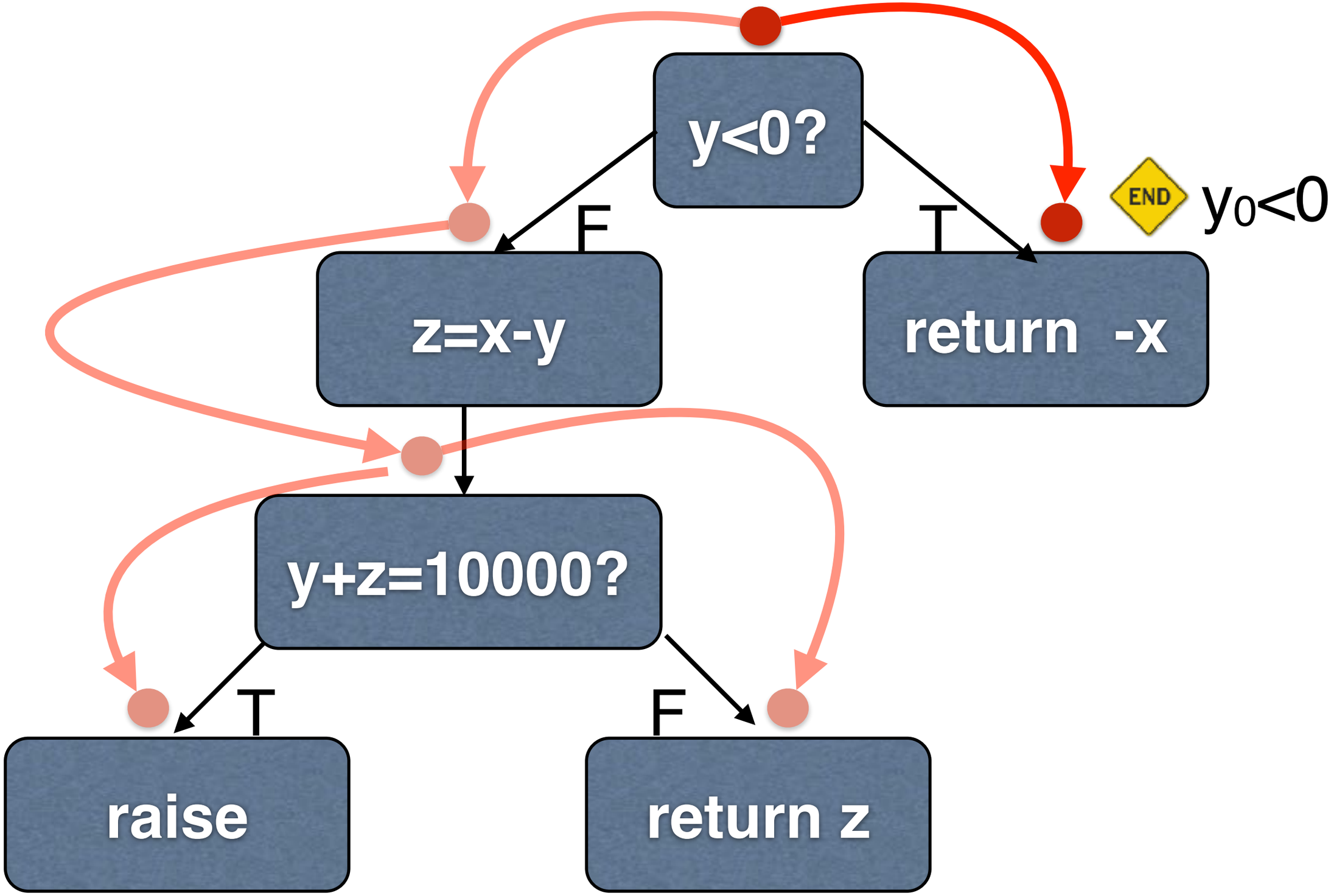
<u>path condition:</u>

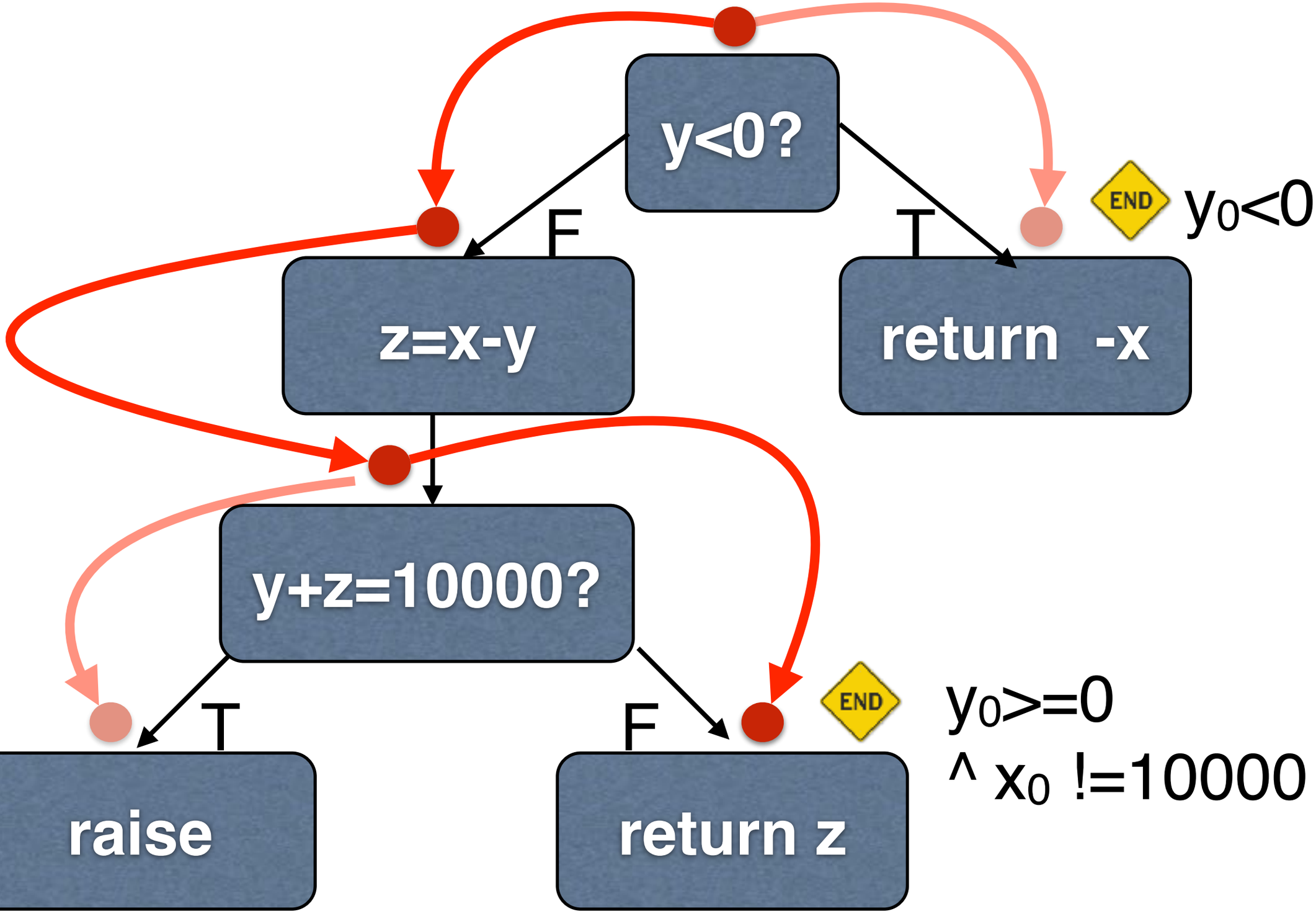
$y_0>=0 \wedge$

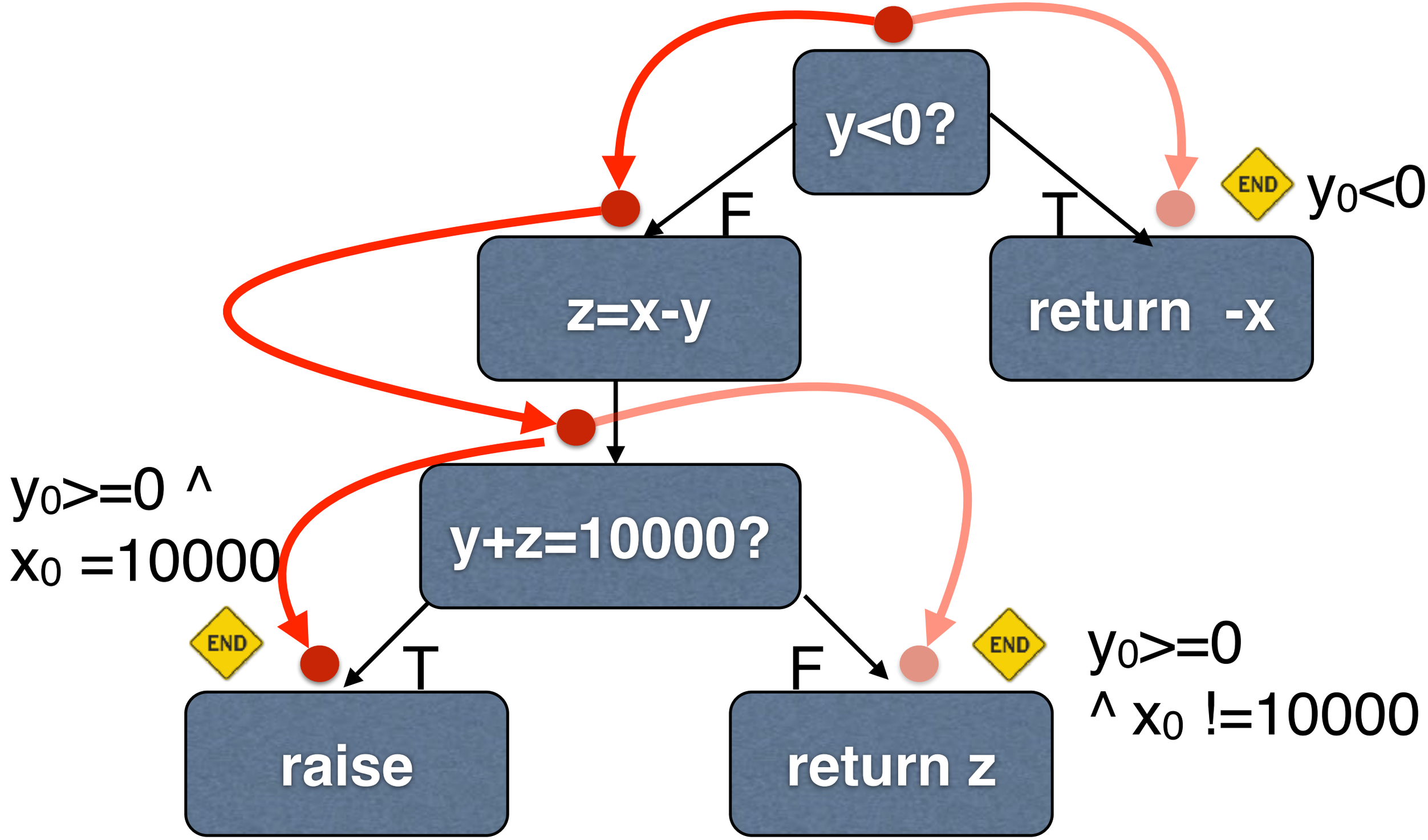
$x_0 =10000$

simple conjunction!




y<0?
F
T
END
y0<0
z=x-y
return -x
y+z=10000?
T
F
raise
return z


y<0?
F
T
END
y0<0
z=x-y
return -x
y+z=10000?
T
F
END
raise
return z
y0>=0
^ x0 !=10000


y<0?
F
T
END
y0<0
z=x-y
return -x
y+z=10000?
y0>=0 ^
x0 =10000
END
T
F
END
y0>=0
^ x0 !=10000
raise
return z

Constraint Solving

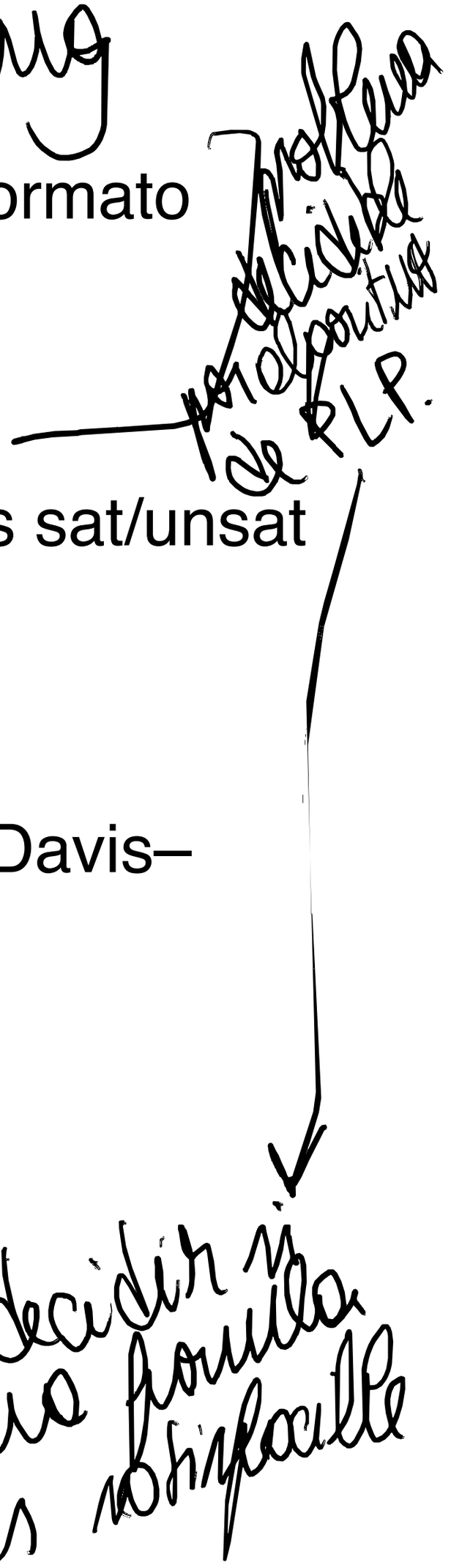
- Es un programa que resuelve fórmulas lógicas descriptas en un lenguaje
  - SAT
  - UNSAT
  - UNKNOWN/TIMEOUT
- Si es SAT, da un valor para cada variable

Determina si una fórmula lógica es válida/satisfacible o no.

SAT-Solvers

∈ Constraint Solving

- Input: Una fórmula proposicional (en formato CNF)
- Output: SAT/UNSAT
- Decidir si una fórmula proposicional es sat/unsat es decidible
- Tarda peor caso O(2^n)
- SAT-Solvers usualmente usan DPLL (Davis–Putnam–Logemann–Loveland)



DPLL

```
Algorithm DPLL
  Input: A set of clauses Φ.
  Output: A Truth Value.
```

```
function DPLL(Φ)
   if Φ is a consistent set of literals
       then return true;
   if Φ contains an empty clause
       then return false;
   for every unit clause {l} in Φ
      Φ ← unit-propagate(l, Φ);
   for every literal l that occurs pure in Φ
      Φ ← pure-literal-assign(l, Φ);
   l ← choose-literal(Φ);
   return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {not(l)});
```

- "←" denotes assignment. For instance, "*largest* ← *item*" means that the value of *largest* changes to the value of *item*.
- "**return**" terminates the algorithm and outputs the following value.

DPLL

- DPLL (Davis–Putnam–Logemann–Loveland)
  Output: SAT/UNSAT
  - Elijo un literal cuyo valor no ha sido fijado
  - Propago el valor del literal hasta no poder eliminar mas literales
  - Si hay conflicto, hago backtrack
  - Si no lo hay, elijo una nueva variable para asignar

DPLL: Ejemplo

A or B or C

and

Not A or B or C

and

Not A or Not B or Not C

DPLL: Ejemplo

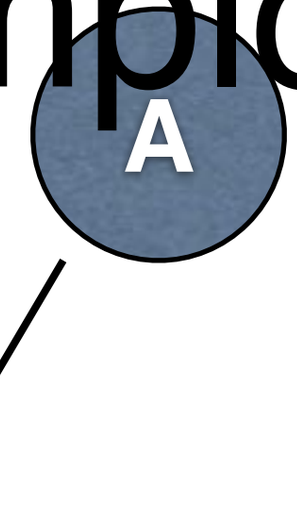
A or B or C

and

Not A or B or C

and

Not A or Not B or Not C



DPLL: Ejemplo

True

True

and

B or C

and

Not B or Not C

DPLL: Ejemplo

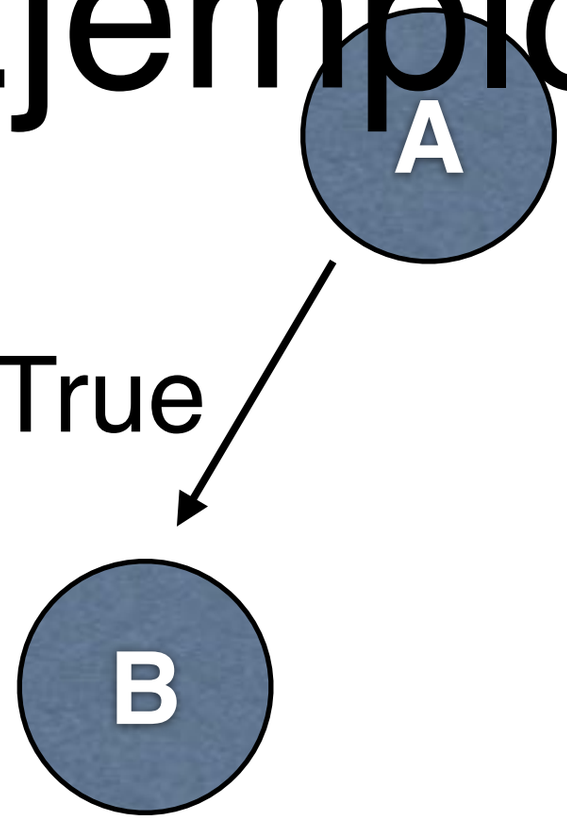
True

and

B or C

and

Not B or Not C



DPLL: Ejemplo

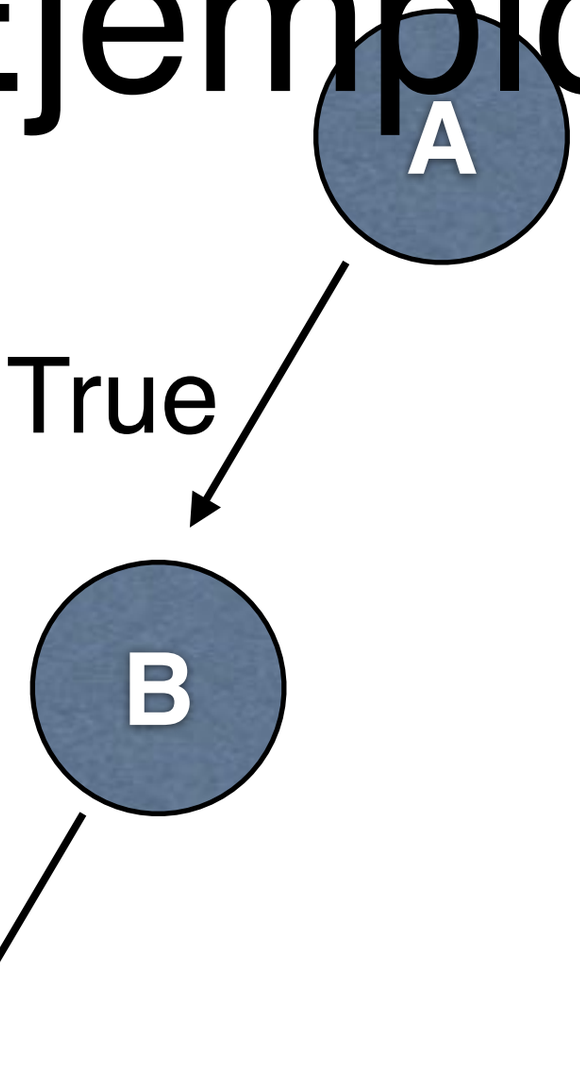
True

and

B or C

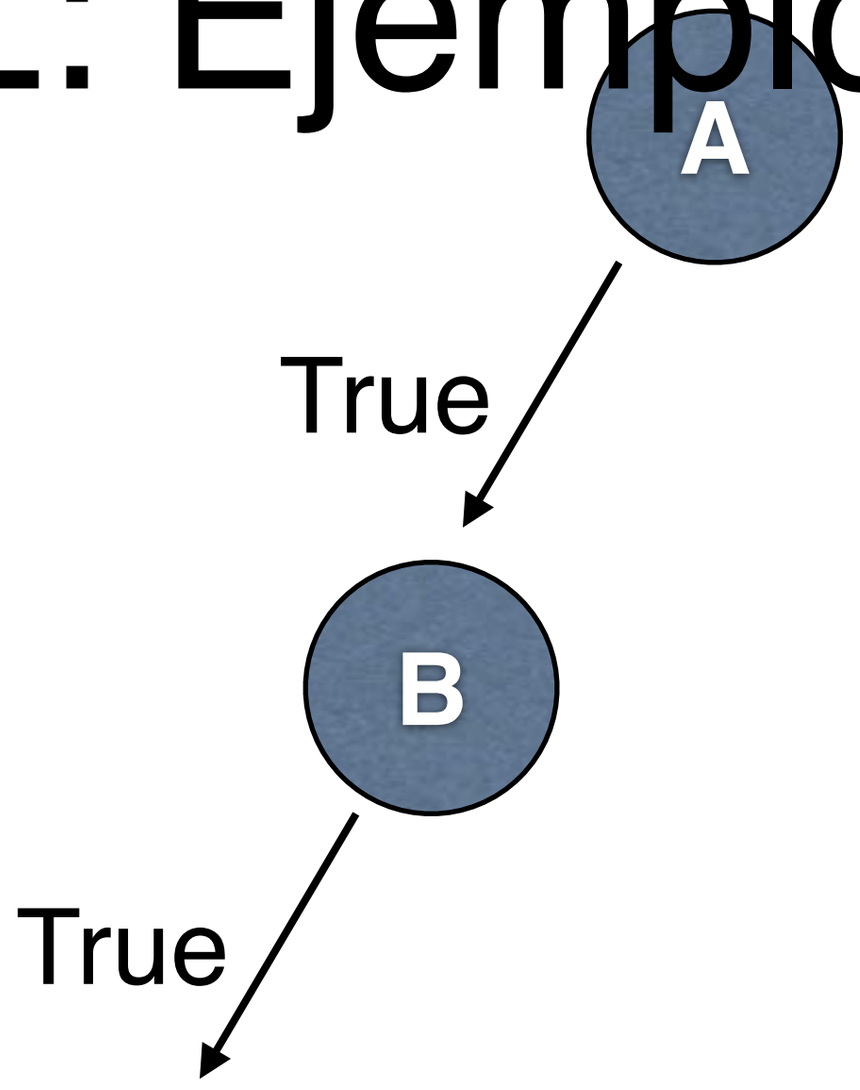
and

Not B or Not C



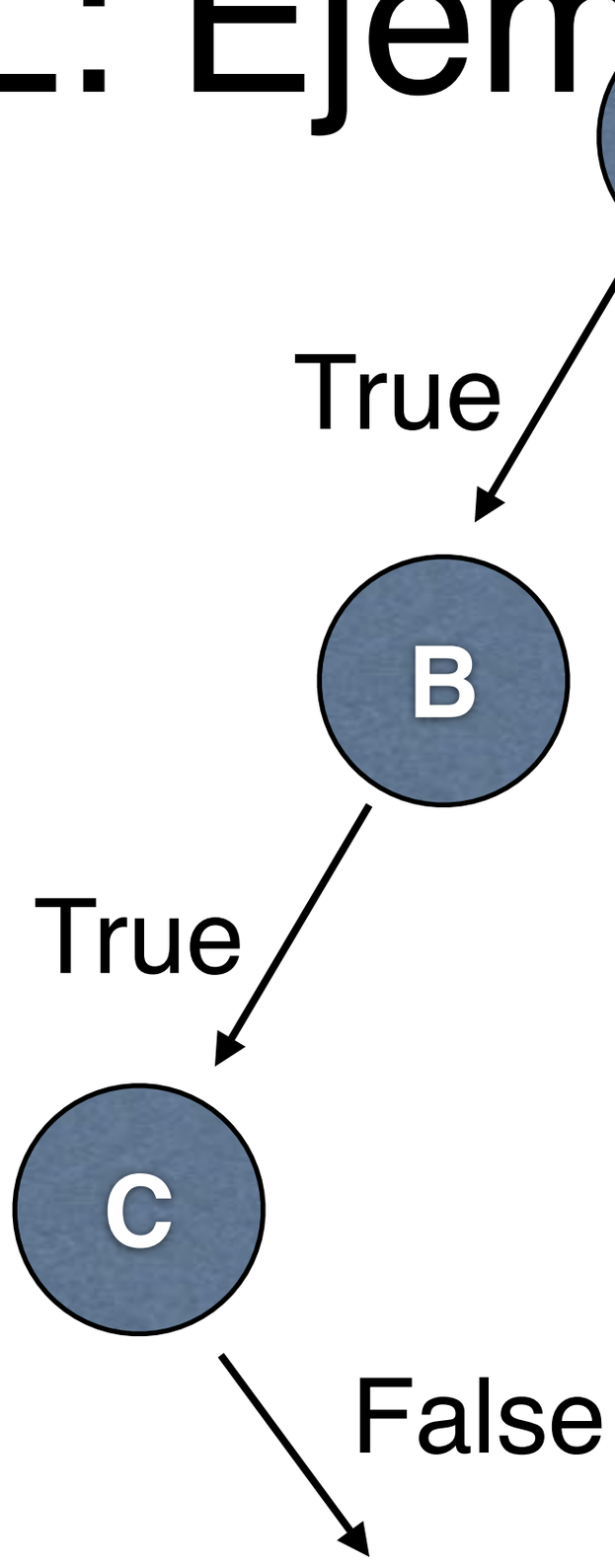
DPLL: Ejemplo

True
and
True
and
Not C

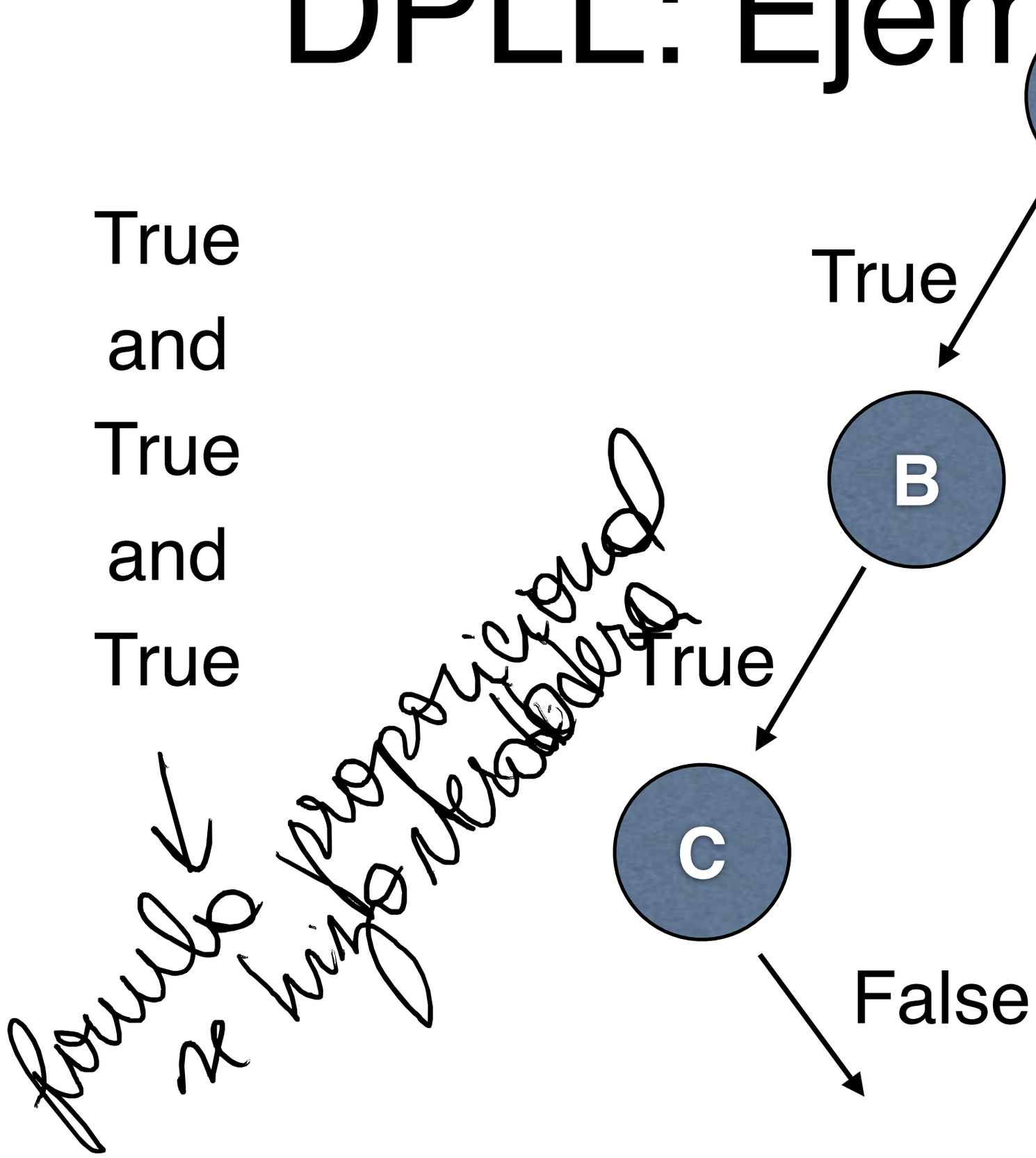


DPLL: Ejemplo

True
and
True
and
Not C



DPLL: Ejemplo



DPLL: Ejemplo

A or B or C

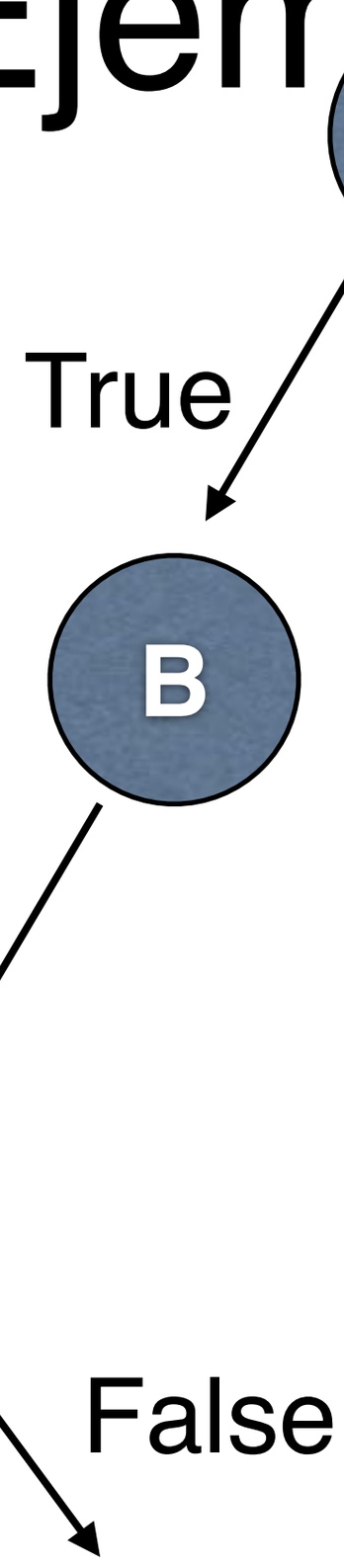
and

Not A or B or C

and

Not A or Not B or Not C

La fórmula
Es satisfacible
(A=true, B=True, C=False)



given a conflict → learn to create clauses

Conflict-Driven Clause Learning

SAT-Solving

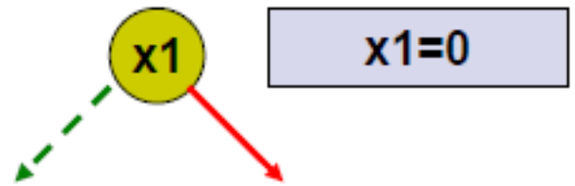
- Mejorar DPLL usando “learning de cláusulas”
- Cuando hay un conflicto, lo “aprendo” como una fórmula nueva que agrego a lo que tengo que cumplir
- Permite encontrar mas rápidamente “contradicciones” dentro de la fórmula

clause captures conflict
& try to solve conflict first
& avoid further issues down the line

CDCL: Ejemplo

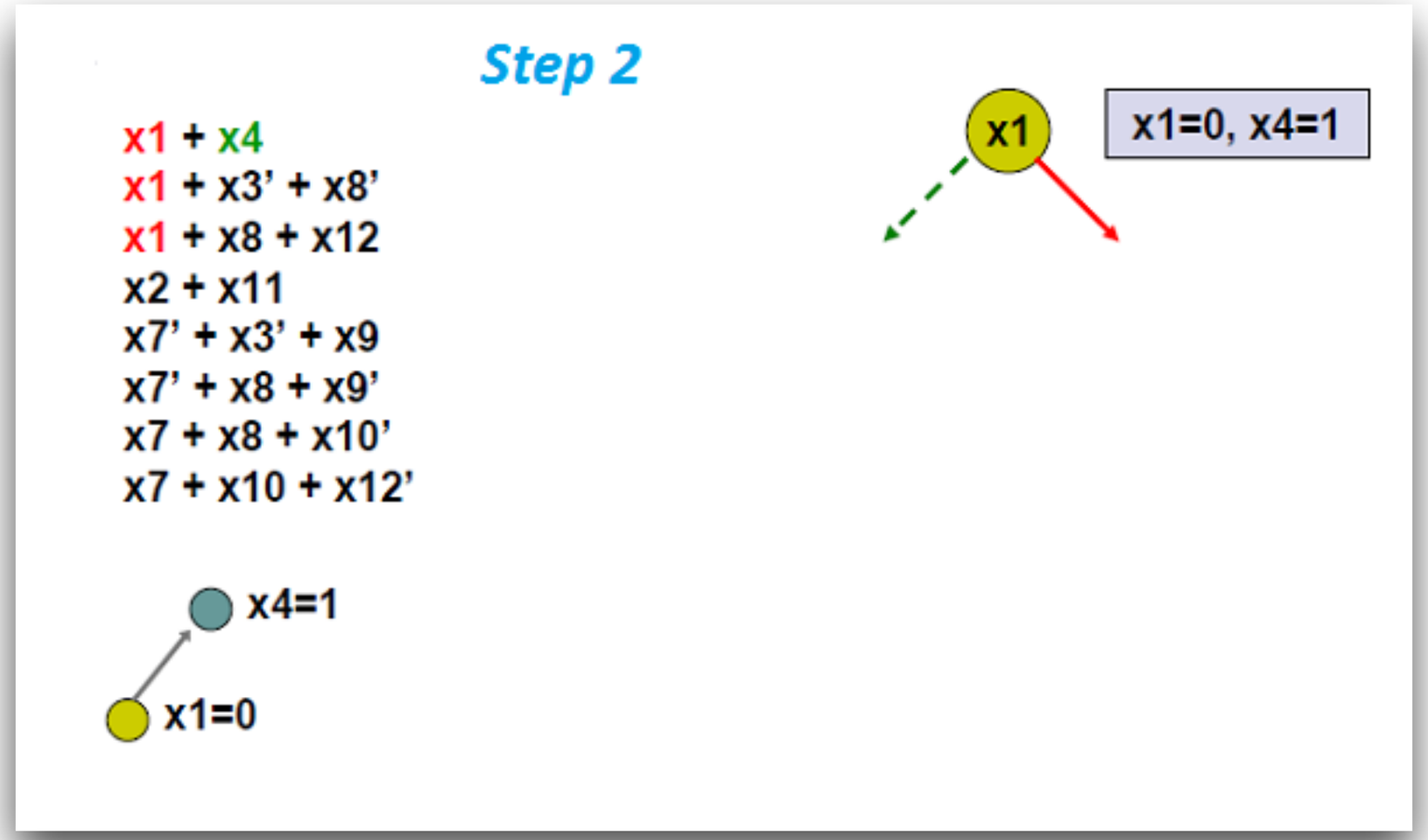
Step 1

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

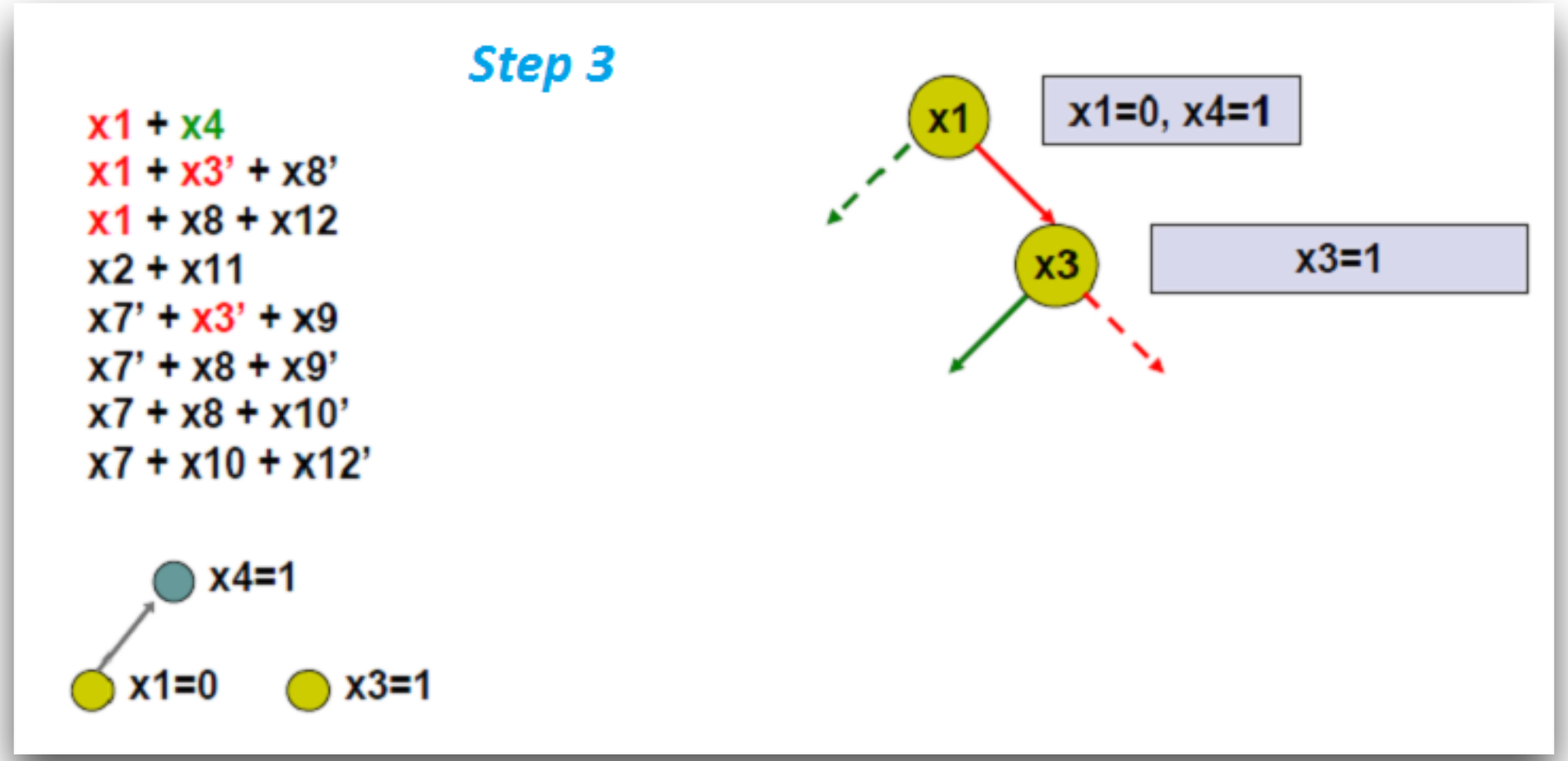


x1=0

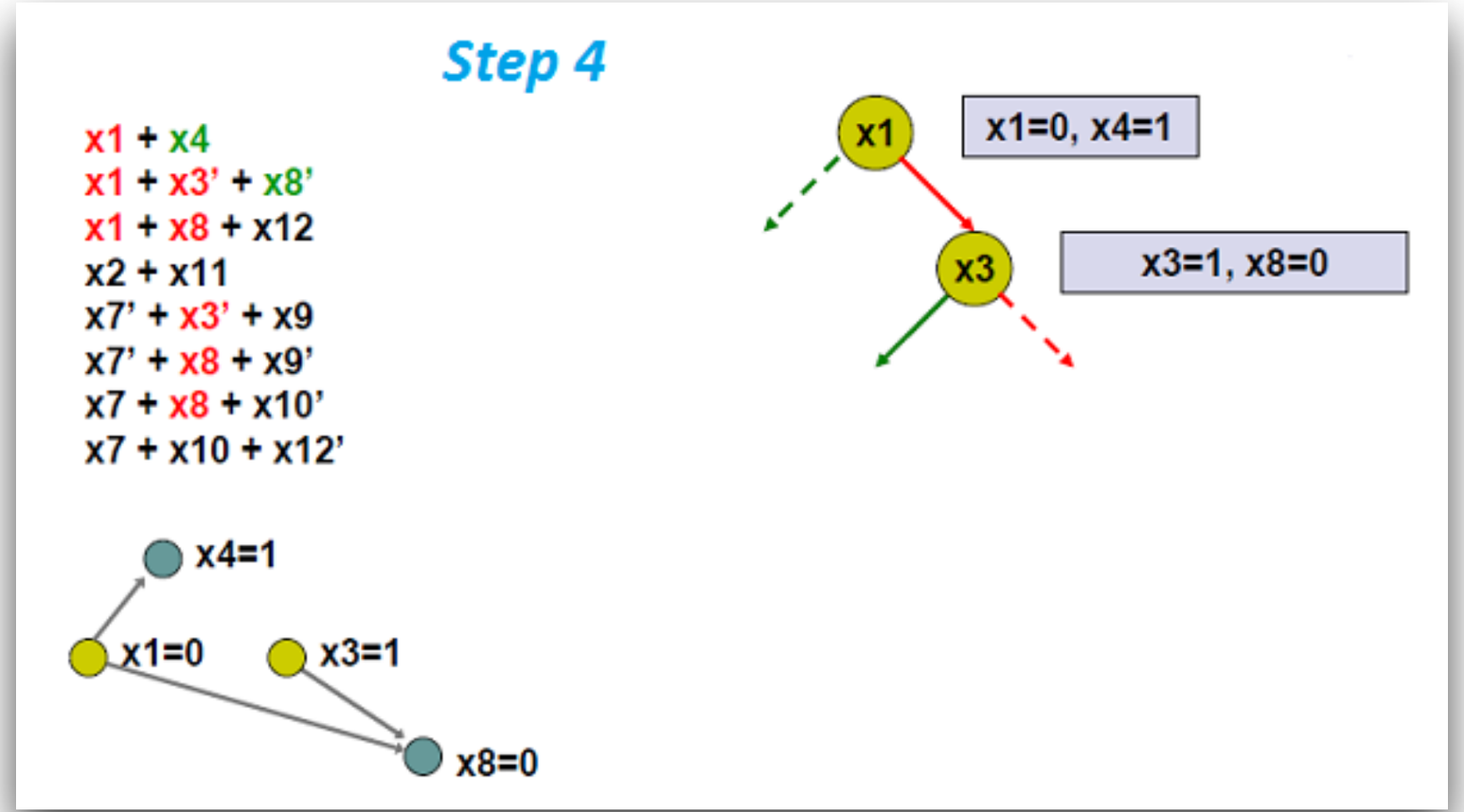
CDCL: Ejemplo



CDCL: Ejemplo



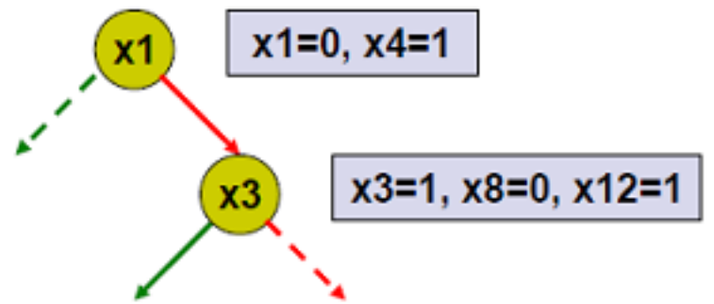
CDCL: Ejemplo

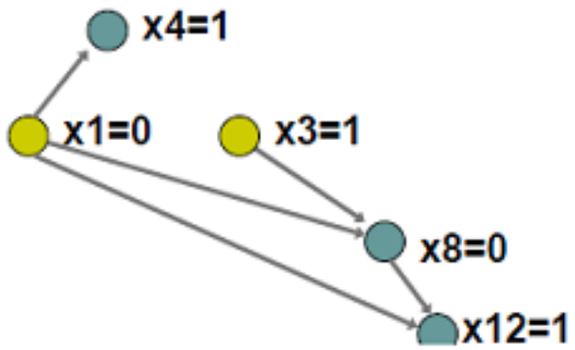


CDCL: Ejemplo

Step 5

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

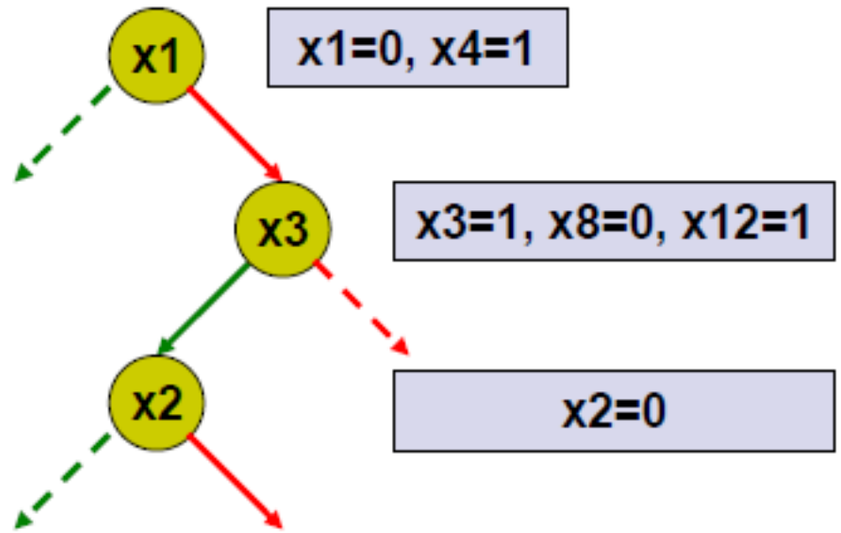


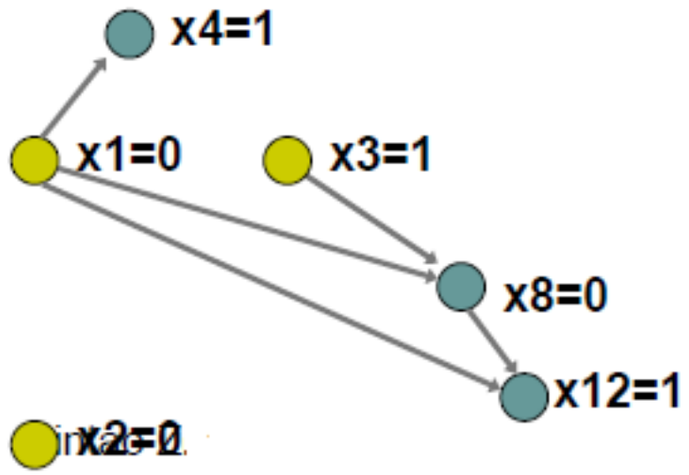


CDCL: Ejemplo

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

Step 6

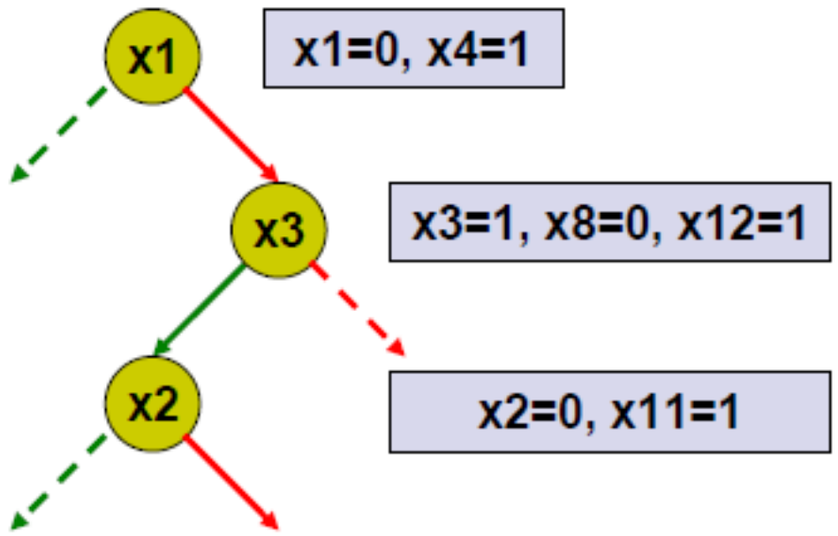


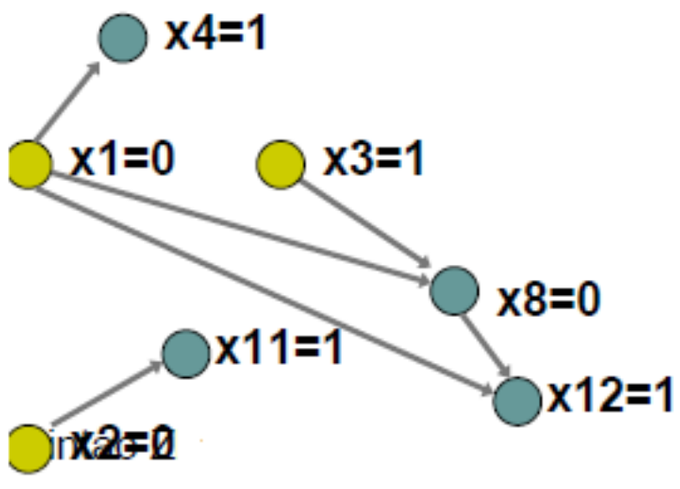


CDCL: Ejemplo

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

Step 7

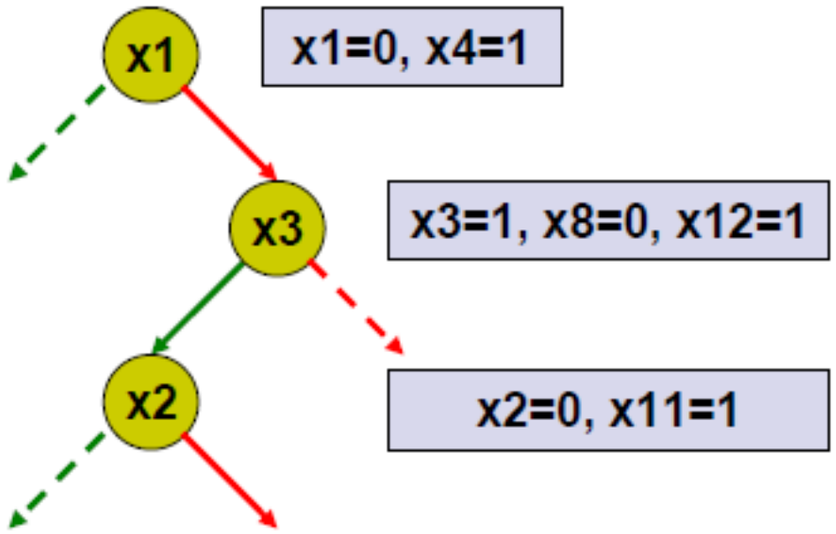


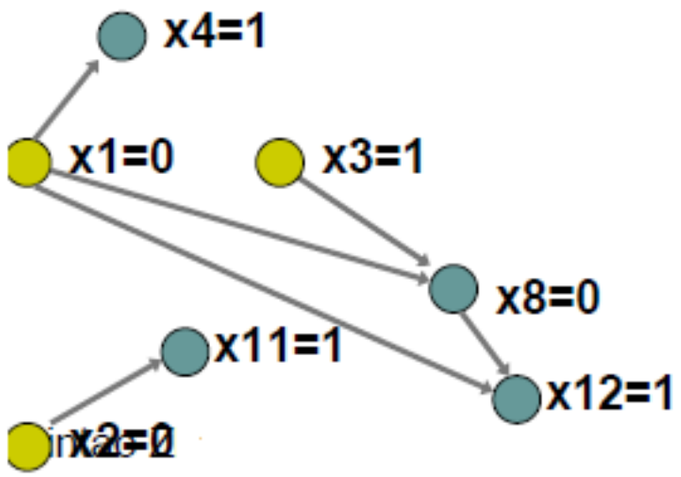


CDCL: Ejemplo

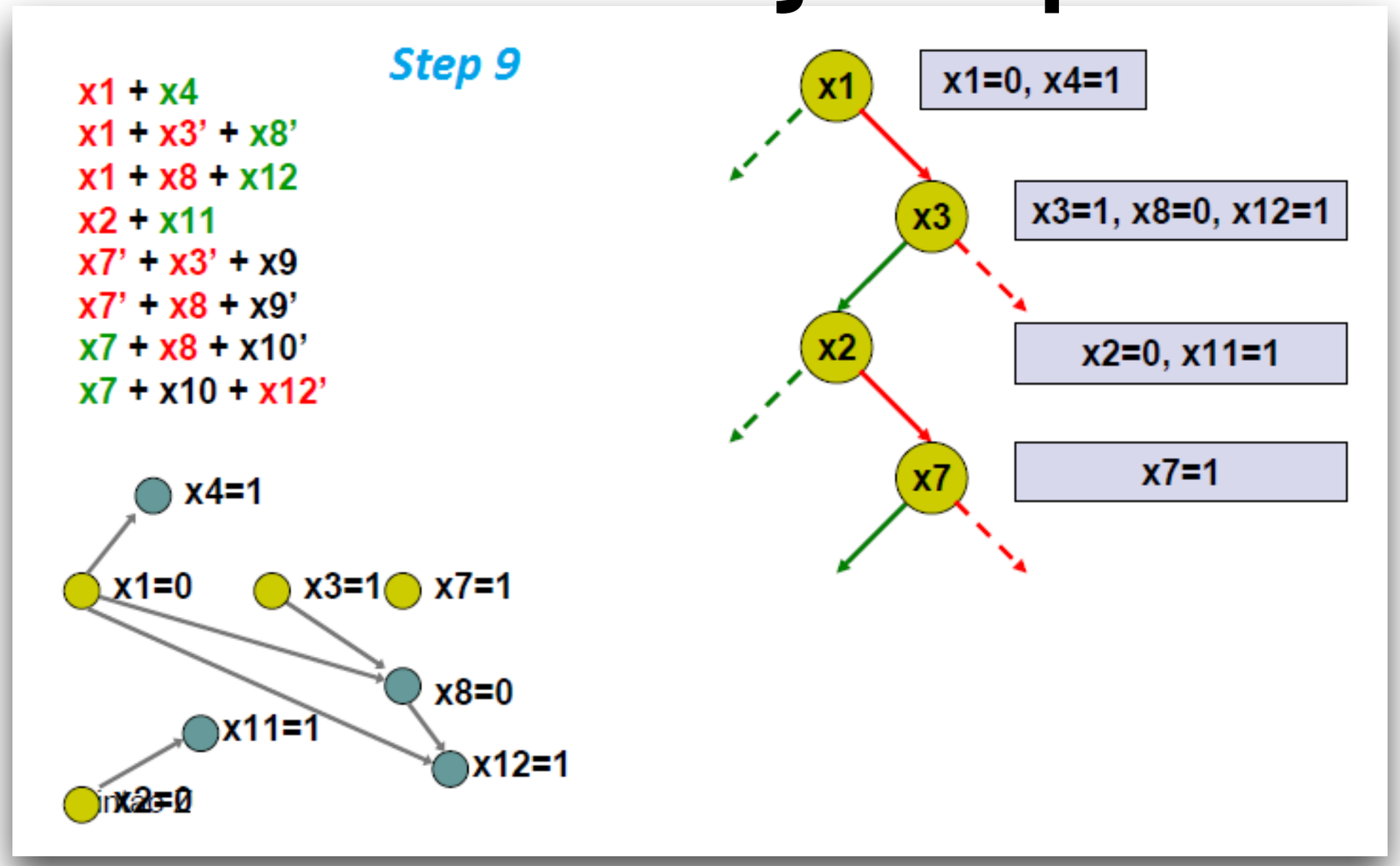
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

Step 8





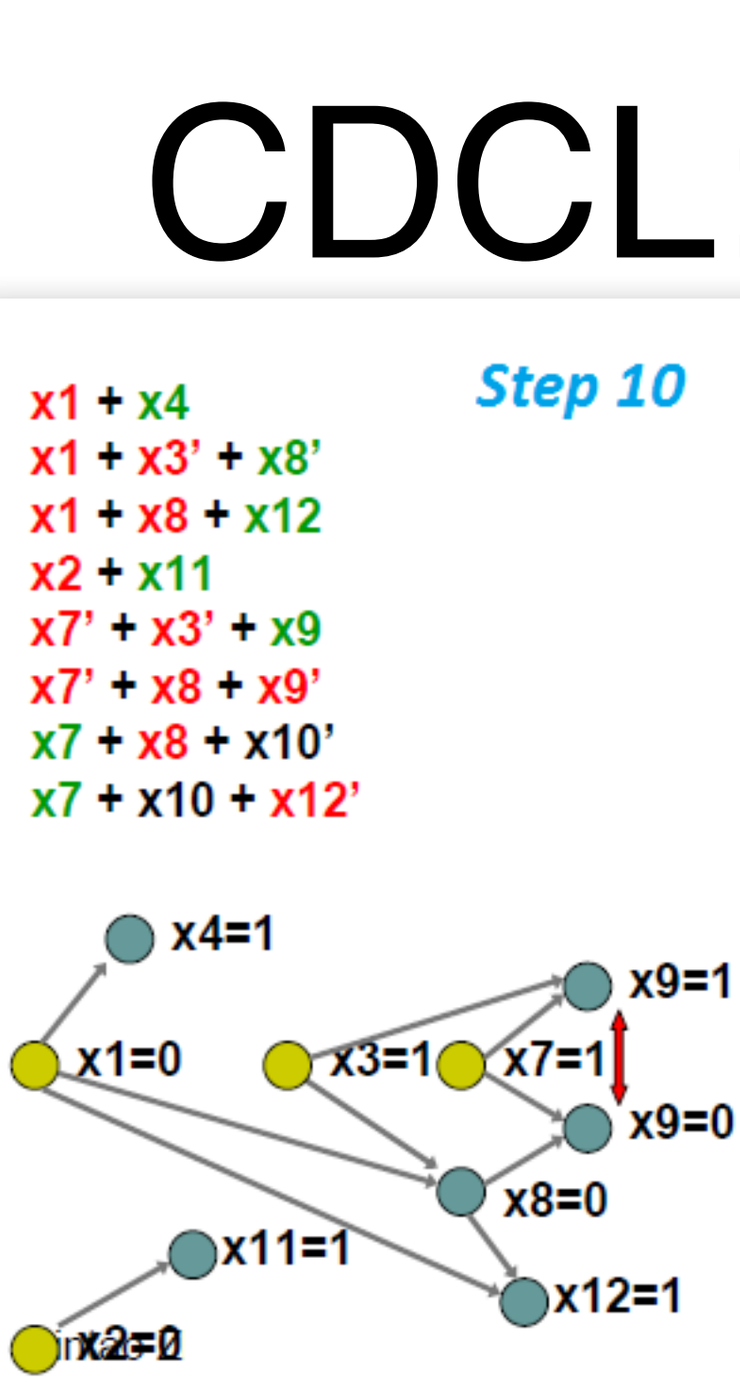
CDCL: Ejemplo

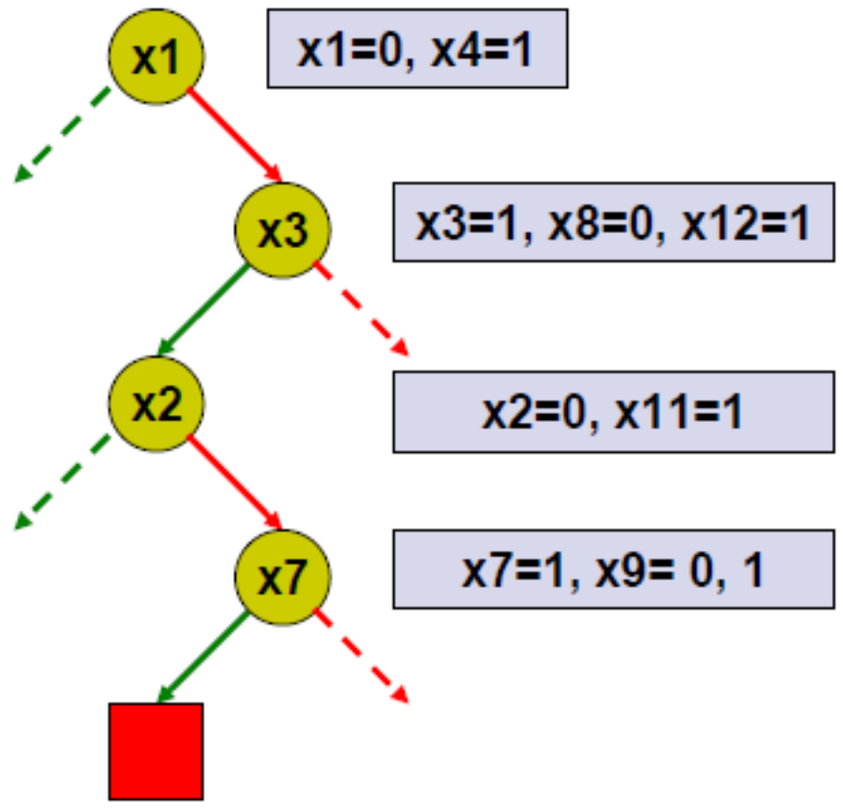


CDCL: Ejemplo

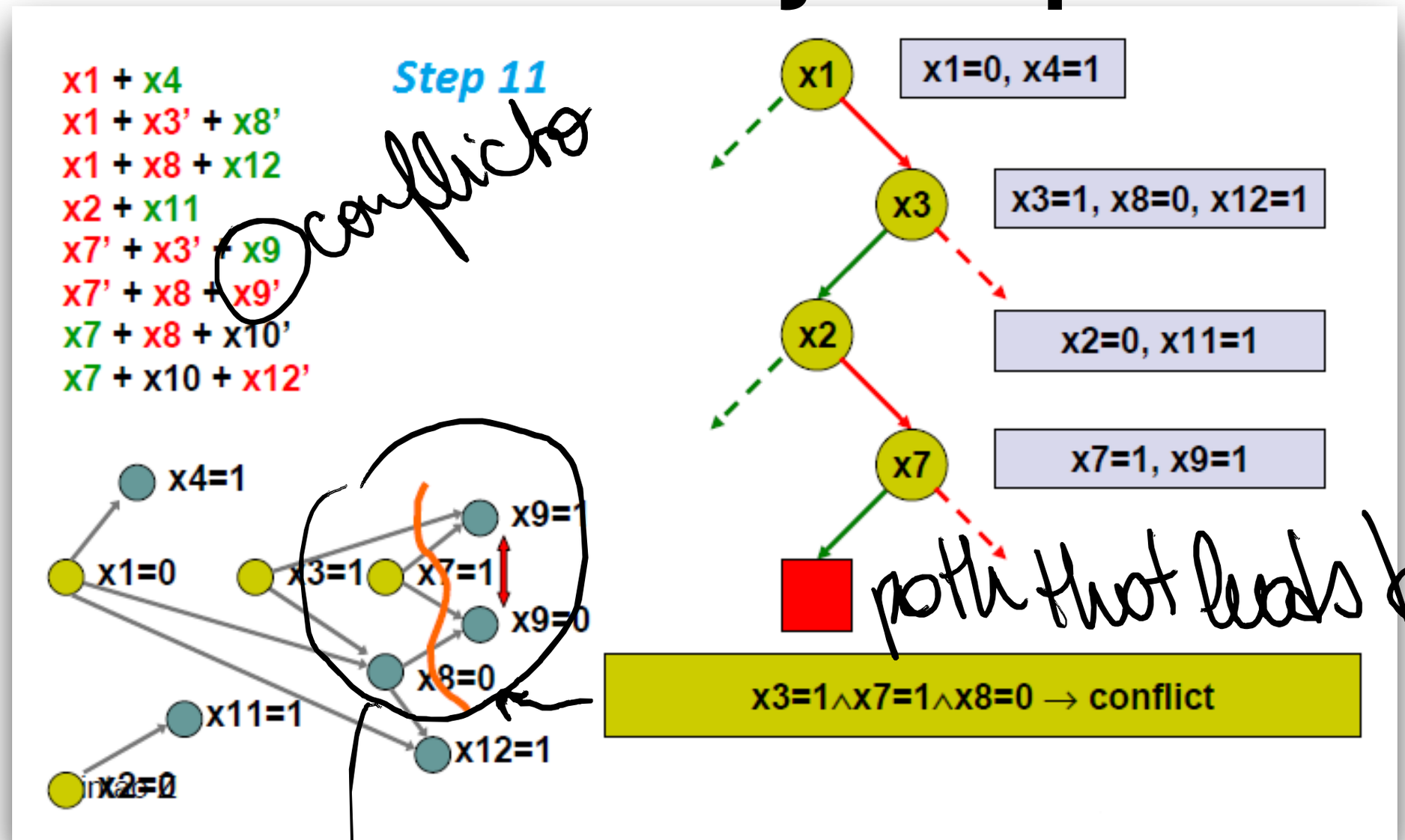
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

Step 10





CDCL: Ejemplo



CDCL: Ejemplo

If a implies b, then b' implies a'

Step 12

$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow$ conflict

Not conflict $\rightarrow (x3=1 \wedge x7=1 \wedge x8=0)'$

true $\rightarrow (x3=1 \wedge x7=1 \wedge x8=0)'$

$(x3=1 \wedge x7=1 \wedge x8=0)'$

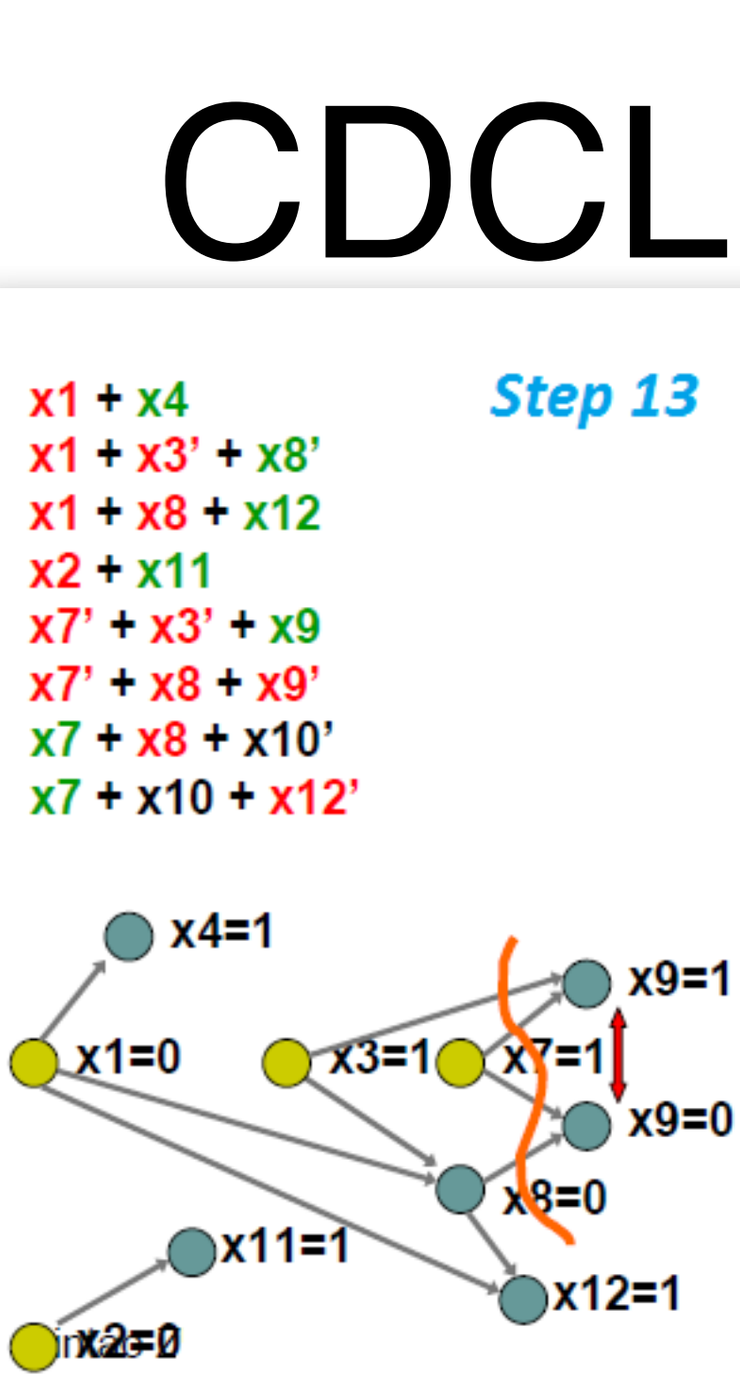
$(x3' + x7' + x8)$

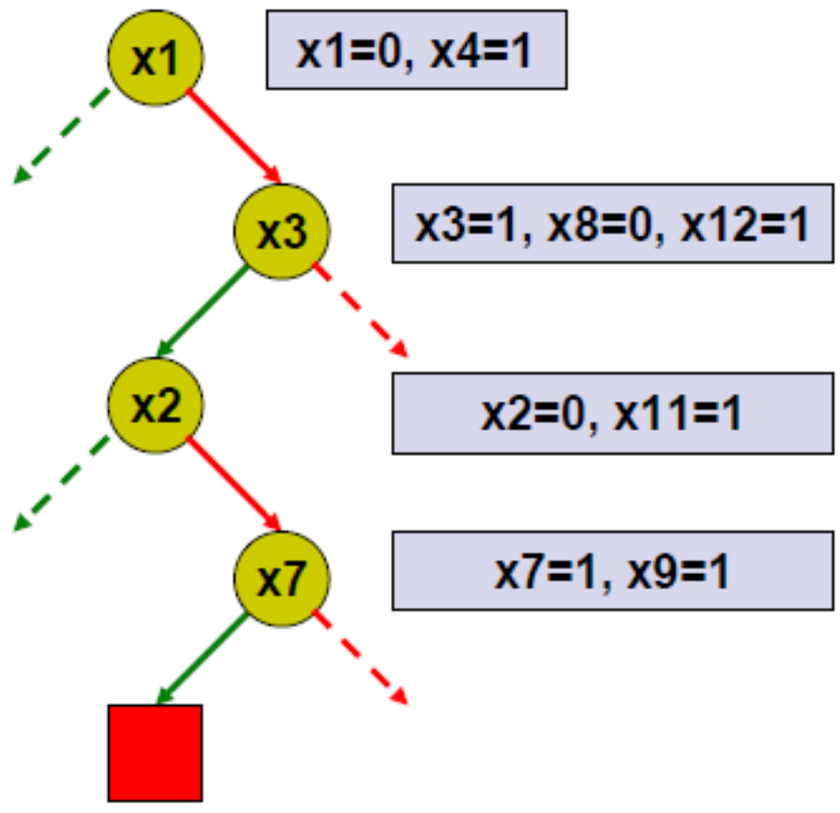
new created clause to prevent conflict from being reached

CDCL: Ejemplo

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

Step 13





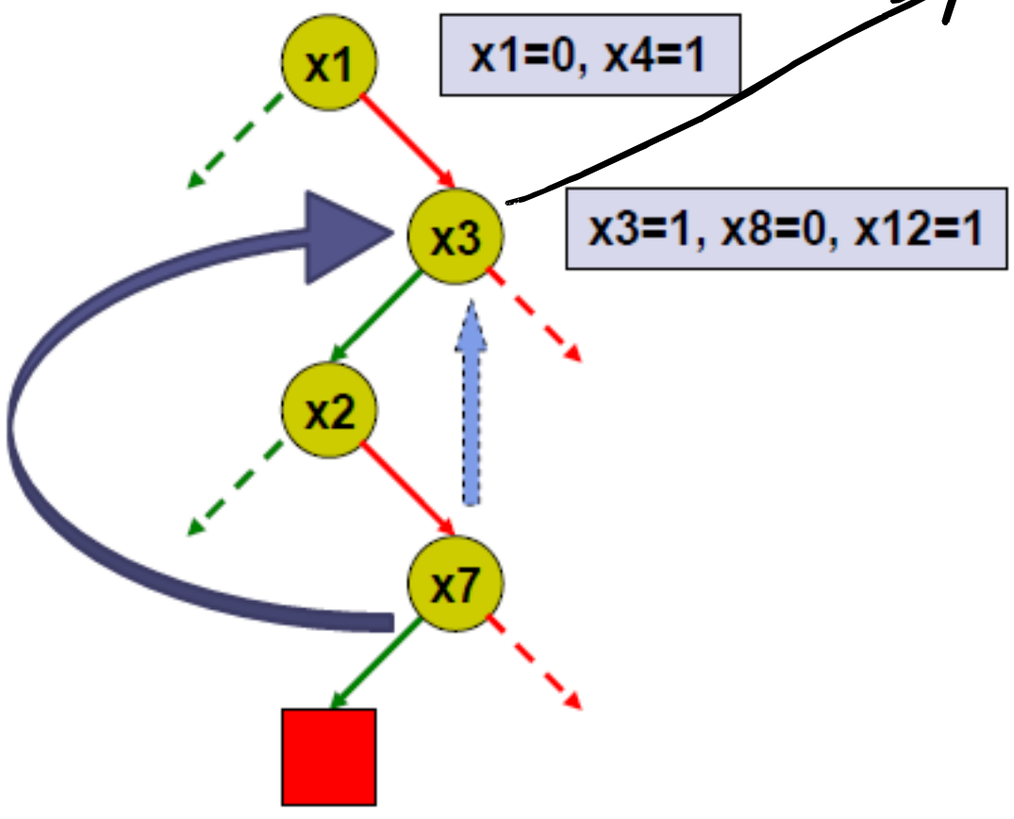
x3=1∧x7=1∧x8=0 → conflict

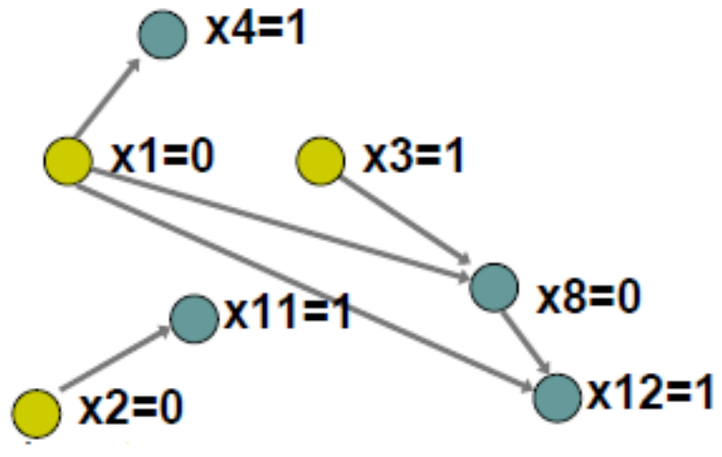
Add conflict clause: x3'+x7'+x8

CDCL: Ejemplo

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'
x3' + x8 + x7'

Step 14





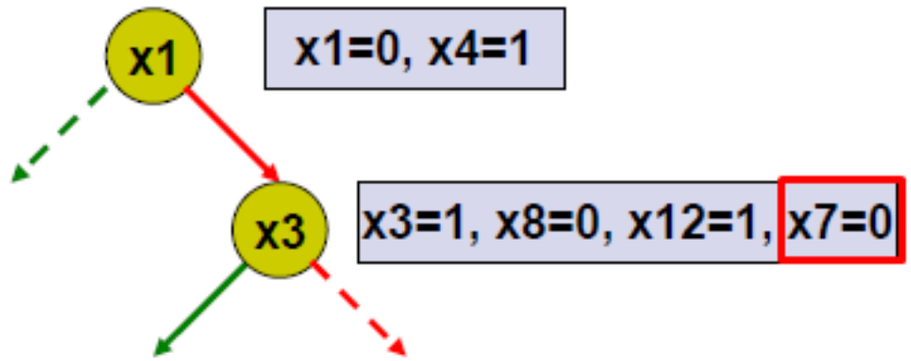
Backtrack to the decision level of x3=1:
x7 = 0

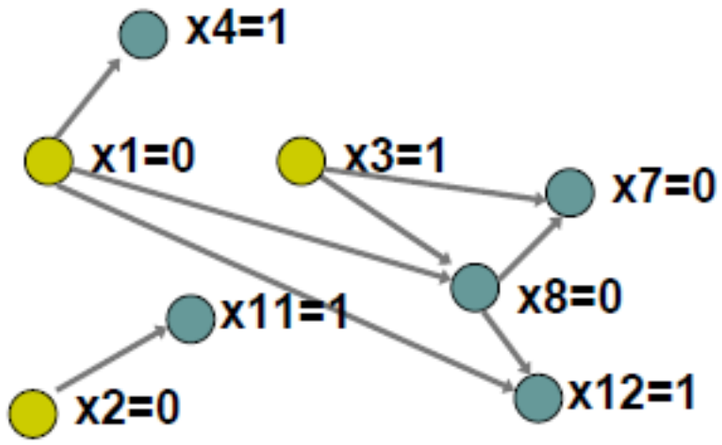
backtrack to earliest of direct decisions

CDCL: Ejemplo

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'
x3' + x8 + x7'

Step 15





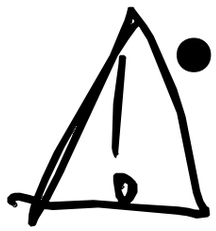
SAT-Solvers

- SAT4J (https://www.sat4j.org)
- MiniSAT (http://minisat.se)
- Lingeling (http://fmv.jku.at/lingeling/)
- MaxSAT (http://sat.inesc-id.pt/open-wbo/)

SMT-Solvers

- El SAT-Solver únicamente resuelve fórmulas proposicionales
- Un SMT-Solver permite resolver fórmulas
  - Con cuantificadores (para todo, existe)
  - Con “teorías” (ej: enteros, arreglos, bitvectors, números reales, strings, gramáticas, etc)

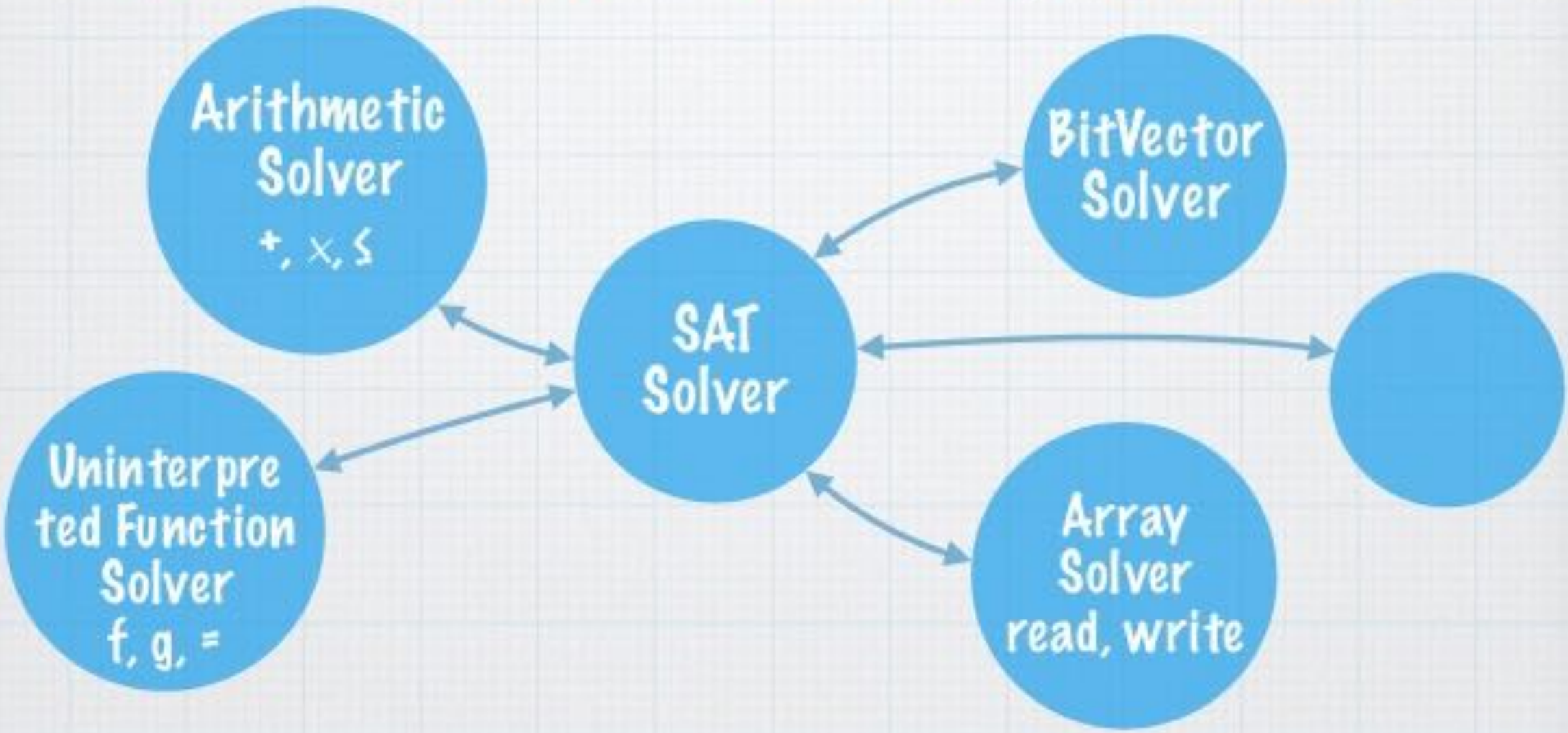
SMT-Solvers



- Pero decidir si una fórmula FOL es sat/unsat **NO** es decidible

- Input: Una fórmula FOL con teorías (arrays, enteros, números reales, funciones no interpretadas, etc.)
- Output: SAT/UNSAT/UNKNOWN (timeout)
- Usa DPLL + solventes específicos para otros problemas

SMT Solver Impl.
SAT Solver + Theory solvers



* SAT solver is responsible for Boolean reasoning
* Theory solvers are responsible for handling specific functions/relations etc.

↳ SMT solver asigna a variables proposicionales del SAT, predicados sobre el dominio de cuerdas.

SMT-Solvers

https://github.com/Z3Prover/z3

https://cvc4.github.io

https://yices.csl.sri.com

SMT-Lib

- Es un lenguaje stándard para escribir fórmulas para un SMT-Solver (lenguaje p/ cualquier SMT-Solver)
- Casi todos los SMT-Solvers poseen un lenguaje nativo + un módulo para traducir de SMT-Lib
- http://smtlib.cs.uiowa.edu
- Existe una especificación del comportamiento esperado de cada función
- Es un lenguaje para máquinas, no para humanos (simil XML)

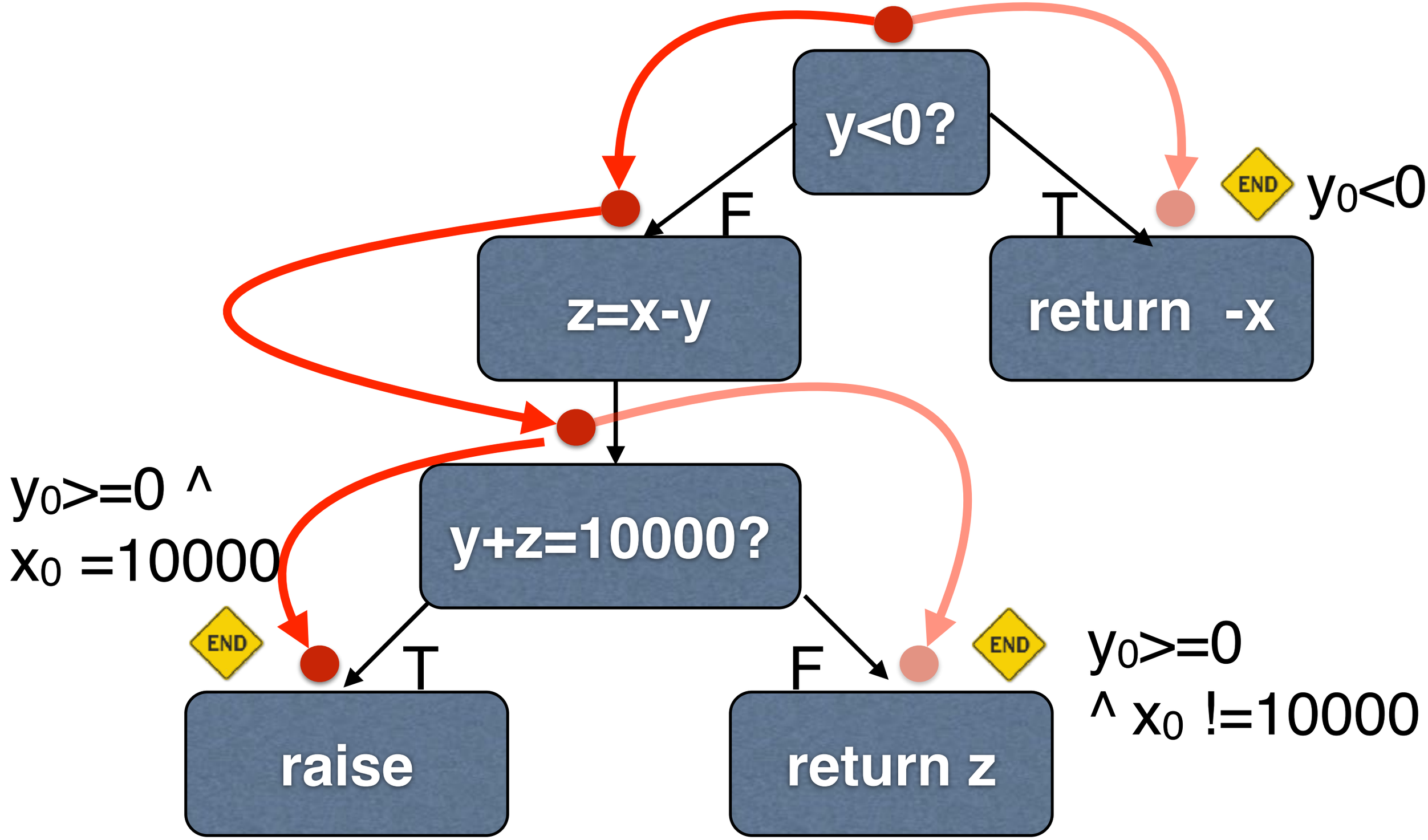
SMT-Lib: Ejemplo

```
(declare-const p Bool)
(assert (and p (not p)))
(check-sat) ; returns 'unsat'
(exit)
```

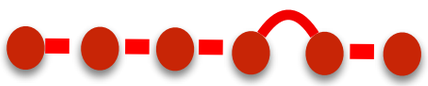
SMT-Lib: Ejemplo

```
(declare-const x Int)
(declare-const y Int)
(assert (= (- x y) (+ x (- y 1))))
(check-sat)
; unsat
(exit)
```

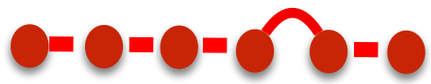
```
(declare-const x_0 (_ BitVec 32))
(declare-const x_1 (_ BitVec 32))
(declare-const x_2 (_ BitVec 32))
(declare-const y_0 (_ BitVec 32))
(declare-const y_1 (_ BitVec 32))
(assert (= x_1 (bvadd x_0 y_0)))
(assert (= y_1 (bvsub x_1 y_0)))
(assert (= x_2 (bvsub x_1 y_1)))

(assert (not
  (and (= x_2 y_0)
       (= y_1 x_0))))
(check-sat)
```


y<0?
F
T
END
y0<0
z=x-y
return -x
y+z=10000?
y0>=0 ^
x0 =10000
END
T
F
END
y0>=0
^ x0 !=10000
raise
return z

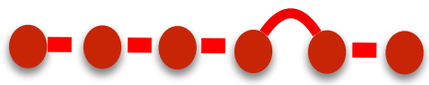
Constraint Solving



$y_0<0$

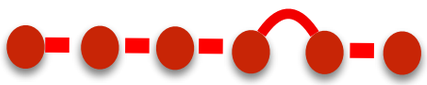


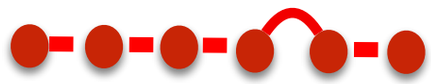
$y_0>=0 \wedge x_0 != 10000$



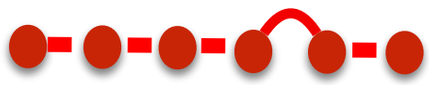
$y_0>=0 \wedge x_0 = 10000$

Constraint Solving



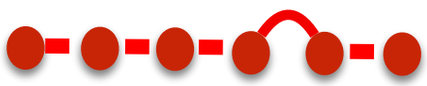


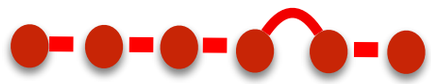
$y_0 >= 0 \wedge x_0 \neq 10000$



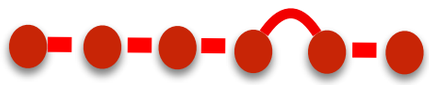
$y_0 >= 0 \wedge x_0 = 10000$

Constraint Solving





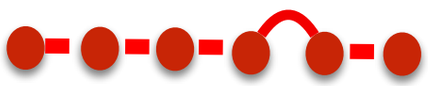
$y_0 >= 0 \wedge x_0 != 10000$



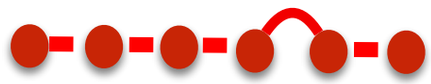
$y_0 >= 0 \wedge x_0 = 10000$

$y_0 = -1$

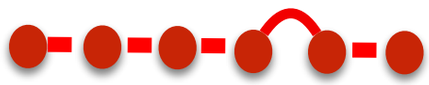
Constraint Solving



y_0=-1

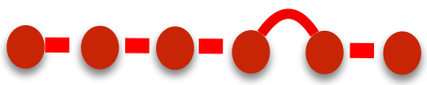


y_0>=0 ^ x_0 !=10000

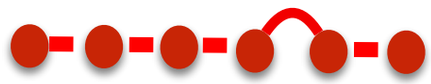


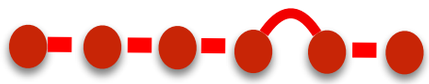
y_0>=0 ^ x_0 =10000

Constraint Solving



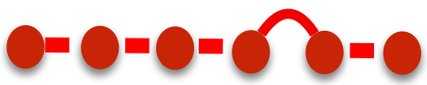
y_0=-1



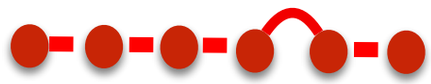


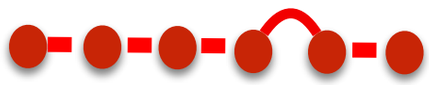
y_0>=0 ^ x_0 =10000

Constraint Solving



y_0=-1

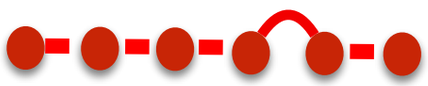




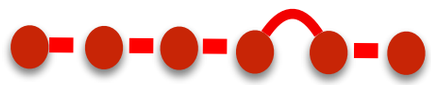
y_0>=0 ^ x_0 =10000

y_0=0, x_0 =0

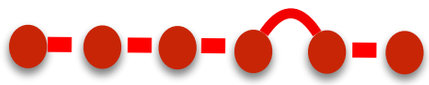
Constraint Solving



y_0=-1

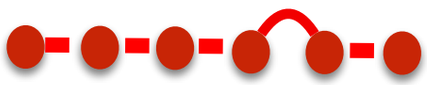


y_0=0, x_0 =0

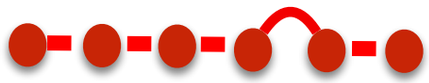


y_0>=0 ^ x_0 =10000

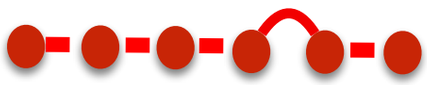
Constraint Solving



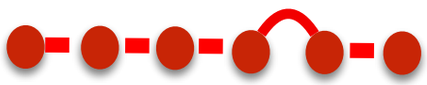
y_0=-1



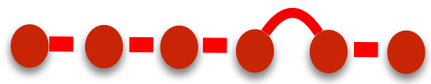
y_0=0, x_0 =0



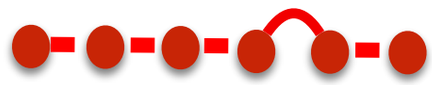
Constraint Solving



y_0=-1

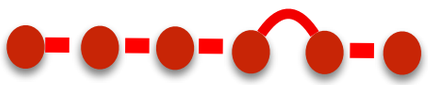


y_0=0, x_0 =0

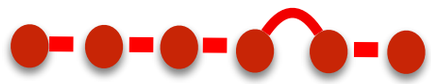


y_0=0, x_0 =10000

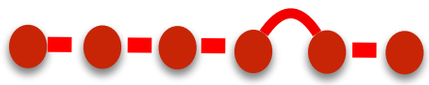
Constraint Solving



y_0=-1



y_0=0, x_0 =0



y_0=0, x_0 =10000

testme(0,-1)
testme(0,0)
testme(10000,0)

] generated 3 testcases from solver from SMT-solver

```
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
```

pseudocódigo pl generadores con

Ejecución Simbólica

```
execution_paths = compute_paths(cfg, limit).iterator()
Para cfg_path En execution_paths
  Si budget es no vacío
    path_condition = exec_symbolic(cfg_path)

    solution = solve(path_condition)

    Si solution is SAT:
      new_test = create_test(entry_f, solution)
      Agregar new_test a testSuite

retornar testSuite
```

```
@Test
public void testIsOdd0(int ) {
  boolean isOdd = isOdd(1);
  assertTrue(isOdd);
}
@Test
public void testIsOdd1() {
  boolean isOdd = isOdd(3);
  assertTrue(isOdd);
}
@Test
public void testIsOdd2() {
  boolean isOdd = isOdd(5);
  assertTrue(isOdd);
}
@Test
public void testIsOdd3() {
  boolean isOdd = isOdd(15);
  assertTrue(isOdd);
}
```

```
@Test
public void testIsOdd0(int ) {
  boolean isOdd = isOdd(1);
  assertTrue(isOdd);
}
@Test
public void testIsOdd1() {
  boolean isOdd = isOdd(3);
  assertTrue(isOdd);
}
@Test
public void testIsOdd2() {
  boolean isOdd = isOdd(5);
  assertTrue(isOdd);
}
@Test
public void testIsOdd3() {
  boolean isOdd = isOdd(15);
  assertTrue(isOdd);
}
```

Test Scenario

Test Data

Test Assertions

Parameterized Unit Tests (PUTs)

```
@ParameterizedTest
@ValueSource(ints = {1,3,5,15})
public void testIsOdd(int x) {
  boolean isOdd = isOdd(x);
  assertTrue(isOdd);
}
```

1 test code multiple data

searched via symbolic execution & SMT test generation constraint solver

SYMBOLIC EXEC → allows to create paths that are used by constraint solver to assign test data values.

Parameterized Unit Tests (PUTs)

```
@ParameterizedTest
@ValueSource(ints = {1,3,5,15})
public void testIsOdd(int x) {
  boolean isOdd = isOdd(x);
  assertTrue(isOdd);
}
```

- Los PUTs separan 2 dimensiones:

Parameterized Unit Tests (PUTs)

```
@ParameterizedTest
@ValueSource(ints = {1,3,5,15})
public void testIsOdd(int x) {
  boolean isOdd = isOdd(x);
  assertTrue(isOdd);
}
```

- Los PUTs separan 2 dimensiones:
  - Especificación del comportamiento esperado (assertions)

Parameterized Unit Tests (PUTs)

```
@ParameterizedTest
@ValueSource(ints = {1,3,5,15})
public void testIsOdd(int x) {
  boolean isOdd = isOdd(x);
  assertTrue(isOdd);
}
```

- Los PUTs separan 2 dimensiones:
  - Especificación del comportamiento esperado (assertions)
  - Selección de datos de test relevantes (coverage)

Parameterized Unit Testing

- Varios test frameworks soportan Parameterised Unit Tests (PUTs):
  - .NET: Soportado por MBUnit, NUnit
  - Java: Soportado a partir de JUnit 4.X
- Existen tools para generar test inputs para PUTs:
  - .NET: Microsoft Visual Studio
  - Java: AgitarOne (http://www.agitar.com)

Parameterized Unit Tests + Ejecución Simbólica

- Podemos pensar al PUT como el entry point de la generación de tests con ejecución simbólica
- Todos los argumentos tienen que ser soportados por el 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

```
@ParameterizedTest
public static void p_test2(int valueX, int valueY) {
  Adder adder = new Adder();
  int rv = adder.performAction(valueX, valueY);
  int expectedValue = valueX + valueY;
  assertEquals(expectedValue, rv);
}
```

Parameterized Unit Tests

```
@ParameterizedTest
public static void p_test2(int[] array) {
  assumeTrue(array!=null); // pre
  sort(array);
  assertTrue(isSorted(array)); // post
}
```

Parameterized Unit Tests

```
@ParameterizedTest
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));
}
```

PUTs: Inputs Integer

```
@ParameterizedTest
public static void p_test2(int len, int a1, int a2) {
   assumeTrue(len>1); // pre
   MiniStack s = new MiniStack(len);
   s.push(a1);
   s.push(a2);
   assertTrue(s.contains(a1);
   assertTrue(s.contains(a2));
   assertTrue(s.isNotEmpty());
}
```

determina QUÉ es lo que se puede saber del dominio con el constraint solver.

Constraint Language Expresiones Enteras

- Dado un programa que opera con enteros, ¿qué expresiones puedo llegar a tener?

```
public int add(int x, int y) {
   if (x==0) {
     Return y;
   } else {
      int temp = x+y;
      return temp;
   }
}
```

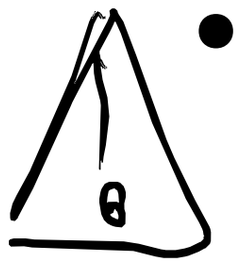
- Expresiones enteras: x+y, Condicionales: x==0

ejemplo del dominio del lenguaje.

Expresiones Enteras

- Expresiones Enteras:
  - Literales: 0,1,2,…
  - Variables Enteras: var0, var1, var
  - Operaciones: E+E, E-E, E*E, E/E, E%E
- Fórmulas:
  - E==E, E!=E, E>E,E>=E,E<E,E<=E

Expresiones Enteras



- Necesitamos un constraint solver que soporte todas las expresiones enteras (y sus fórmulas) para poder resolver constraints enteras

Constraint Language Expresiones Enteras

Var0>0
Var0<Var1
Var1<Var2
Var3==Var2

Var0: 10
Var1: 11
Var2: 12
Var3: 12

PUTs: Inputs String

```
@ParameterizedTest
public static void p_test3(String id, String
emailAddress) {
  assumeTrue(id!=null);
  assumeTrue(emailAddress!=null);
  assumeTrue(isValid(emailAddress));
  Customer s = new Customer(id);
  s.setEmailAddress(emailAddress);
  assertTrue(s.hasValidEmailAddress());
}
```

PUTs: Inputs String

- Dado un programa que opera con strings, ¿qué expresiones puedo llegar a tener?

```
public int checkId(String id) {
  If (id.startWith(“hello”)
      Return 1;
  } else if (id.endsWith(“world”) {
      Return 2;
  } else {
      String s = id.toLowerCase();
      Return s.equals(“hola mundo”);
  }
}
```

Constraint Language
Cadenas de Strings

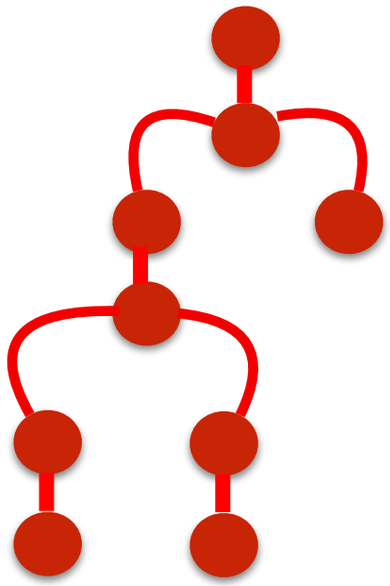
- Expresiones sobre Strings:
  - Valores literales: “”, “A”, “B”, …
  - Variables string: var0, var1, var
  - Expresiones: E+E, E.substring(int,int), etc.
- Constraints:
  - E.equals(E), !E.equals(E), E.contains(E), etc

Constraint Language
Cadenas de Strings

Var0.endsWith(“Mundo”)
Not equals(Substring(Var0,0,4),”Hola”)
EqualsIgnoreCase(Substring(Var0,0,4),”Hola”)

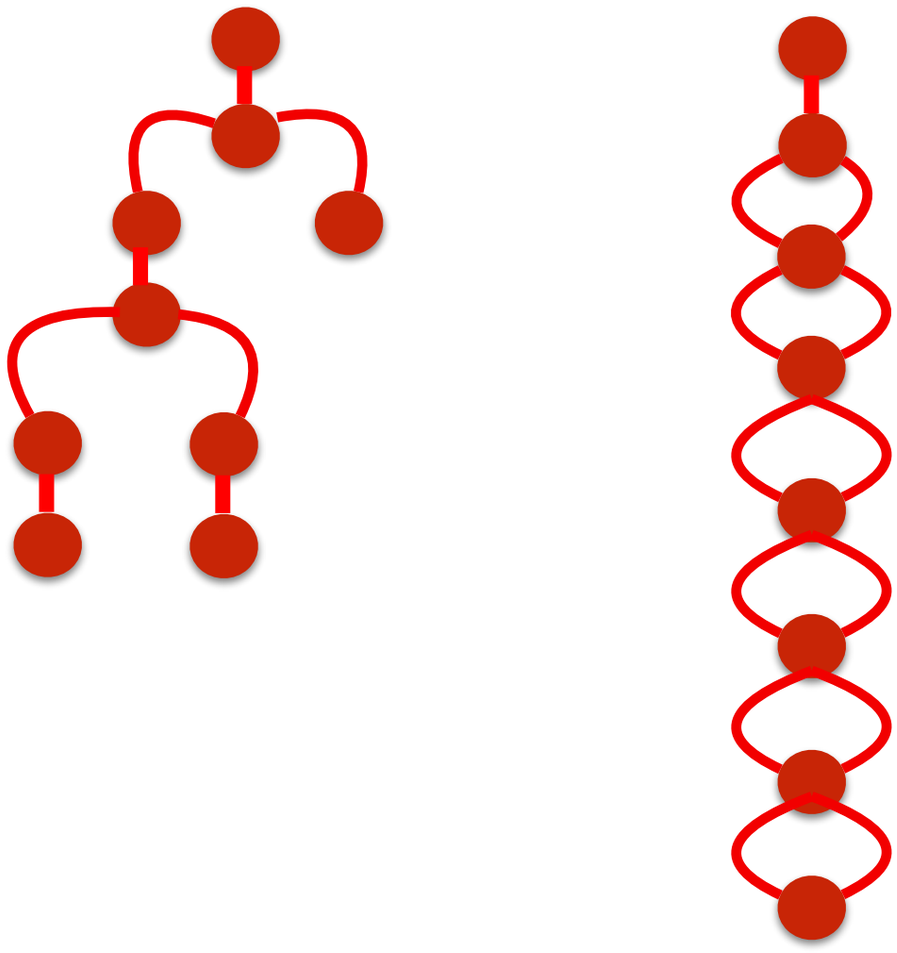
Var0: “holA Mundo”

Ejecución Simbólica: Limitaciones



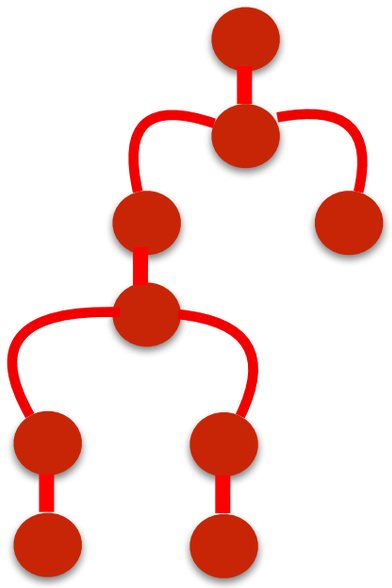
3 caminos

Ejecución Simbólica: Limitaciones

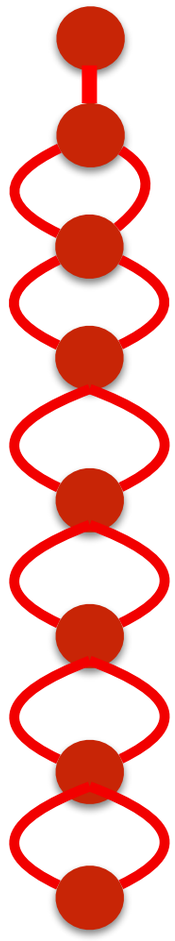


3 caminos

Ejecución Simbólica: Limitaciones

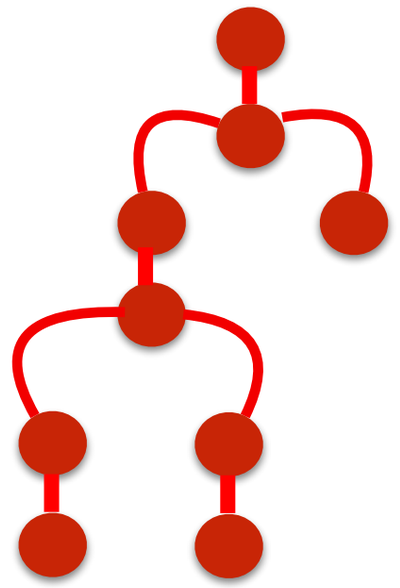


3 caminos

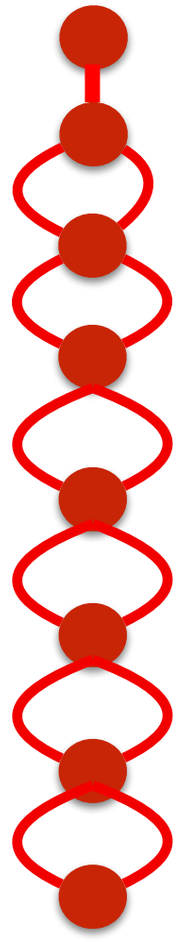


2^n caminos

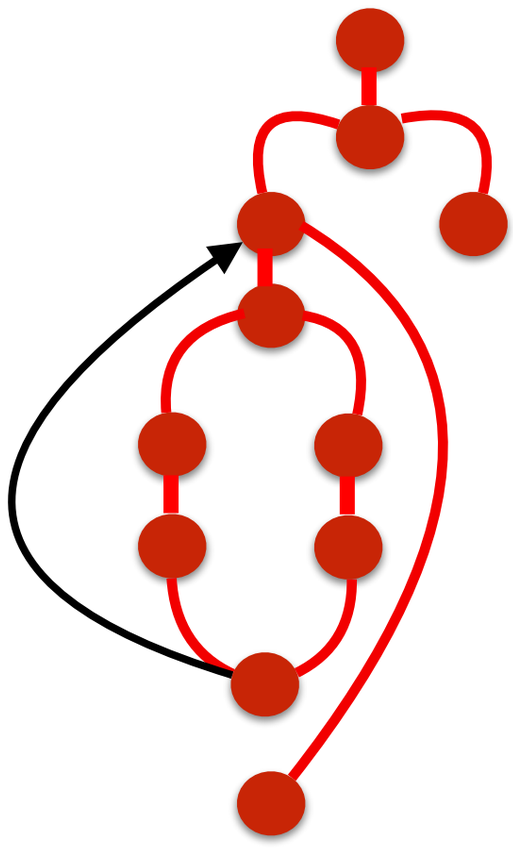
Ejecución Simbólica: Limitaciones



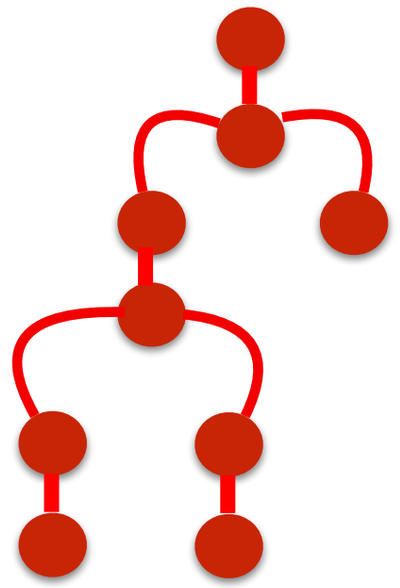
3 caminos



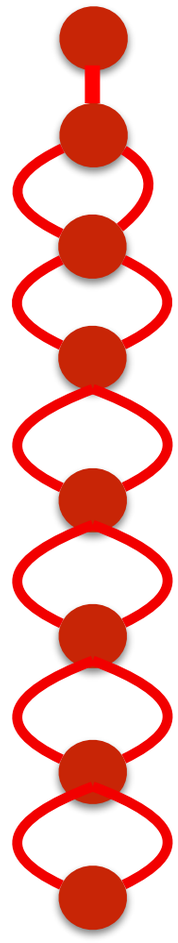
2^n caminos



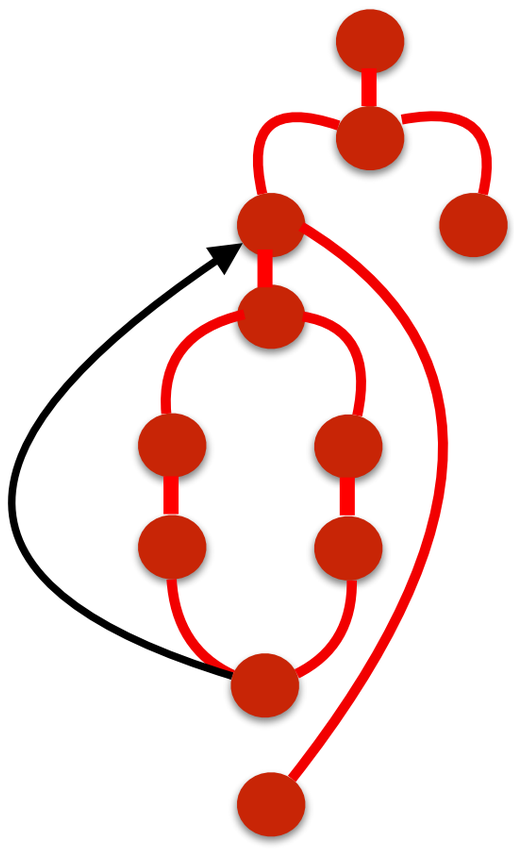
Ejecución Simbólica: Limitaciones : #cauuilon



3 caminos



2^n caminos



¡Infinitos caminos!

Ejecución Simbólica: Limitaciones

: complejidad de exploración

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

```
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: Limitaciones

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

```
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
```

Requiere aritmética No-lineal

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?

Caminos no satisfacibles

```
public User createUser(final String id) {
    If (id!=null && id.contains(“User”)) {
     …
    }
    doSomething();
    If (id!=null && id.contains(“User”)) {
      …
    }
    Return … //
}
```

- ¿Puede existir un camino de ejecución que entre al primer if y no entre al segundo if?

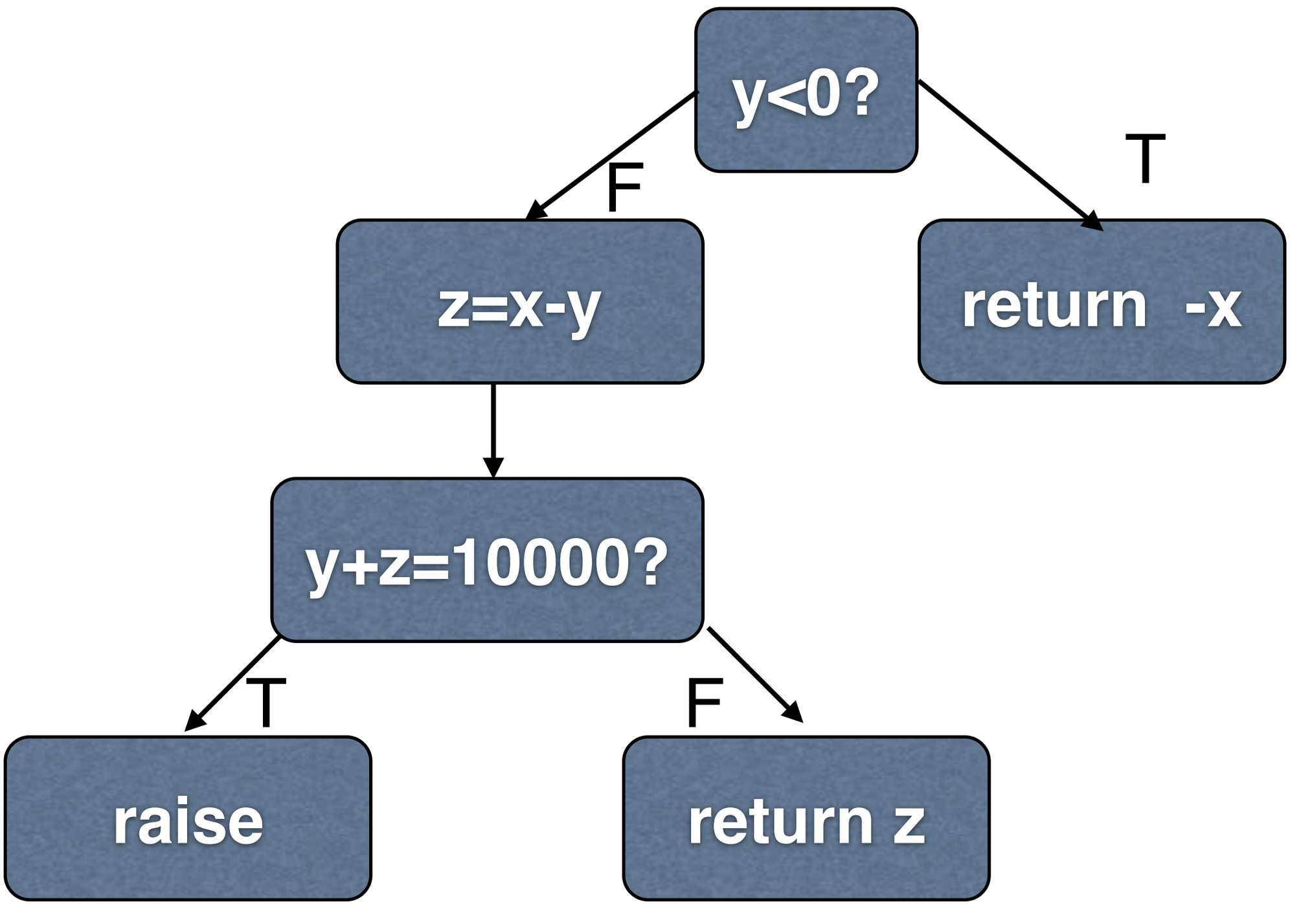
Información de Runtime

```
URL url = new URL(“http://www.something.com");
is = new InputStreamReader(url.openStream())
br = new BufferedReader(is);
line = br.readLine();
If line.equals(…) {
  // do something
} else {
  // do something else
}
```

Symbolic Exec → po over paths & pc values.

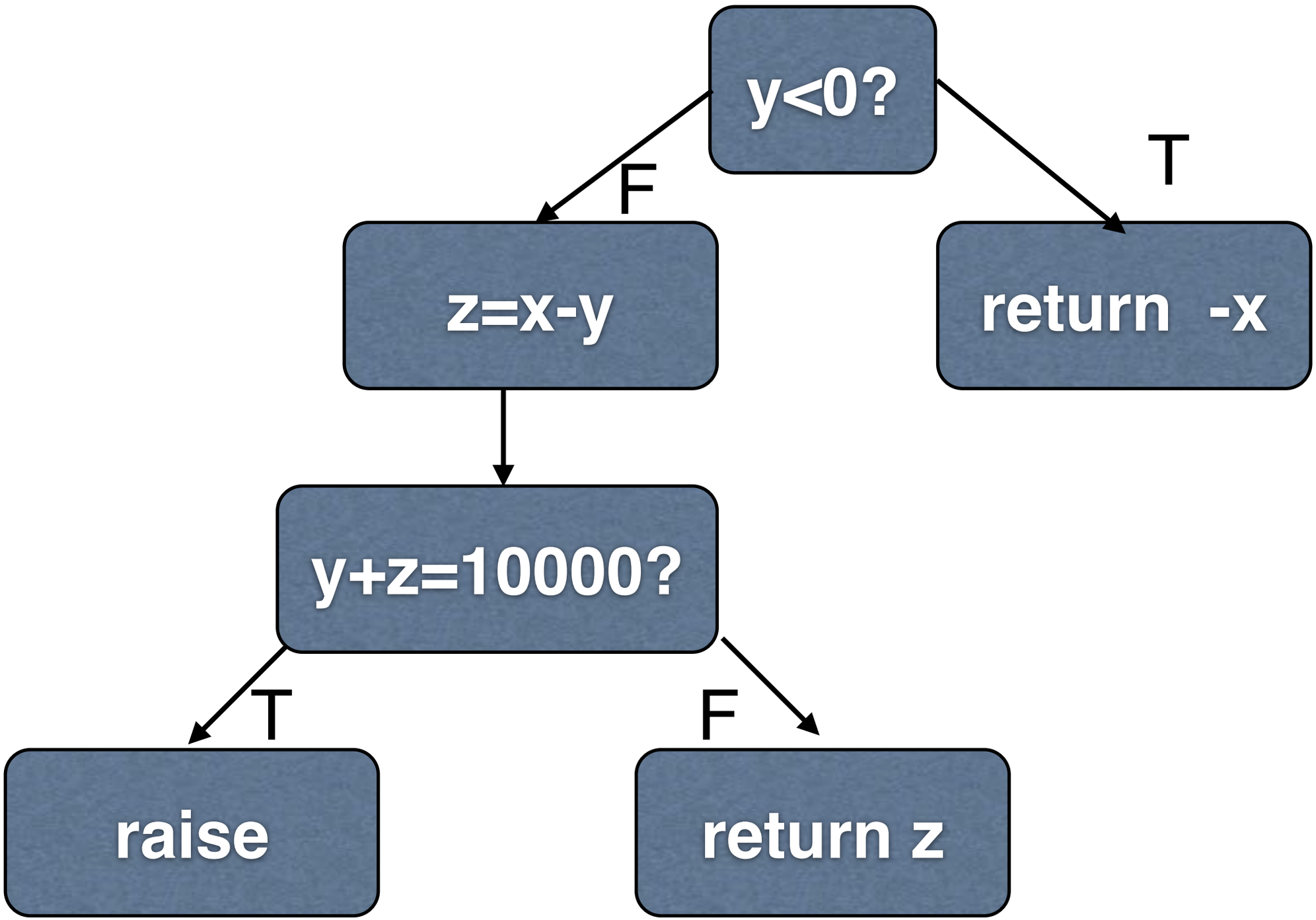
vs

Ejecución Simbólica Dinámica

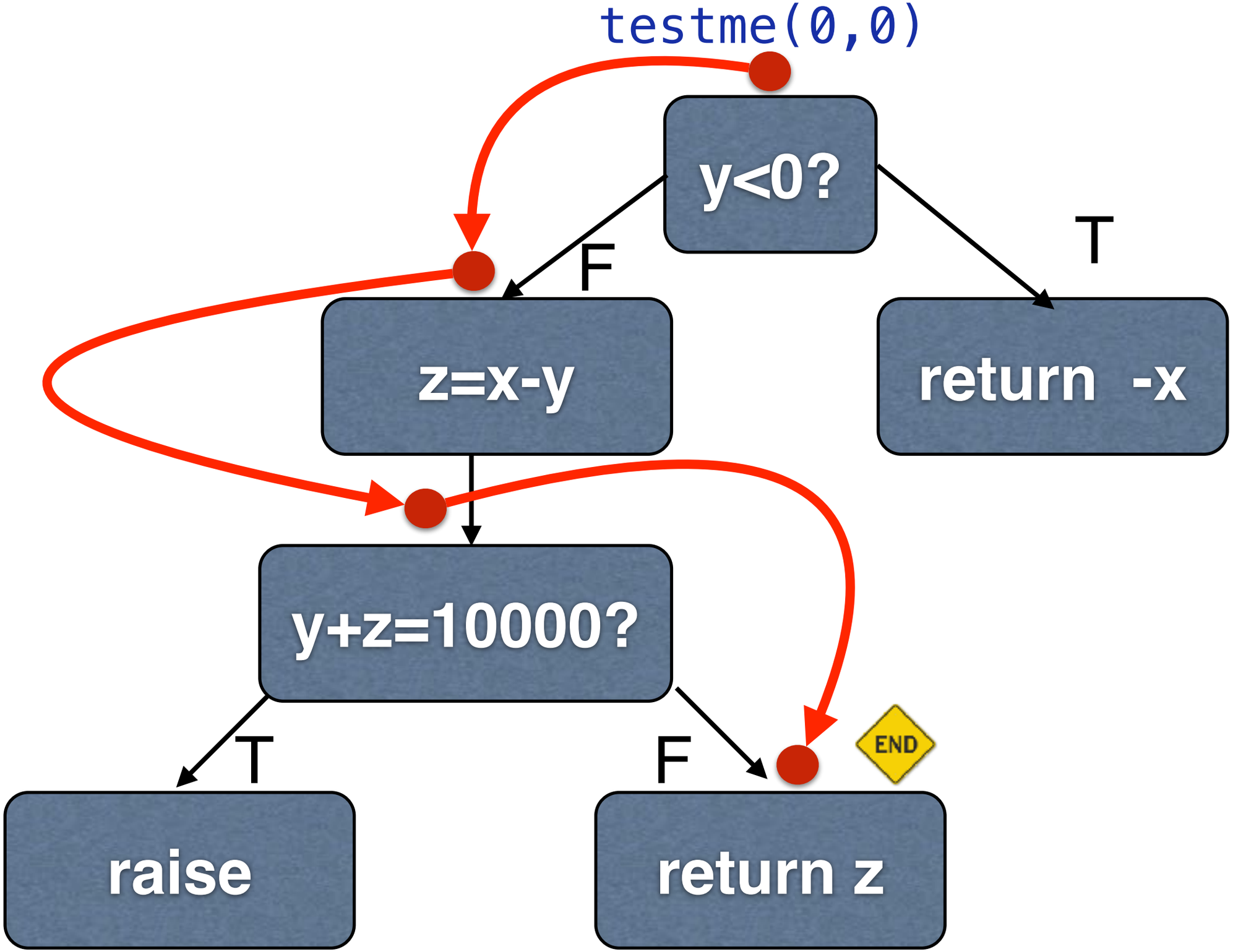


Ejecución Simbólica Dinámica

```
testme(0,0)
```

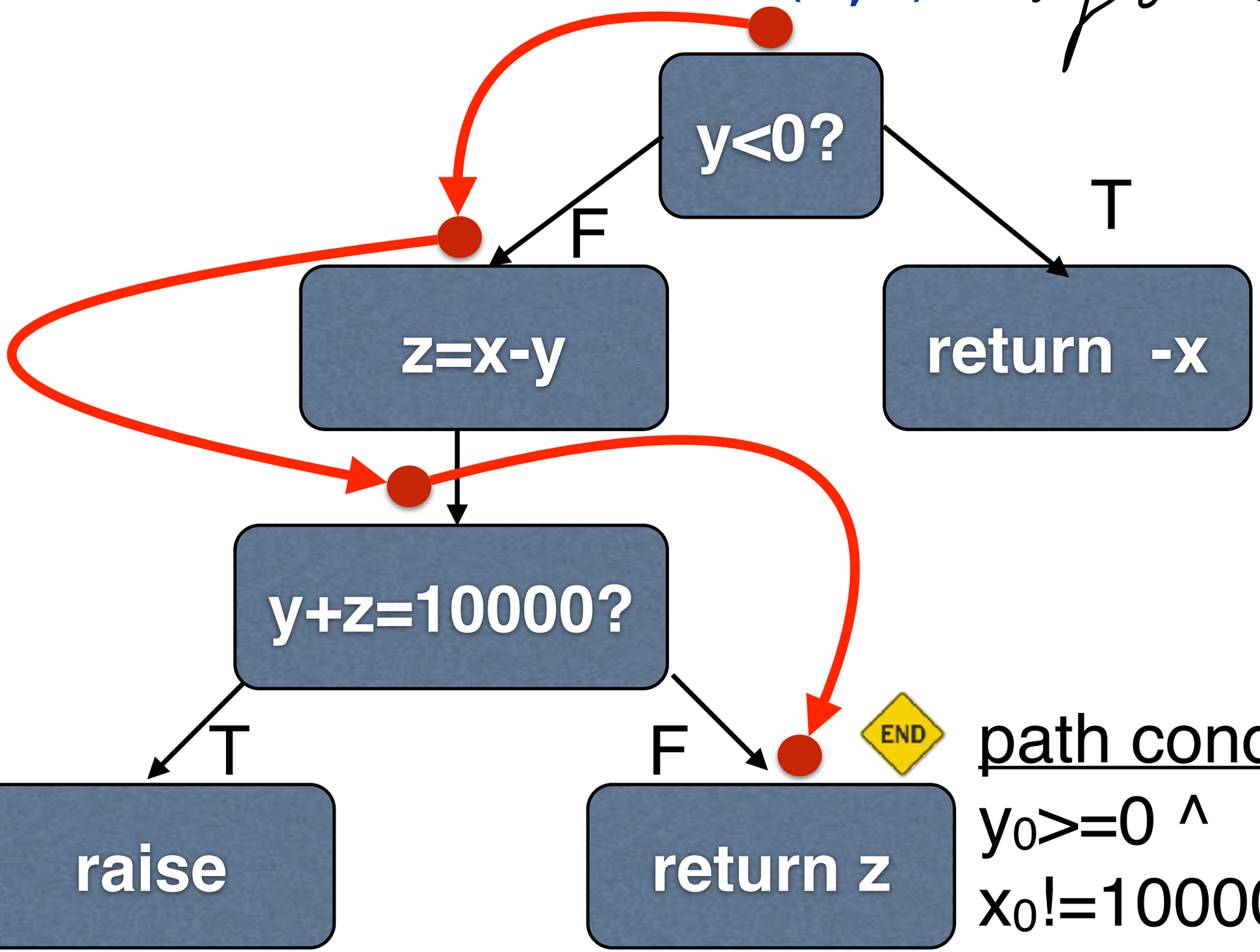


Ejecución Simbólica Dinámica



Ejecución Simbólica Dinámica

testme(0,0) → primer test

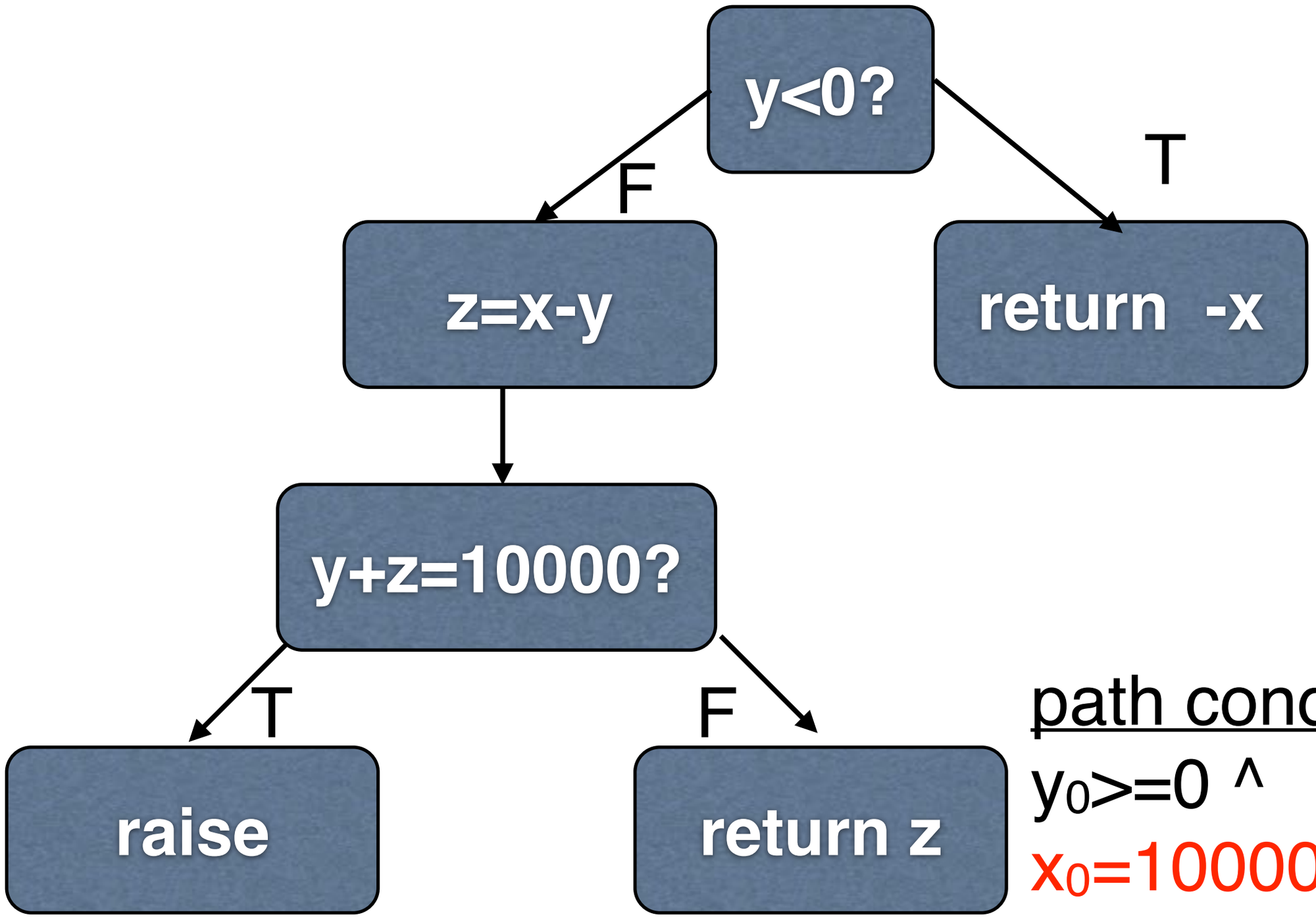


path condition:

$y_0 >= 0$ ^

$x_0 != 10000$

Ejecución Simbólica Dinámica

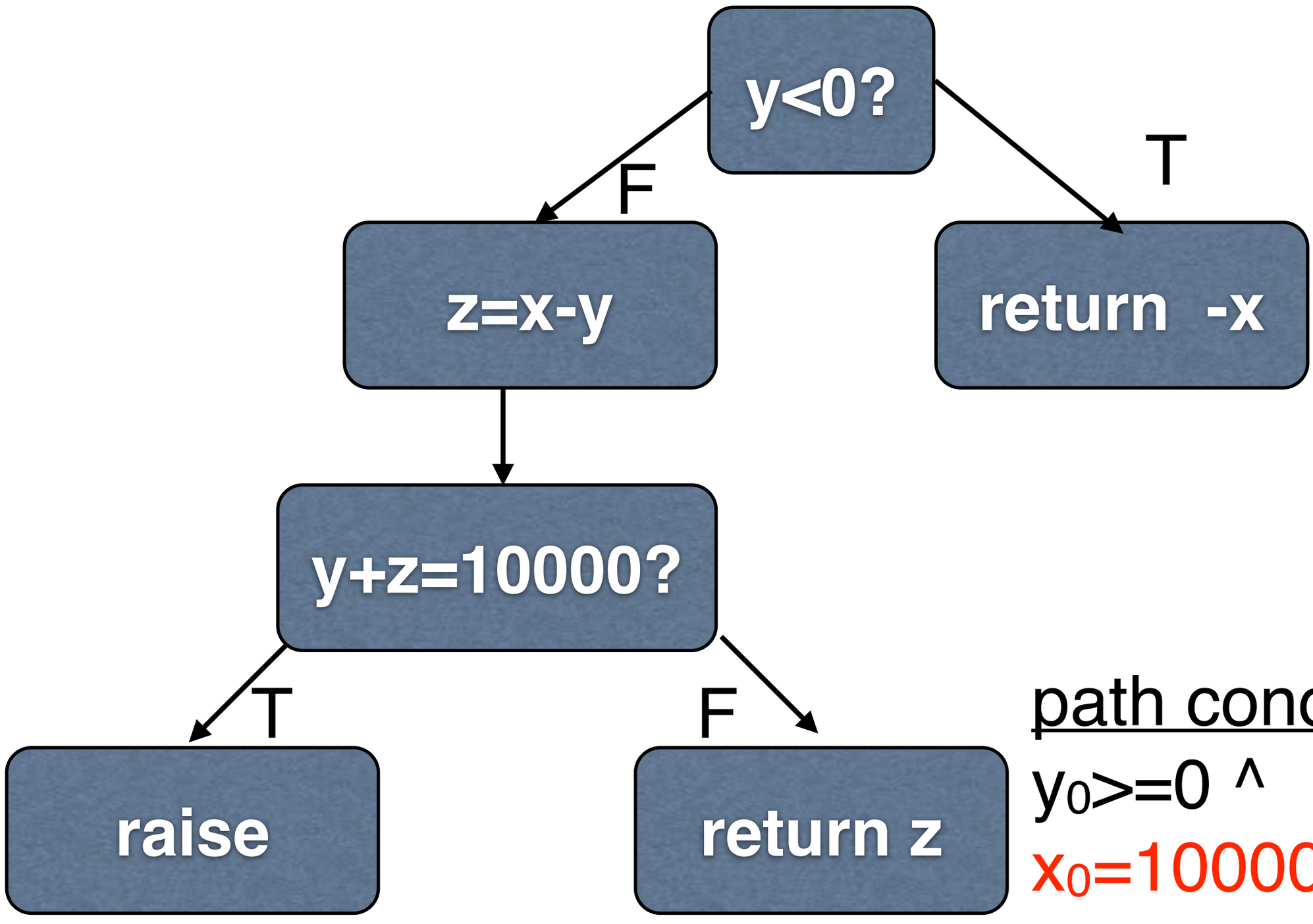


path condition:

$y_0 >= 0$ ^

$x_0 = 10000$

Ejecución Simbólica Dinámica



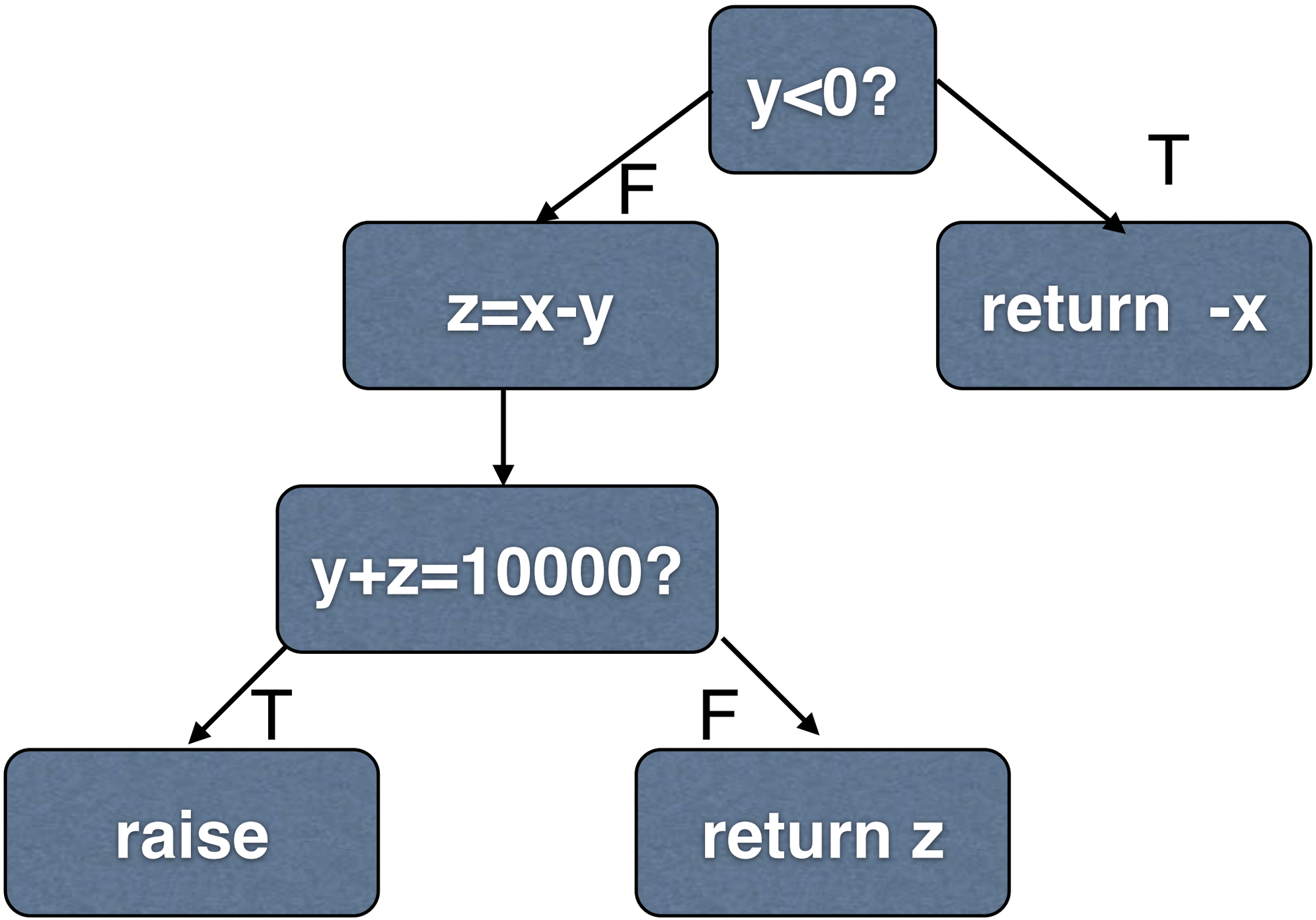
path condition:

$y_0 >= 0$ ^

$x_0 = 10000$

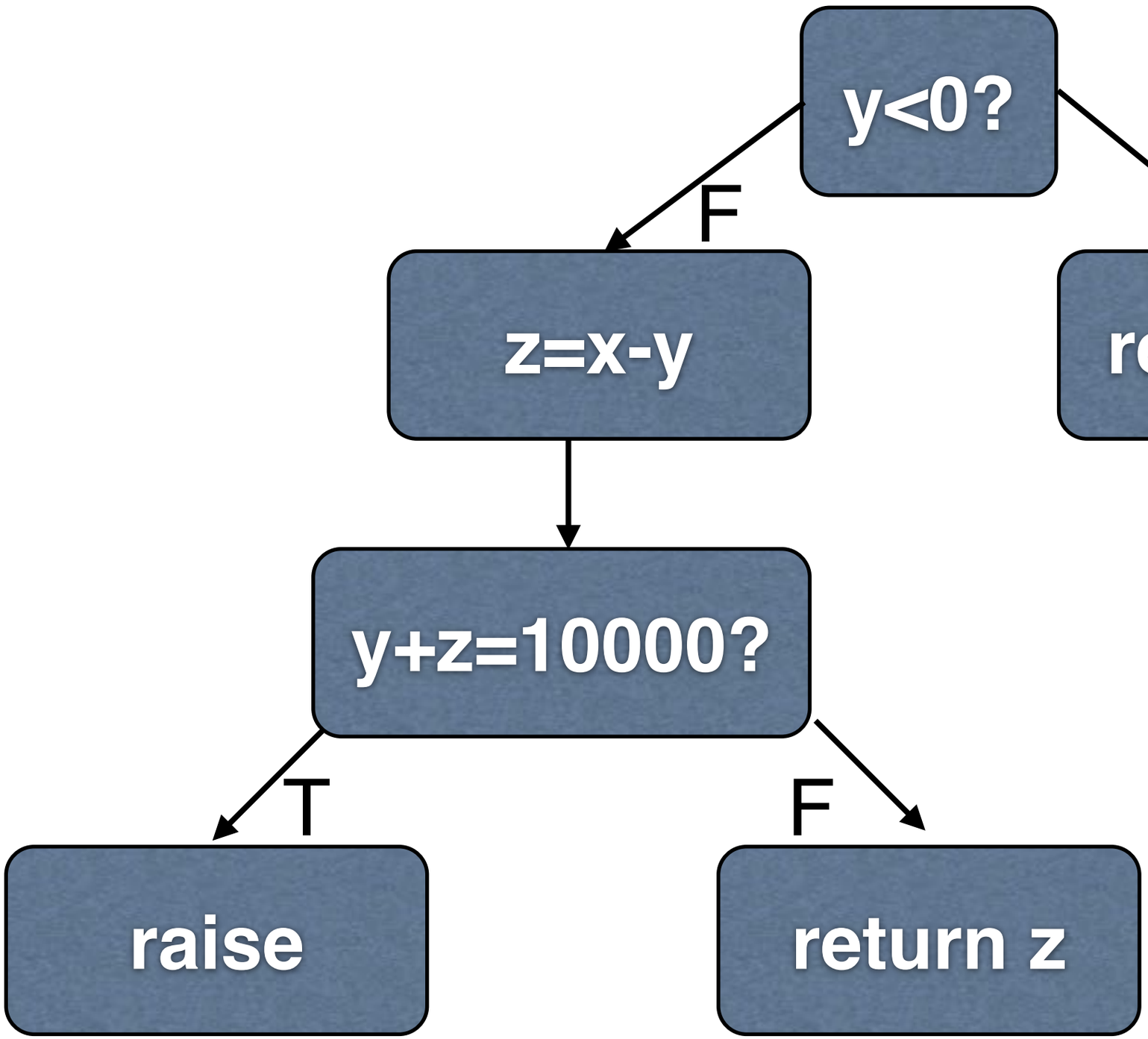
we want to go to the closest uncovered path → negate closest path condition addition

Ejecución Simbólica Dinámica

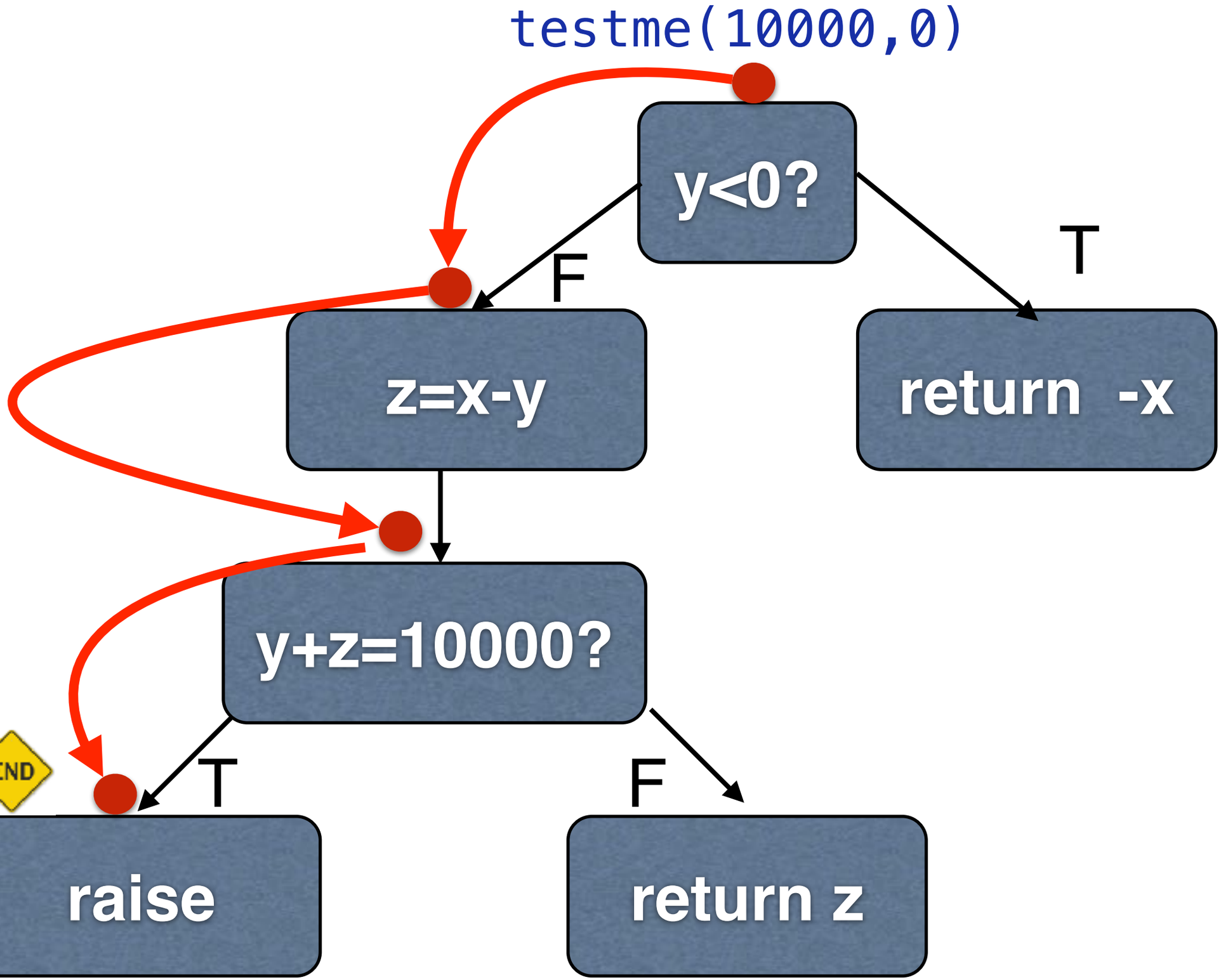


Ejecución Simbólica Dinámica

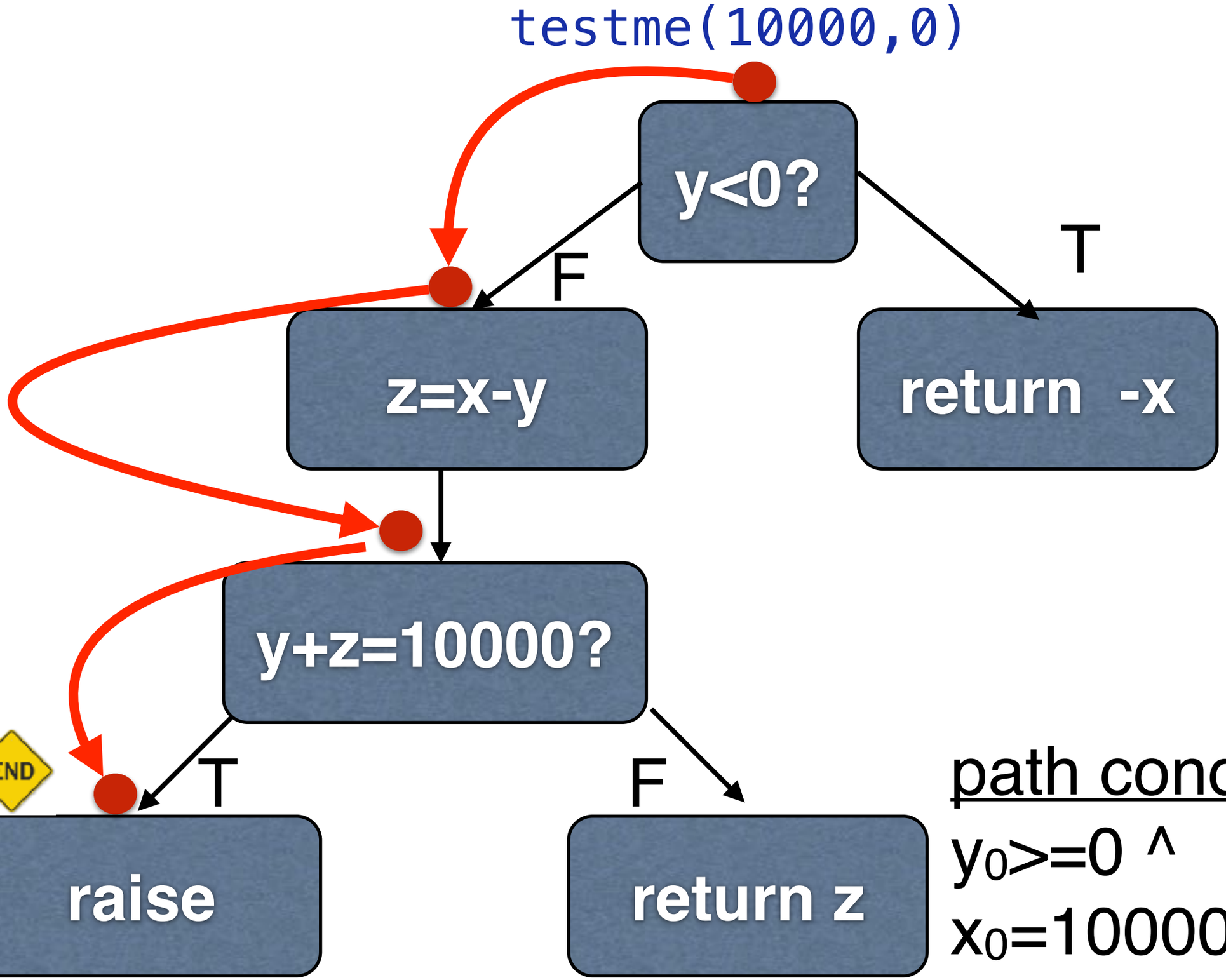
```
testme(10000,0)
```



Ejecución Simbólica Dinámica



Ejecución Simbólica Dinámica

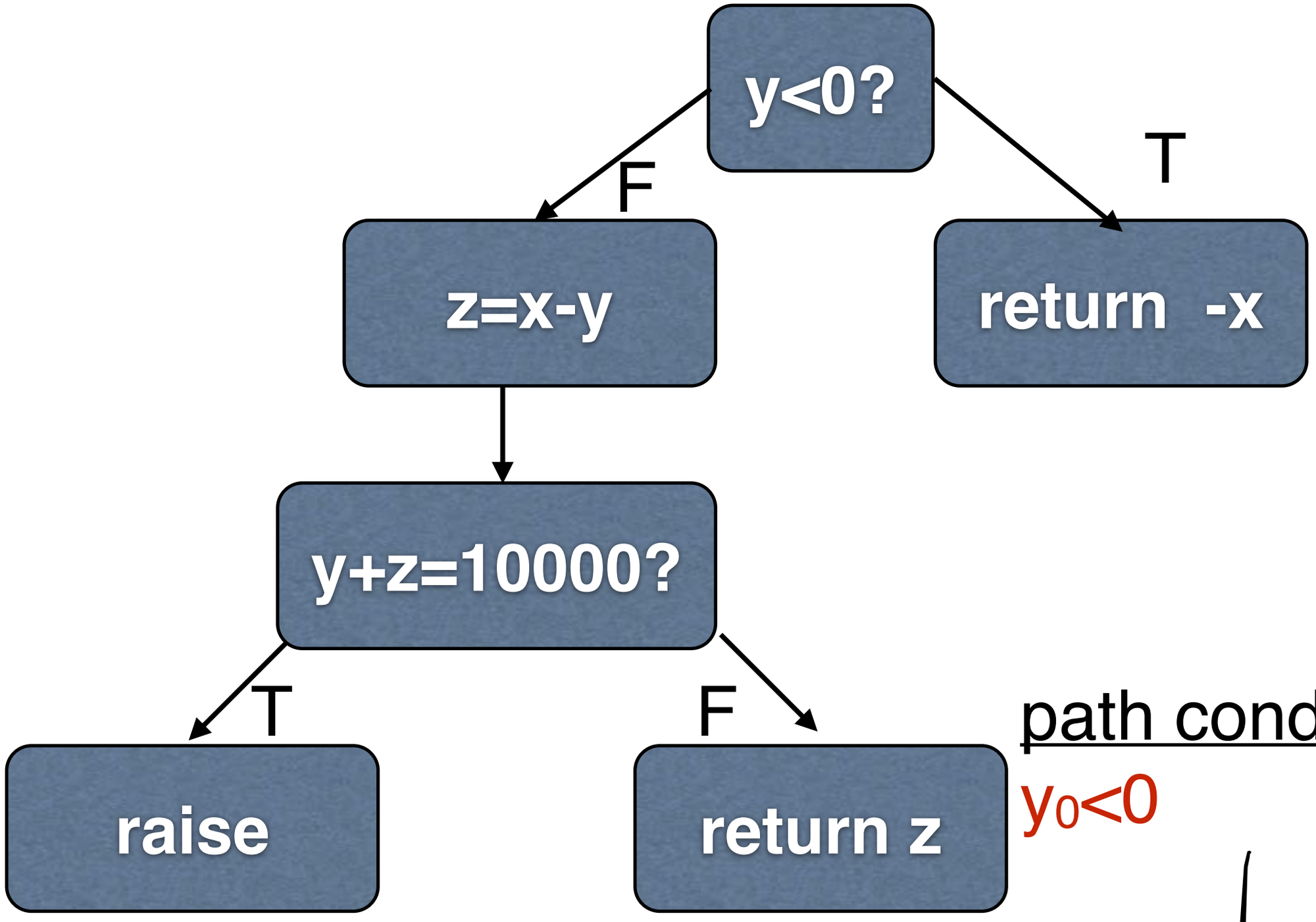


path condition:

$y_0 >= 0$ ^

$x_0 = 10000$

Ejecución Simbólica Dinámica



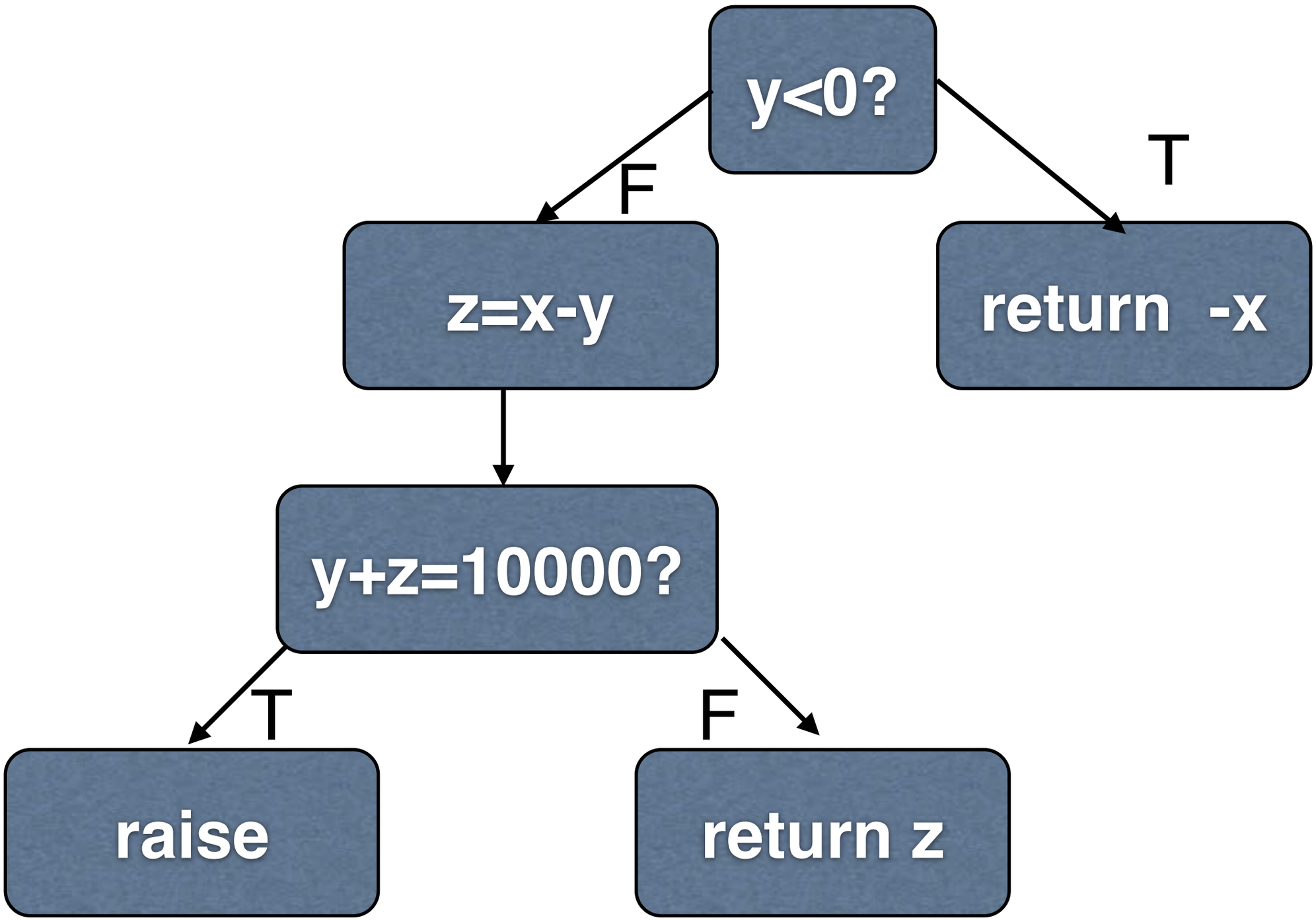
path condition:

$y_0 < 0$

removed & negated last cond

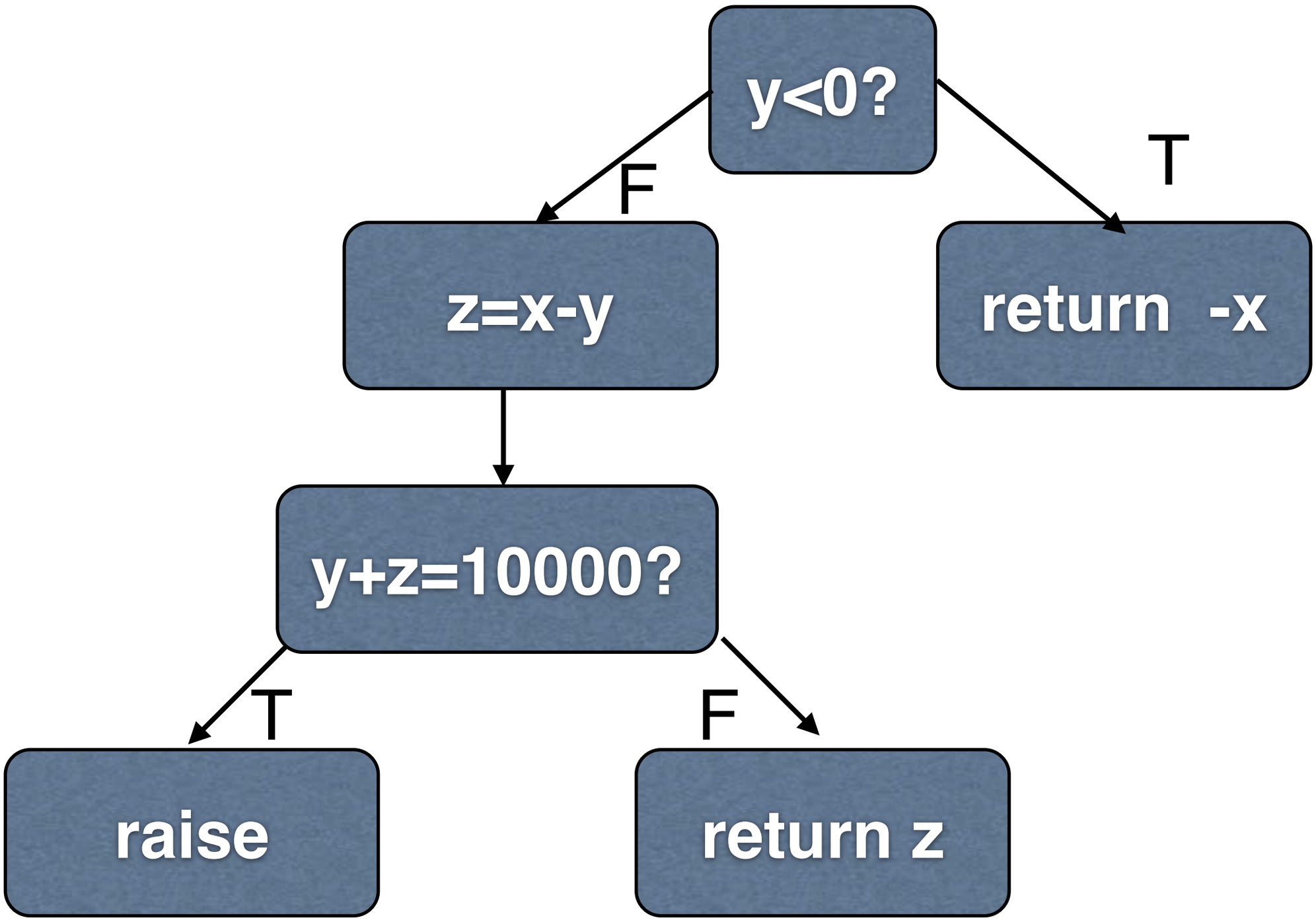
reordering condition

Ejecución Simbólica Dinámica

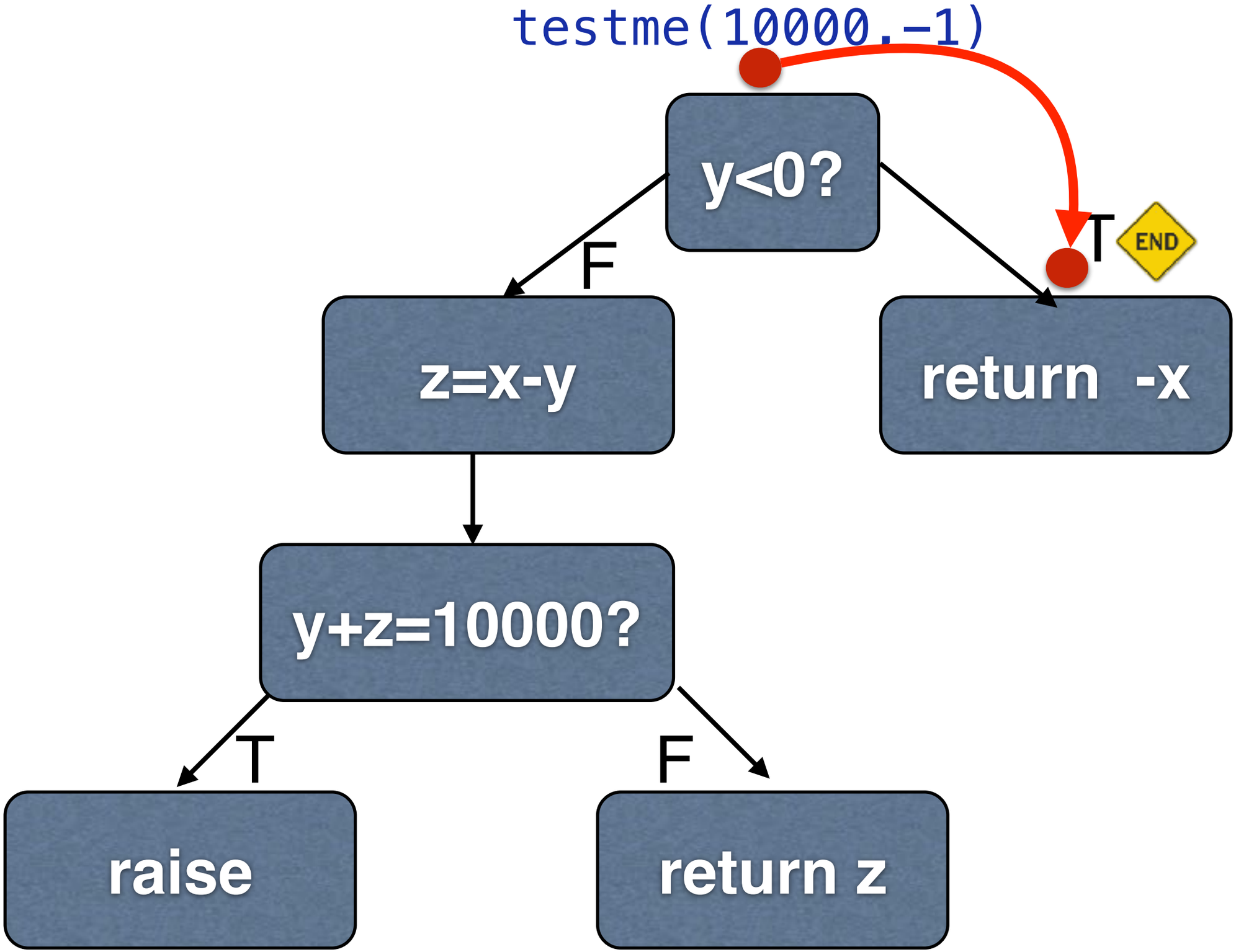


Ejecución Simbólica Dinámica

```
testme(10000,-1)
```



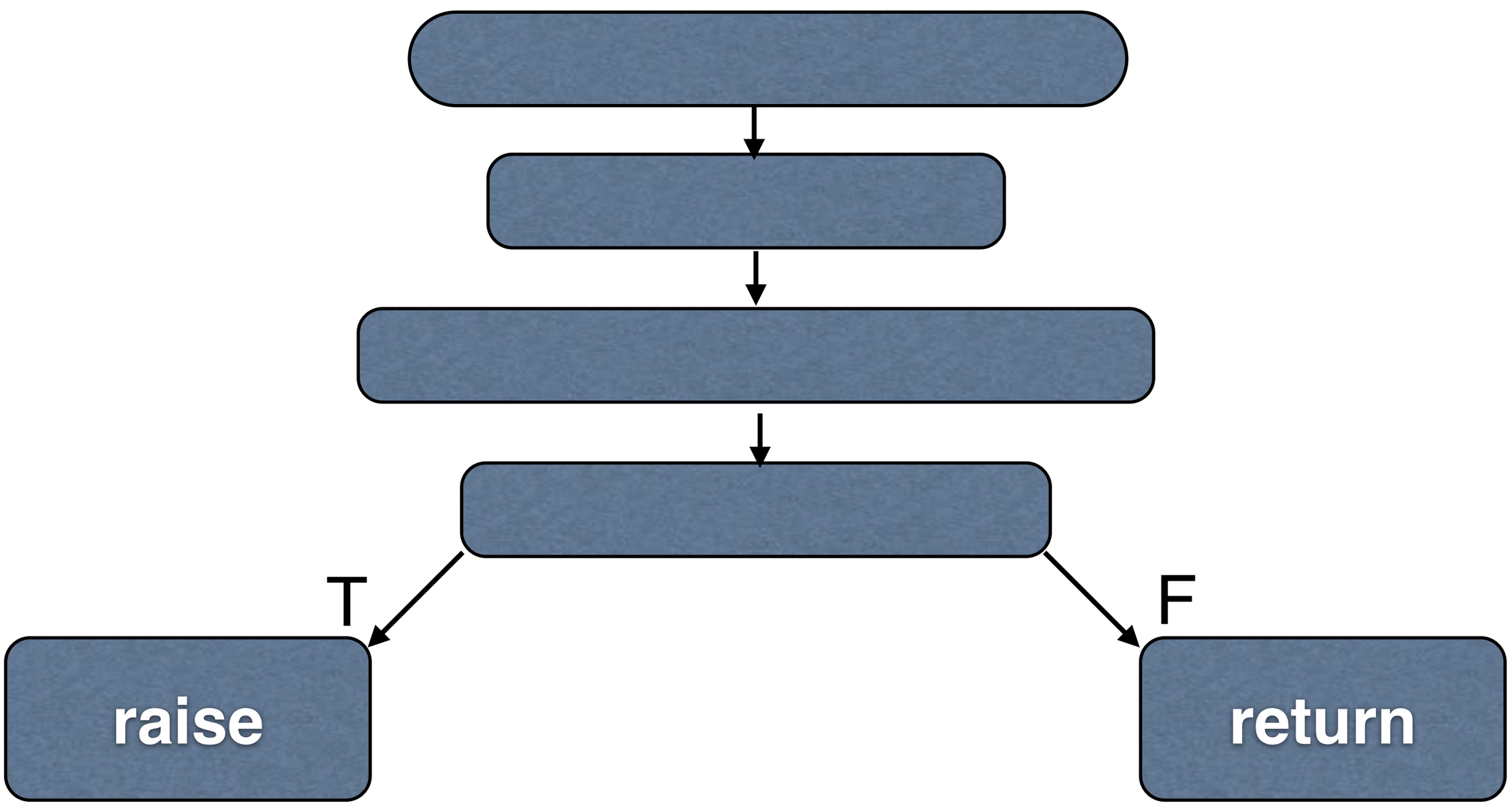
Ejecución Simbólica Dinámica



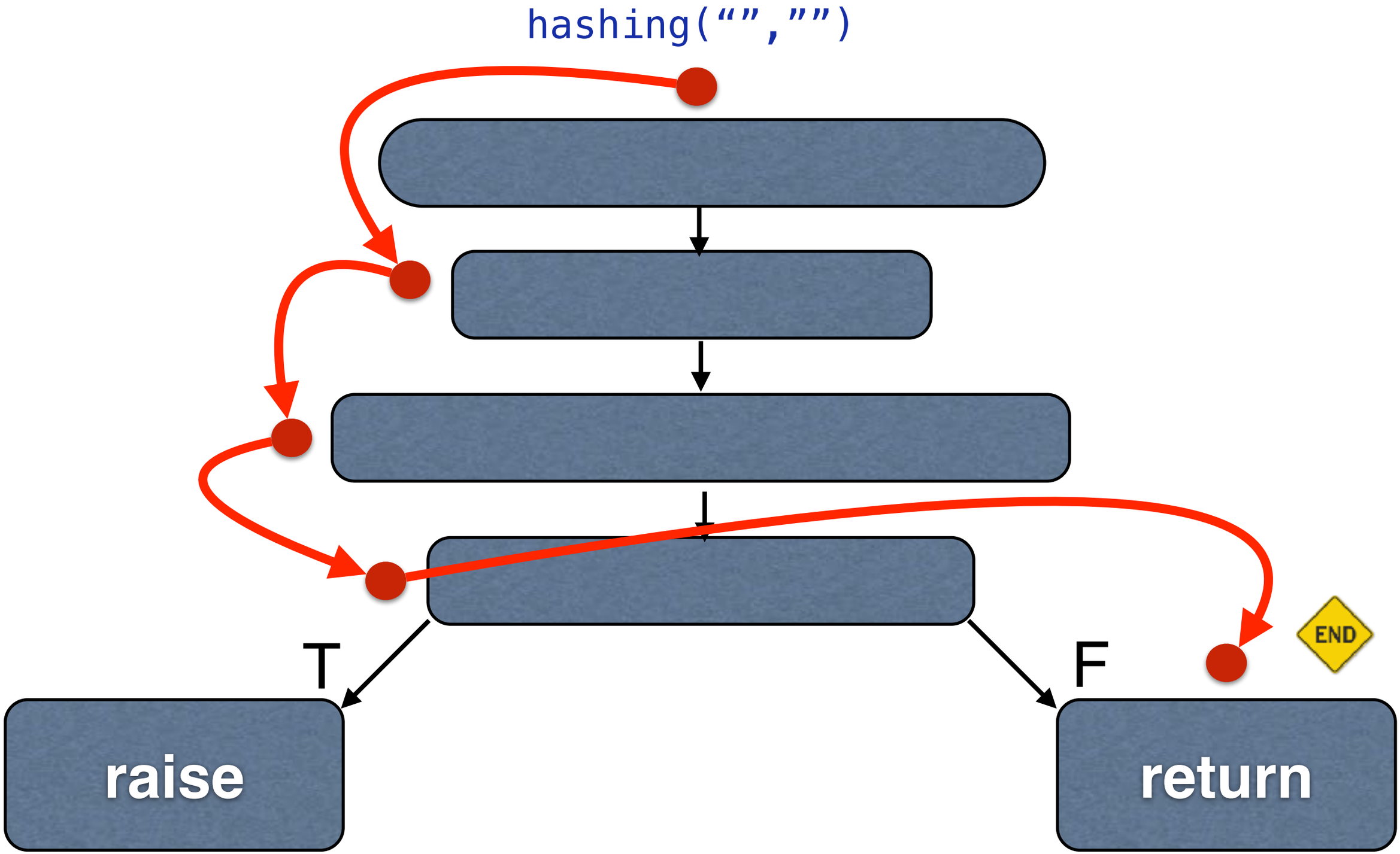
Ejecución Simbólica Dinámica

```
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

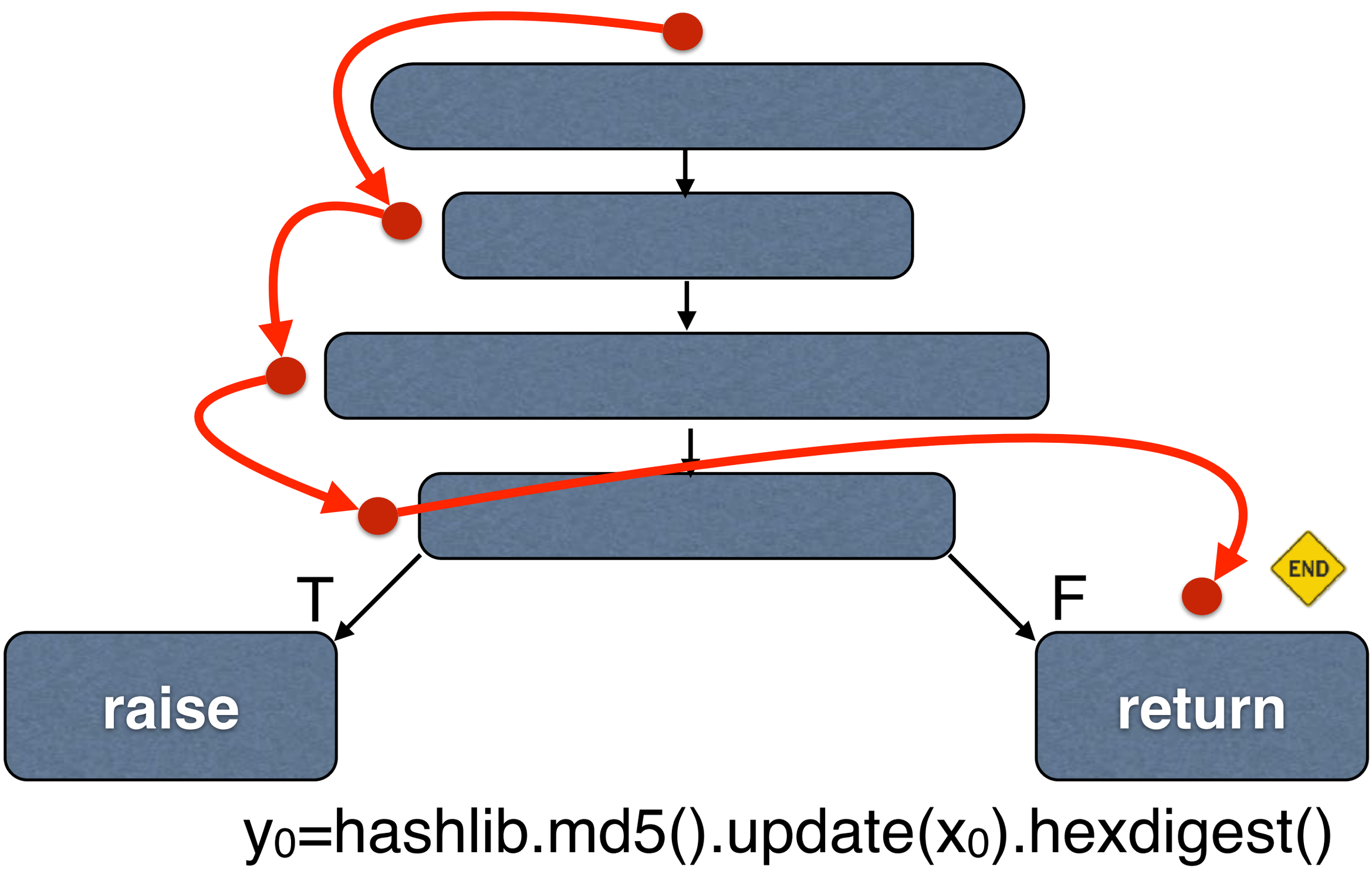


Ejecución Simbólica Dinámica



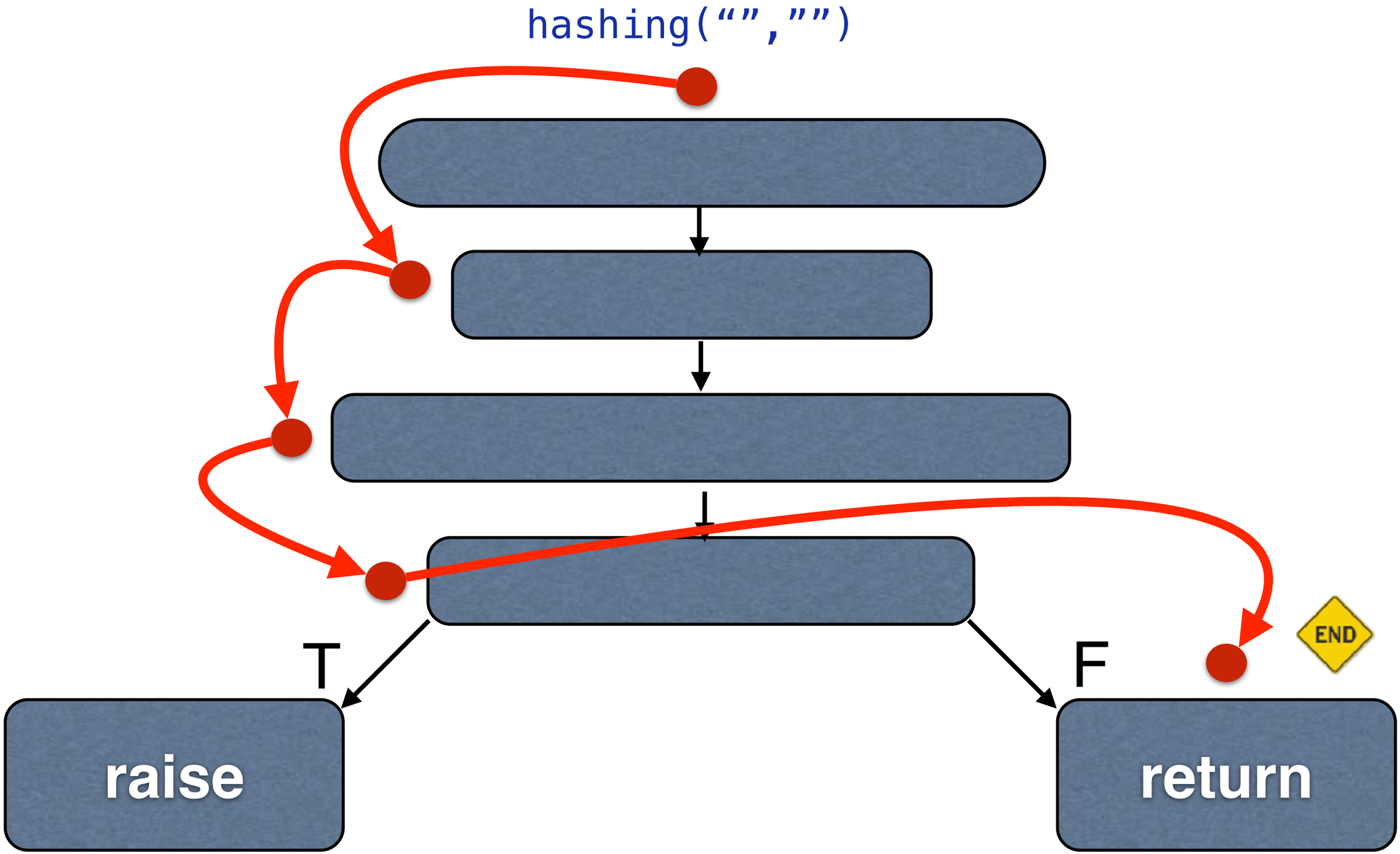
ejecución simbólica se hace al mismo tiempo que ejecución concreta. (CONCOLIC = CONcrete + SYMBOLIC)

Ejecución Simbólica Dinámica

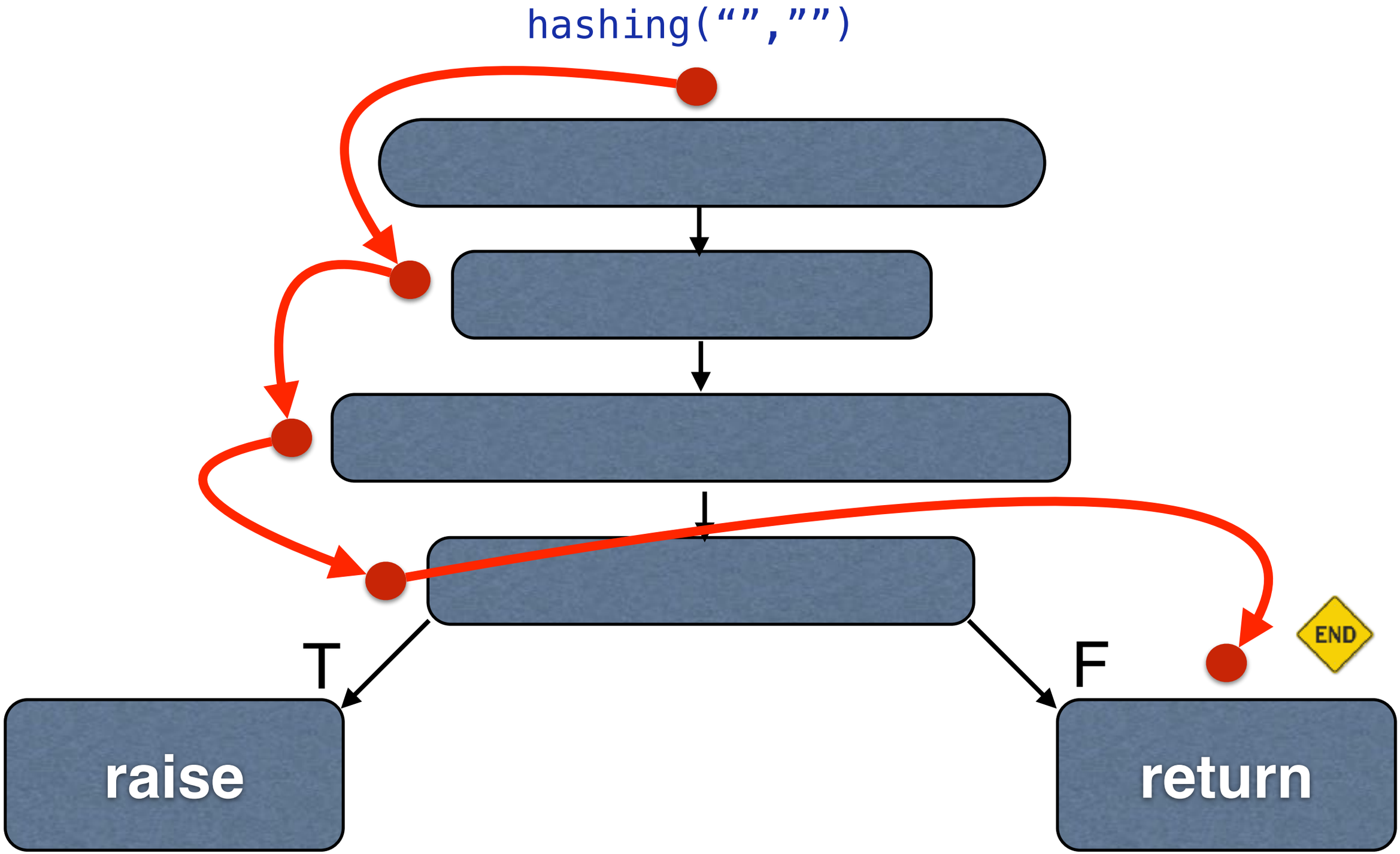


y_0=hashlib.md5().update(x_0).hexdigest()

Ejecución Simbólica Dinámica



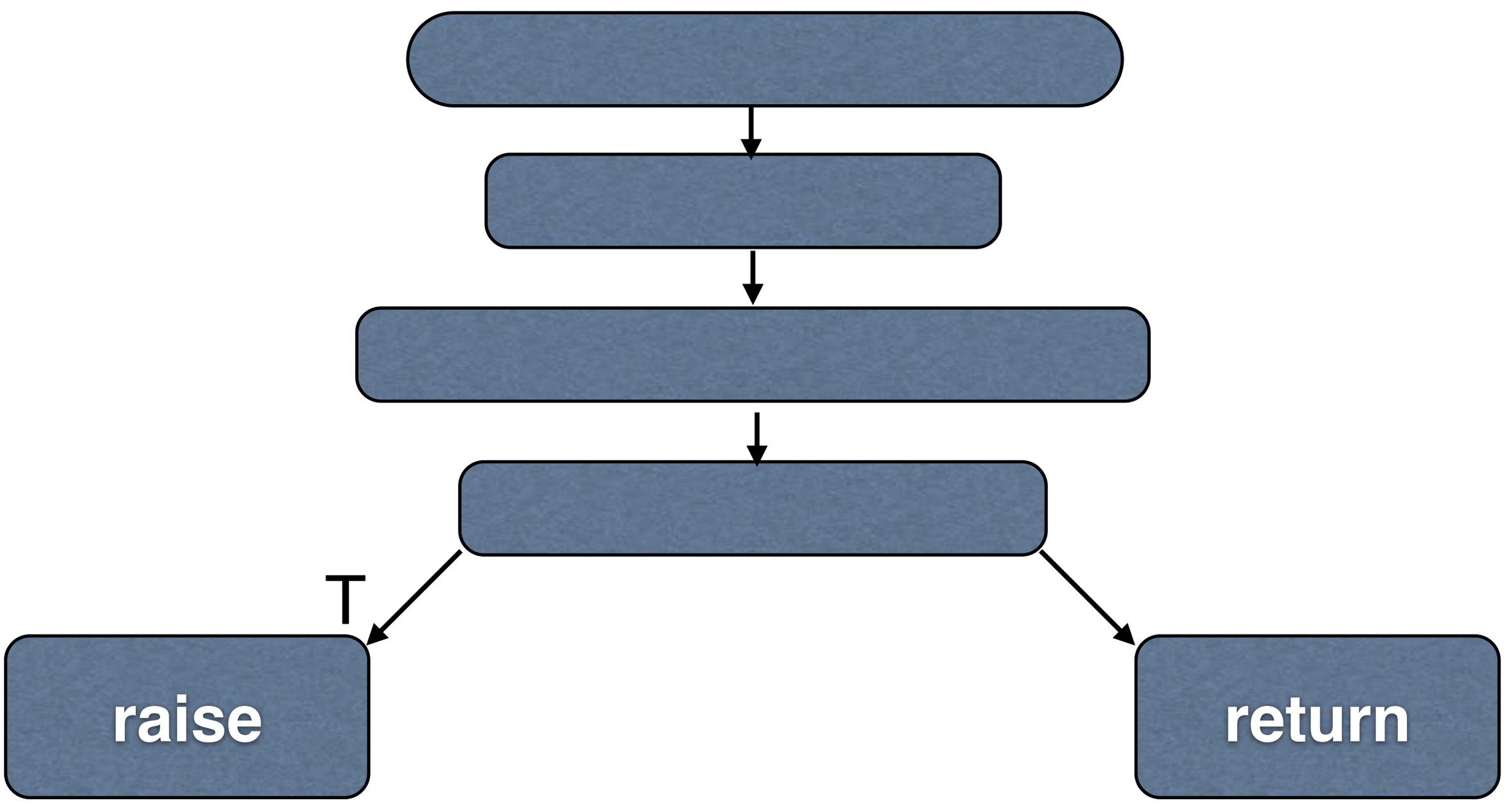
Ejecución Simbólica Dinámica



y_0=“d41d8cd98f00b204e9800998ecf8427e”

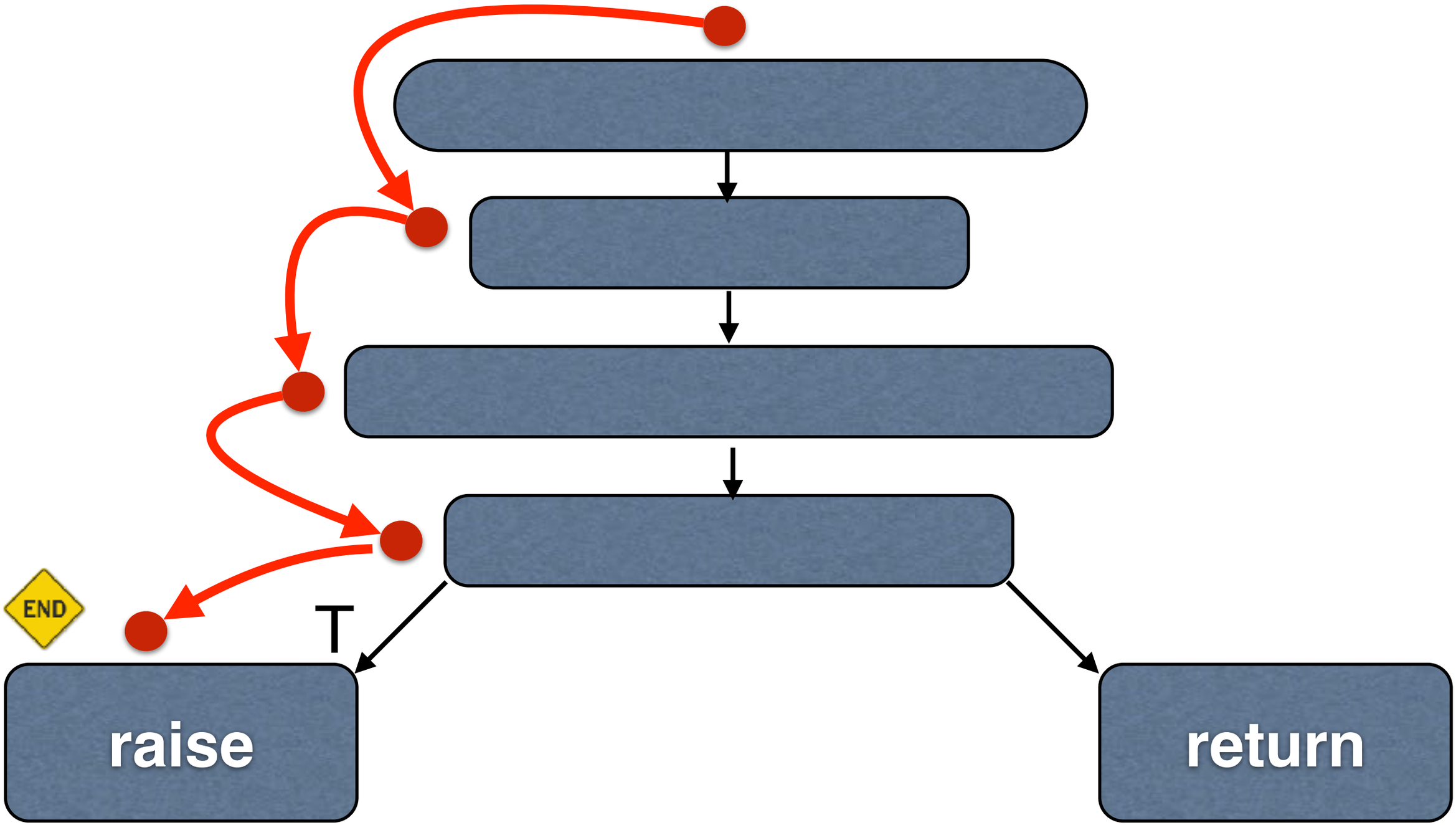
Ejecución Simbólica Dinámica

```
hashing(“”,”d41d8cd98f00b204e9800998ecf8427e”)
```



Ejecución Simbólica Dinámica

```
hashing(“”,”d41d8cd98f00b204e9800998ecf8427e”)
```



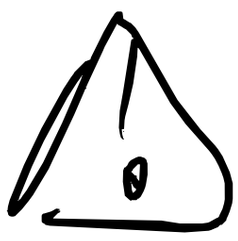
pseudocódigo de generador de tests c/

Ejecución Simbólica Dinámica

```
t = create_test(M)

Mientras budget es no vacío:
 path_condition = exec_concolic(t)
 Para i desde path_condition.size()-1 hasta 0
   branch = path_condition[i]
   Si neg(branch) is not covered:
     query = path_condition[0..i-1] && neg(branch)
     solution = solve(query)
     Si solution is SAT:
       t = create_test(M, solution)
       Agregar t a testSuite
       break

retornar testSuite
```

Ejecución Simbólica Dinámica

- La exploración (i.e. nuevas path conditions a resolver) es guiada por ejecuciones concretas
  - Concolic Execution: Concrete+Symbolic
- Algunas limitaciones 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?

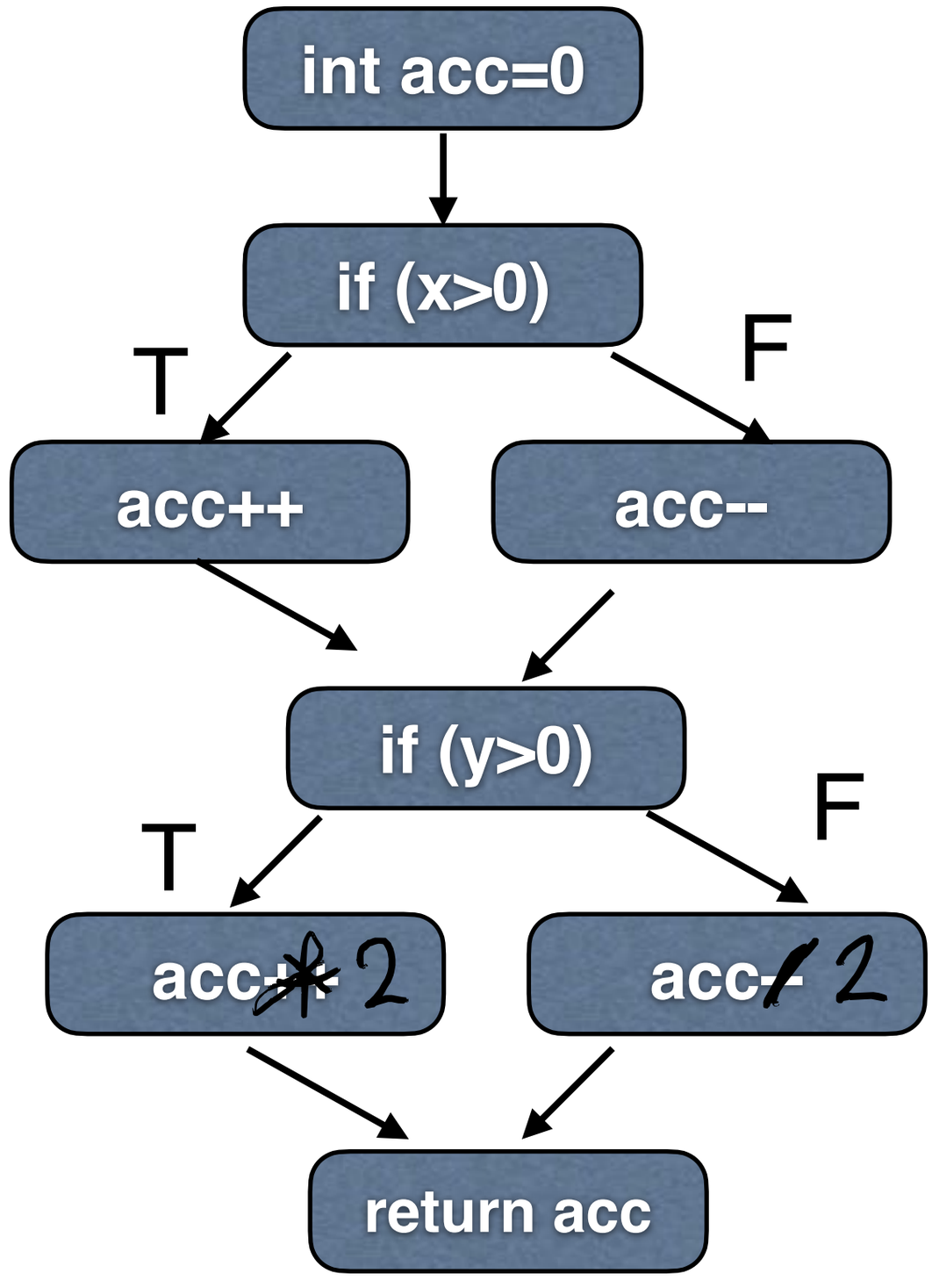
- Problema: si ejecutamos el input y solo obtenemos el camino recorrido en el CFG, entonces no sabemos cuales fueron los valores concretos en cada estado intermedio
- Por eso, si necesitamos "fallar" a un valor concreto, en ese momento no lo tenemos
- exec_concolic(t) hace ambas cosas a la vez!

Arbol de Cómputo

- Es una representación de la exploración del algoritmo de DSE (dynamic symbolic exec)
- Cada nodo representa una decisión (if, while, for). La rama izquierda es el "FALSE", la rama derecha es el "TRUE".
- Cada hoja puede ser:
  - Resultado UNSAT/UNKNOWN del solver
  - "X" si el camino está cerrado o ejecutado

CFG de test_me ≠ árbol

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc/2;
   }
   return acc;
}
```



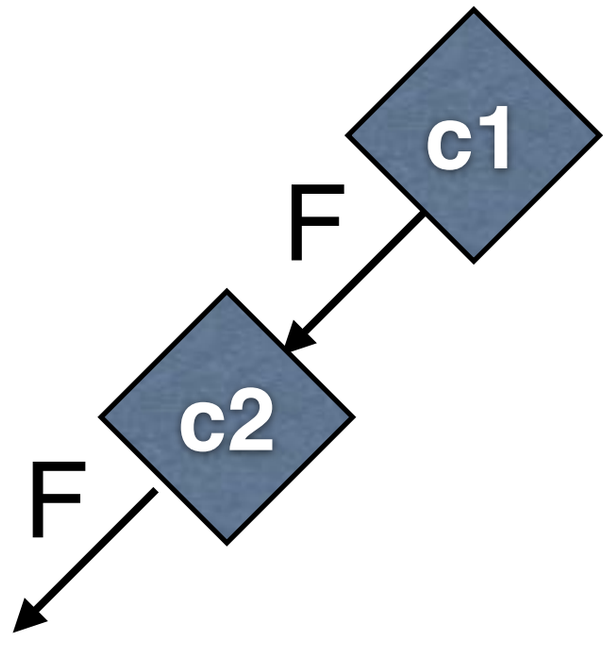
```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

árbol de cómputo explorado

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0			
2				
3				
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

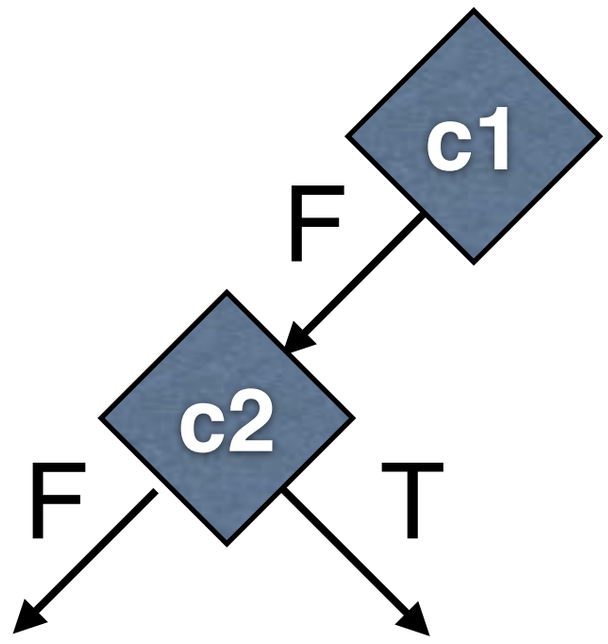
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)		
2				
3				
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

árbol de cómputo explorado

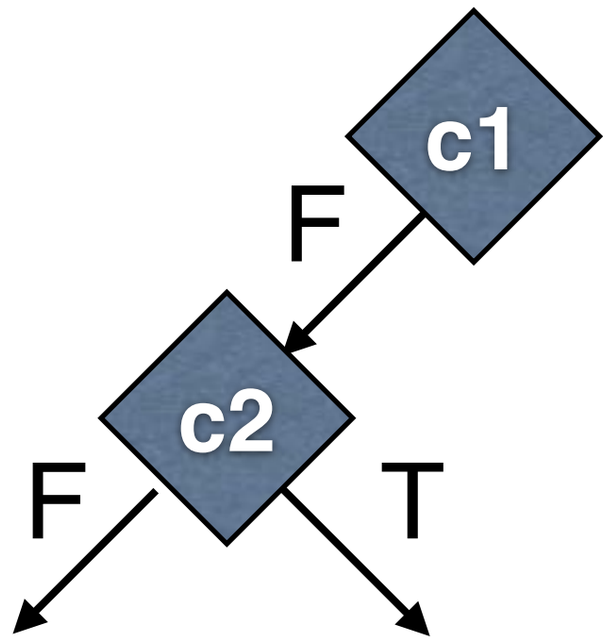


X ?

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	?	
2				
3				
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

árbol de cómputo explorado

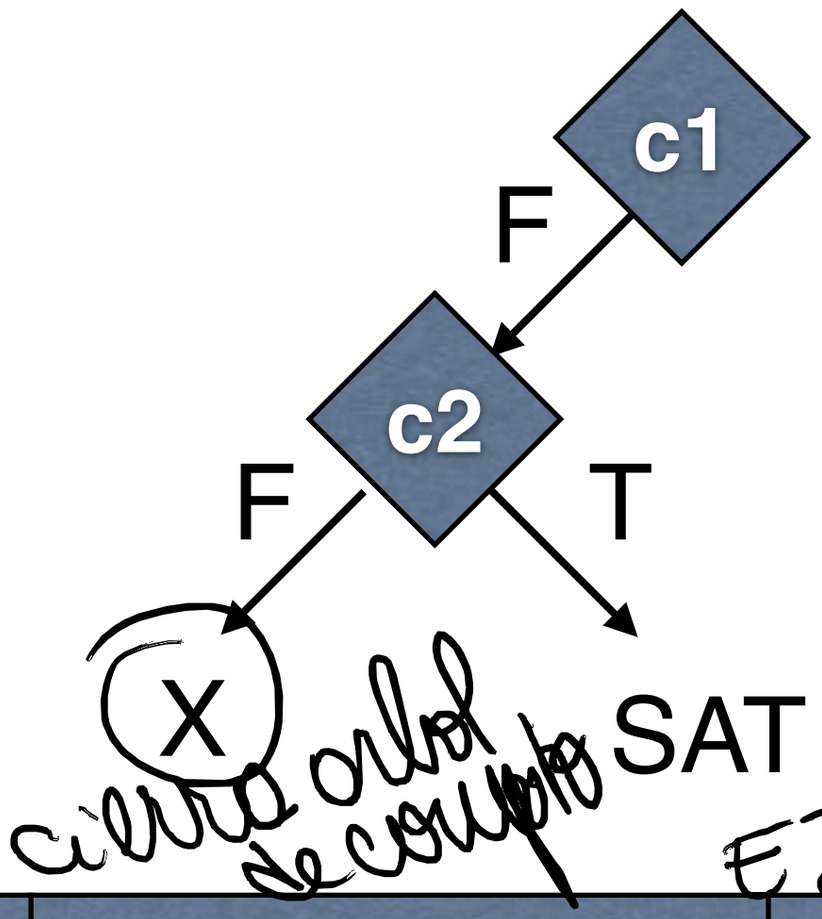


X ?

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	?
2				
3				
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

árbol de cómputo explorado

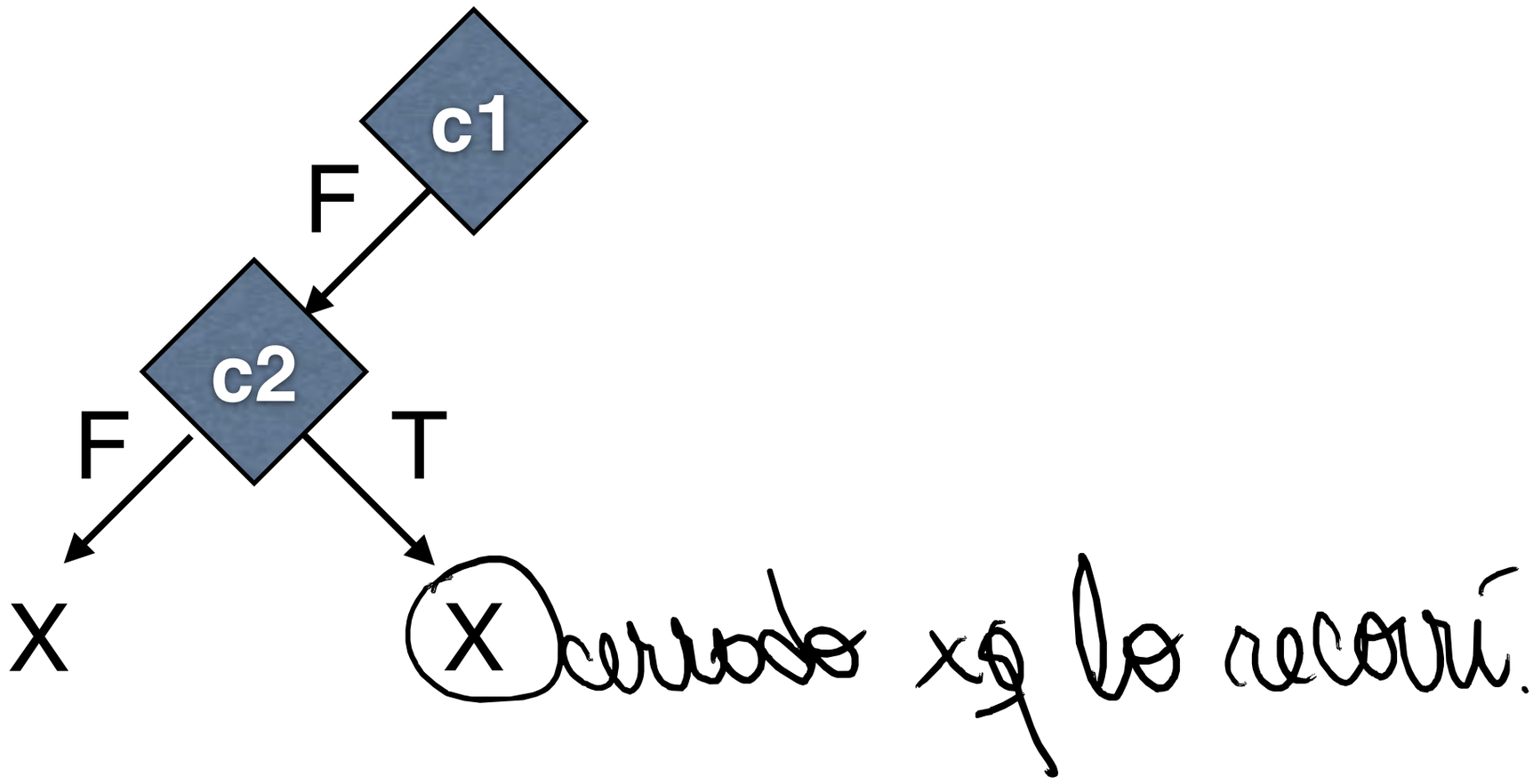


EJECUCIÓN CONCRETA

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	x0=0,y0=1
2				
3				
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

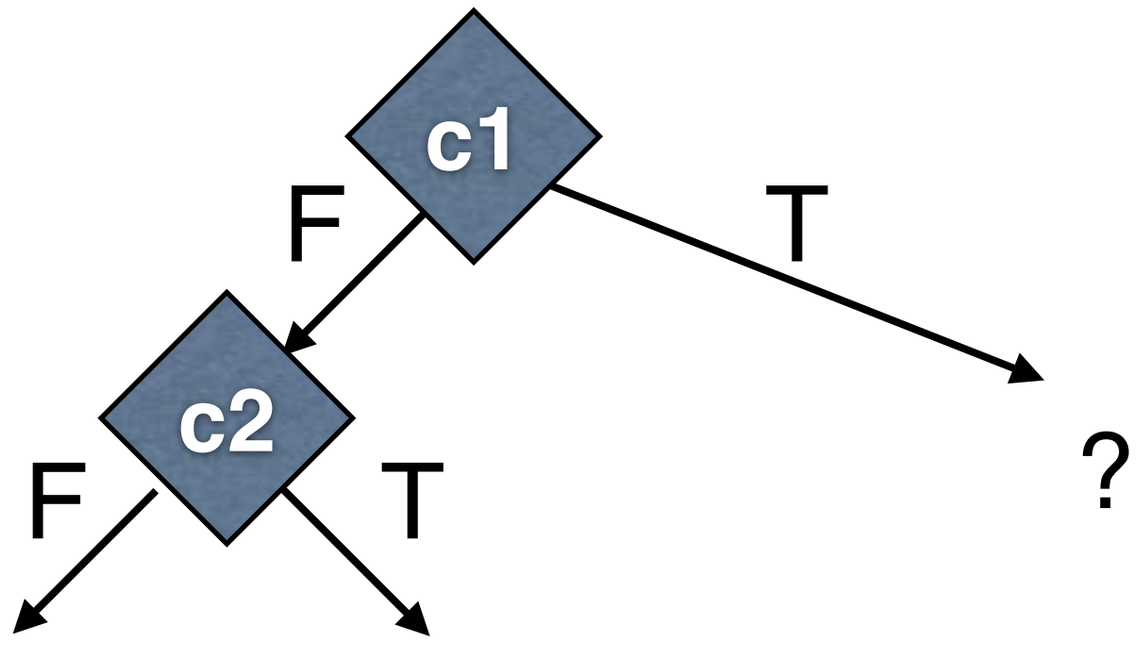
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	x0=0,y0=1
2	x=0,y=1	!(x0>0) && (y0>0)		
3				
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

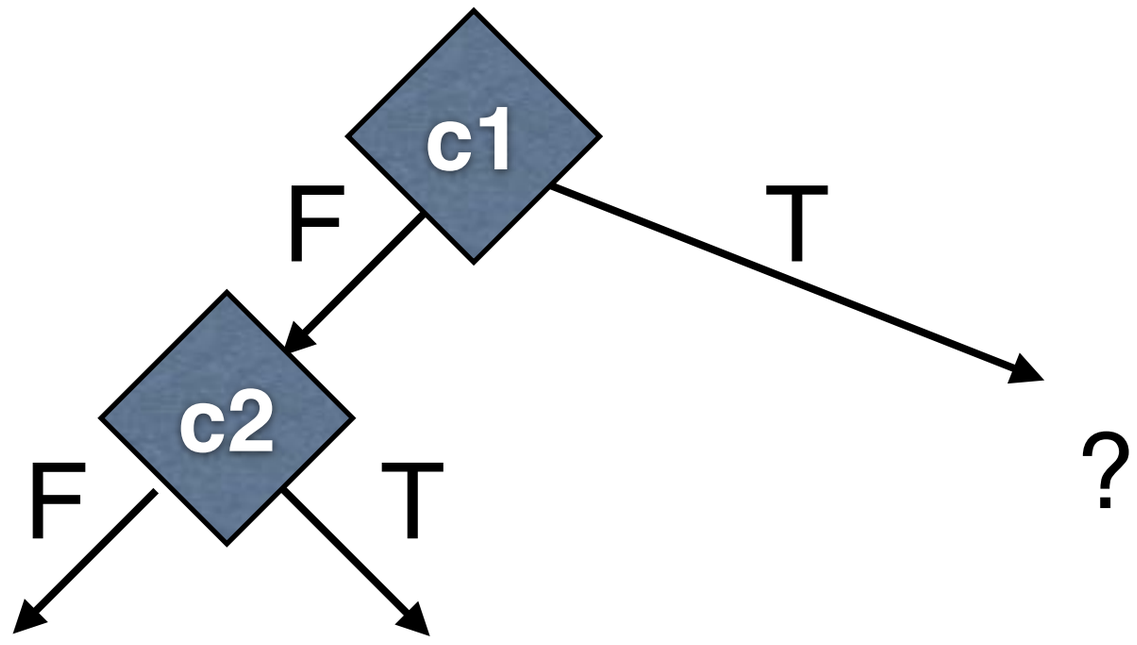
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	x0=0,y0=1
2	x=0,y=1	!(x0>0) && (y0>0)	?	
3				
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

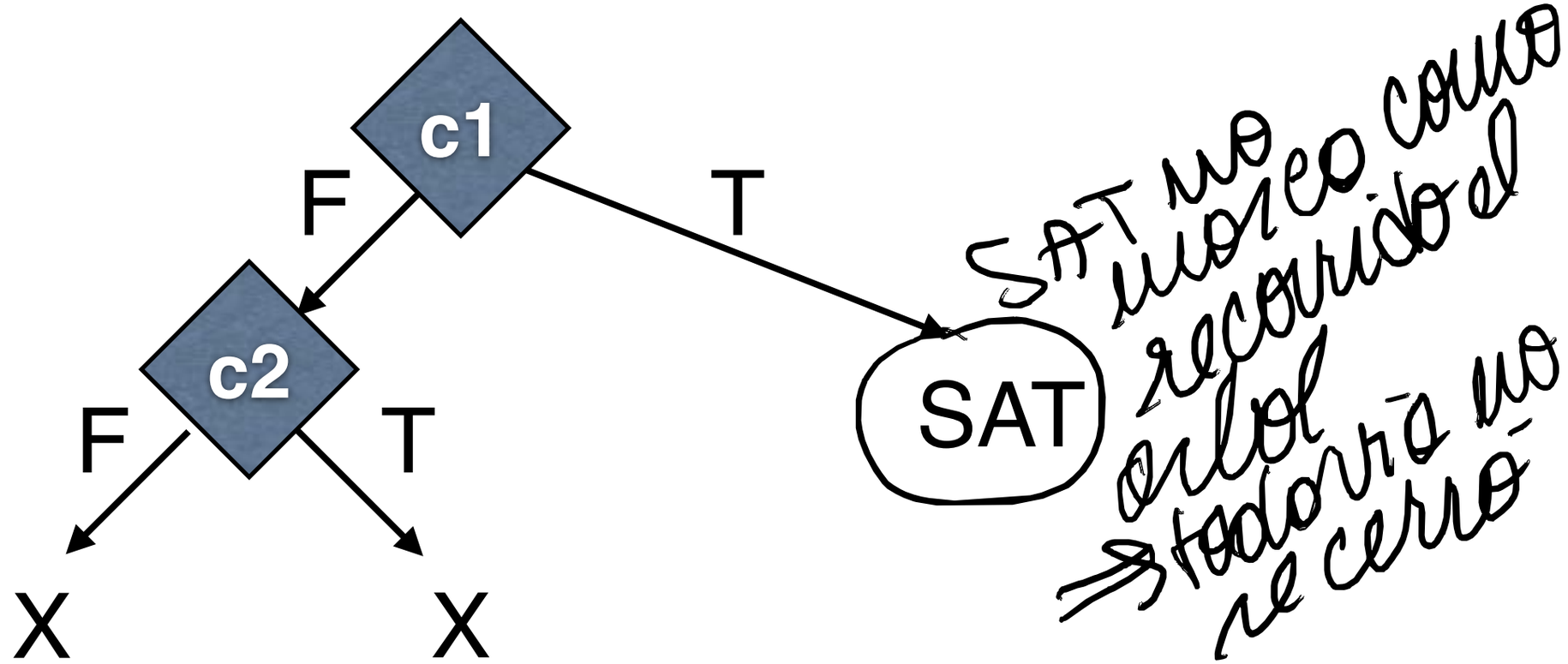
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	x0=0,y0=1
2	x=0,y=1	!(x0>0) && (y0>0)	(x0>0)	?
3				
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

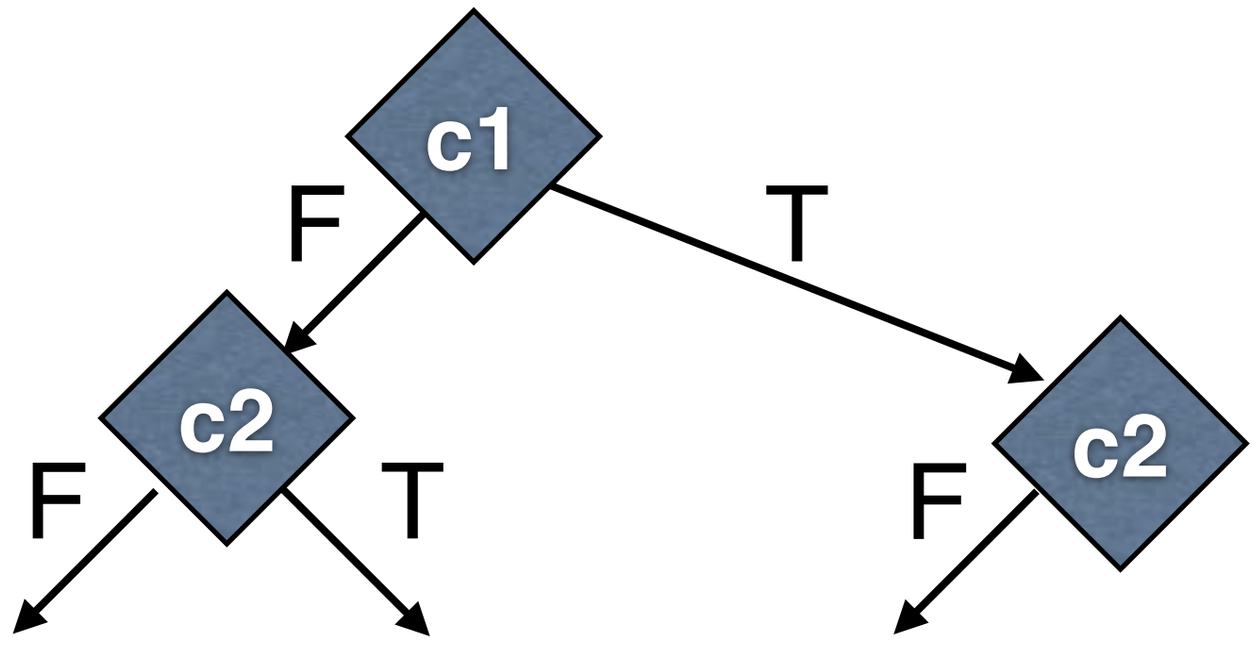
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	x0=0,y0=1
2	x=0,y=1	!(x0>0) && (y0>0)	(x0>0)	x0=1,y0=0
3				
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

árbol de cómputo explorado

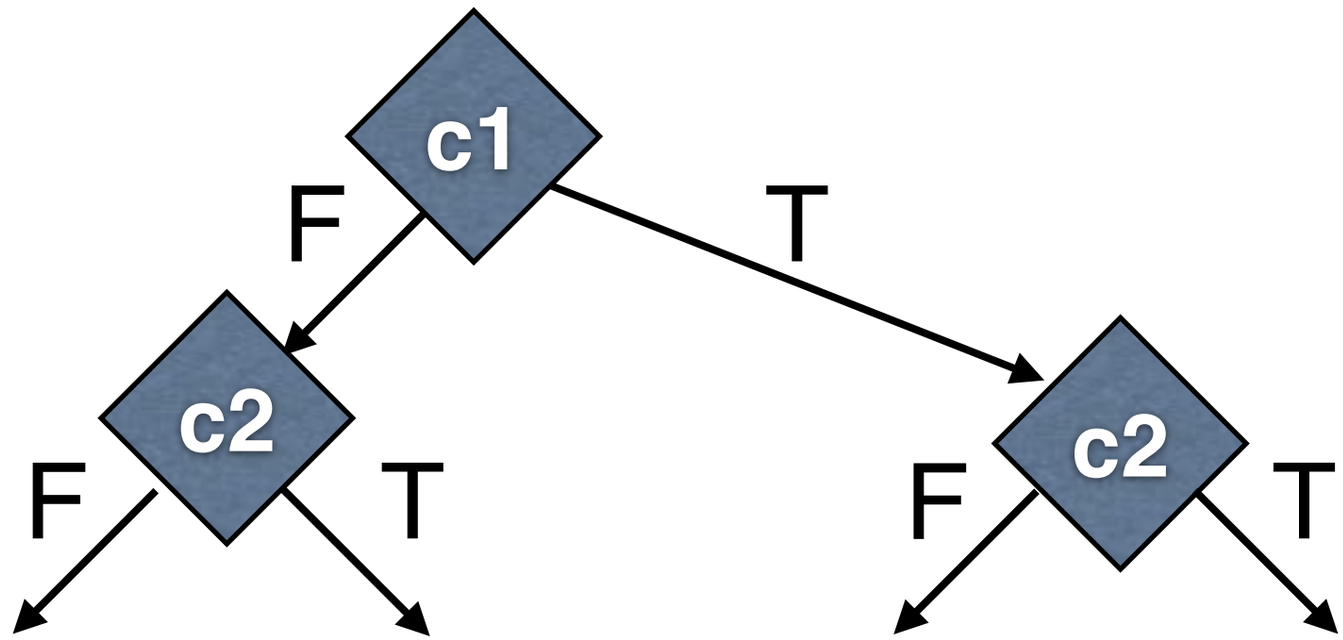


X X X

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	x0=0,y0=1
2	x=0,y=1	!(x0>0) && (y0>0)	(x0>0)	x0=1,y0=0
3	x=1,y=0	(x0>0) && !(y0>0)		
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

árbol de cómputo explorado

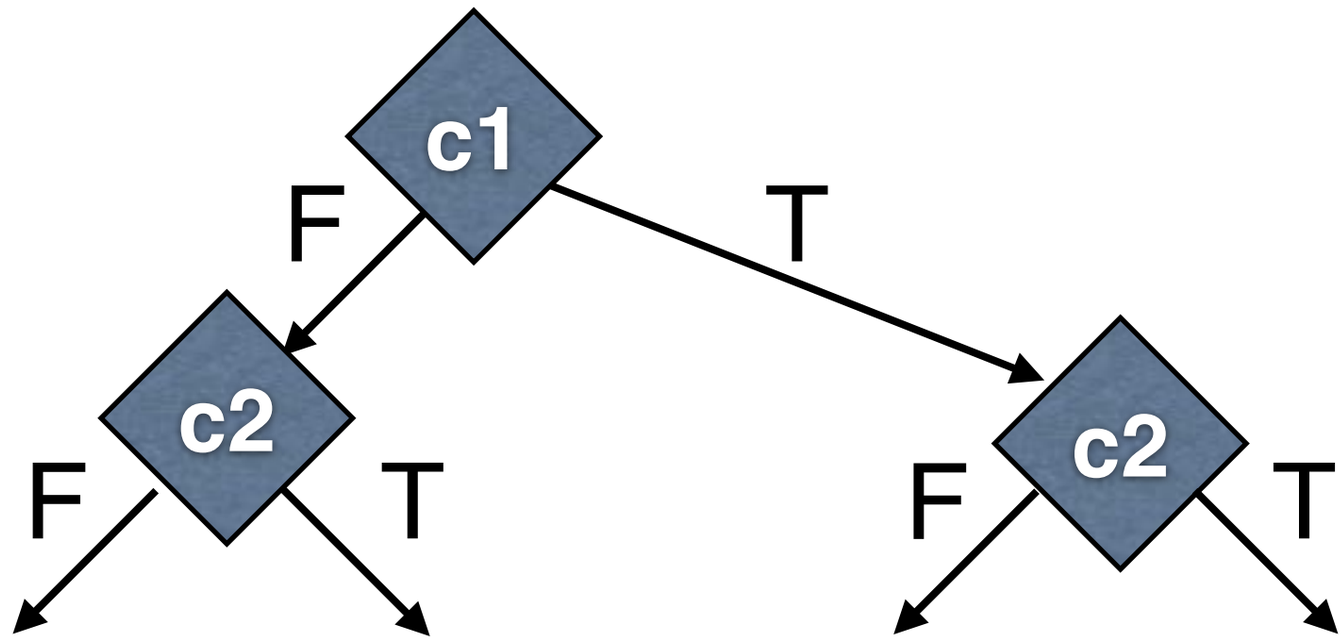


X X X ?

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	x0=0,y0=1
2	x=0,y=1	!(x0>0) && (y0>0)	(x0>0)	x0=1,y0=0
3	x=1,y=0	(x0>0) && !(y0>0)	?	
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

árbol de cómputo explorado

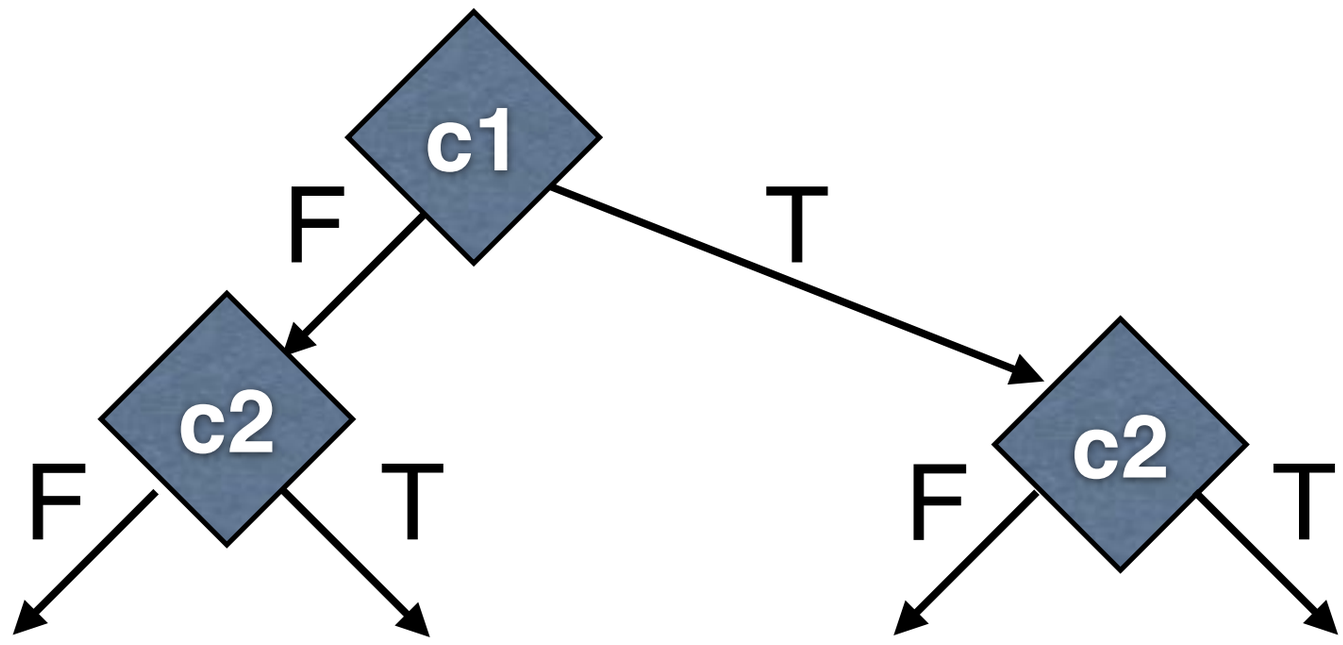


X X X ?

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	x0=0,y0=1
2	x=0,y=1	!(x0>0) && (y0>0)	(x0>0)	x0=1,y0=0
3	x=1,y=0	(x0>0) && !(y0>0)	(x0>0) &&(y0>0)	?
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

árbol de cómputo explorado

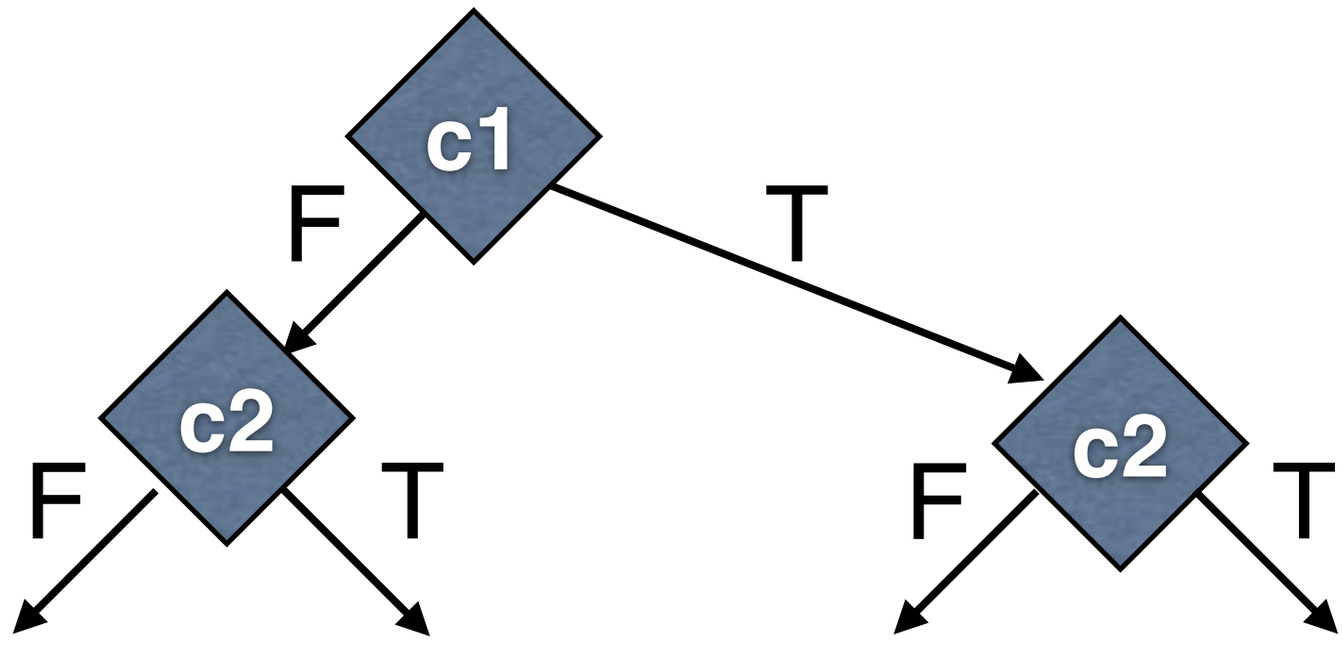


X X X SAT

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	x0=0,y0=1
2	x=0,y=1	!(x0>0) && (y0>0)	(x0>0)	x0=1,y0=0
3	x=1,y=0	(x0>0) && !(y0>0)	(x0>0) &&(y0>0)	x0=1,y0=1
4				

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

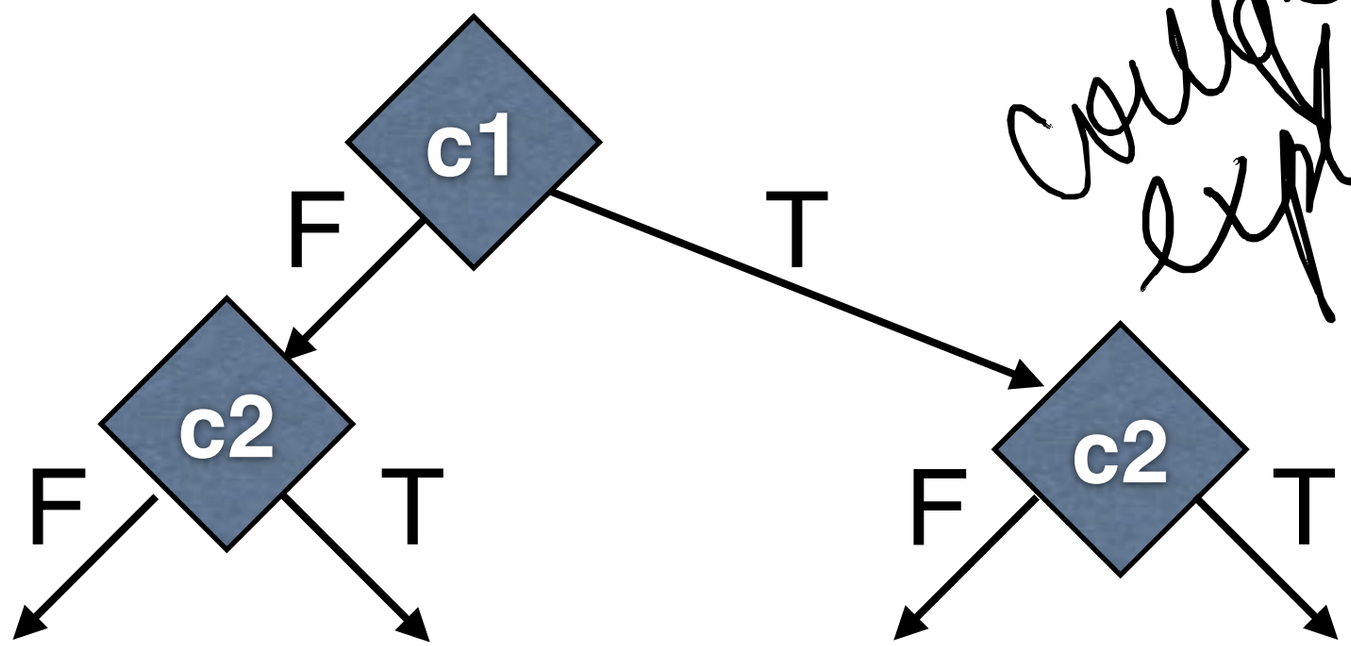
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	x0=0,y0=1
2	x=0,y=1	!(x0>0) && (y0>0)	(x0>0)	x0=1,y0=0
3	x=1,y=0	(x0>0) && !(y0>0)	(x0>0) &&(y0>0)	x0=1,y0=1
4	x=1,y=1	(x0>0) && (y0>0)		

```
int test_me(int x, int y ) {
   int acc=0;
   if (x>0) { // c1
      acc++;
   } else {
      acc--;
   }
   if (y>0) { // c2
      acc=acc*2;
   } else {
      acc=acc*2;
   }
}
```

árbol de cómputo explorado



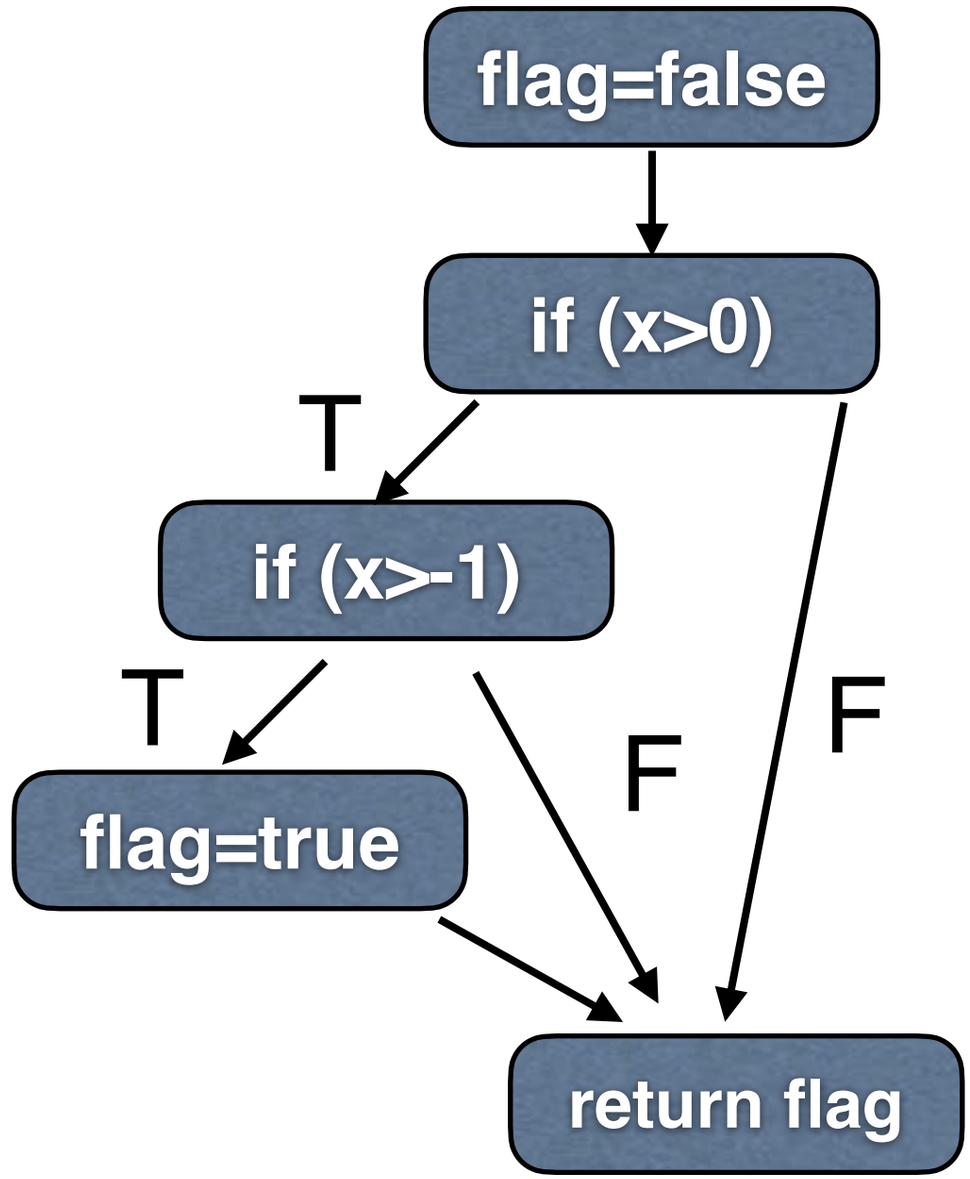
caminos explorados

X X X X

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,y=0	!(x0>0) && !(y0>0)	!(x0>0) && (y0>0)	x0=0,y0=1
2	x=0,y=1	!(x0>0) && (y0>0)	(x0>0)	x0=1,y0=0
3	x=1,y=0	(x0>0) && !(y0>0)	(x0>0) &&(y0>0)	x0=1,y0=1
4	x=1,y=1	(x0>0) && (y0>0)	FIN	

CFG de test_me

```
boolean test_me(int x) {
   boolean flag=false;
   if (x>0) { // c1
      if (x>-1) { // c2
         flag=true;
      }
   }
  return flag;
}
```



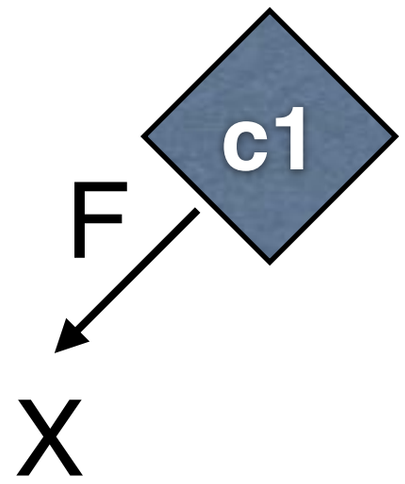
árbol de cómputo explorado

```
boolean test_me(int x) {
   boolean flag=false;
   if (x>0) { // c1
      if (x>-1) { // c2
         flag=true;
      }
   }
  return flag;
}
```

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0			
2				
3				
4				

```
boolean test_me(int x) {
   boolean flag=false;
   if (x>0) { // c1
      if (x>-1) { // c2
         flag=true;
      }
   }
  return flag;
}
```

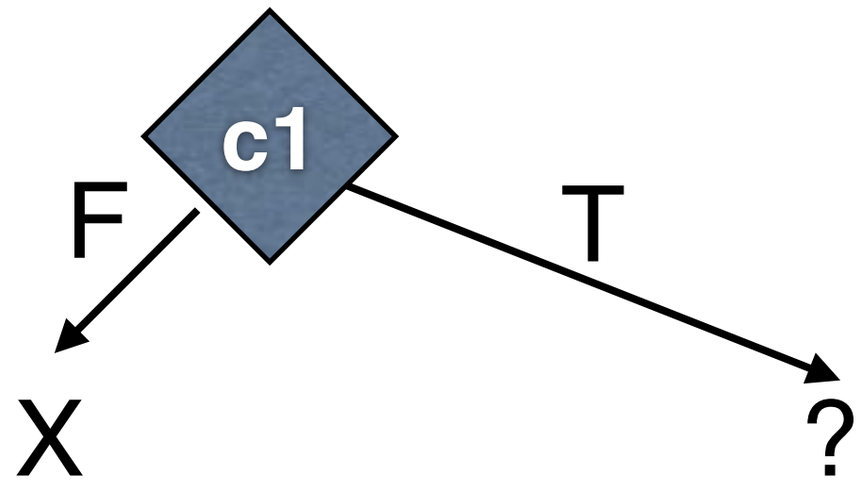
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0	!(x0>0)		
2				
3				
4				

```
boolean test_me(int x) {
   boolean flag=false;
   if (x>0) { // c1
      if (x>-1) { // c2
         flag=true;
      }
   }
  return flag;
}
```

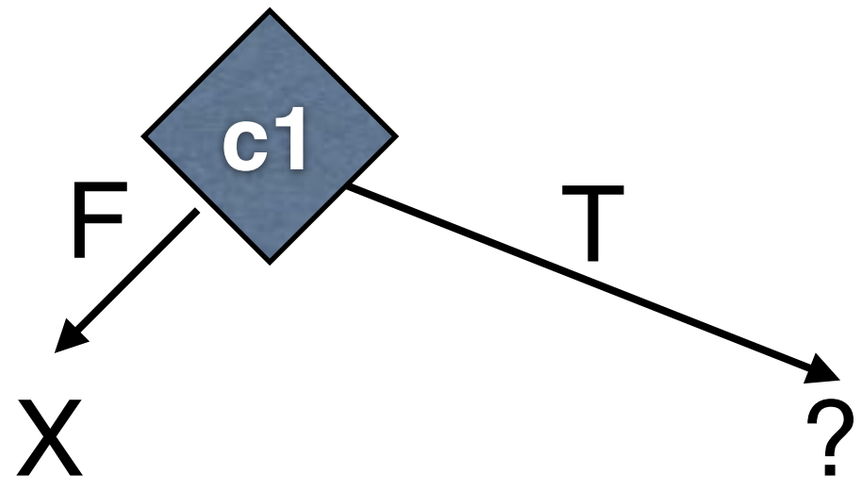
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0	!(x0>0)		
2				
3				
4				

```
boolean test_me(int x) {
   boolean flag=false;
   if (x>0) { // c1
      if (x>-1) { // c2
         flag=true;
      }
   }
  return flag;
}
```

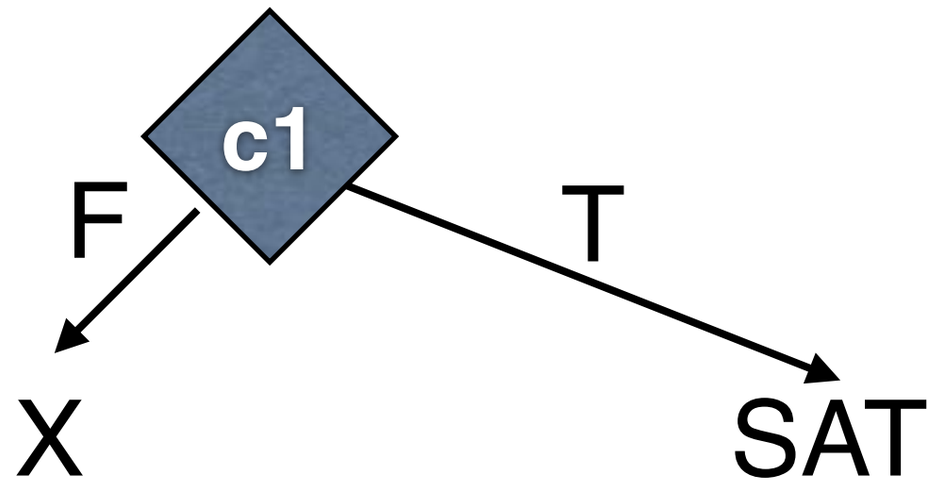
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0	!(x0>0)	x0>0	?
2				
3				
4				

```
boolean test_me(int x) {
   boolean flag=false;
   if (x>0) { // c1
      if (x>-1) { // c2
         flag=true;
      }
   }
  return flag;
}
```

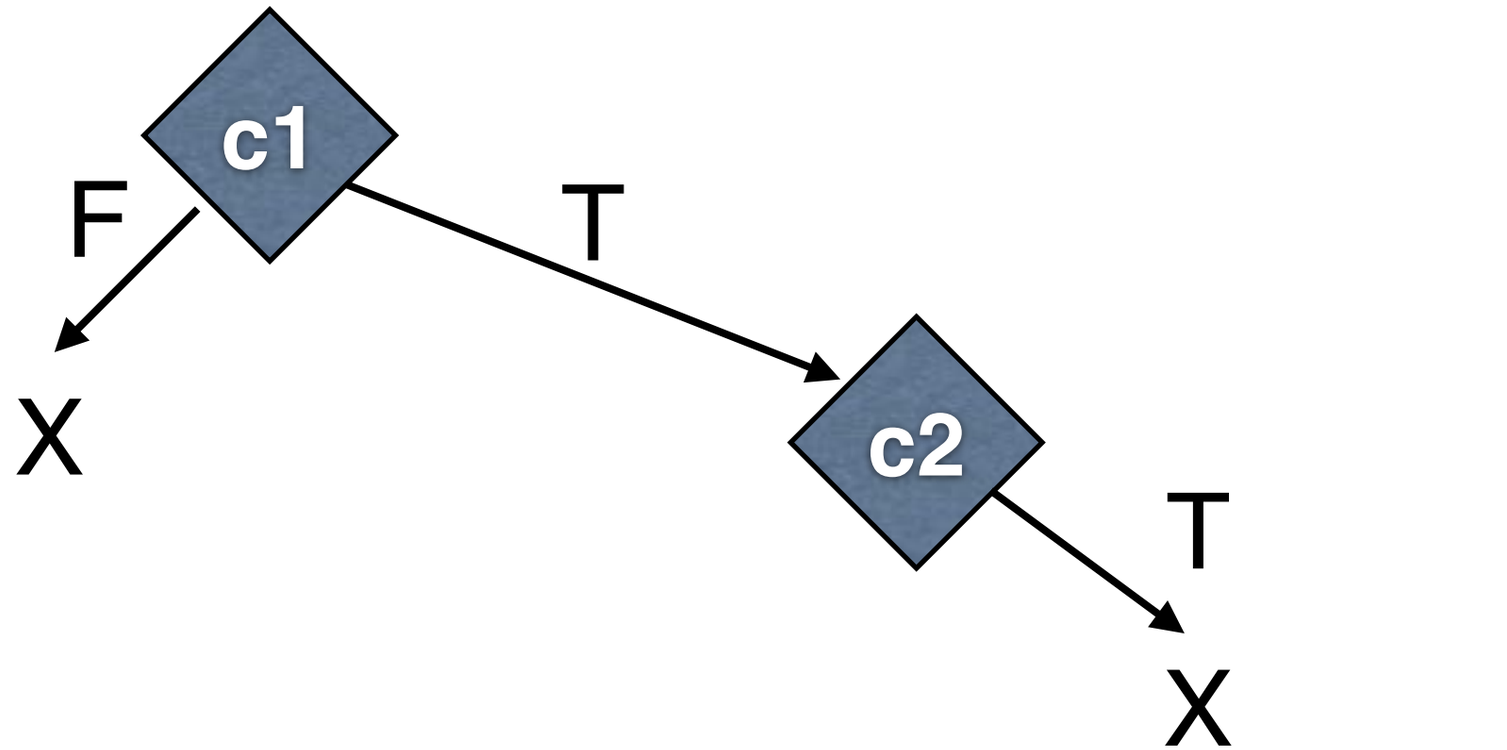
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0	!(x0>0)	x0>0	x0=1
2				
3				
4				

```
boolean test_me(int x) {
   boolean flag=false;
   if (x>0) { // c1
      if (x>-1) { // c2
         flag=true;
      }
   }
  return flag;
}
```

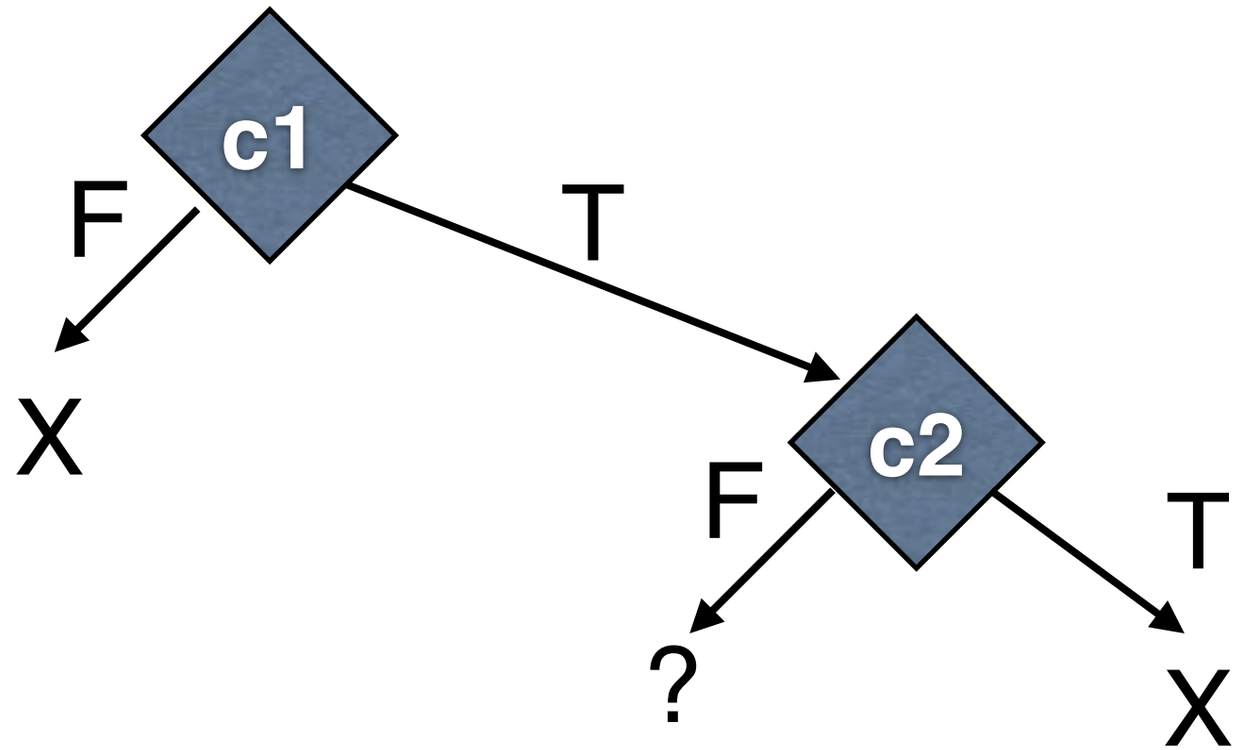
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0	!(x0>0)	x0>0	x0=1
2	x=1	(x0>0) && (x0>-1)		
3				
4				

```
boolean test_me(int x) {
   boolean flag=false;
   if (x>0) { // c1
      if (x>-1) { // c2
         flag=true;
      }
   }
  return flag;
}
```

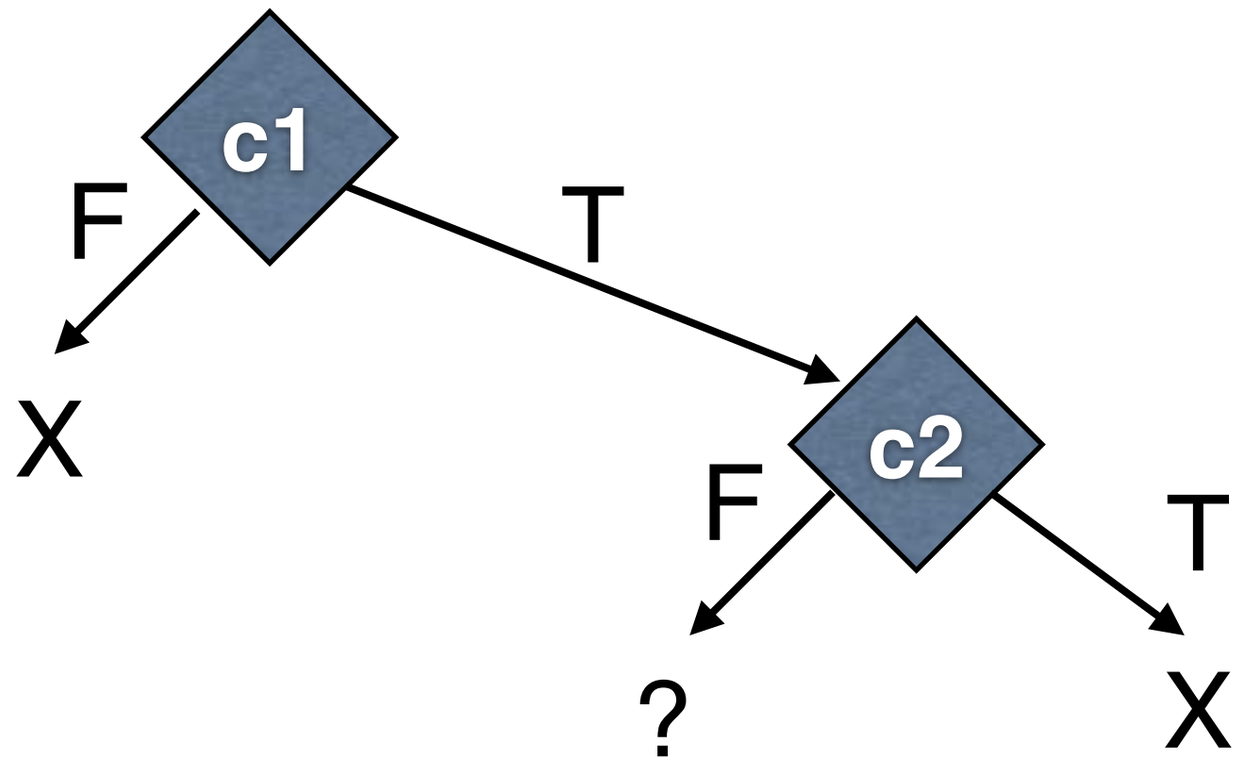
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0	!(x0>0)	x0>0	x0=1
2	x=1	(x0>0) && (x0>-1)	?	
3				
4				

```
boolean test_me(int x) {
   boolean flag=false;
   if (x>0) { // c1
      if (x>-1) { // c2
         flag=true;
      }
   }
  return flag;
}
```

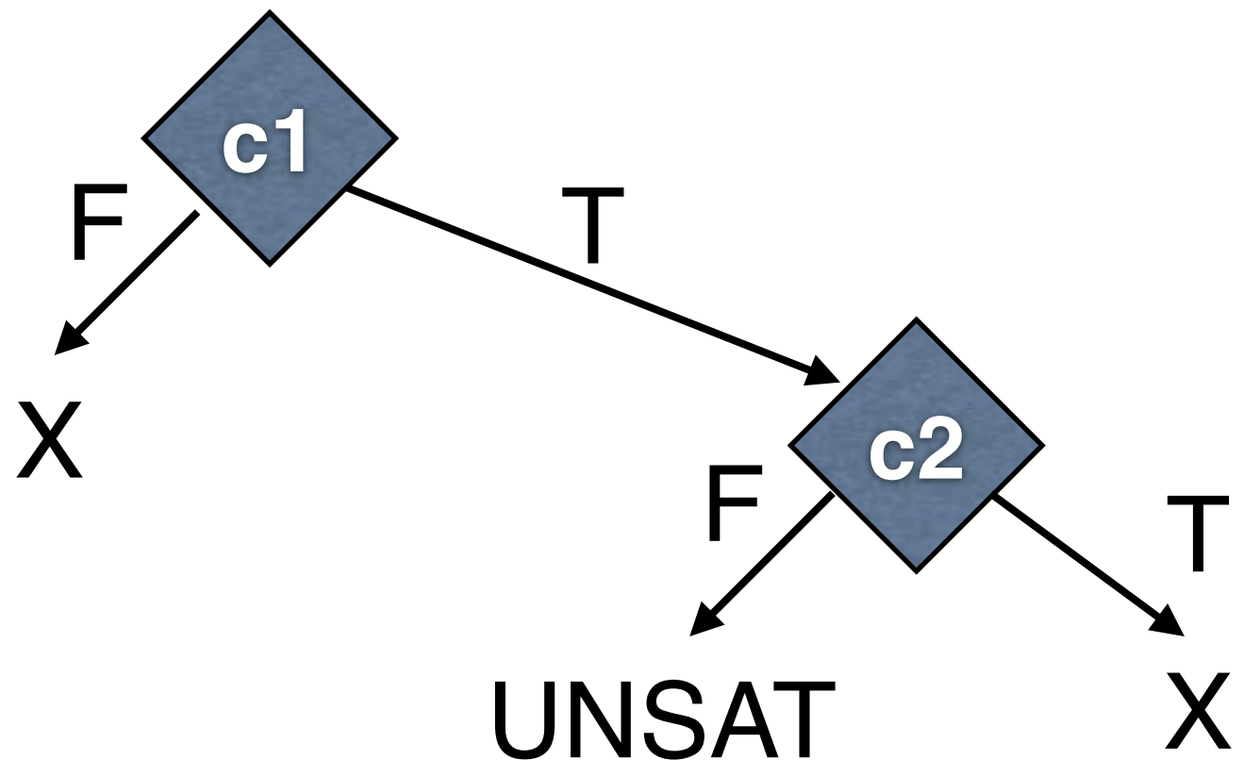
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0	!(x0>0)	x0>0	x0=1
2	x=1	(x0>0) && (x0>-1)	(x0>0) && !(x0>-1)	?
3				
4				

```
boolean test_me(int x) {
   boolean flag=false;
   if (x>0) { // c1
      if (x>-1) { // c2
         flag=true;
      }
   }
  return flag;
}
```

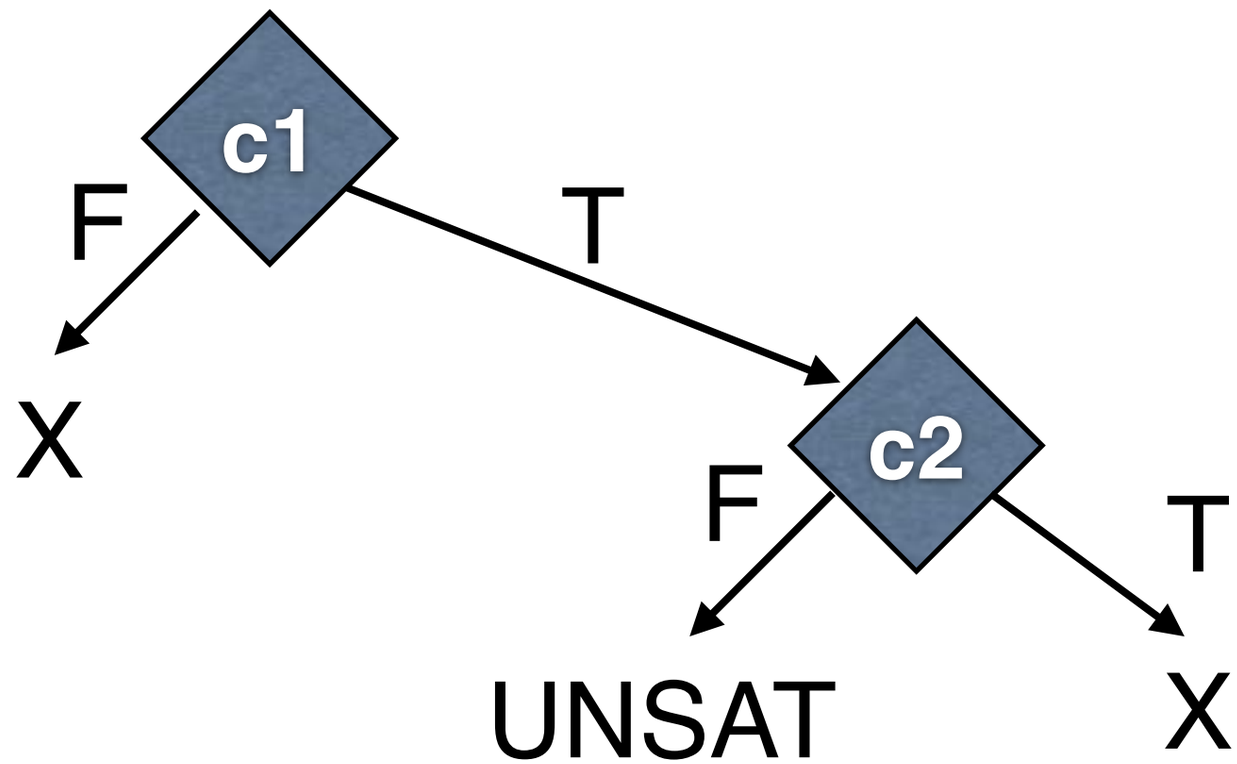
árbol de cómputo explorado



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0	!(x0>0)	x0>0	x0=1
2	x=1	(x0>0) && (x0>-1)	(x0>0) && !(x0>-1)	UNSAT
3				
4				

```
boolean test_me(int x) {
   boolean flag=false;
   if (x>0) { // c1
      if (x>-1) { // c2
         flag=true;
      }
   }
  return flag;
}
```

árbol de cómputo explorado

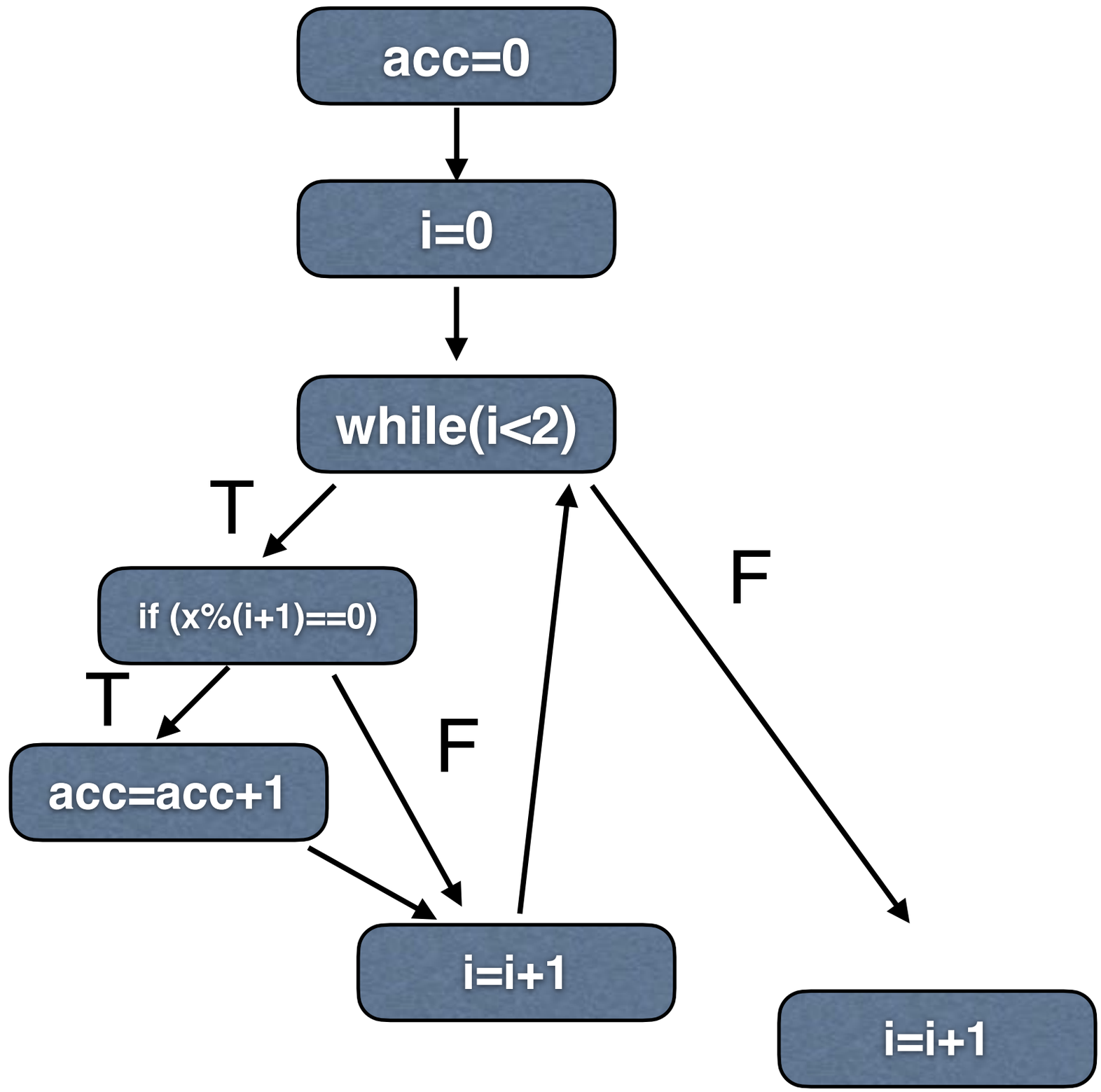


Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0	!(x0>0)	x0>0	x0=1
2	x=1	(x0>0) && (x0>-1)	(x0>0) && !(x0>-1)	UNSAT
3	FIN			
4				

con ciclos

CFG de test_me

```
int test_me(int x) {
  int acc=0;
  int i=0;
  while (i<2) { //c1
     if (x%(i+1))==0 { //c2
        acc=acc+1;
     }
     i=i+1;
  }
  return acc;
}
```



Ciclos/Recursión

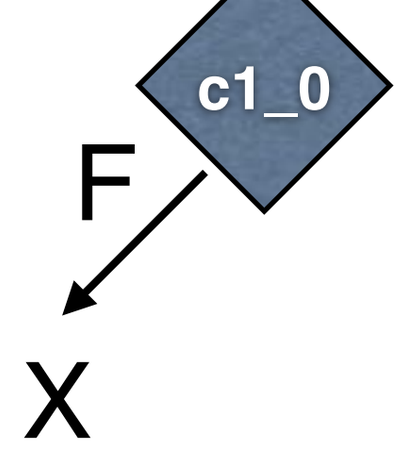
- El árbol de cómputo de un programa con ciclos/recursión es potencialmente infinito.
- Por este motivo, debemos limitar la cantidad de veces que un ciclo se ejecuta.
- Cada vez que tengamos un ciclo/recursión, limitaremos la profundidad del árbol de cómputo durante la exploración.
- Por ejemplo, si U=1, a lo sumo un ciclo se puede ejecutar una sola vez.
- Para esto, contamos la cantidad de veces que el arco TRUE de un ciclo se ejecutó en el árbol de cómputo

```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
     if (x%(i+1))==0 { //c2
        acc=acc+1;
     }
     i=i+1;
  }
  return acc;
}
```

Max u(c1)=1

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0			
2				
3				
4				

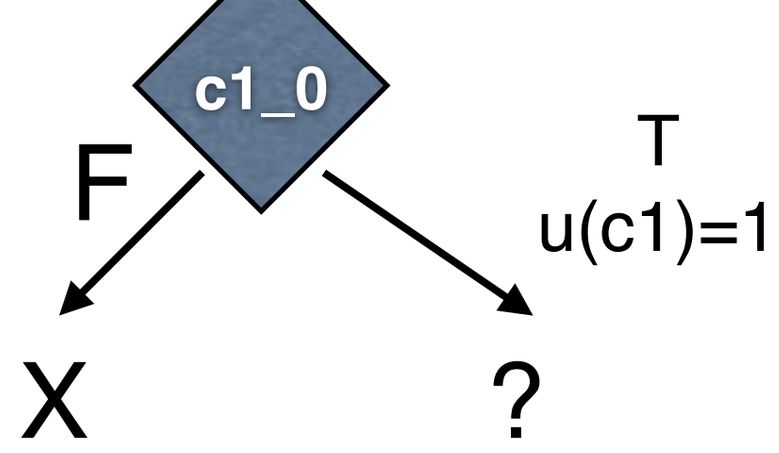
```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
     if (x%(i+1))==0 { //c2
        acc=acc+1;
     }
     i=i+1;
  }
  return acc;
}
```



Max u(c1)=1

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0	!(0<k0)		
2				
3				
4				

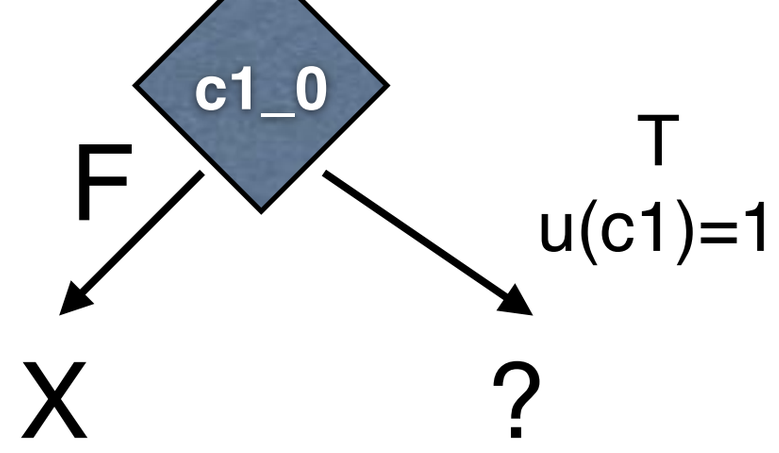
```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
     if (x%(i+1))==0 { //c2
        acc=acc+1;
     }
     i=i+1;
  }
  return acc;
}
```



Max u(c1)=1

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0	!(0<k0)	?	
2				
3				
4				

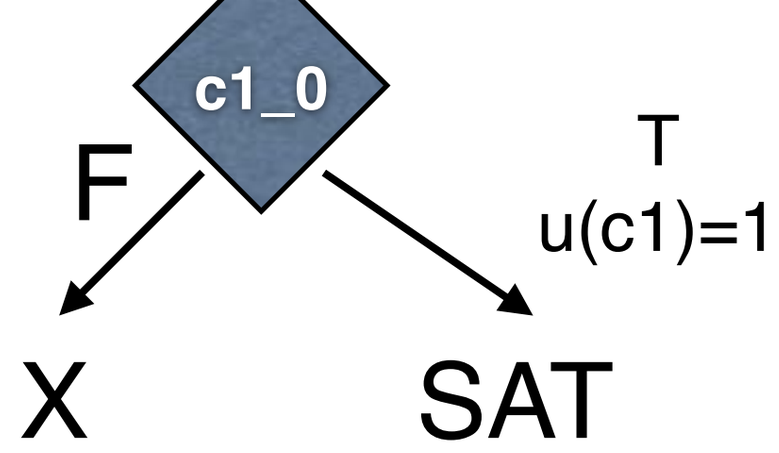
```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
    if (x%(i+1))==0 { //c2
      acc=acc+1;
    }
    i=i+1;
  }
  return acc;
}
```



Max u(c1)=1

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0	!(0<k0)	0<k0	?
2				
3				
4				

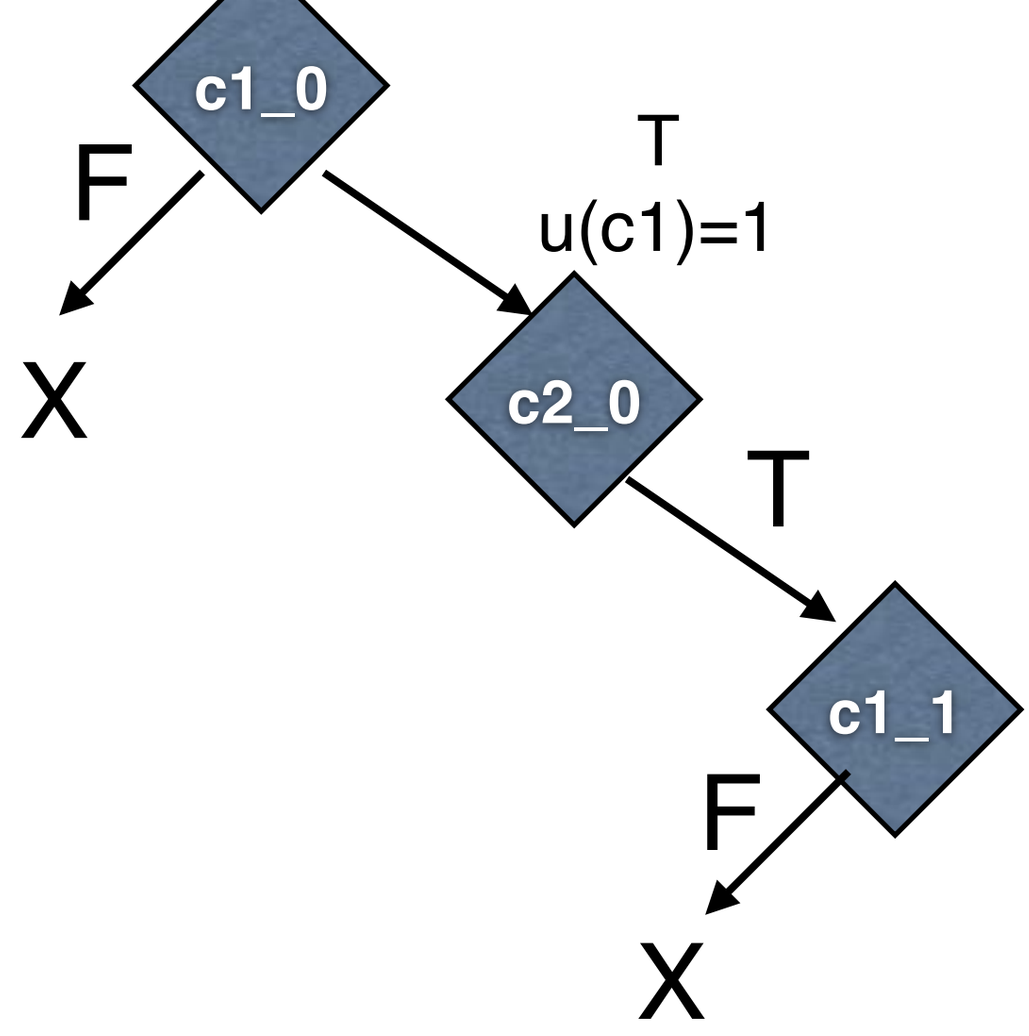
```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
    if (x%(i+1))==0 { //c2
      acc=acc+1;
    }
    i=i+1;
  }
  return acc;
}
```



Max u(c1)=1

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0	!(0<k0)	0<k0	k0=1
2				
3				
4				

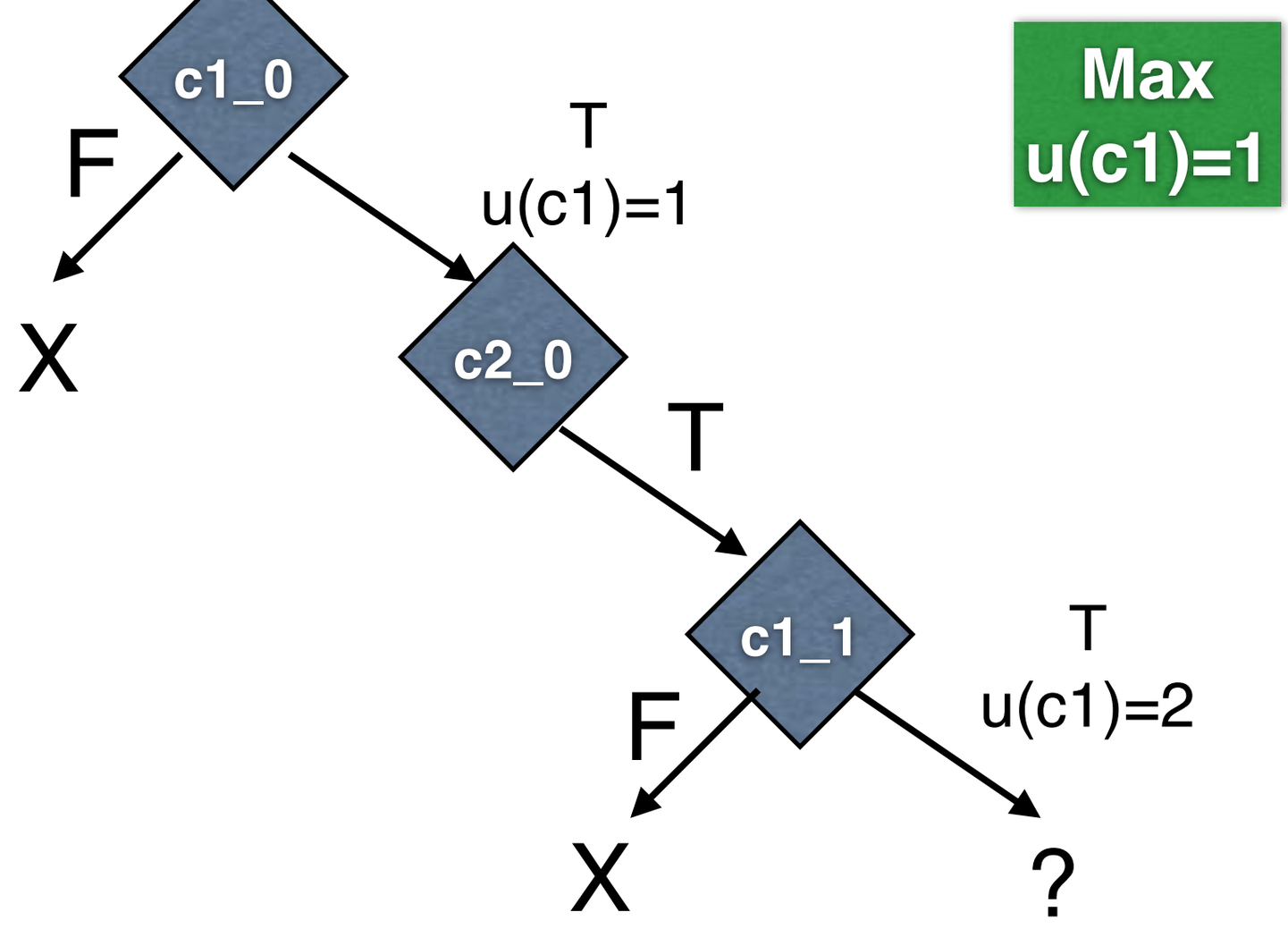
```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
     if (x%(i+1))==0 { //c2
        acc=acc+1;
     }
     i=i+1;
  }
  return acc;
}
```



Max u(c1)=1

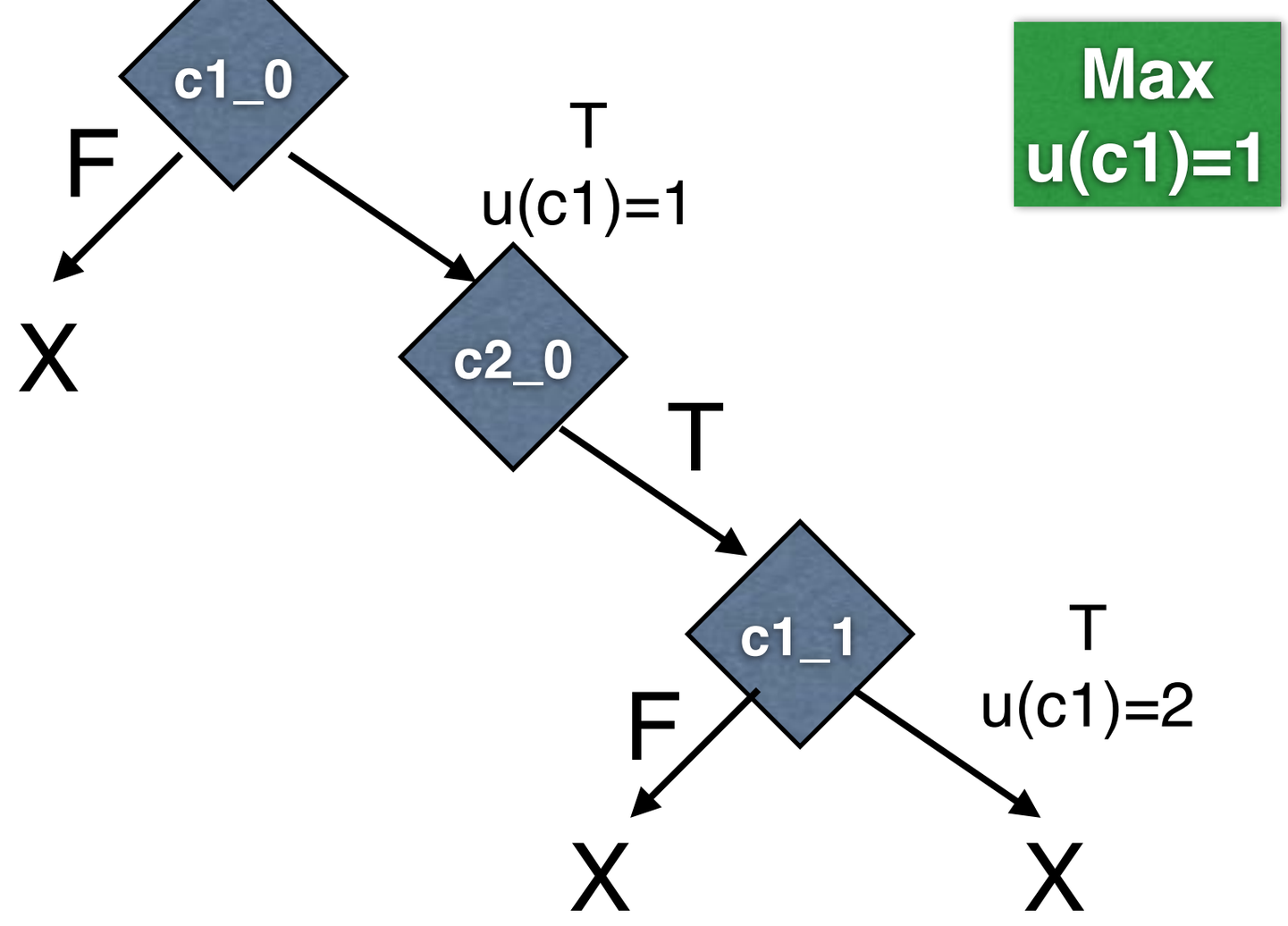
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0	!(0<k0)	0<k0	k0=1
2	x=0,k=1	(0<k0) && (x0%1==0) && !(1<k0)		
3				
4				

```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
     if (x%(i+1))==0 { //c2
        acc=acc+1;
     }
     i=i+1;
  }
  return acc;
}
```



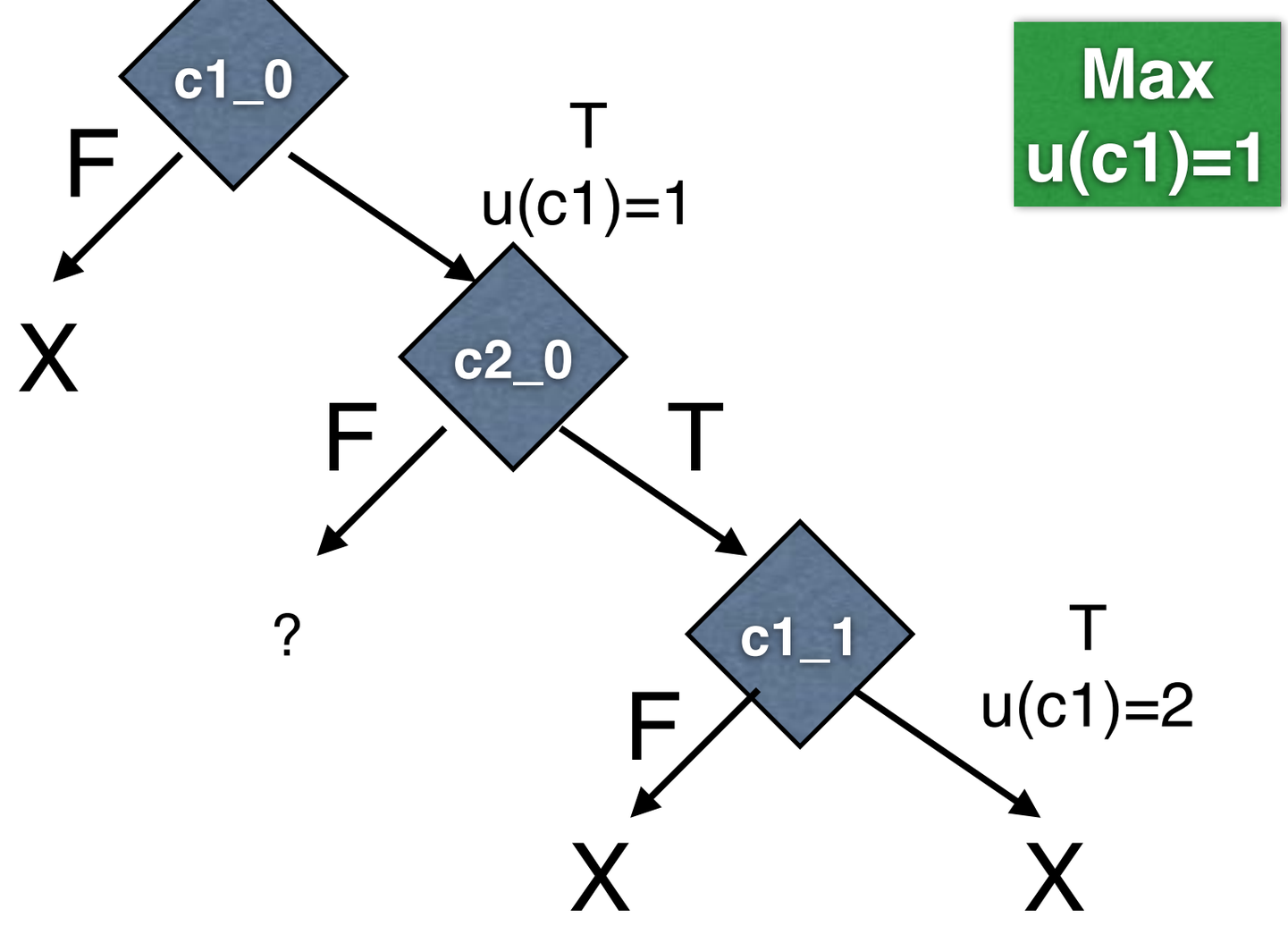
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0	!(0<k0)	0<k0	k0=1
2	x=0,k=1	(0<k0) && (x0%1==0) && !(1<k0)		
3				
4				

```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
     if (x%(i+1))==0 { //c2
        acc=acc+1;
     }
     i=i+1;
  }
  return acc;
}
```



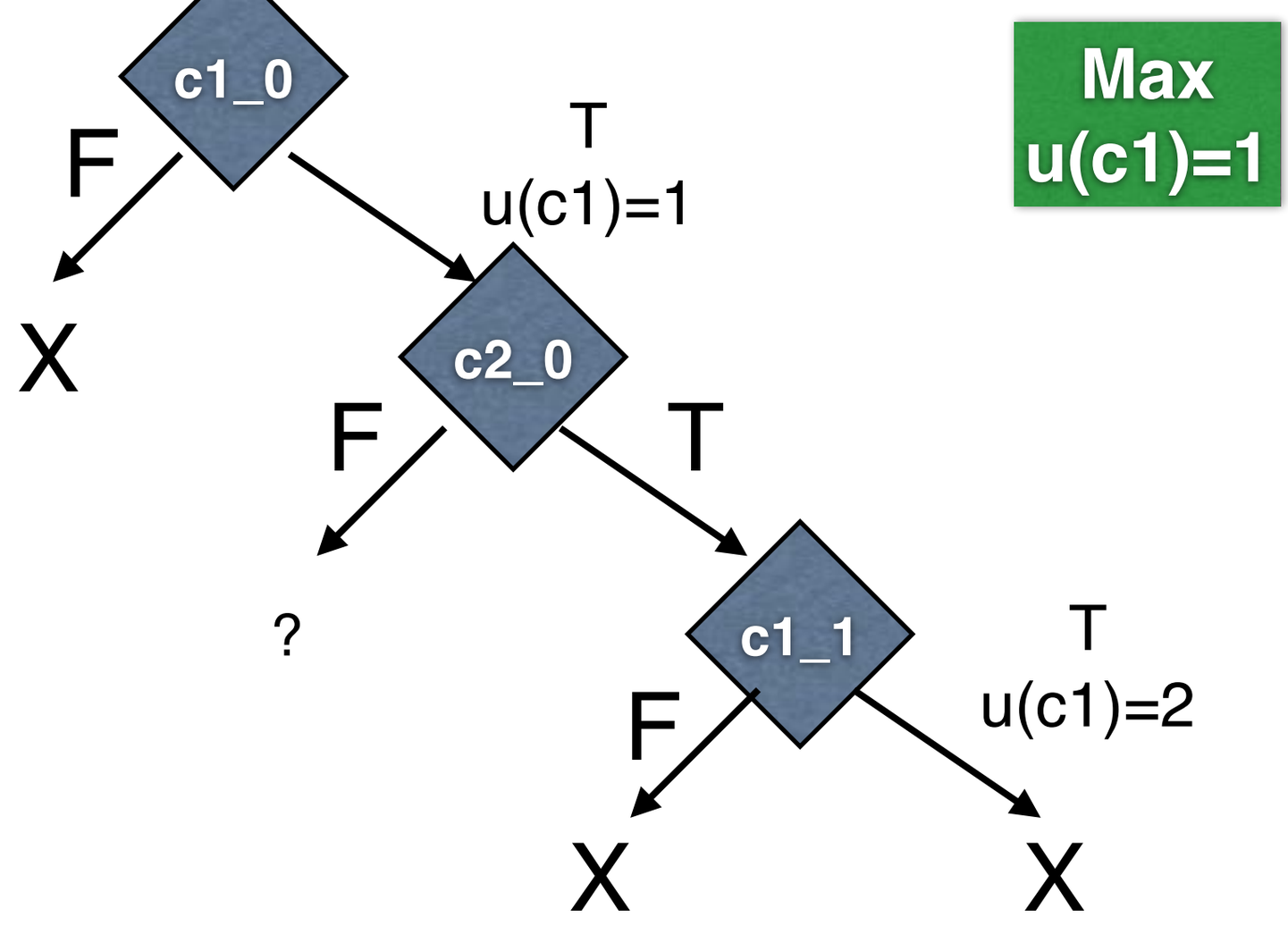
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0	!(0<k0)	0<k0	k0=1
2	x=0,k=1	(0<k0) && (x0%1==0) && !(1<k0)		
3				
4				

```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
     if (x%(i+1))==0 { //c2
        acc=acc+1;
     }
     i=i+1;
  }
  return acc;
}
```



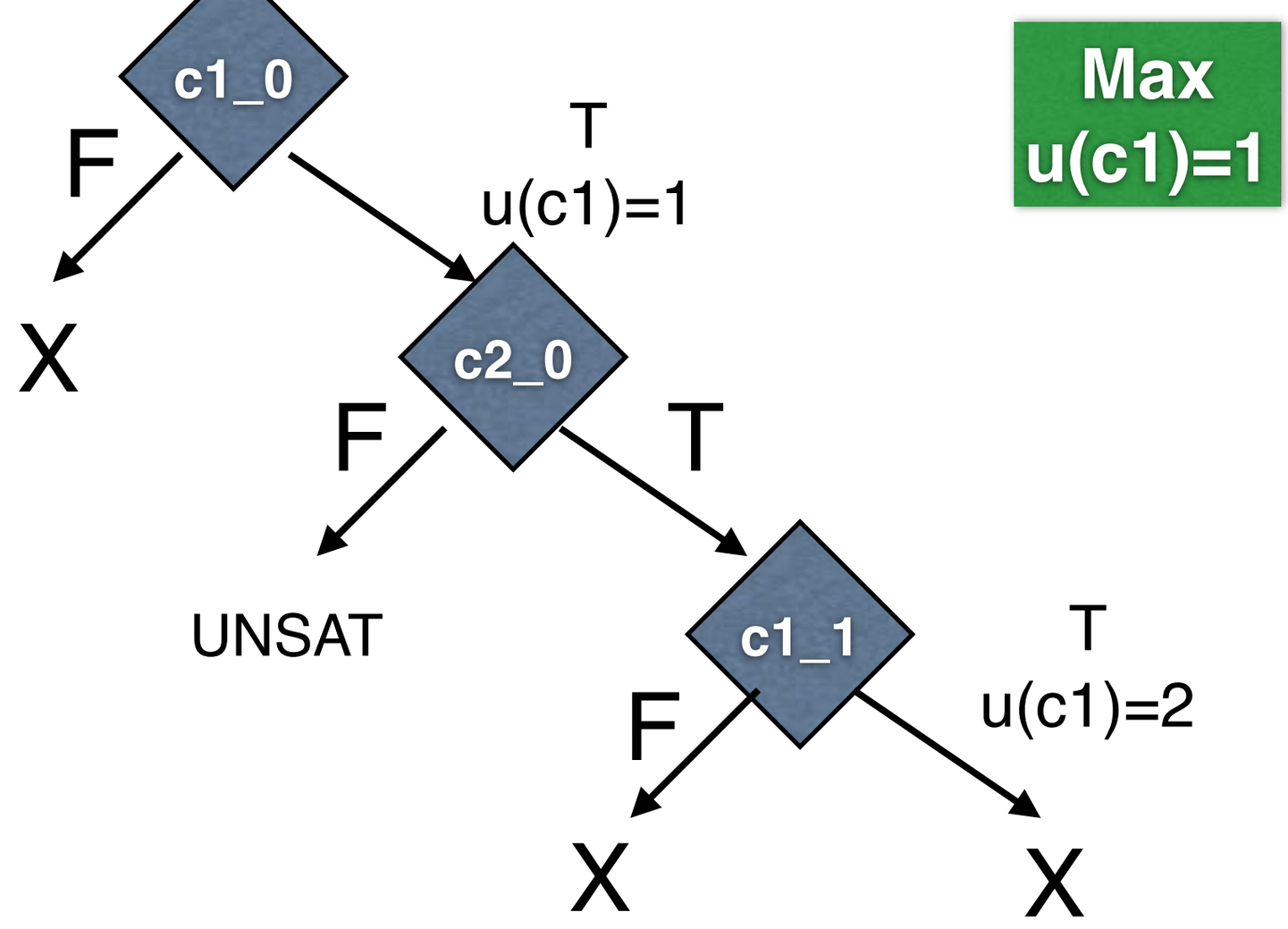
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0	!(0<k0)	0<k0	k0=1
2	x=0,k=1	(0<k0) && (x0%1==0) && !(1<k0)	?	
3				
4				

```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
     if (x%(i+1))==0 { //c2
        acc=acc+1;
     }
     i=i+1;
  }
  return acc;
}
```



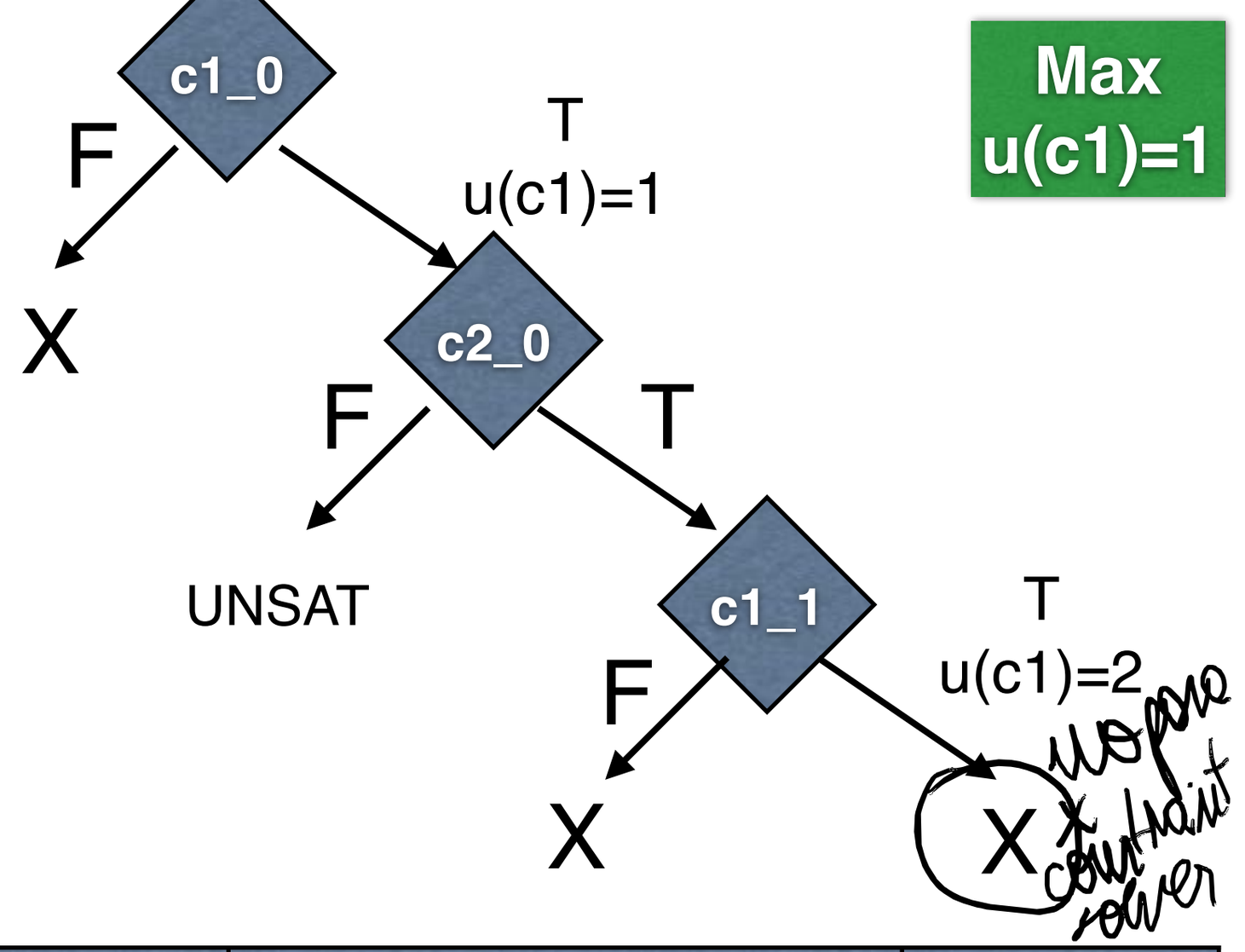
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0	!(0<k0)	0<k0	k0=1
2	x=0,k=1	(0<k0) && (x0%1==0) && !(1<k0)	(0<k0) && !(x0%1==0)	?
3				
4				

```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
     if (x%(i+1))==0 { //c2
        acc=acc+1;
     }
     i=i+1;
  }
  return acc;
}
```



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0	!(0<k0)	0<k0	k0=1
2	x=0,k=1	(0<k0) && (x0%1==0) && !(1<k0)	(0<k0) && !(x0%1==0)	UNSAT
3				
4				

```
int test_me(int x, int k) {
  int acc=0;
  int i=0;
  while (i<k) { //c1
    if (x%(i+1))==0 { //c2
      acc=acc+1;
    }
    i=i+1;
  }
  return acc;
}
```



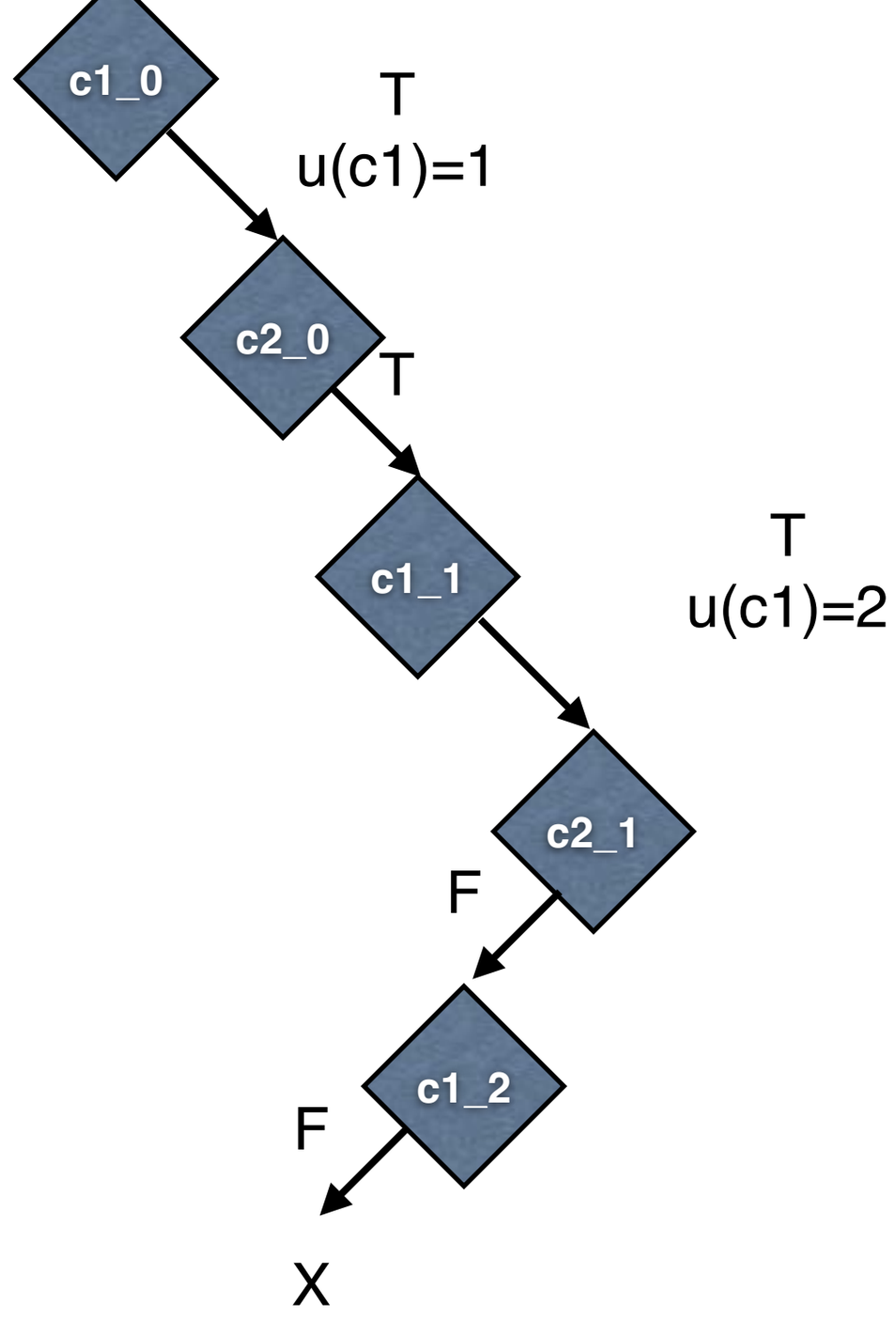
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	x=0,k=0	!(0<k0)	0<k0	k0=1
2	x=0,k=1	(0<k0) && (x0%1==0) && !(1<k0)	(0<k0) && !(x0%1==0)	UNSAT
3	FIN			
4				

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```

Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	?		

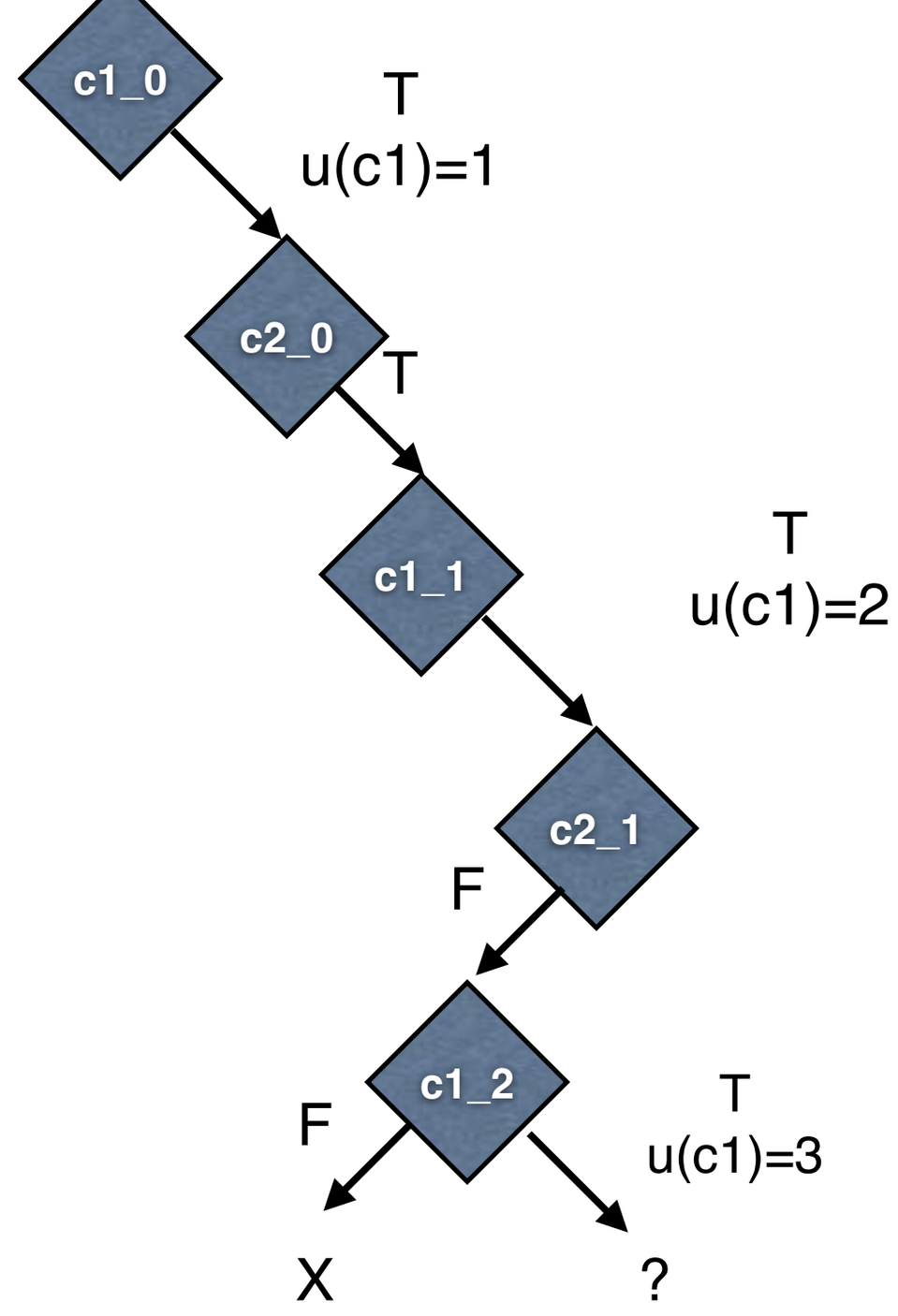
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)		

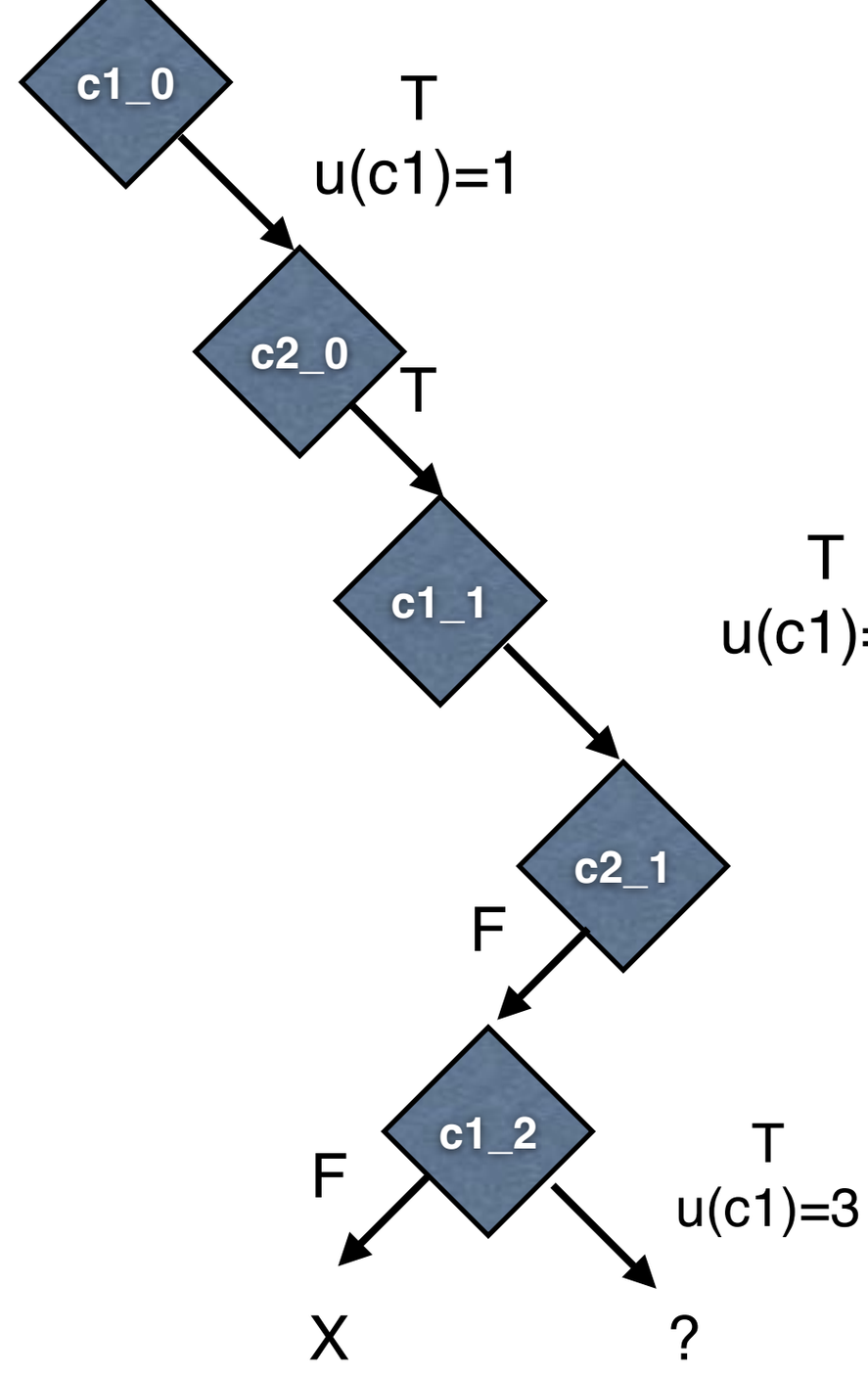
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	?	

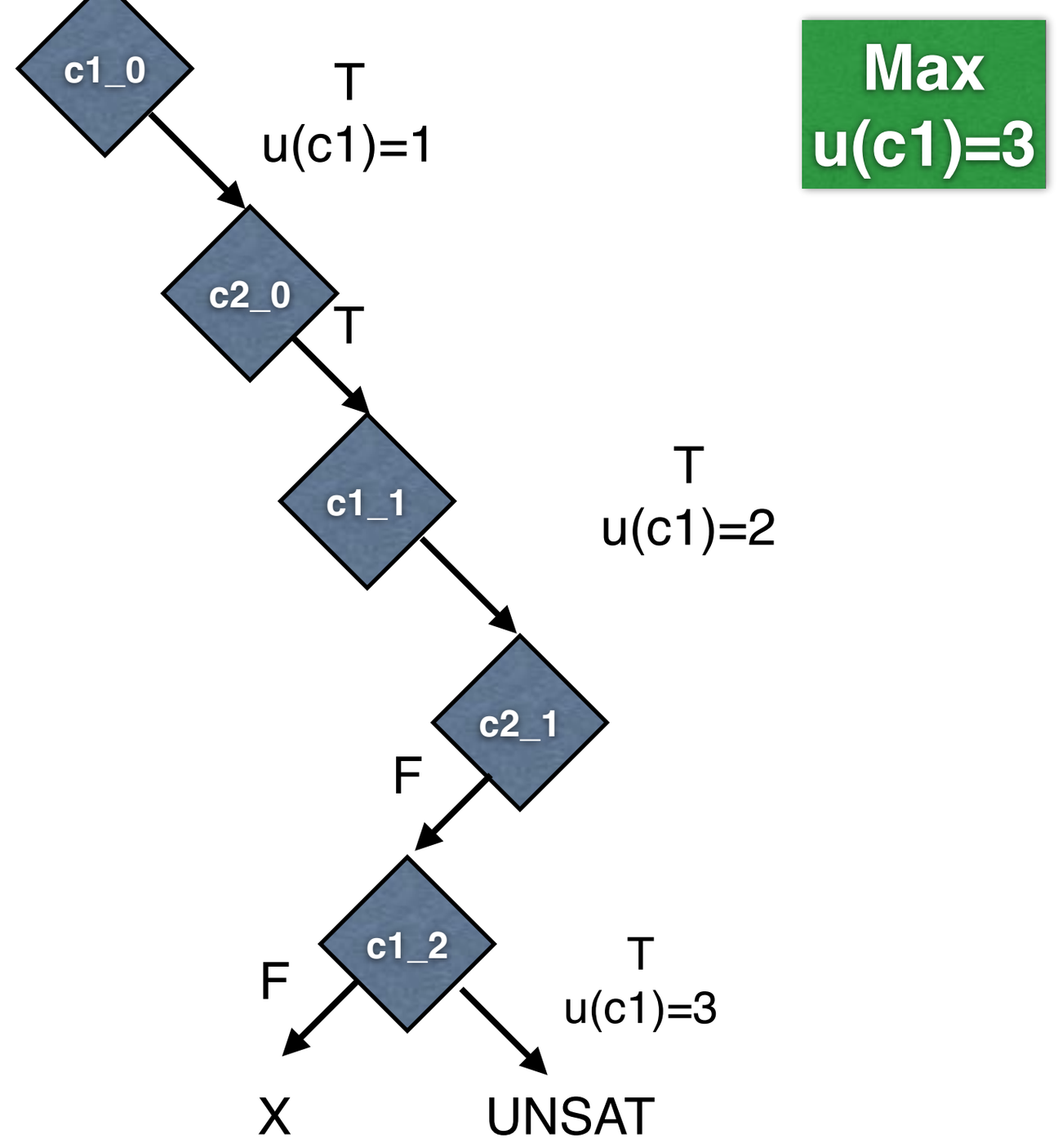
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	?

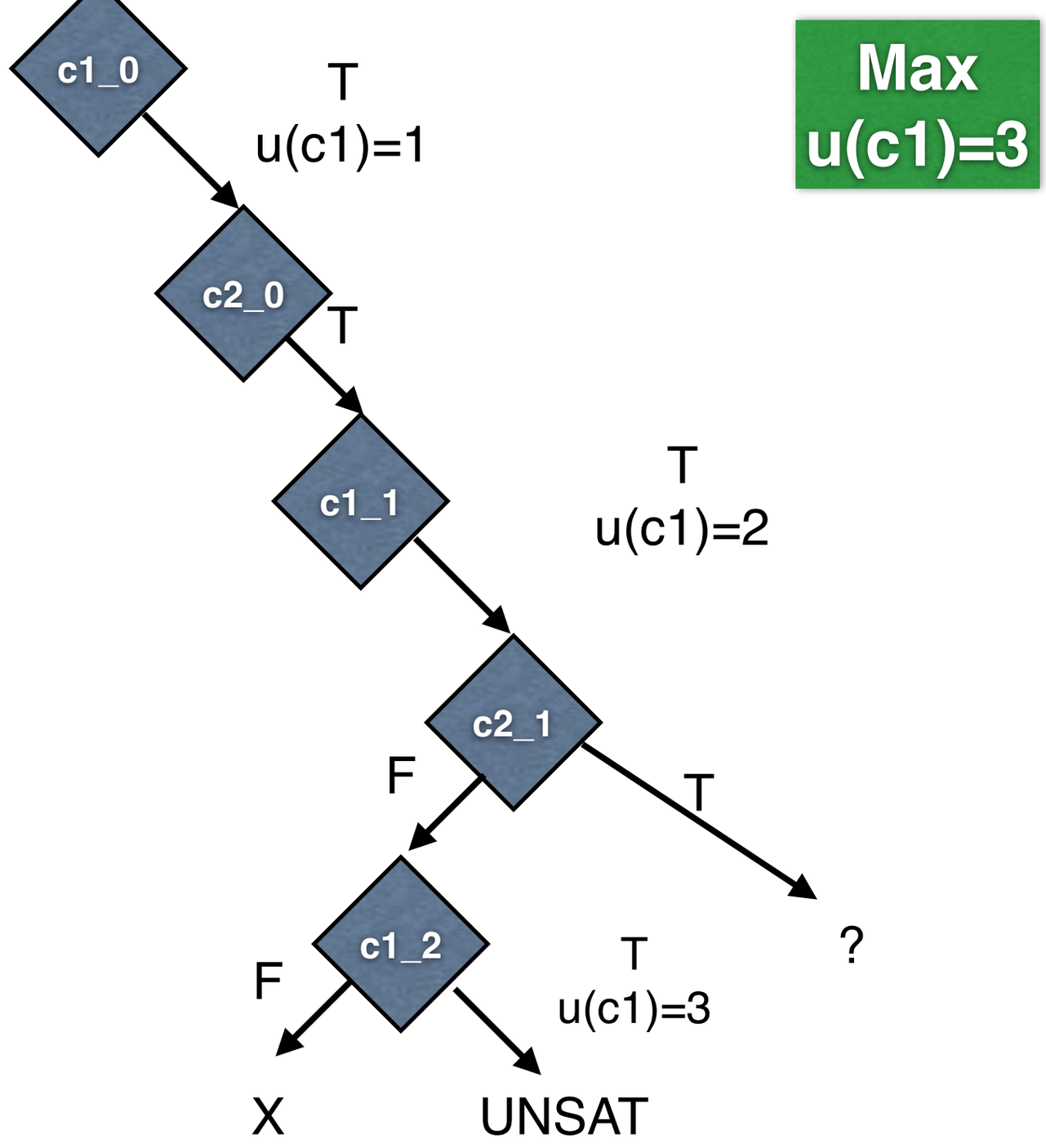
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT

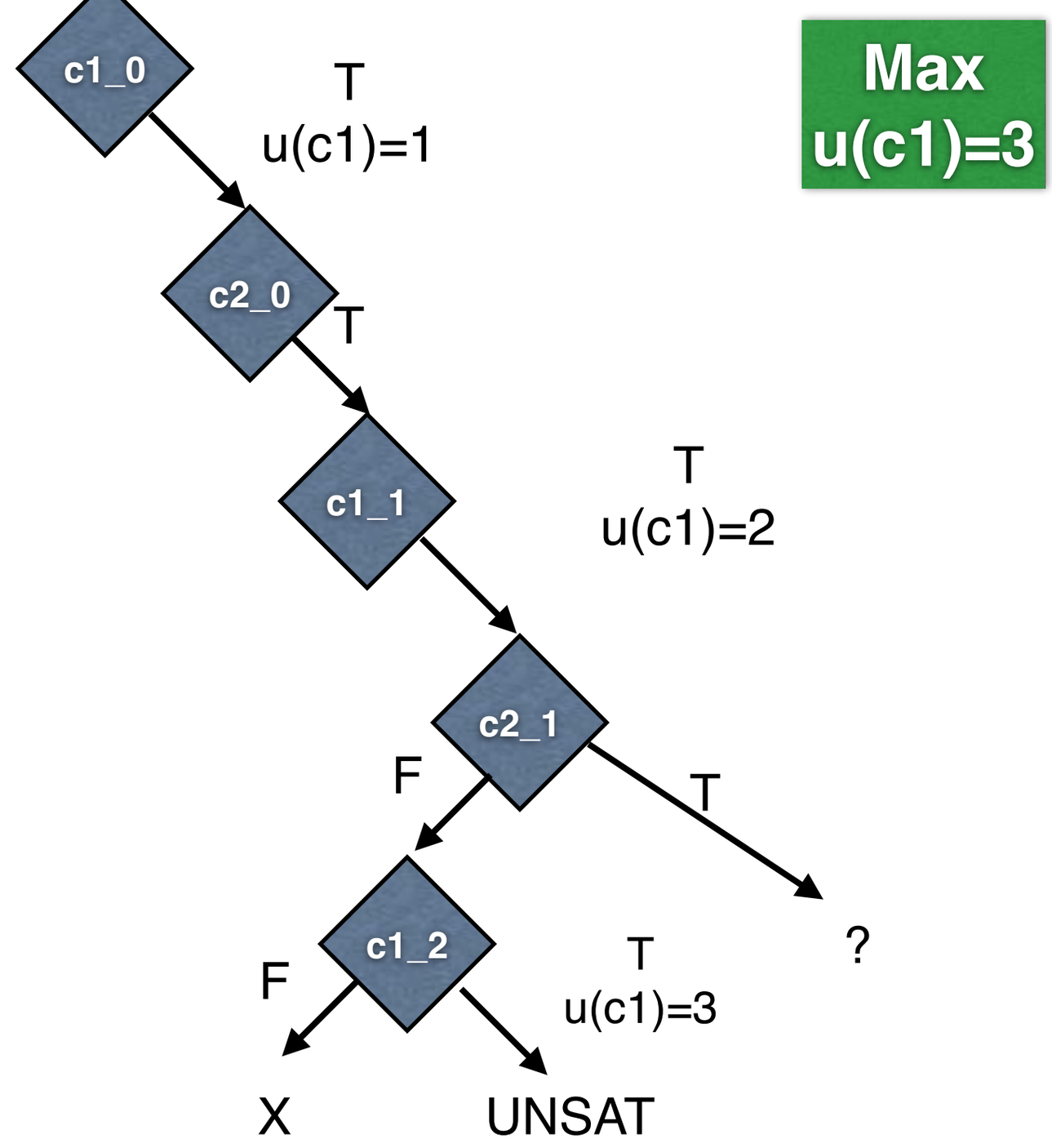
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			?	

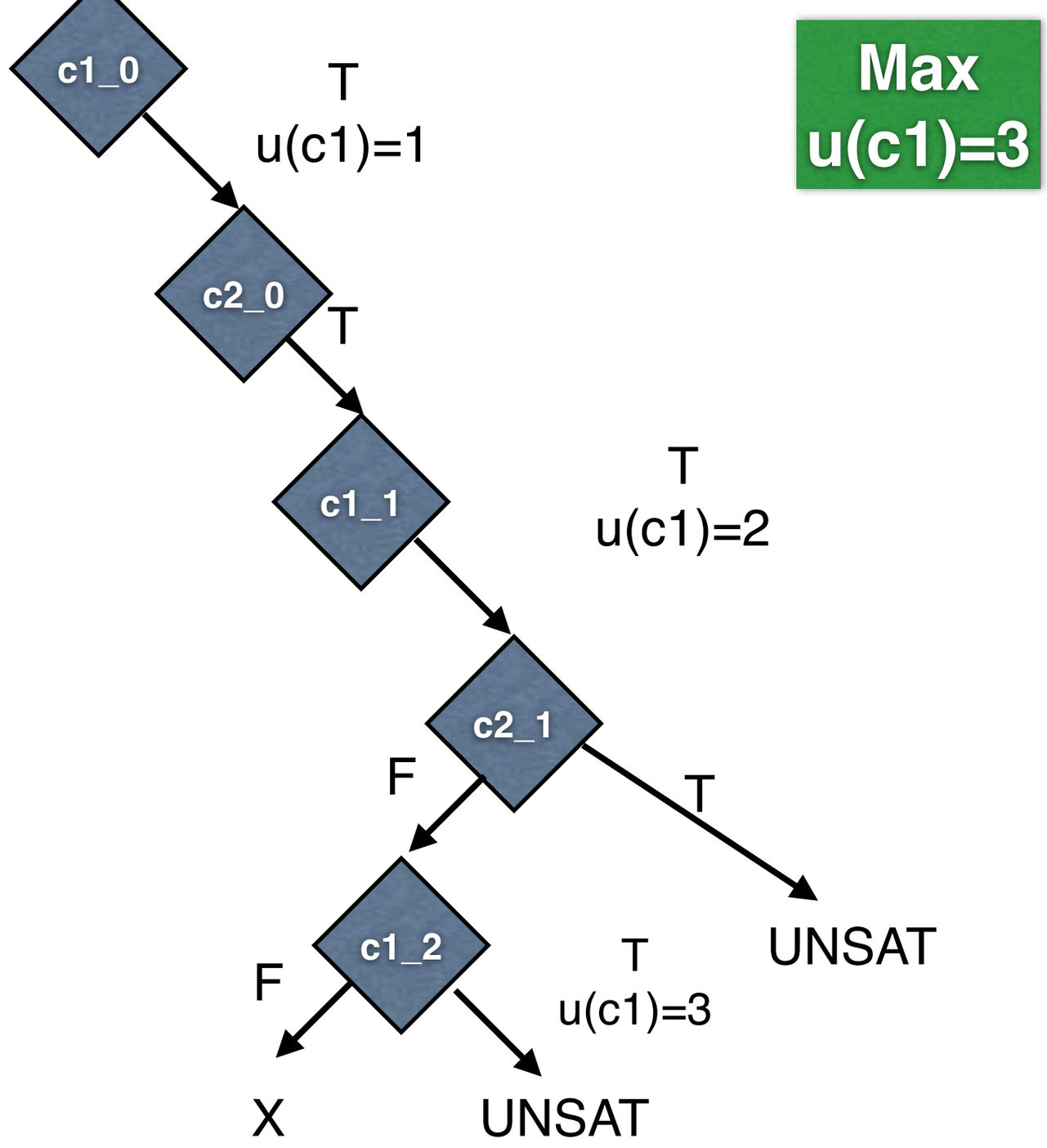
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			(0<2)&&(0==j0)&&(1<2)&&(1==j0)	?

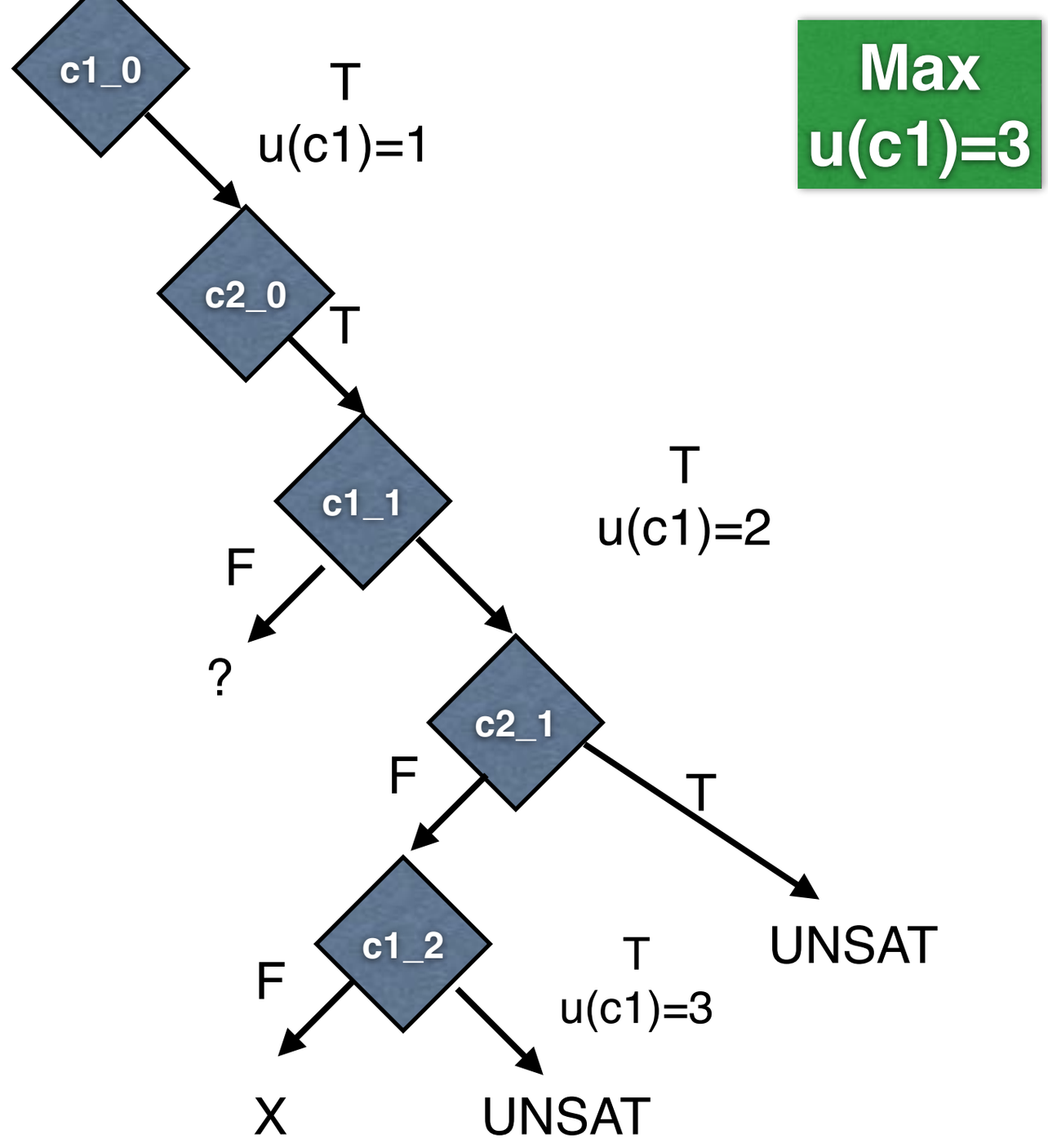
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			(0<2)&&(0==j0)&&(1<2)&&(1==j0)	UNSAT

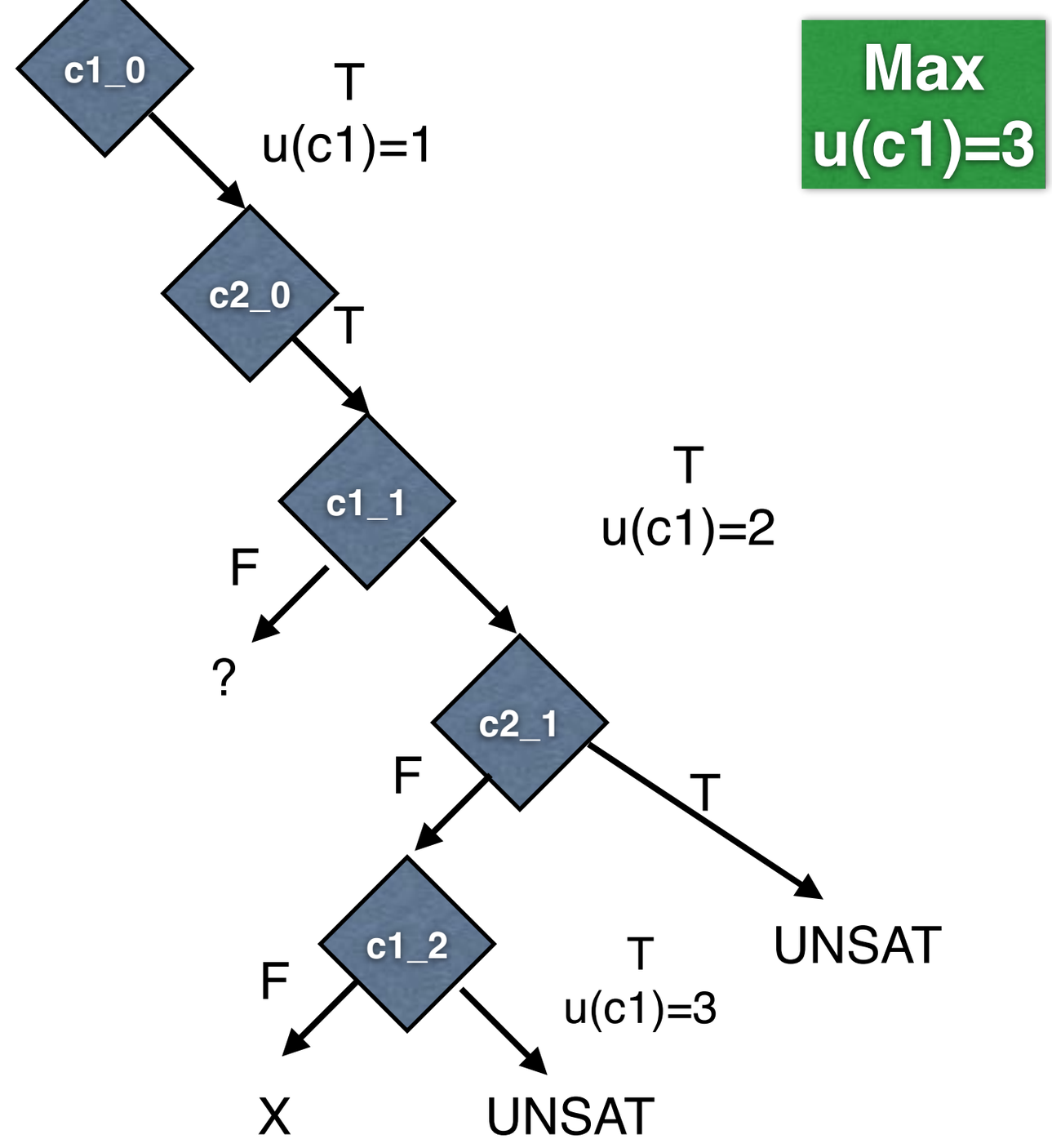
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			(0<2)&&(0==j0)&&(1<2)&&(1==j0)	UNSAT
			?	

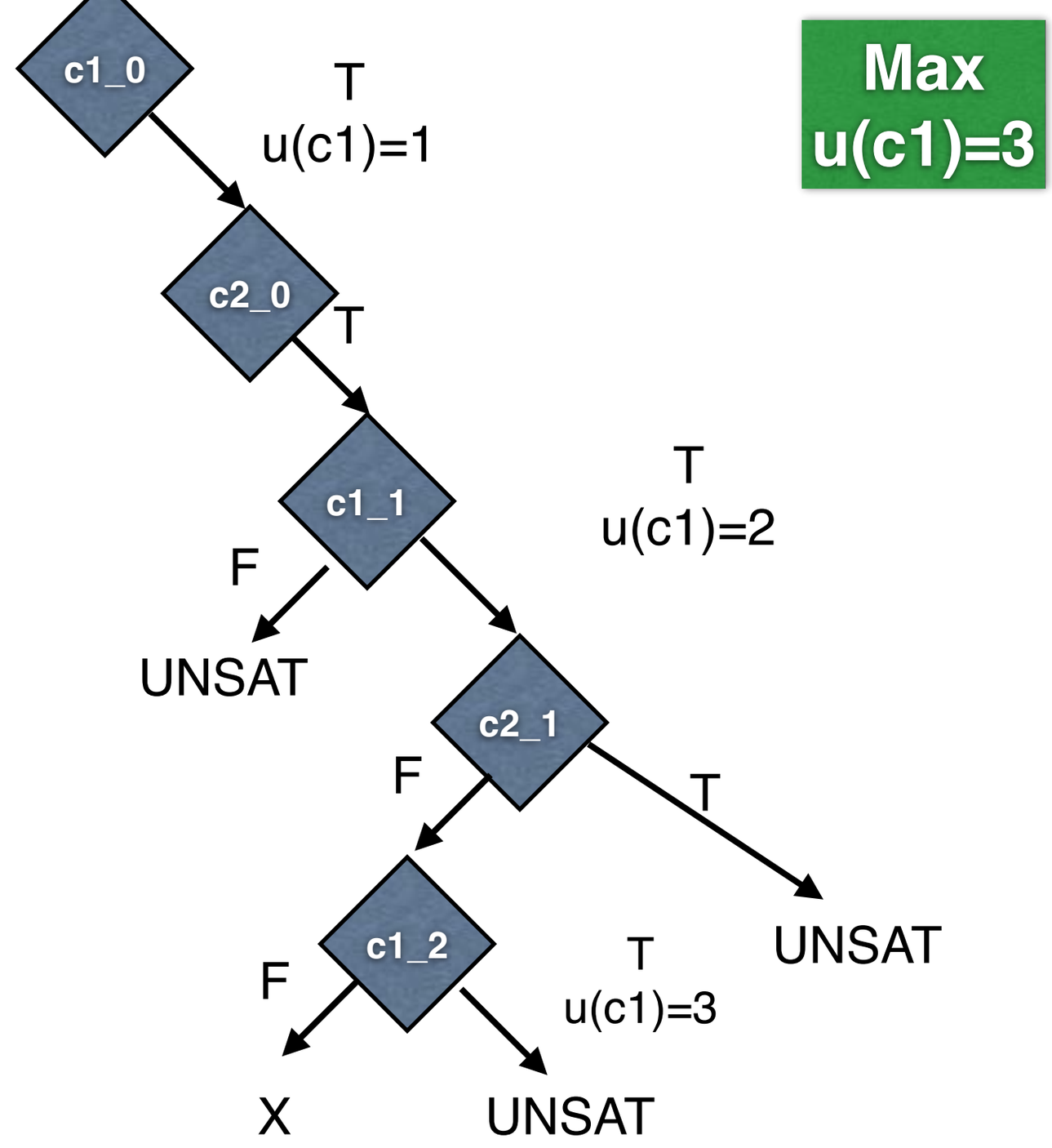
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			(0<2)&&(0==j0)&&(1<2)&&(1==j0)	UNSAT
			(0<2)&&(0==j0)&&!(1<2)	?

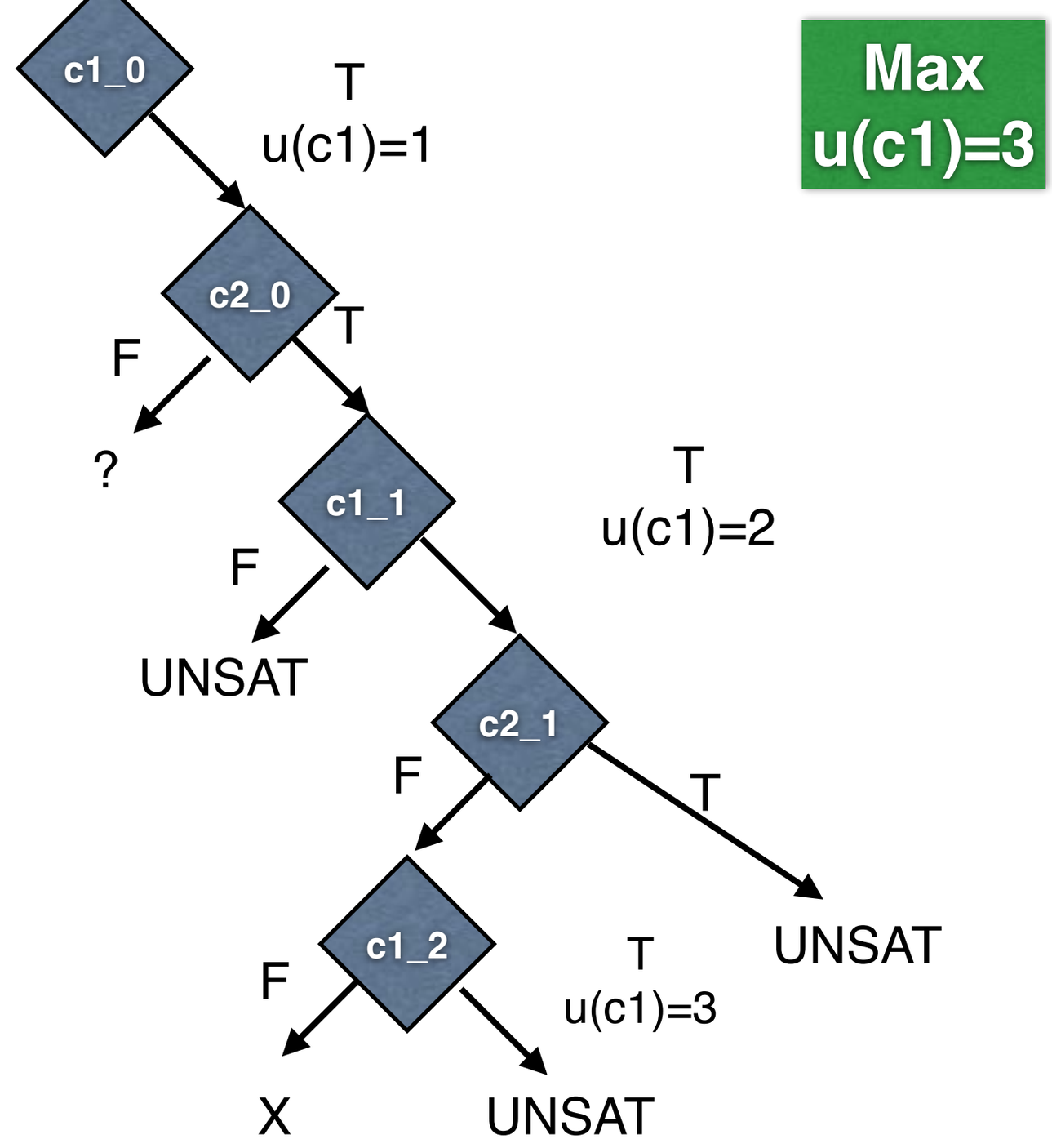
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(1<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(1<2)	UNSAT
			(0<2)&&(0==j0)&&(1<2)&&(1==j0)	UNSAT
			(0<2)&&(0==j0)&&!(1<2)	UNSAT

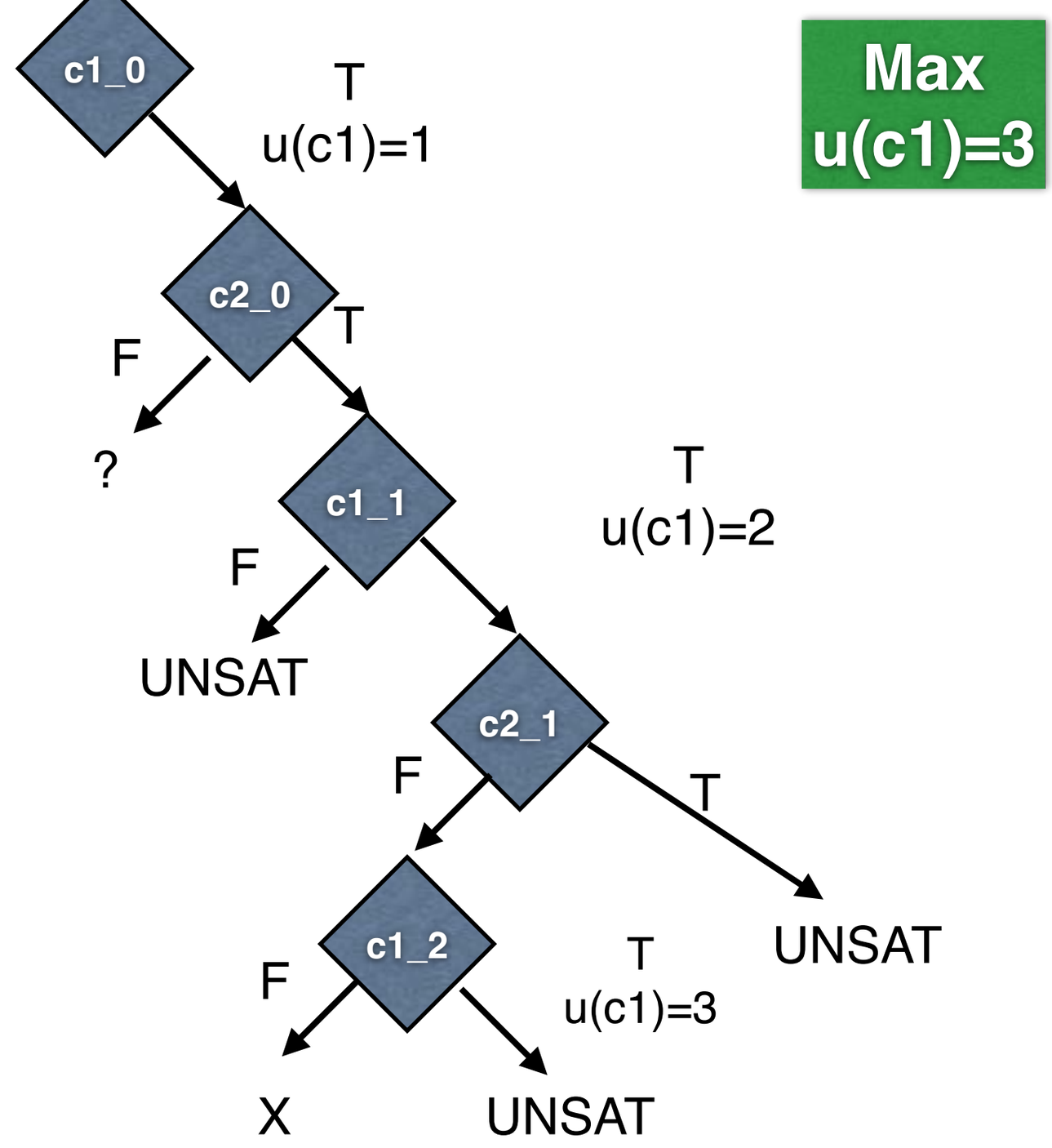
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			(0<2)&&(0==j0)&&(1<2)&&(1==j0)	UNSAT
			(0<2)&&(0==j0)&&!(1<2)	UNSAT
			?	

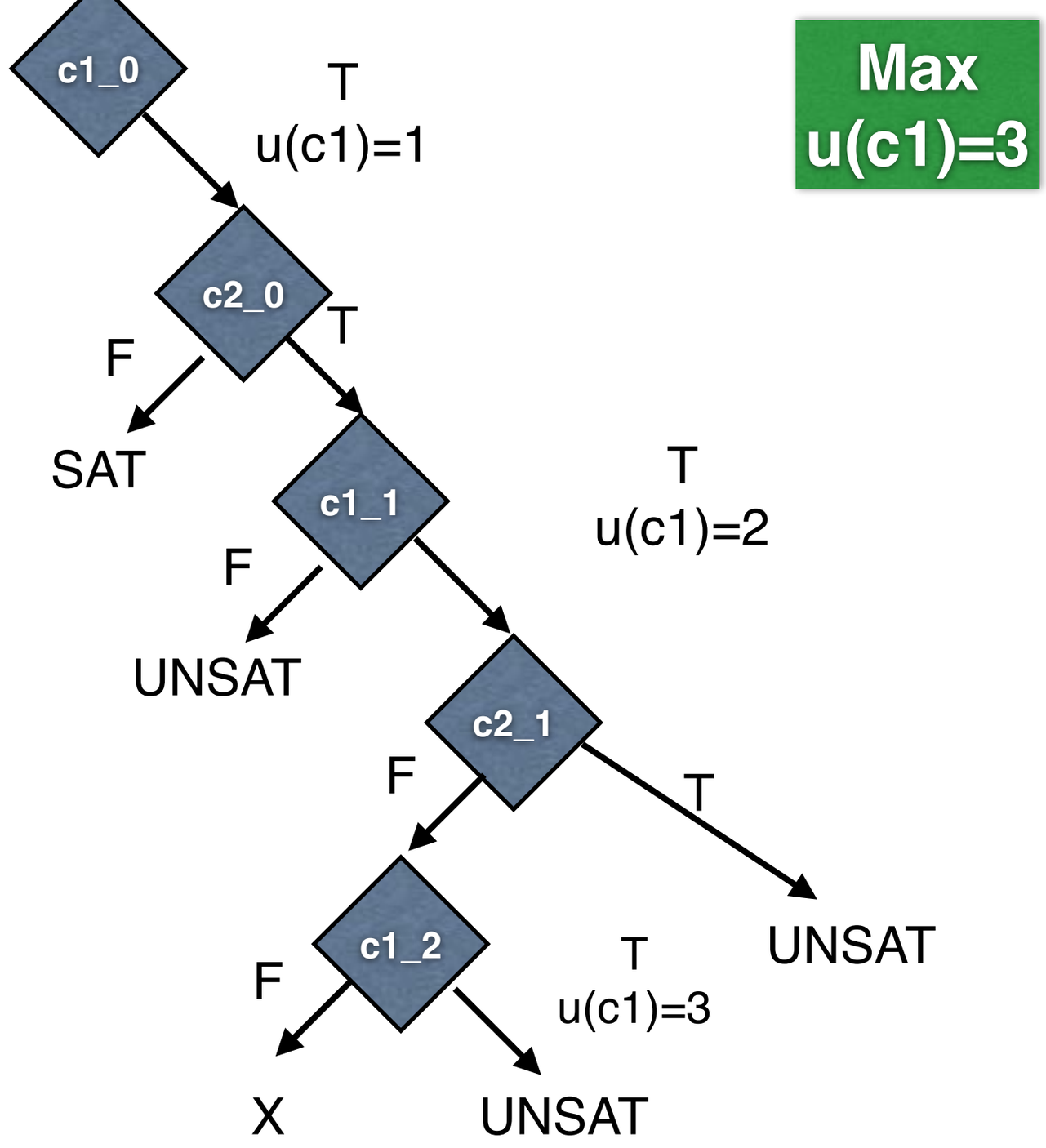
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			(0<2)&&(0==j0)&&(1<2)&&(1==j0)	UNSAT
			(0<2)&&(0==j0)&&!(1<2)	UNSAT
			(0<2)&&!(0==j0)	?

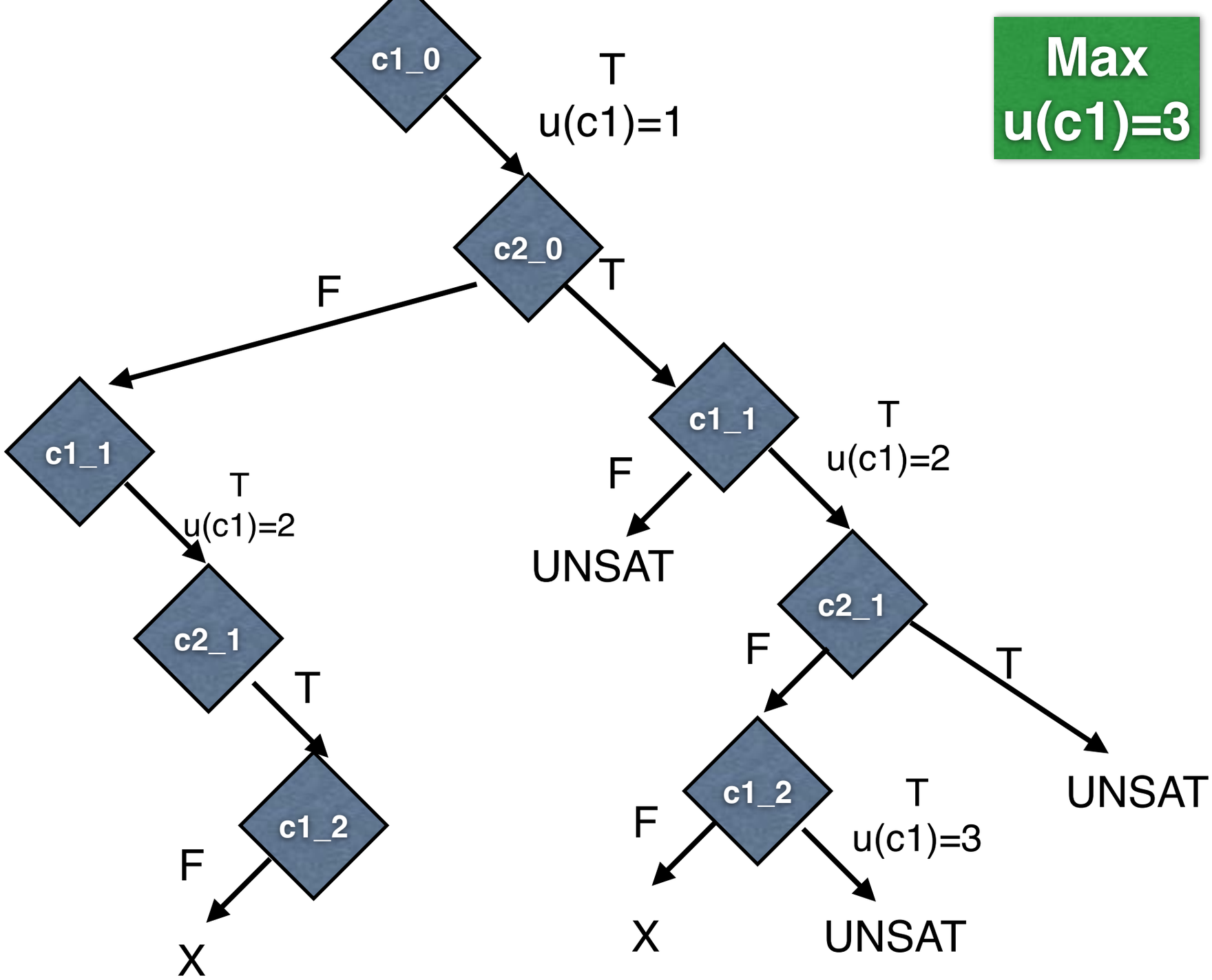
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Max u(c1)=3

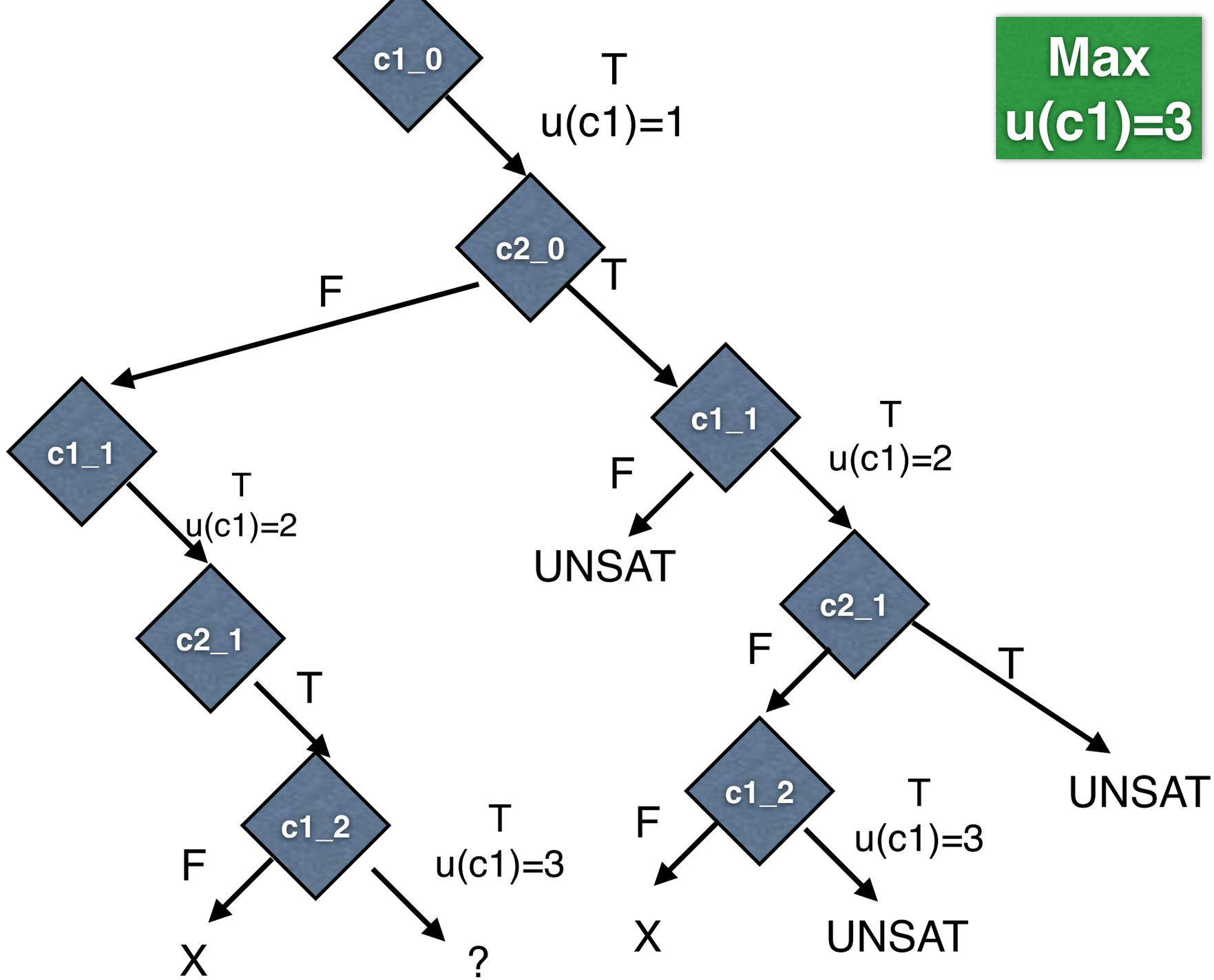
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(1<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(1<2)	UNSAT
			(0<2)&&(0==j0)&&(1<2)&&(1==j0)	UNSAT
			(0<2)&&(0==j0)&&!(1<2)	UNSAT
			(0<2)&&!(0==j0)	j0=1

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



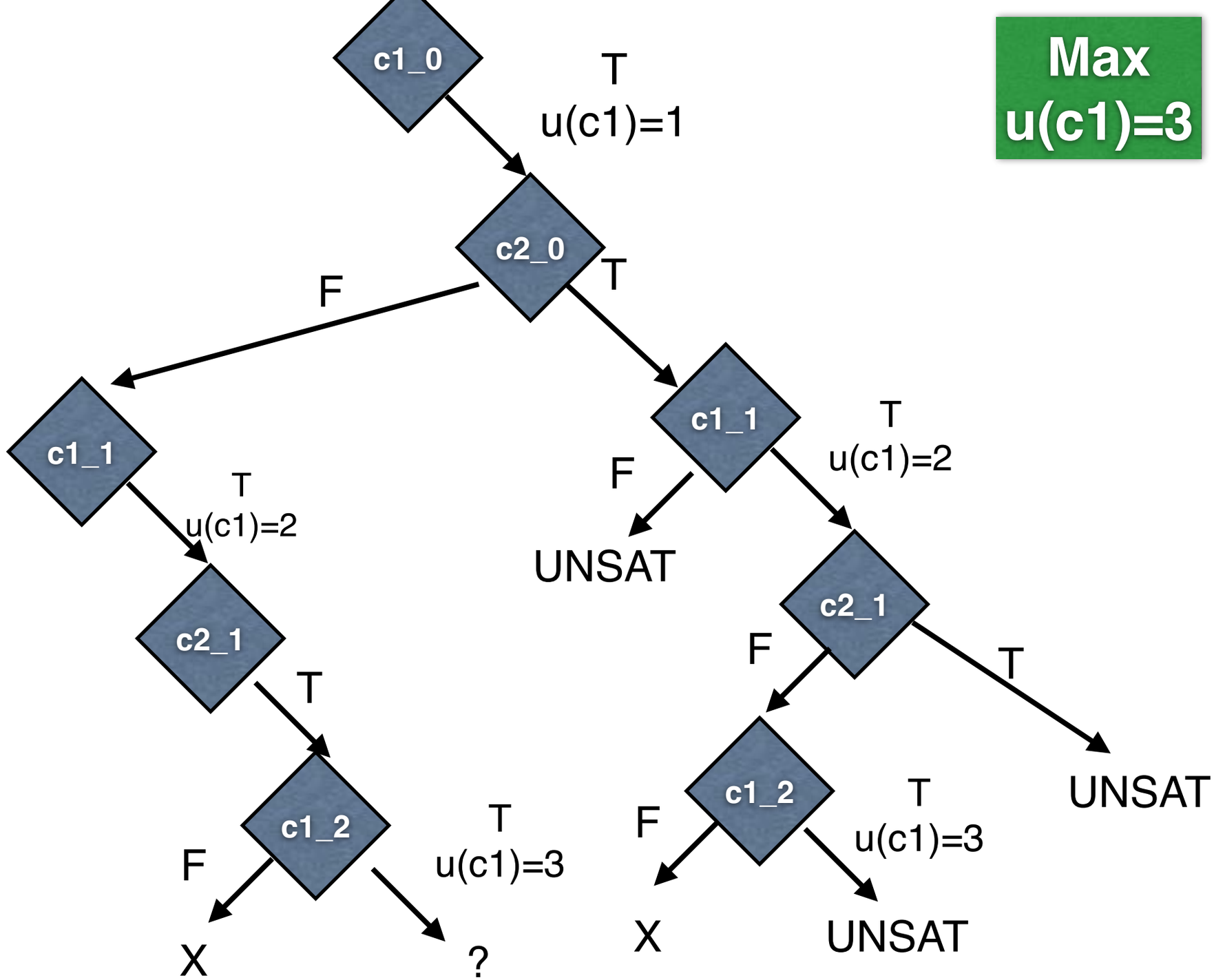
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
2	j=1	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)		

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



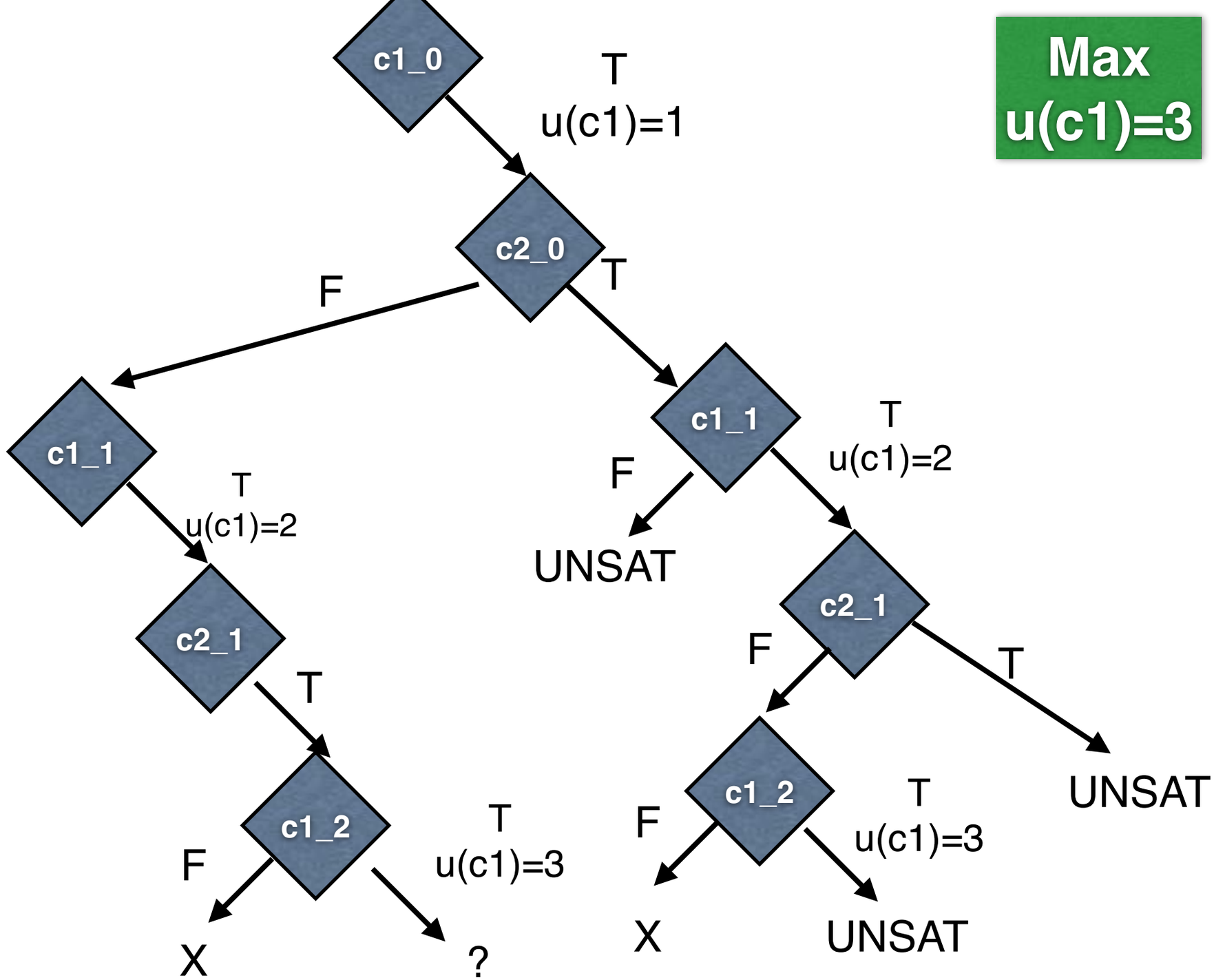
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
2	j=1	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)		

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



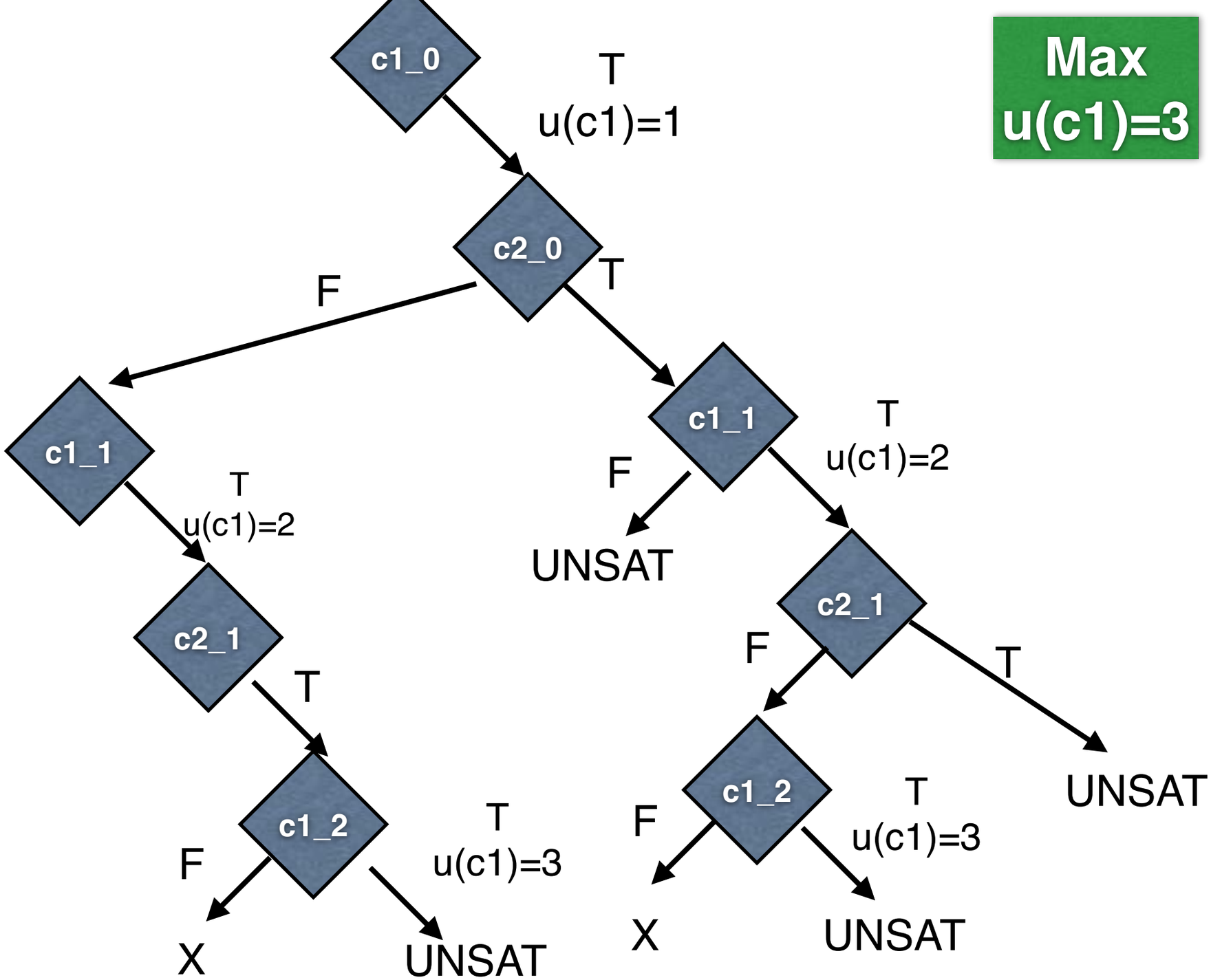
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
2	j=1	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)	?	

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



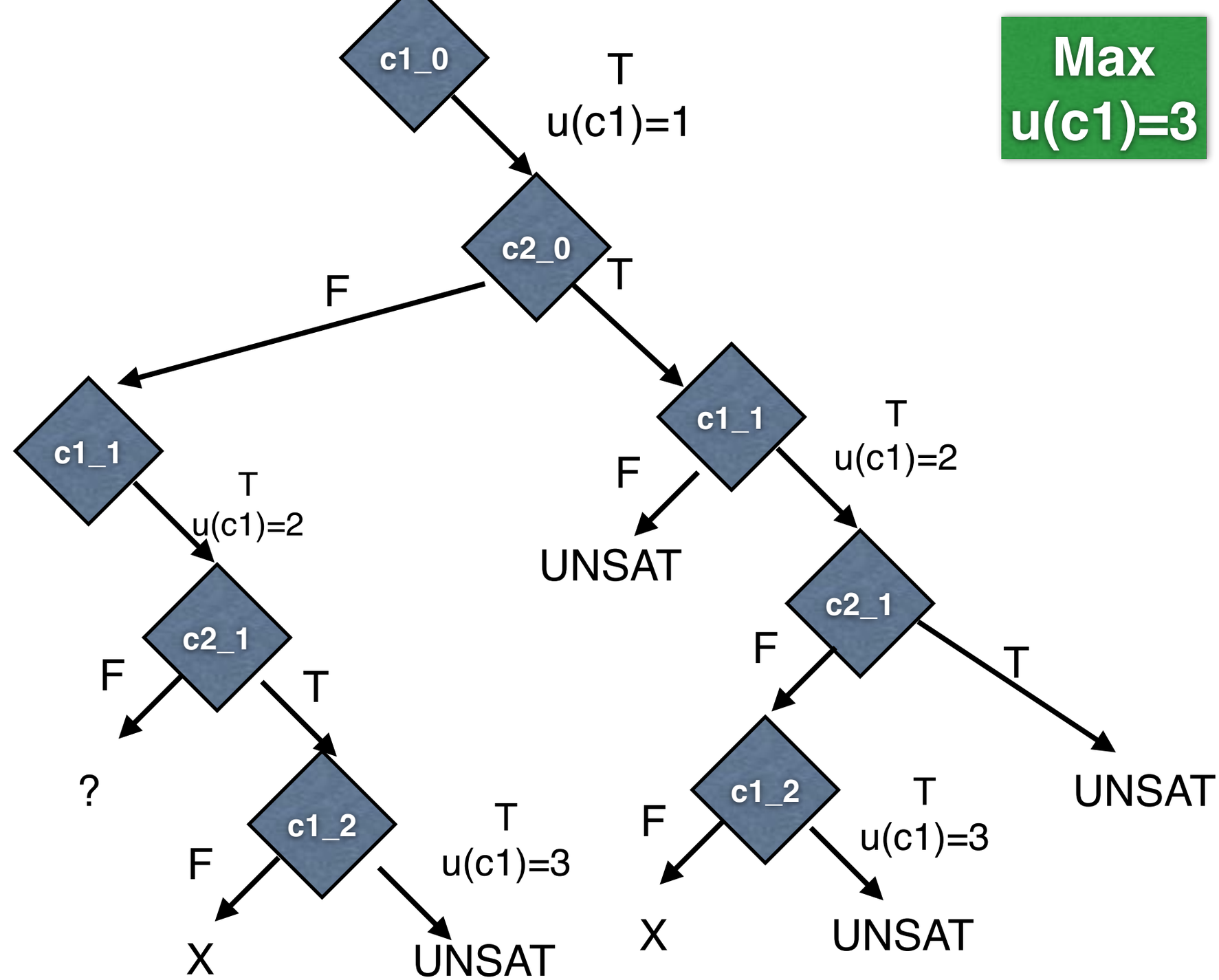
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
2	j=1	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&(2<2)	?

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



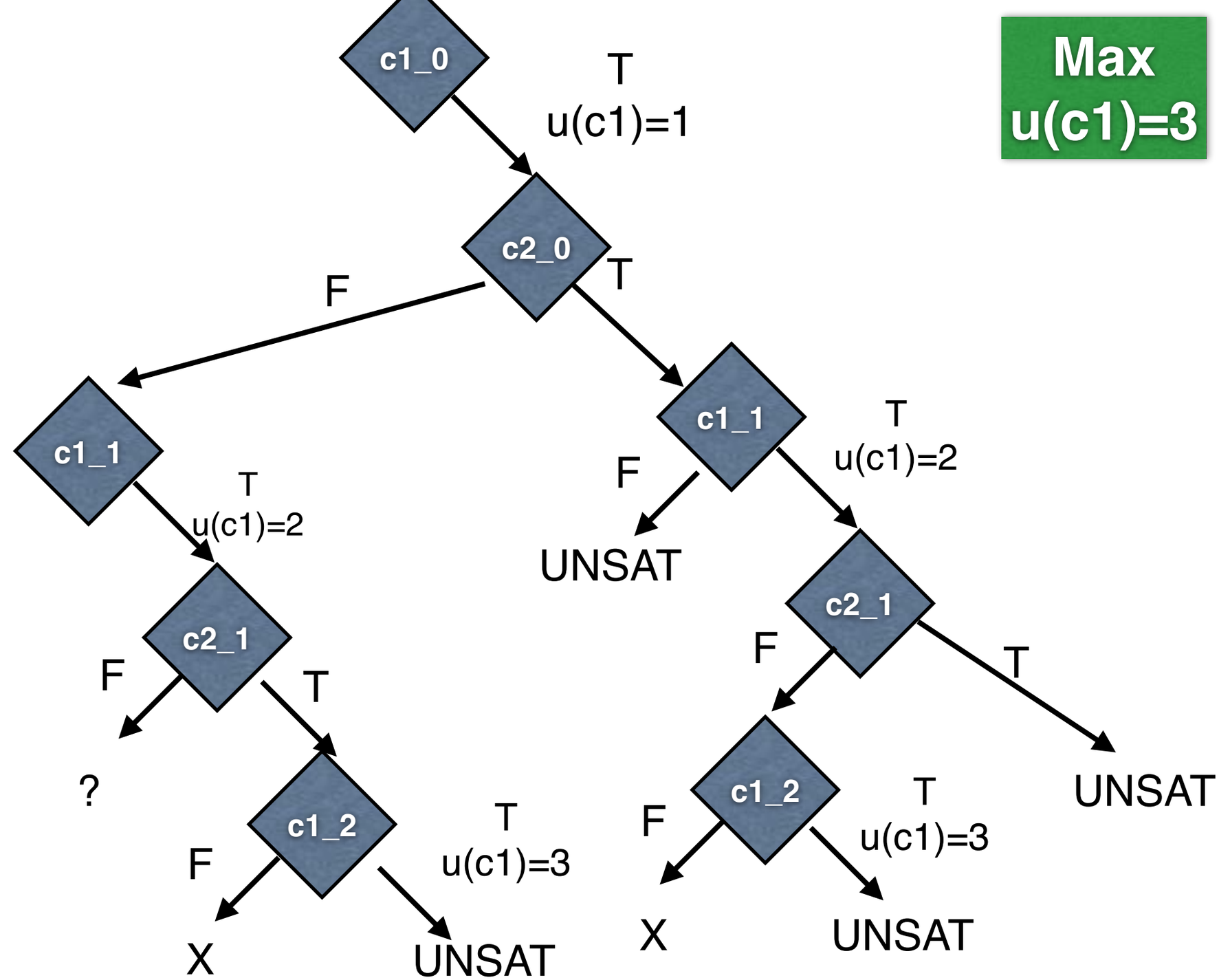
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
2	j=1	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&(2<2)	UNSAT

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



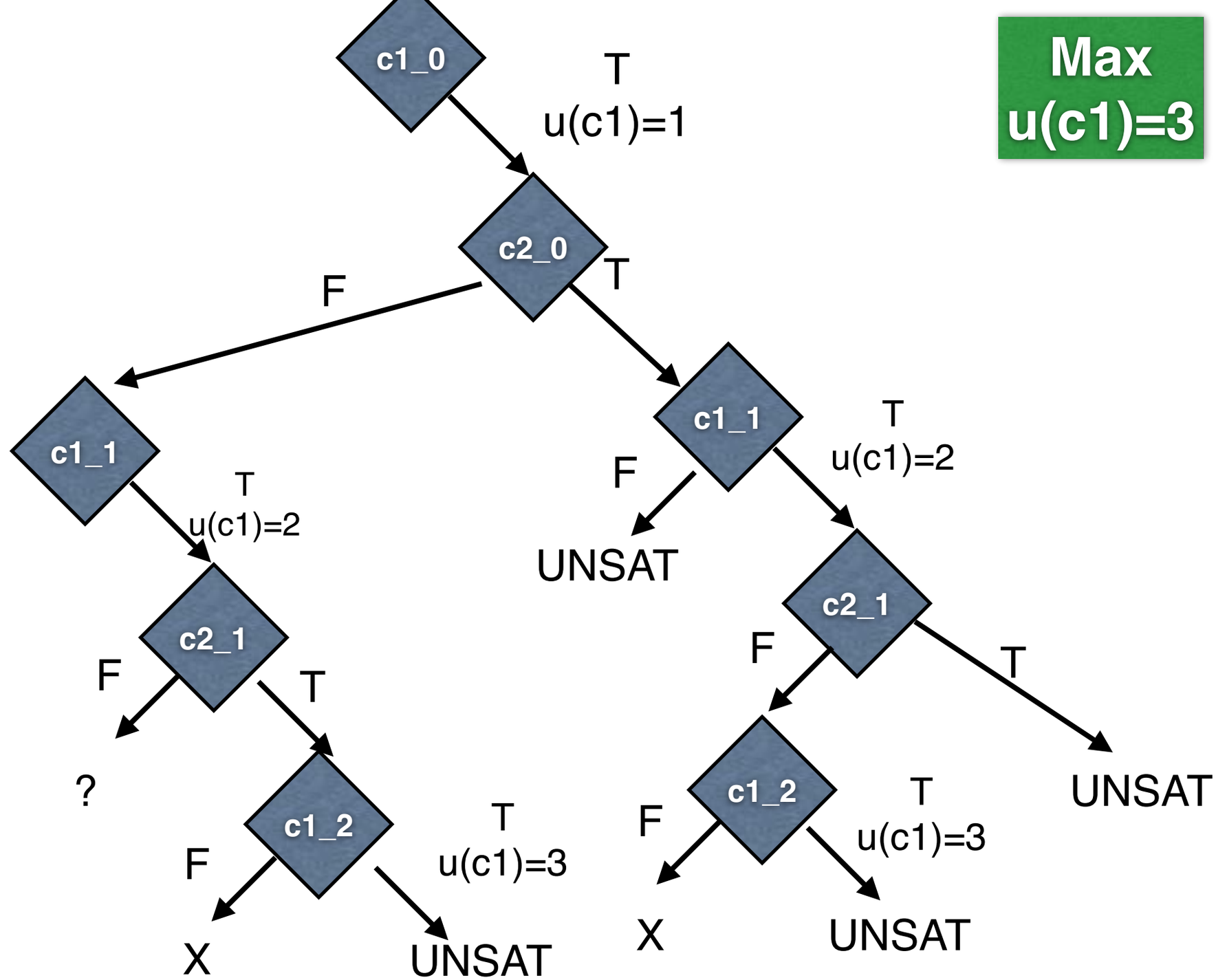
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
2	j=1	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&(2<2)	UNSAT

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



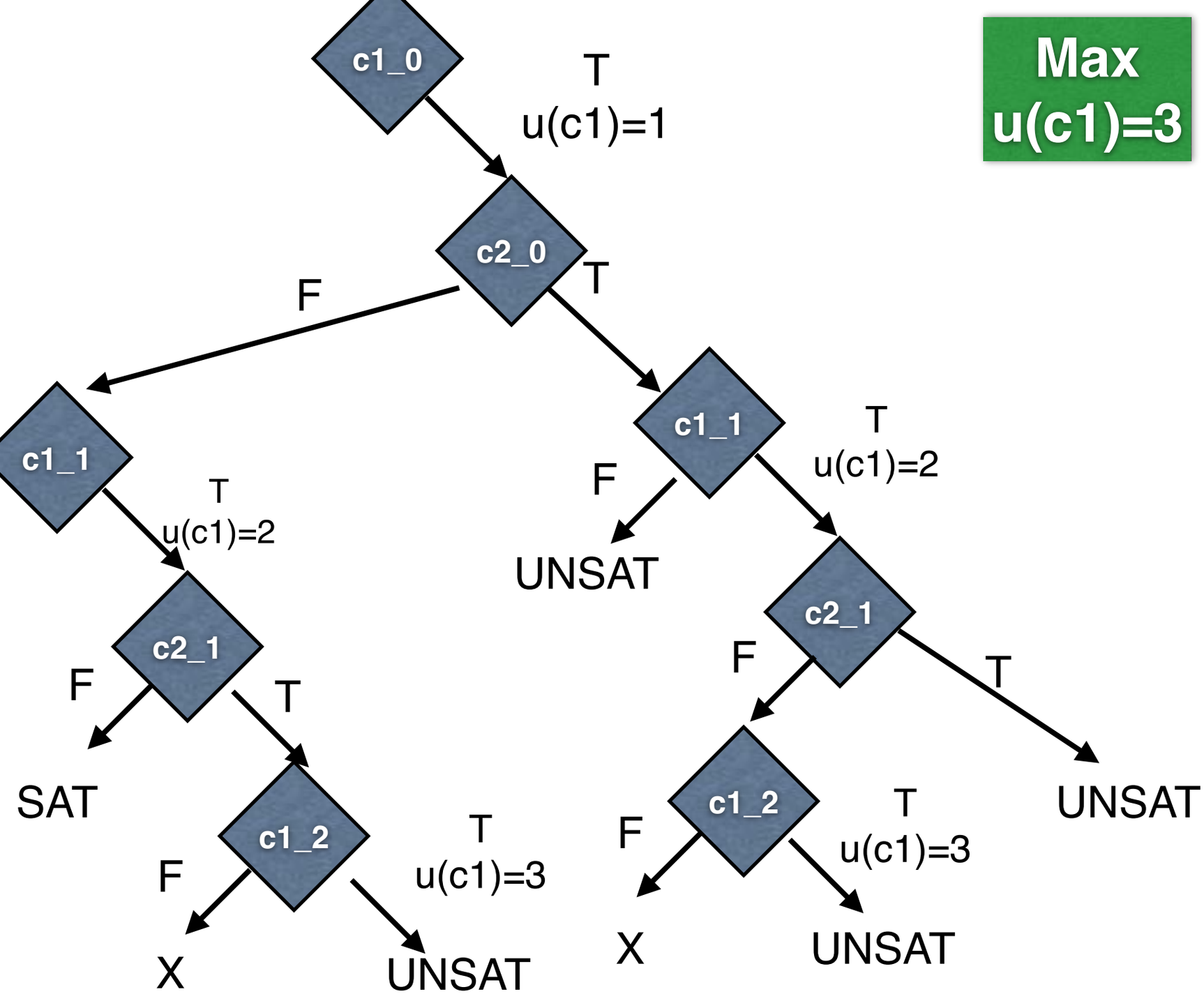
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
2	j=1	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&(2<2)	UNSAT
			?	

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
    if (i==j) { //c2
      r=r+1;
    }
    i=i+1;
  }
  return r;
}
```



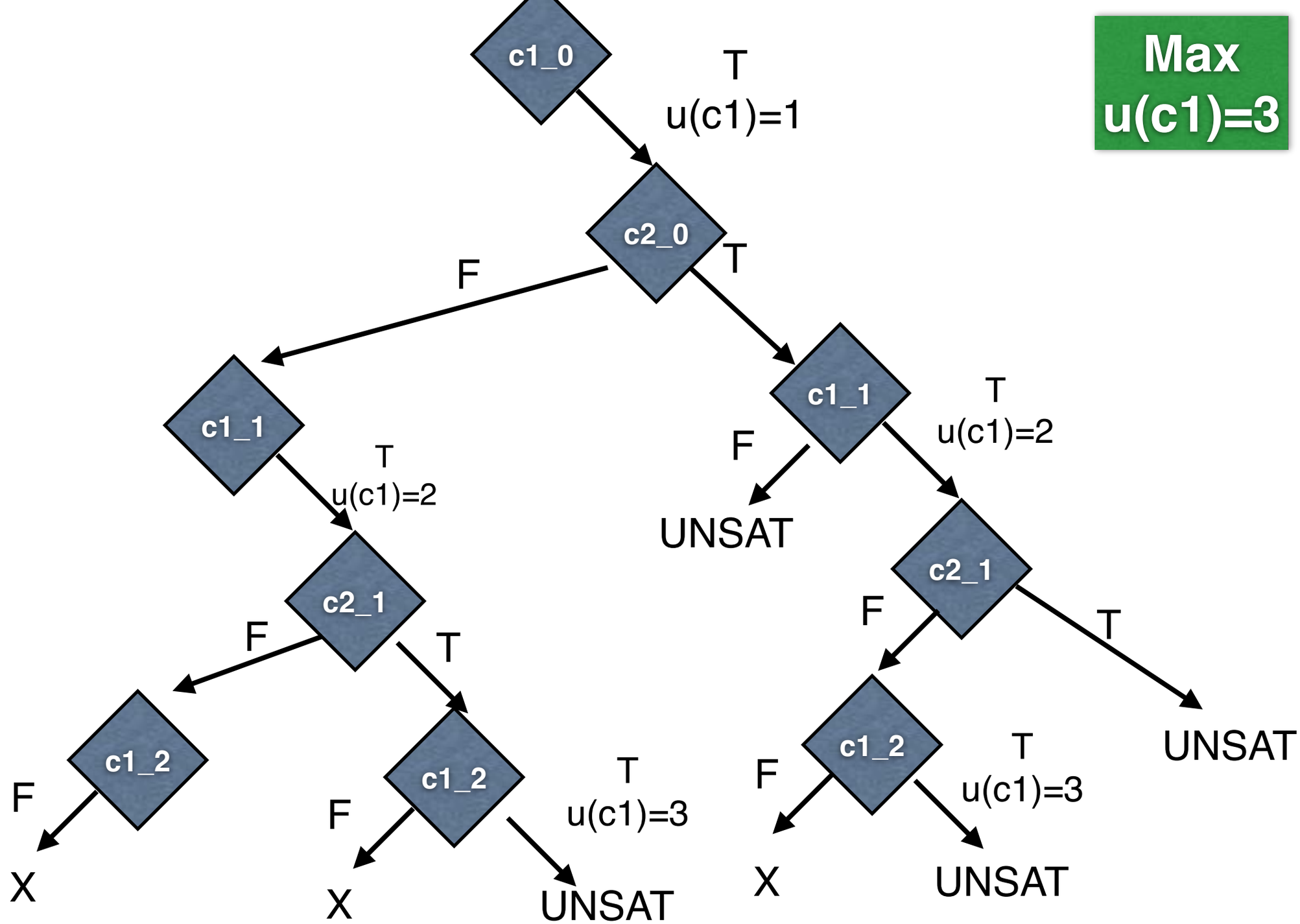
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
2	j=1	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&(2<2)	UNSAT
			(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)	?

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



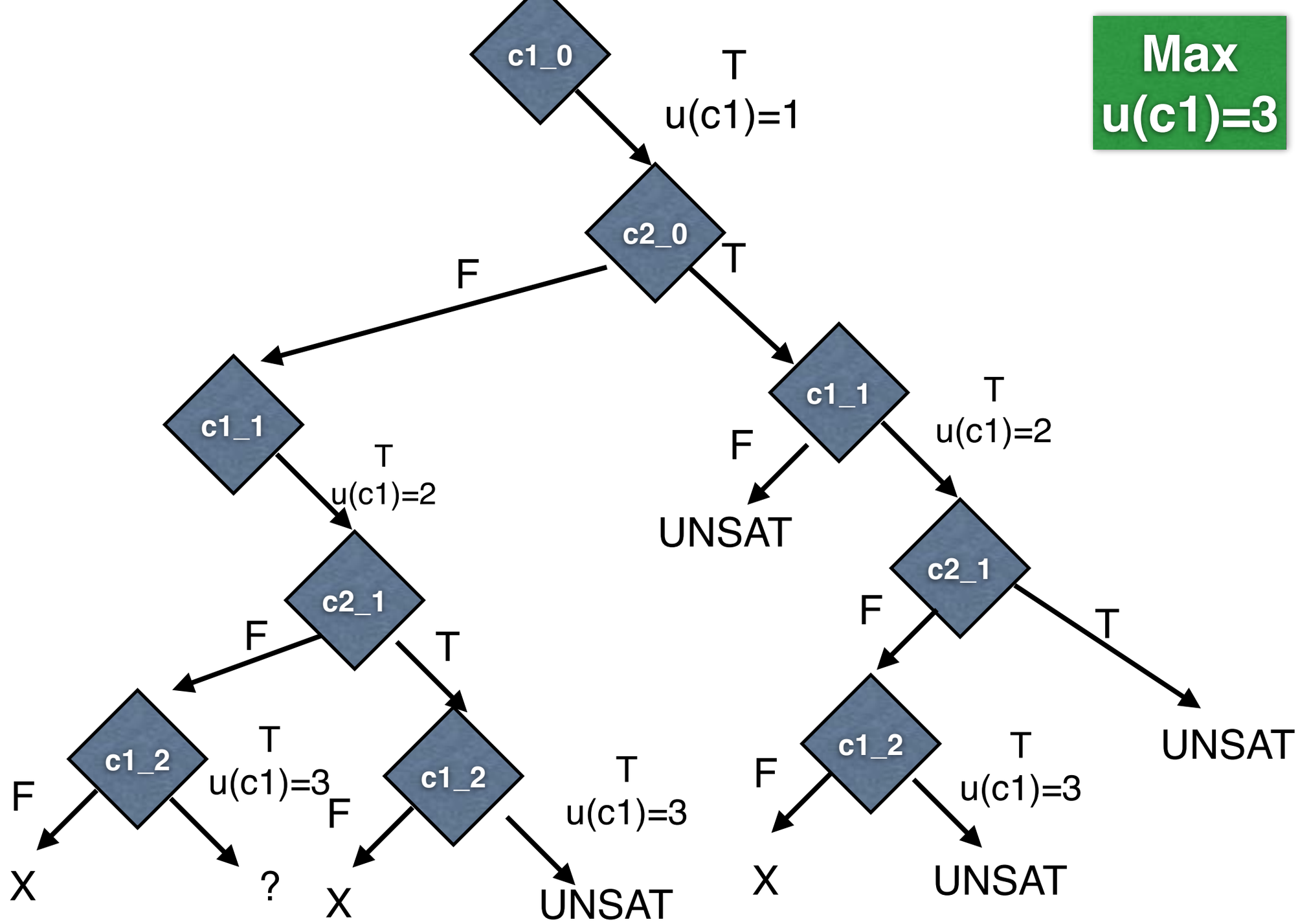
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
2	j=1	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&(2<2)	UNSAT
			(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)	j0=2

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



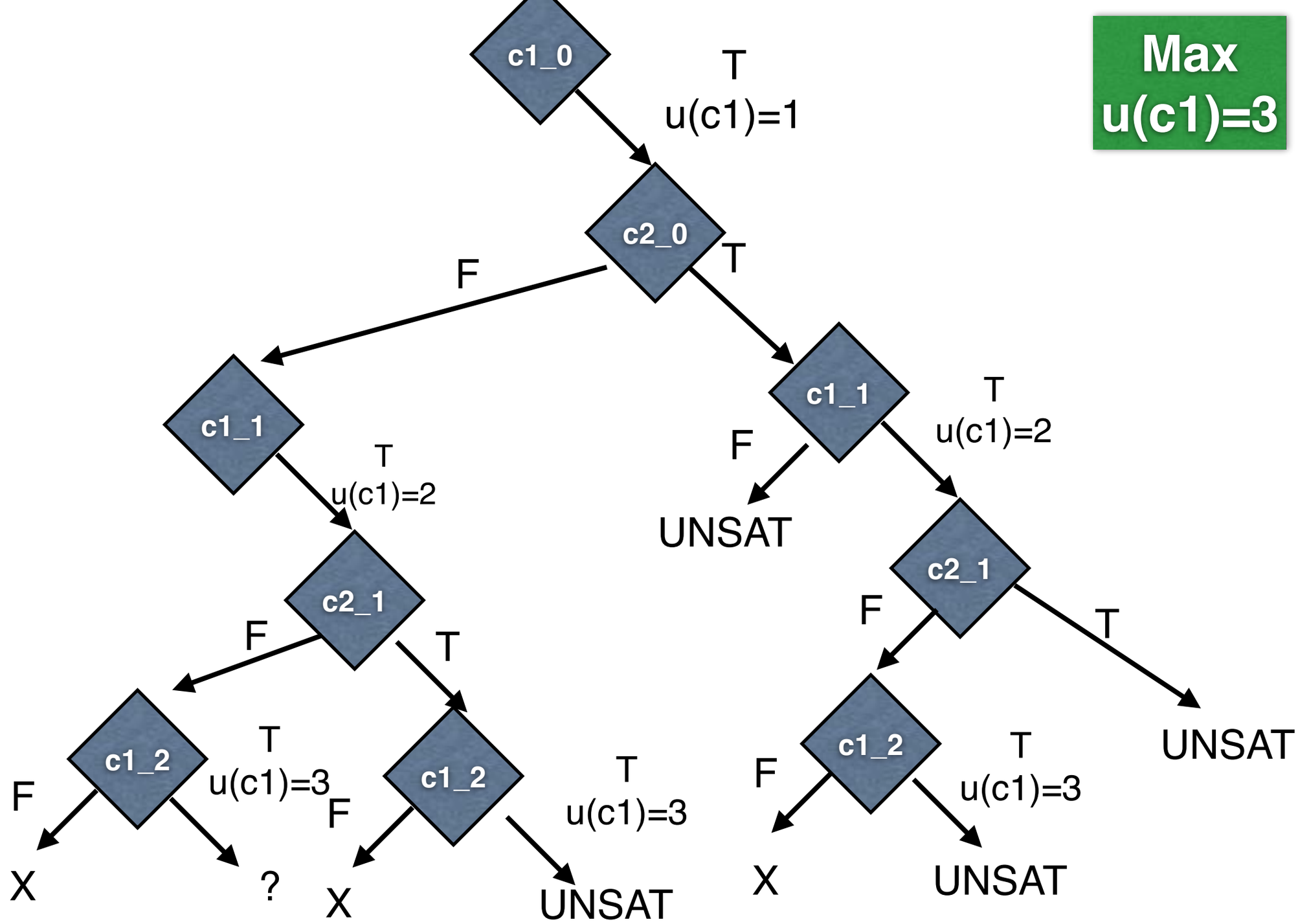
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=2	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)		

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



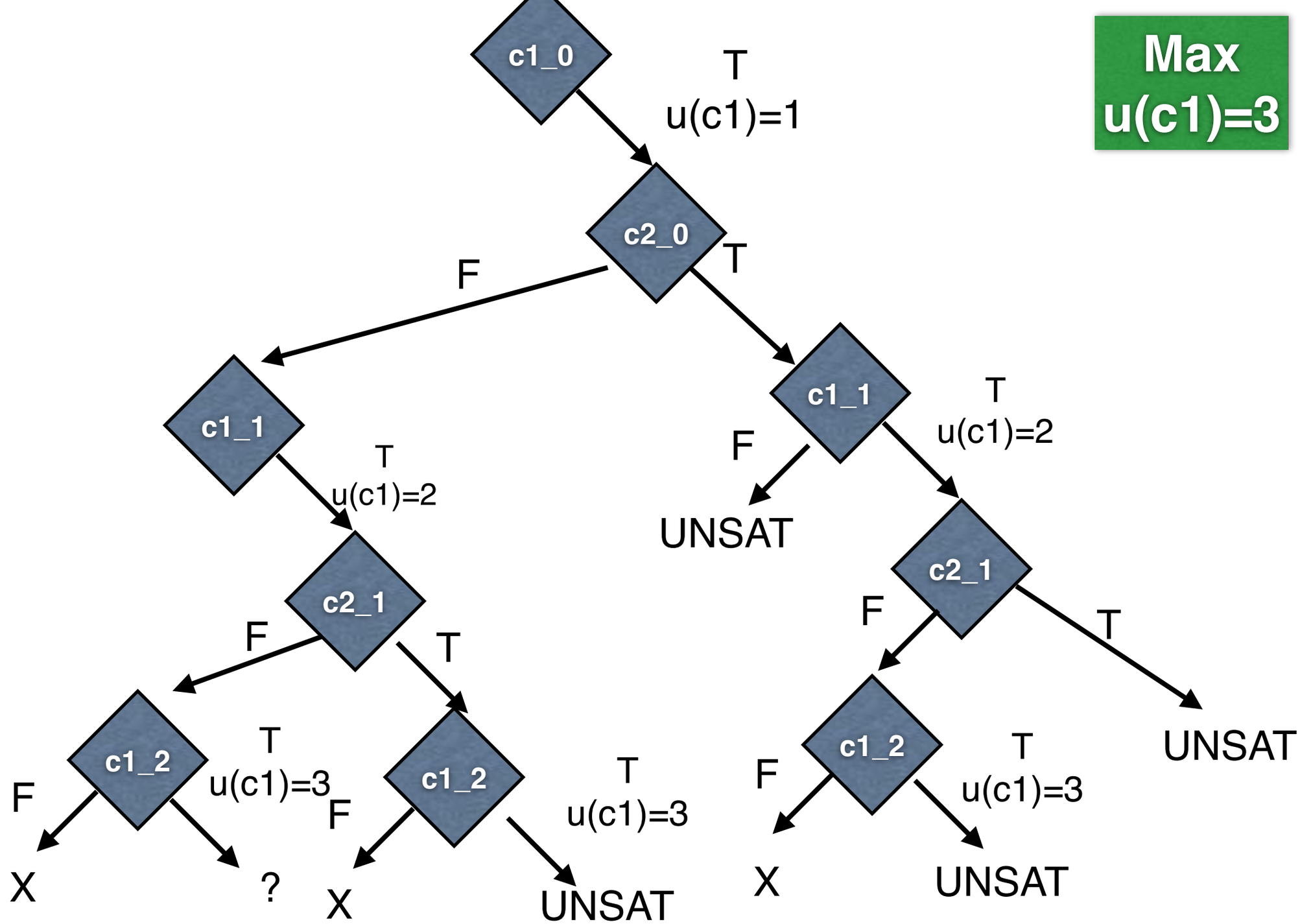
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=2	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)		

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



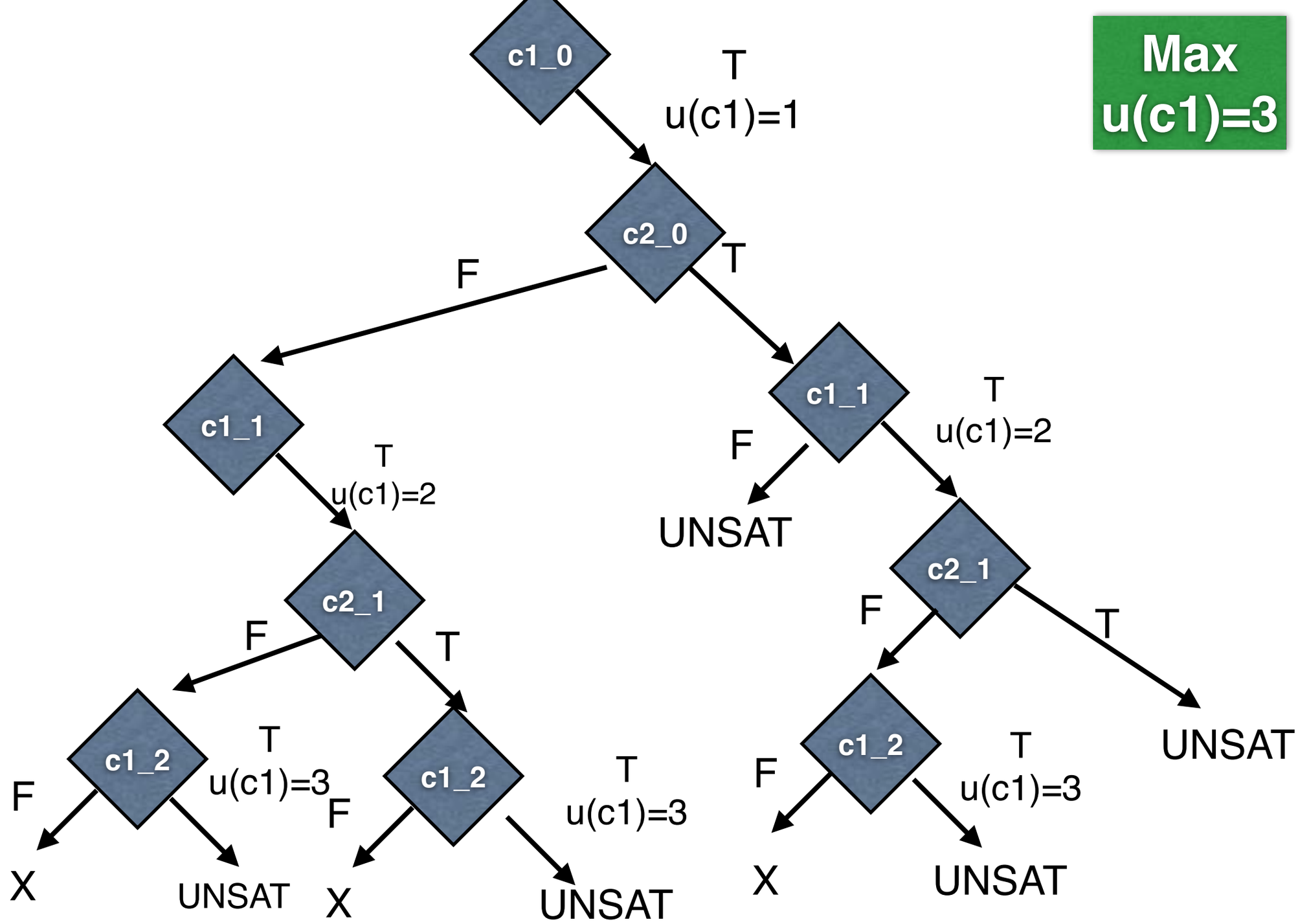
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=2	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	?	

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



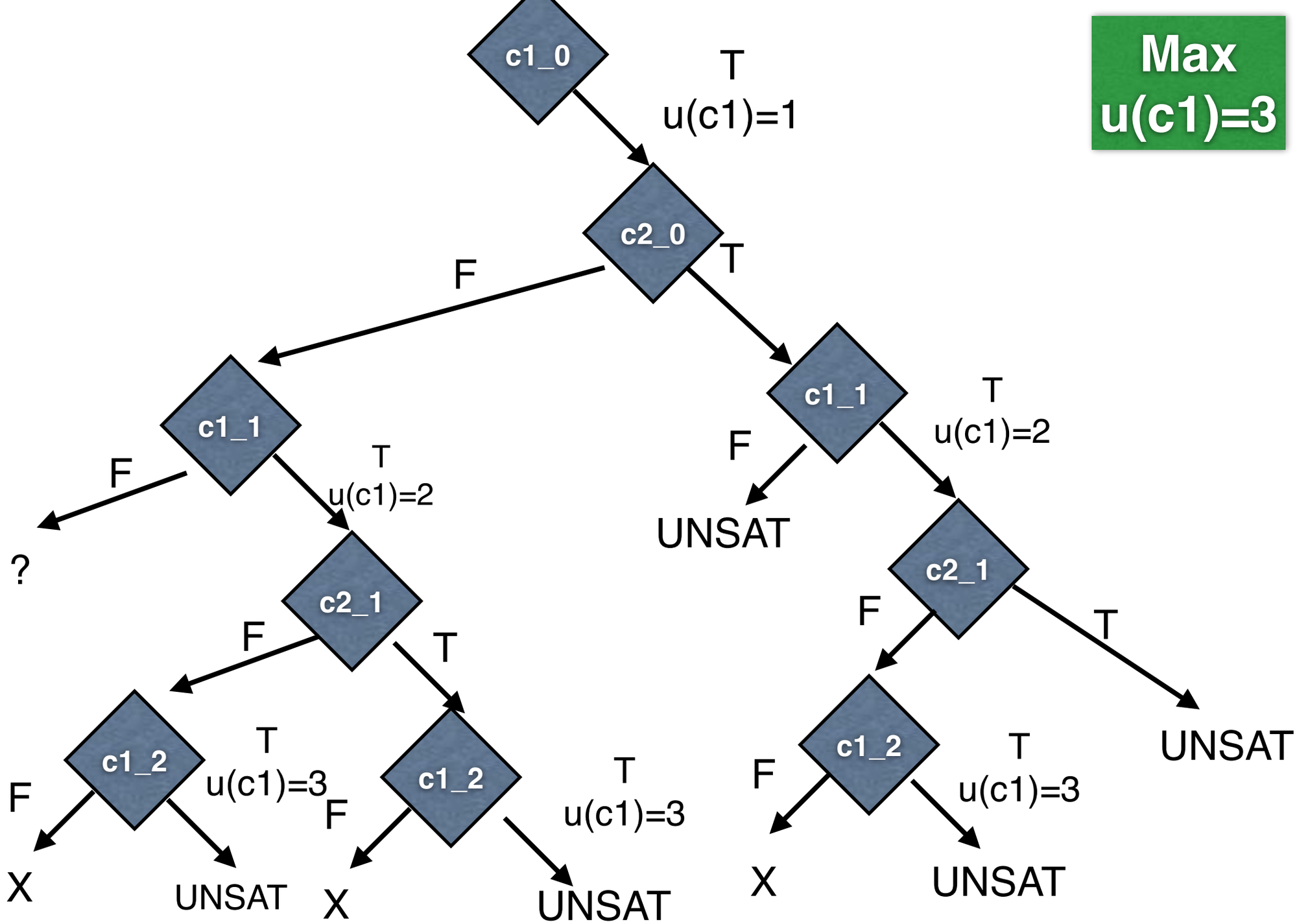
Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=2	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	?

```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
     if (i==j) { //c2
        r=r+1;
     }
     i=i+1;
  }
  return r;
}
```



Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=2	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT

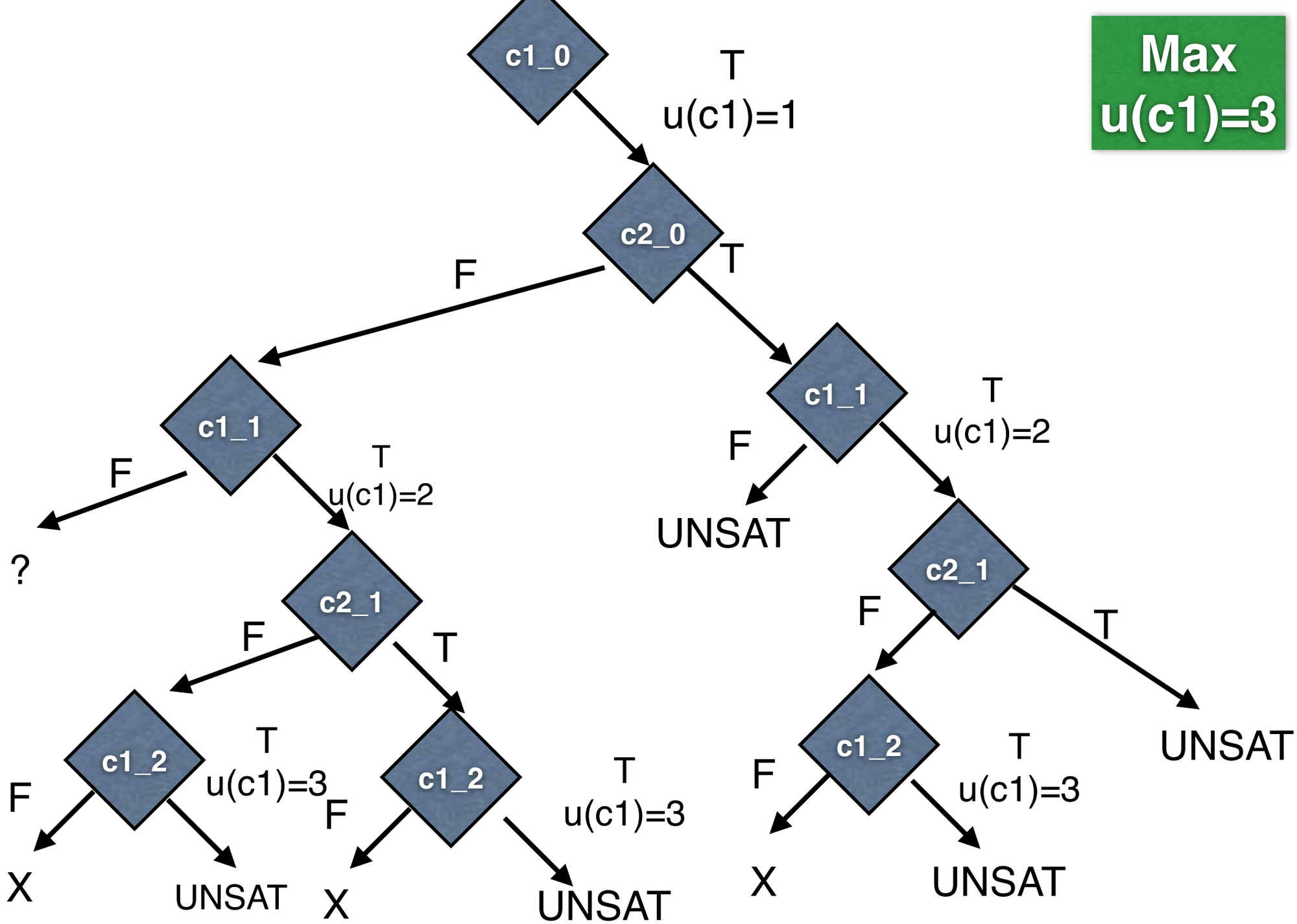
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
    if (i==j) { //c2
      r=r+1;
    }
    i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=2	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			?	

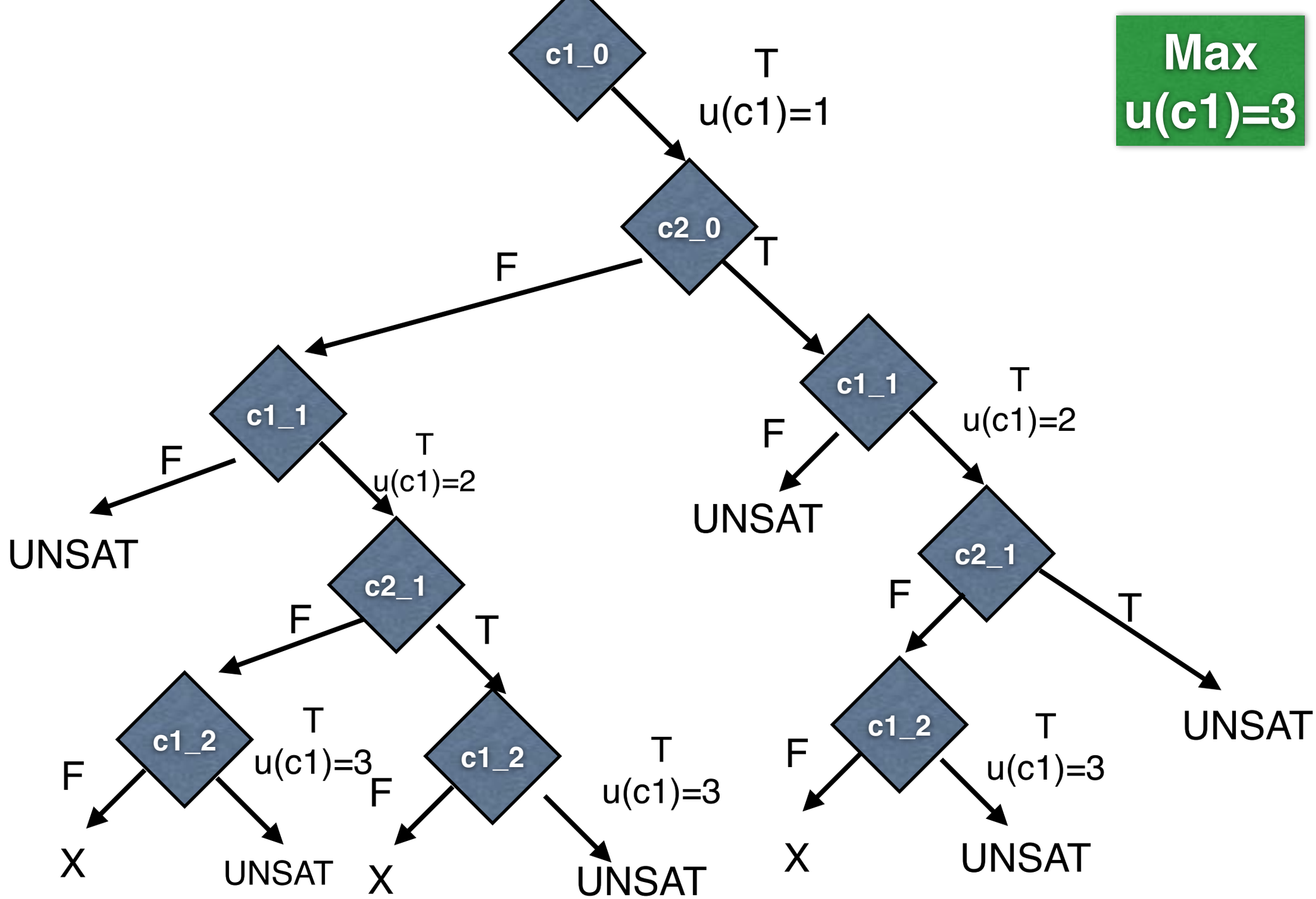
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
    if (i==j) { //c2
      r=r+1;
    }
    i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=2	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			(0<2)&&!(0==j0)&&!(1<2)	?

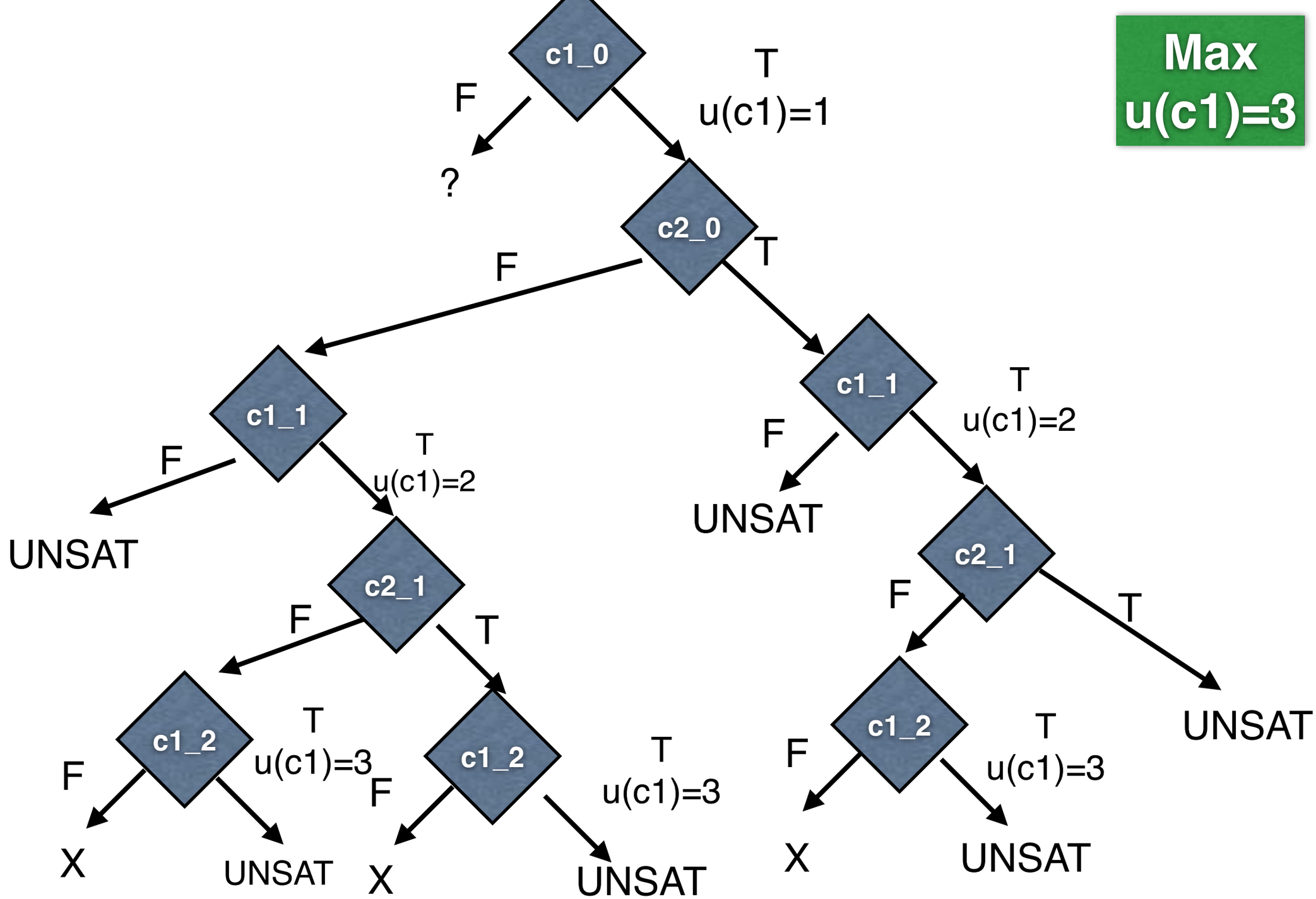
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
    if (i==j) { //c2
      r=r+1;
    }
    i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=2	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			(0<2)&&!(0==j0)&&!(1<2)	UNSAT

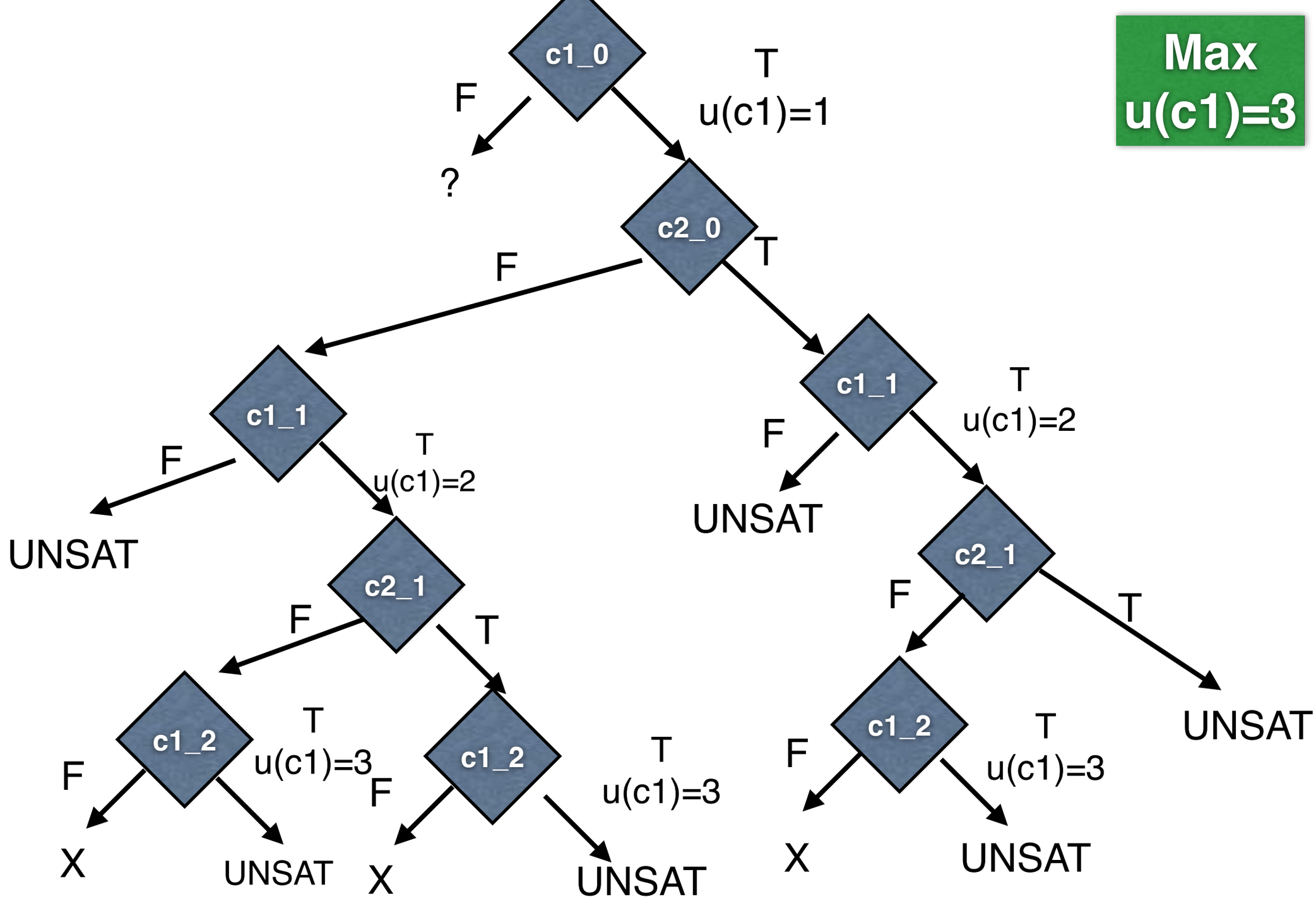
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
    if (i==j) { //c2
      r=r+1;
    }
    i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=2	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			(0<2)&&!(0==j0)&&!(1<2)	UNSAT
			?	

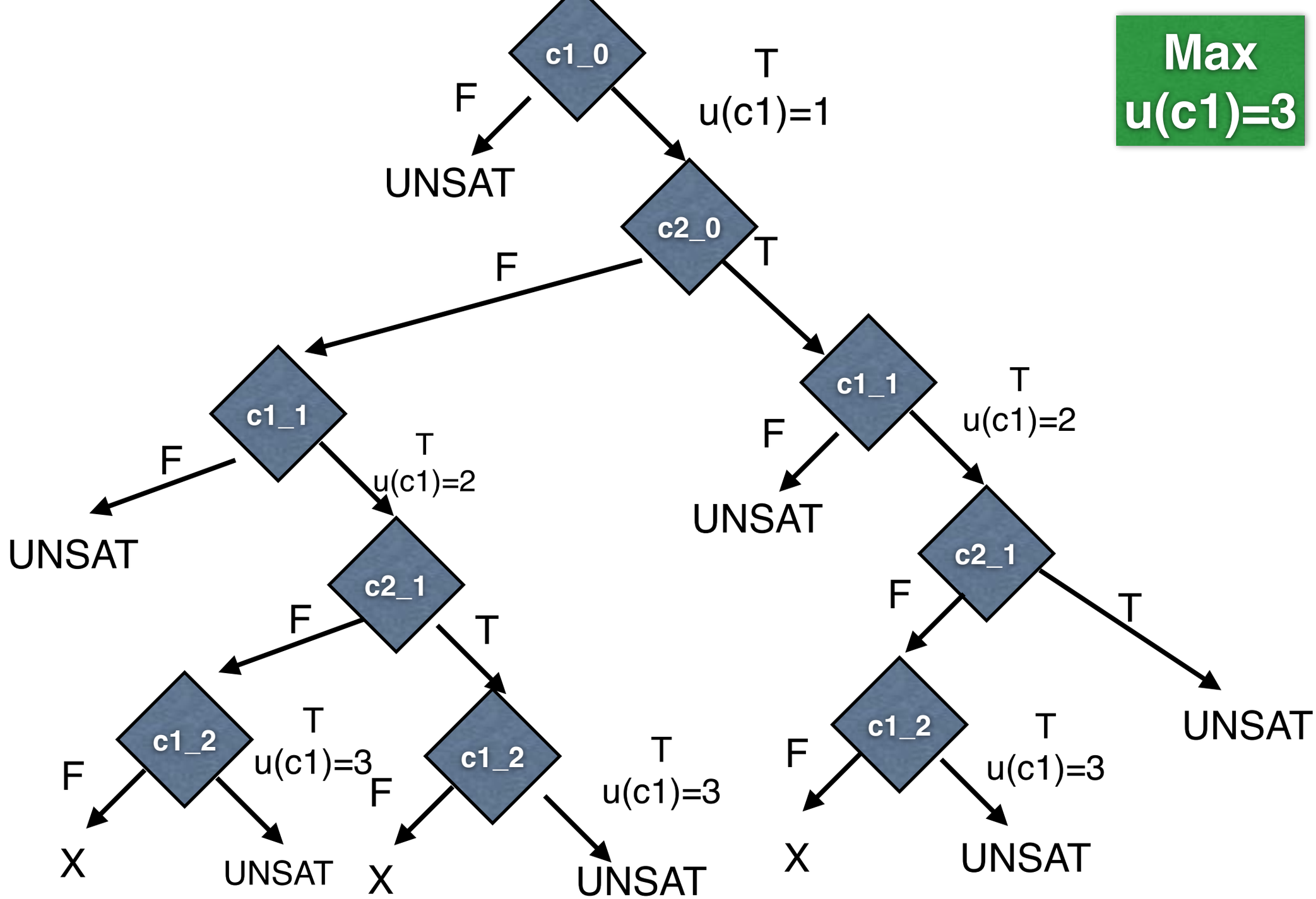
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
    if (i==j) { //c2
      r=r+1;
    }
    i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=2	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			(0<2)&&!(0==j0)&&!(1<2)	UNSAT
			!(0<2)	?

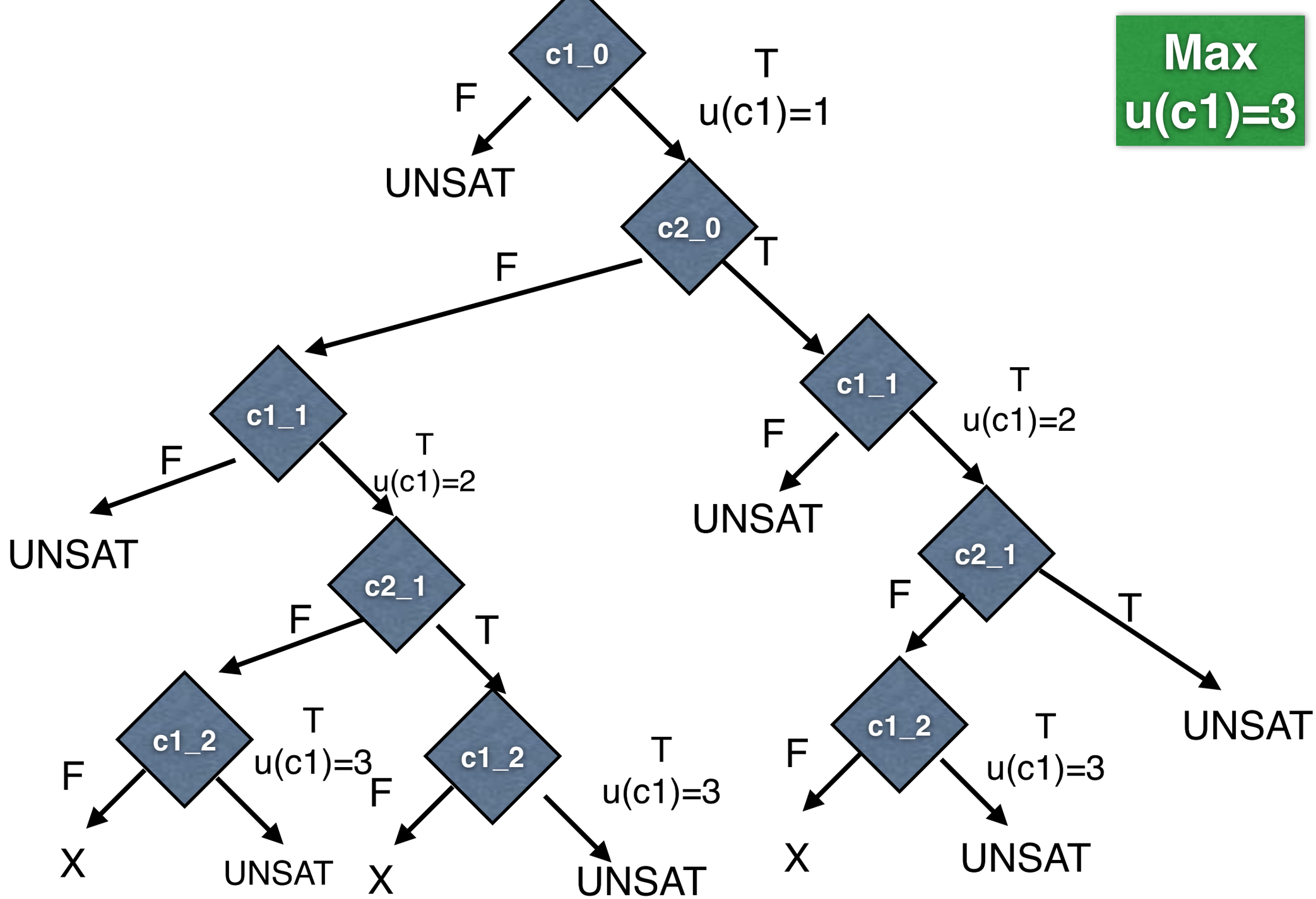
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
    if (i==j) { //c2
      r=r+1;
    }
    i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=2	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)&&(2<2)	UNSAT
			(0<2)&&!(0==j0)&&!(1<2)	UNSAT
			!(0<2)	UNSAT

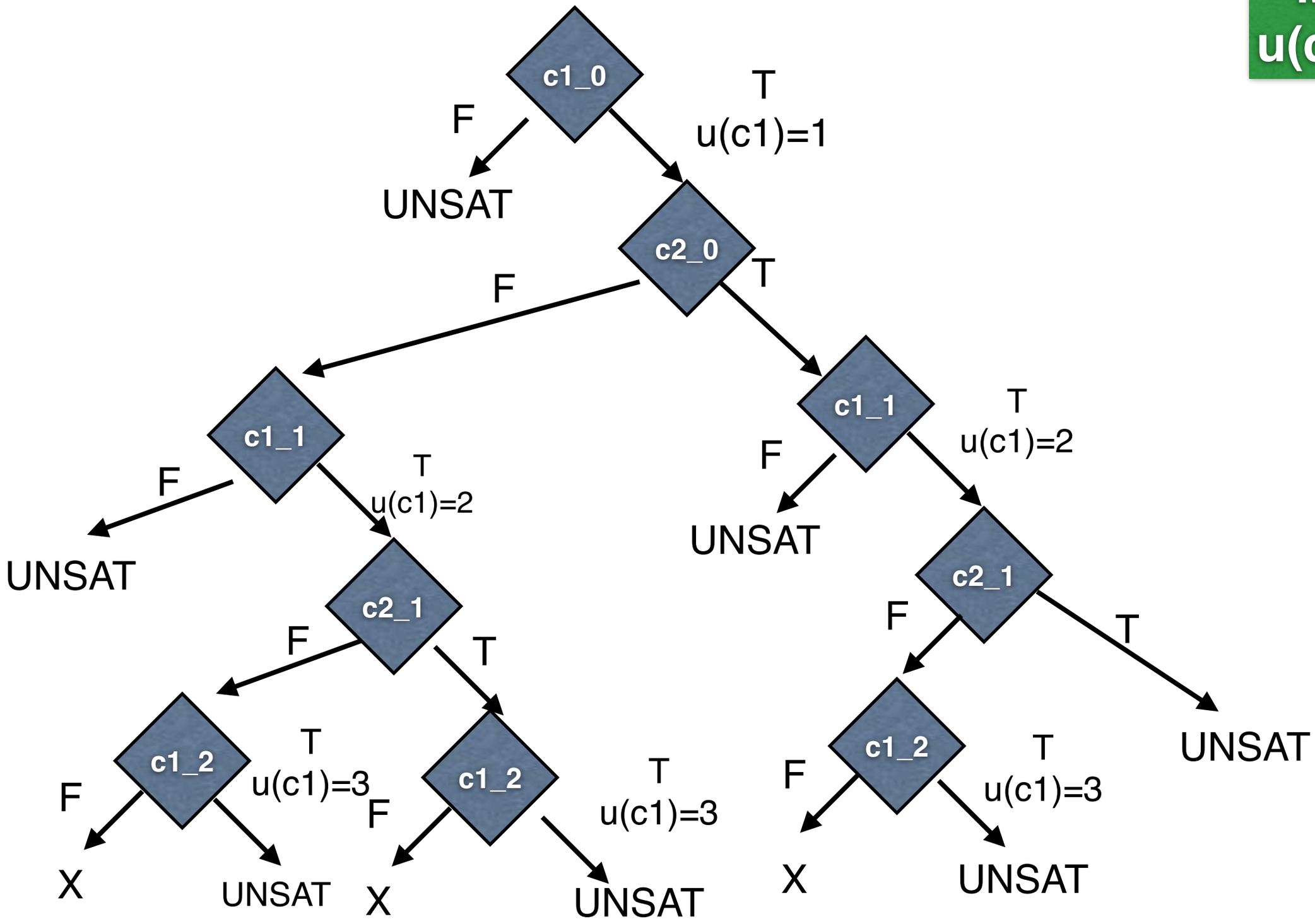
```
int test_me(int j) {
  int r=0;
  int i=0;
  while (i<2) { //c1
    if (i==j) { //c2
      r=r+1;
    }
    i=i+1;
  }
  return r;
}
```



Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=~~0~~ 2	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&(2<2)	UNSAT
			(0<2)&&!(0==j0)&&!(1<2)	UNSAT
			!(0<2)	UNSAT
			FIN	

Max u(c1)=3



Árbol de Cómputo explorado (i.e., no queda nada por explorar con Max u(c1)=3)

Max u(c1)=3

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
1	j=0	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&!(1<2)	(0<2)&&(0==j0)&&(1<2)&&!(1==j0)&&(1<2)	UNSAT
			(0<2)&&(0==j0)&&(1<2)&&(1==j0)	UNSAT
			(0<2)&&(0==j0)&&!(1<2)	UNSAT
			(0<2)&&!(0==j0)	j0=1

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
2	j=1	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&(2<2)	UNSAT
			(0<2)&&!(0==j0)&&(1<2)&&!(1==j0)	j0=2

Iter	Input concreto	Condición de Ruta	Fórmula enviada al demostrador	Resultado posible
3	j=~~1~~ 2	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&!(2<2)	(0<2)&&!(0==j0)&&(1<2)&&(1==j0)&&(2<2)	UNSAT
			(0<2)&&!(0==j0)&&!(1<2)	UNSAT
			!(0<2)	UNSAT
			FIN	

Tabla de ejecución simbólica dinámica finalizada

Imprecisión

¿Qué pasa cuando ejecutamos foo con x=567 e y=42 ?

```
int foo(int x, int y) {
   if (x==hash(y)) {
      …
      if (y==10) return -1; // error
   }
   …
}
```

Imprecisión

```
int foo(int x, int y) {
  if (x==hash(y)) {
    …
    if (y==10) return -1; // error
  }
  …
}
```

Input: x=567 and y=42 ⟶ PC: x==567 and y!=10

Solve(x==567 and y=10) ⟶ Input: x=567 and y=10

PERO: x=567, y=10 no alcanza return -1

De hecho, ni siquiera cumple que x==hash(y)

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

Cambios en la Generación si hay Divergencia

- Conservamos una lista de todas las path conditions que generamos con ejecución simbólica

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

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

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

hay que modificar el pseudocódigo p/ verificar que se recorrió el camino esperado (p/ detectar divergencias)

Heap Constraints

- ¿Qué podemos hacer cuando tenemos un PUT que recibe un tipo no primitivo?

```
public void test(MiniStack s, int value) {
  assumeTrue(s!=null);
  assumeFalse(s.isFull());
  s.push(value);
  int rv = s.pop();
  assertEquals(value, rv);
}
```

Heap Constraints

- Muchas veces tenemos constraints que predican 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)

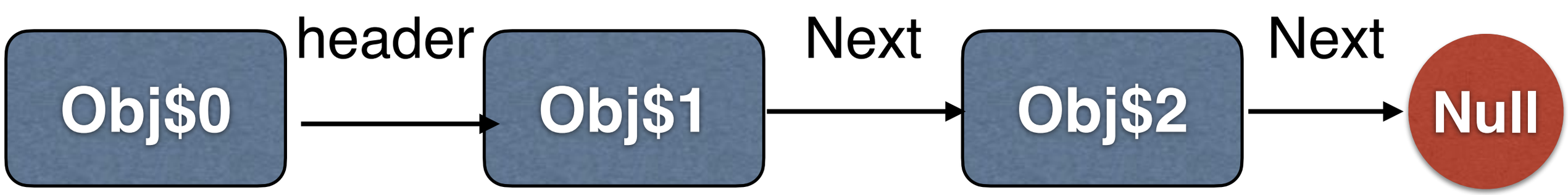
Constraint Language

Referencias a Objetos

(extensión p/ poder manejar objetos)

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

Var0!=null
Var0.header!=Var0
Var0.header!=null
Var0.header.next!=null
Var0.header.next.next==null



Público vs. Privado

```
class MiniStack {
  public int readIndex;
  public Object[] elements;
  …
}
```

Si todos los campos que necesitamos actualizar son de acceso público no hay problema

¿qué pasa cuando hay campos que no son accesibles y necesitamos modificarlos?

```
class MiniStack {
  private int readIndex;
  private Object[] elements;
  …
}
```

Heap Constraints

var0=Obj$1
var0.*header*=Obj$2
var0.*header*.*next*=Obj$3
var1=Obj$2

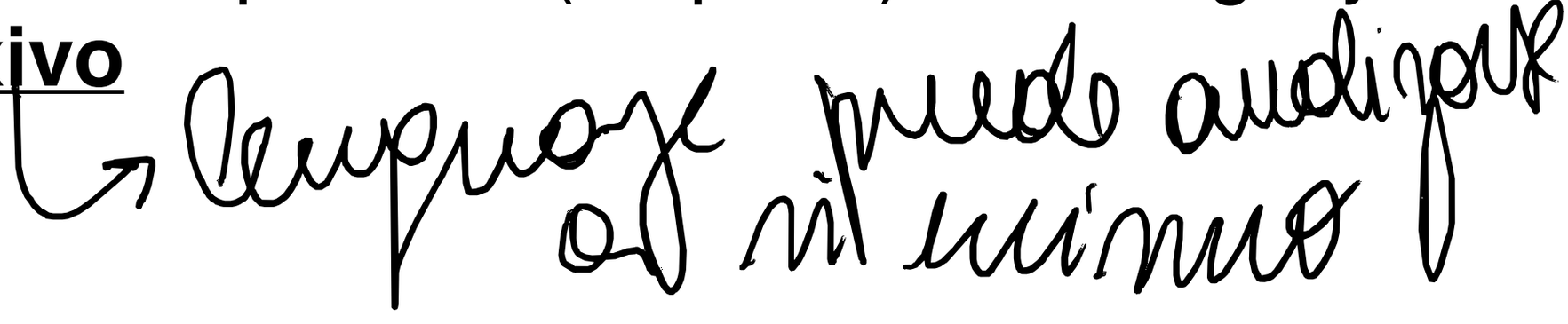
Heap Constraints

var0=Obj$1
var0.*header*=Obj$2
var0.*header*.*next*=Obj$3
var1=Obj$2

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

Heap Constraints

- Muchas veces tenemos constraints que predican 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

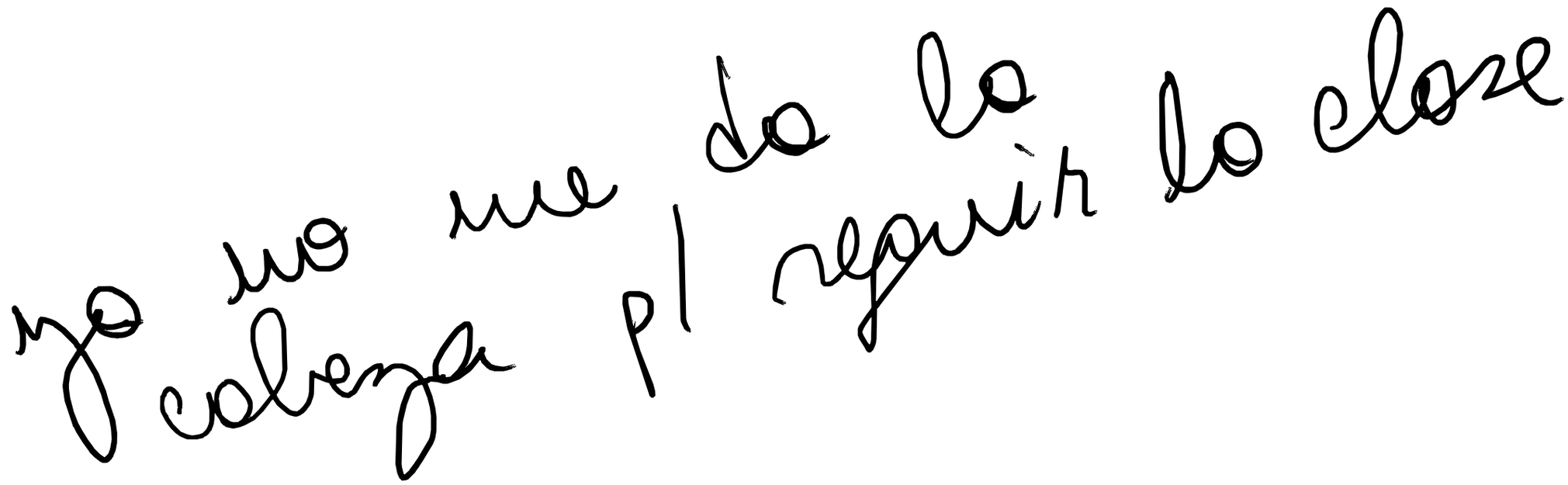
- Nos permiten introspectivamente analizar el mismo programa (ej: Python)
- En Java, podemos:
  - Crear objetos (aunque no haya constructor por defecto)
  - Asignar un valor a campos privados

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);
```

↑ cobertura violenta invocando todos los métodos públicos e interfaces de clases

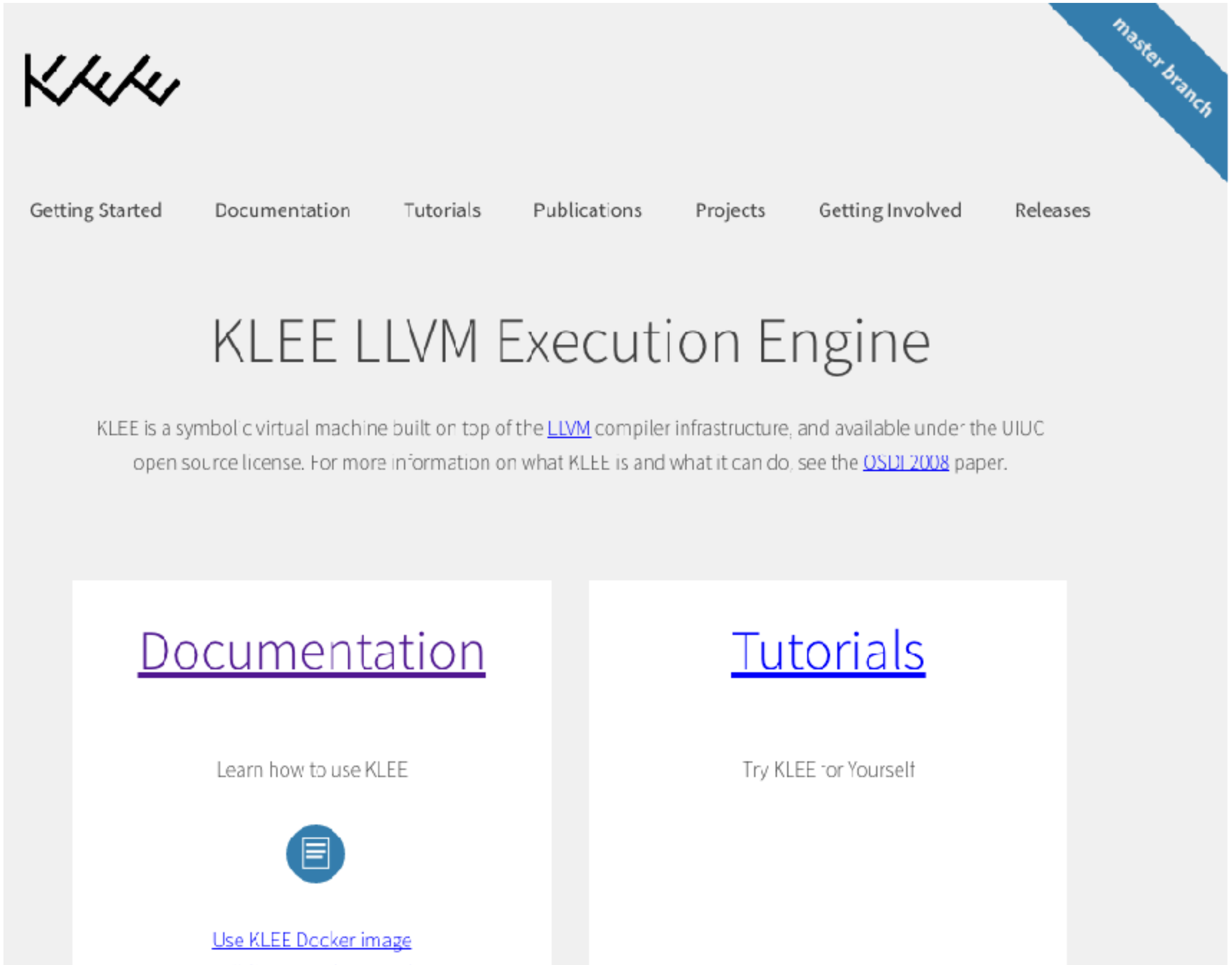
ok



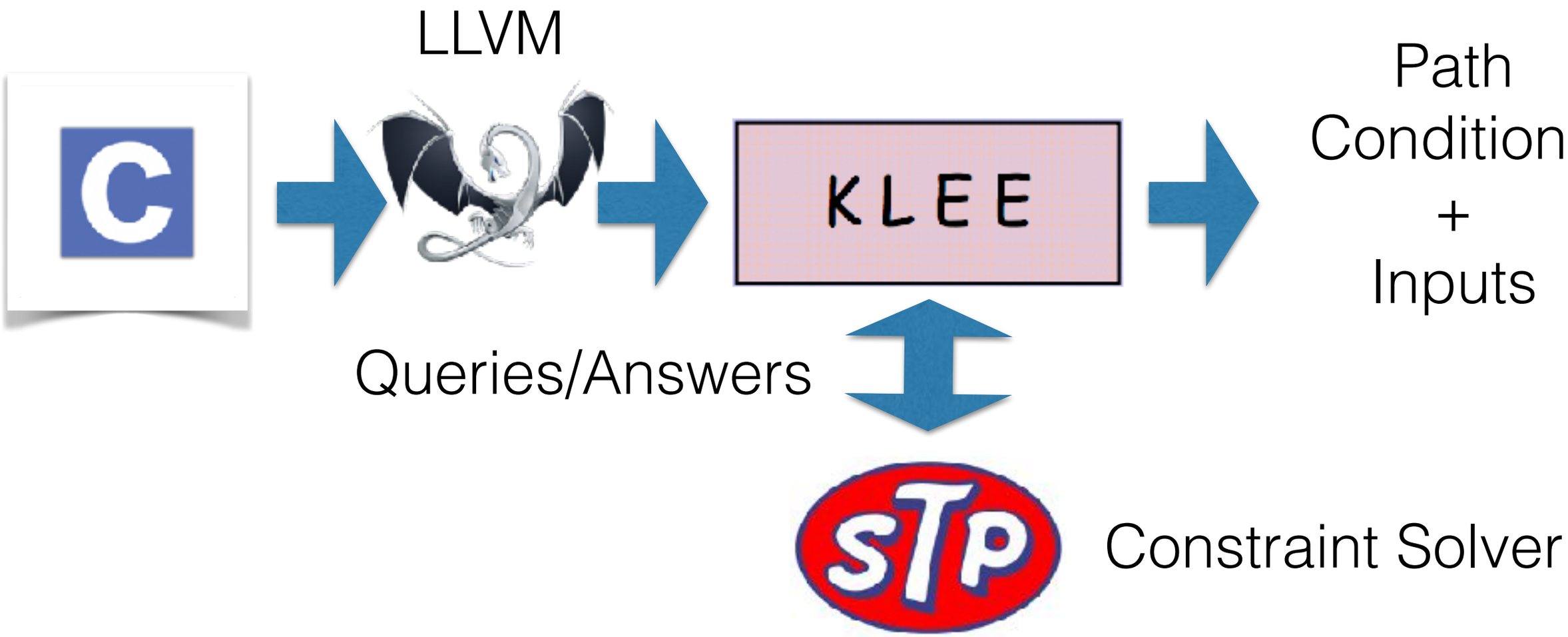
Otros Desafíos Abiertos para ejecución simbólica

- 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)

http://klee.github.io/klee/



KLEE



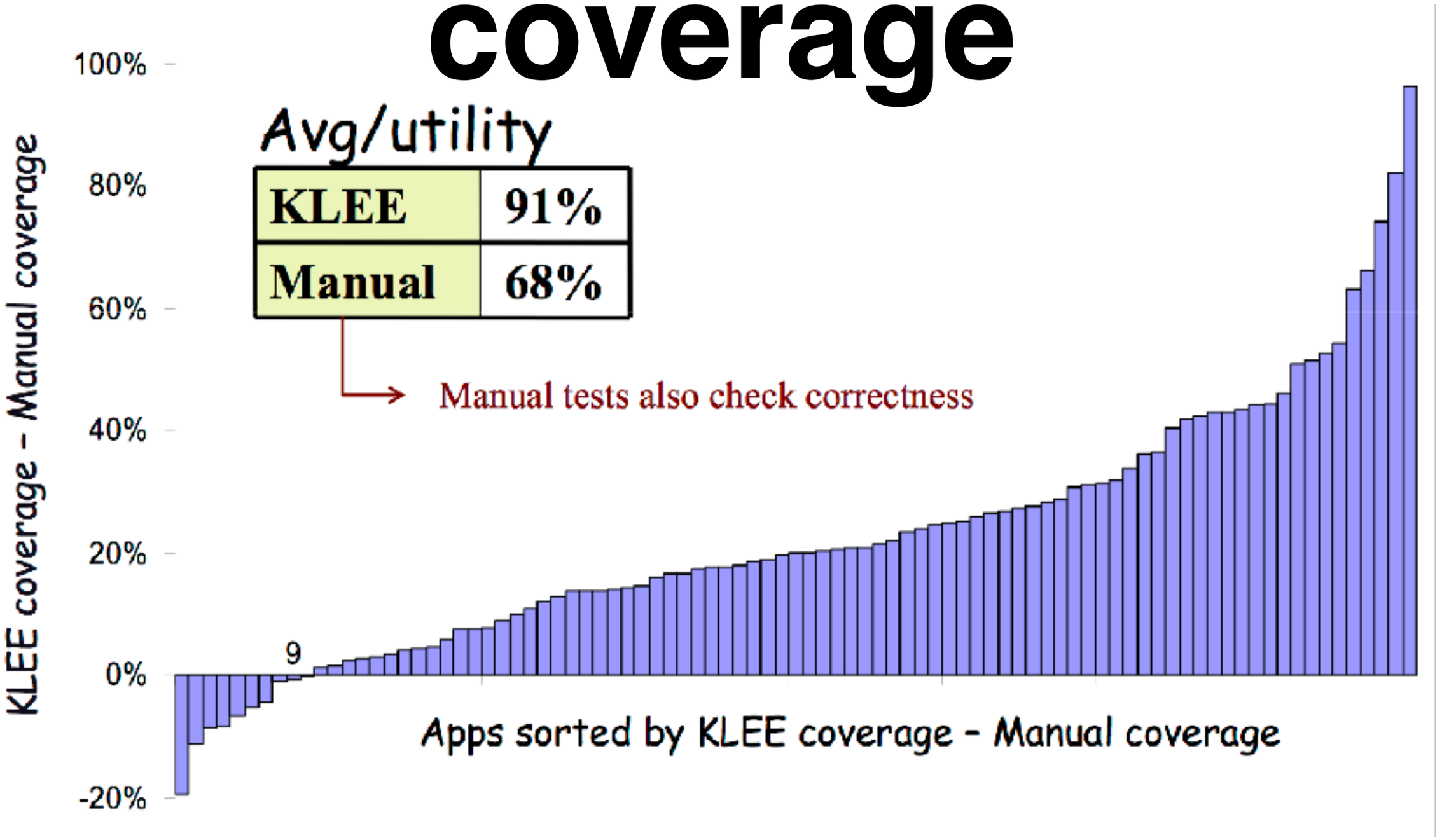
Cristian Cadar, Daniel Dunbar and Dawson Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI 2008

GNU Coreutils application suite

- Core user-level apps installed on UNIX systems (i.e. ls, mkdir, seq, etc.)
- Roughly ~180 KLOC
- Heavily tested, mature code
- Variety of functions, authors, intensive interaction with environment

Cristian Cadar, Daniel Dunbar and Dawson Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI 2008

KLEE: Code coverage



Cristian Cadar, Daniel Dunbar and Dawson Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI 2008

KLEE: Bug finding

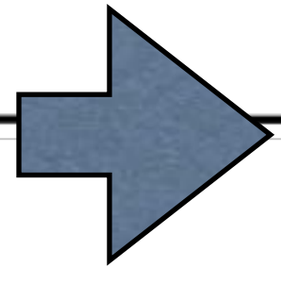
`md5sum -c t1.txt`	`pr -e t2.txt`
`mkdir -Z a b`	`tac -r t3.txt t3.txt`
`mkfifo -Z a b`	`paste -d\\ abcdefghijklmnopqrstuvwxyz`
`mknod -Z a b p`	`ptx -F\\ abcdefghijklmnopqrstuvwxyz`
`seq -f %0 1`	`ptx x t4.txt`

t1.txt: \t \tMD5(

t2.txt: \b\b\b\b\b\b\b\t

t3.txt: \n

t4.txt: A



KLEE: Bug finding

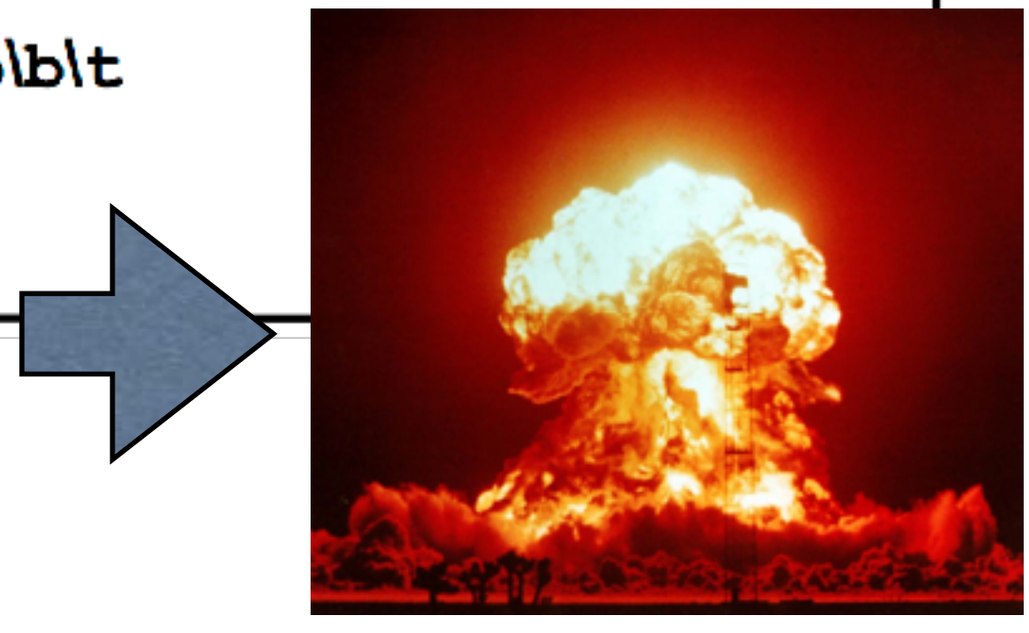
`md5sum -c t1.txt`	`pr -e t2.txt`
`mkdir -Z a b`	`tac -r t3.txt t3.txt`
`mkfifo -Z a b`	`paste -d\\ abcdefghijklmnopqrstuvwxyz`
`mknod -Z a b p`	`ptx -F\\ abcdefghijklmnopqrstuvwxyz`
`seq -f %0 1`	`ptx x t4.txt`

t1.txt: `\t \tMD5(`

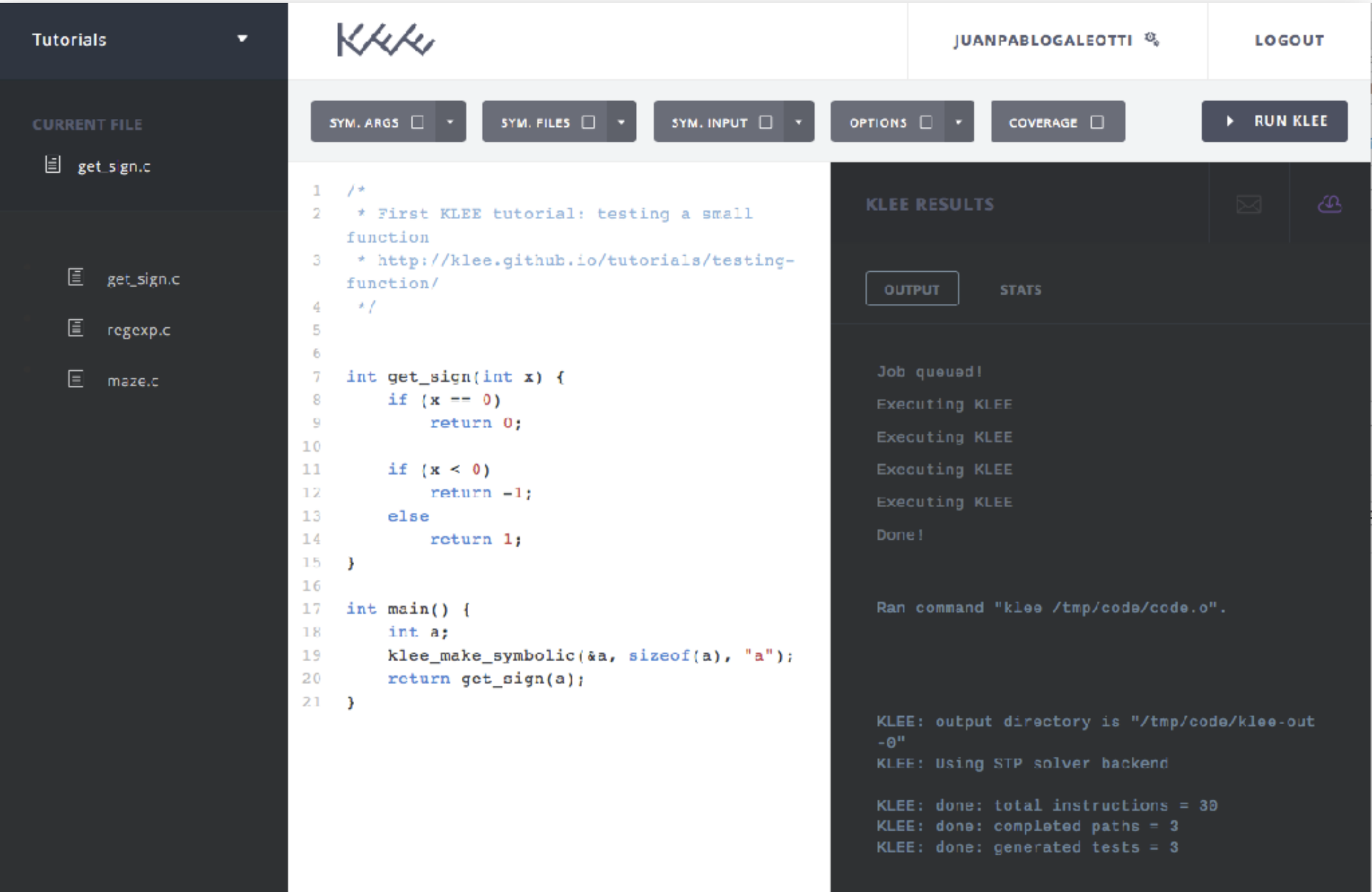
t2.txt: `\b\b\b\b\b\b\b\t`

t3.txt: `\n`

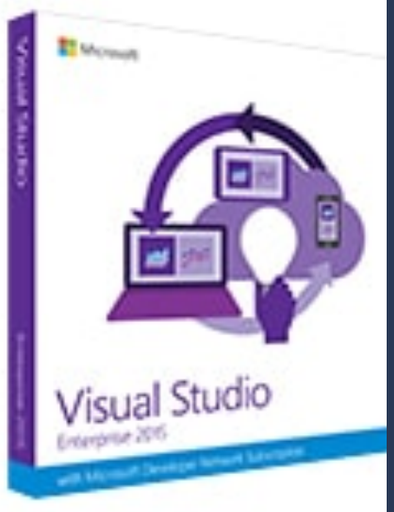
t4.txt: `A`

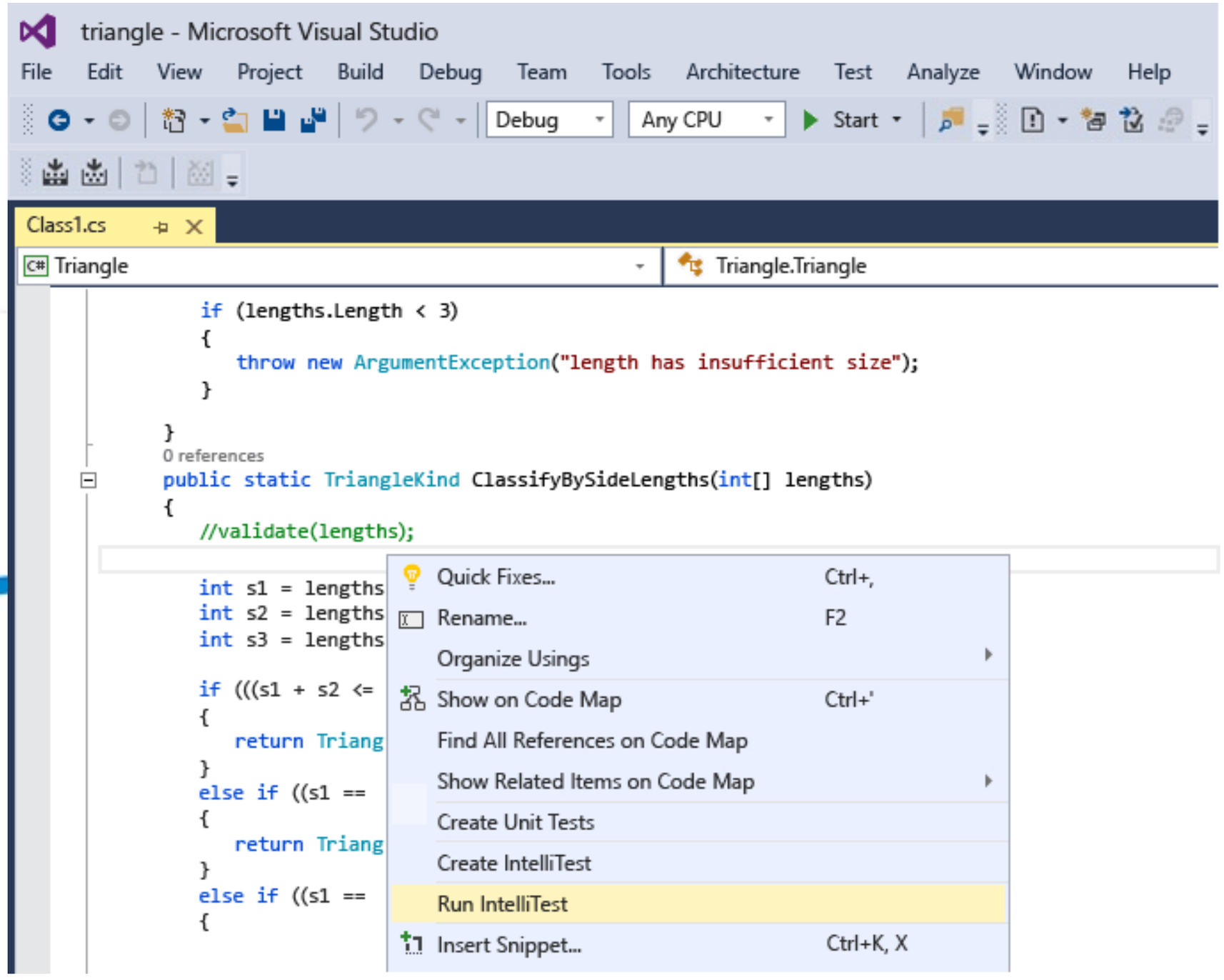


http://klee.doc.ic.ac.uk/



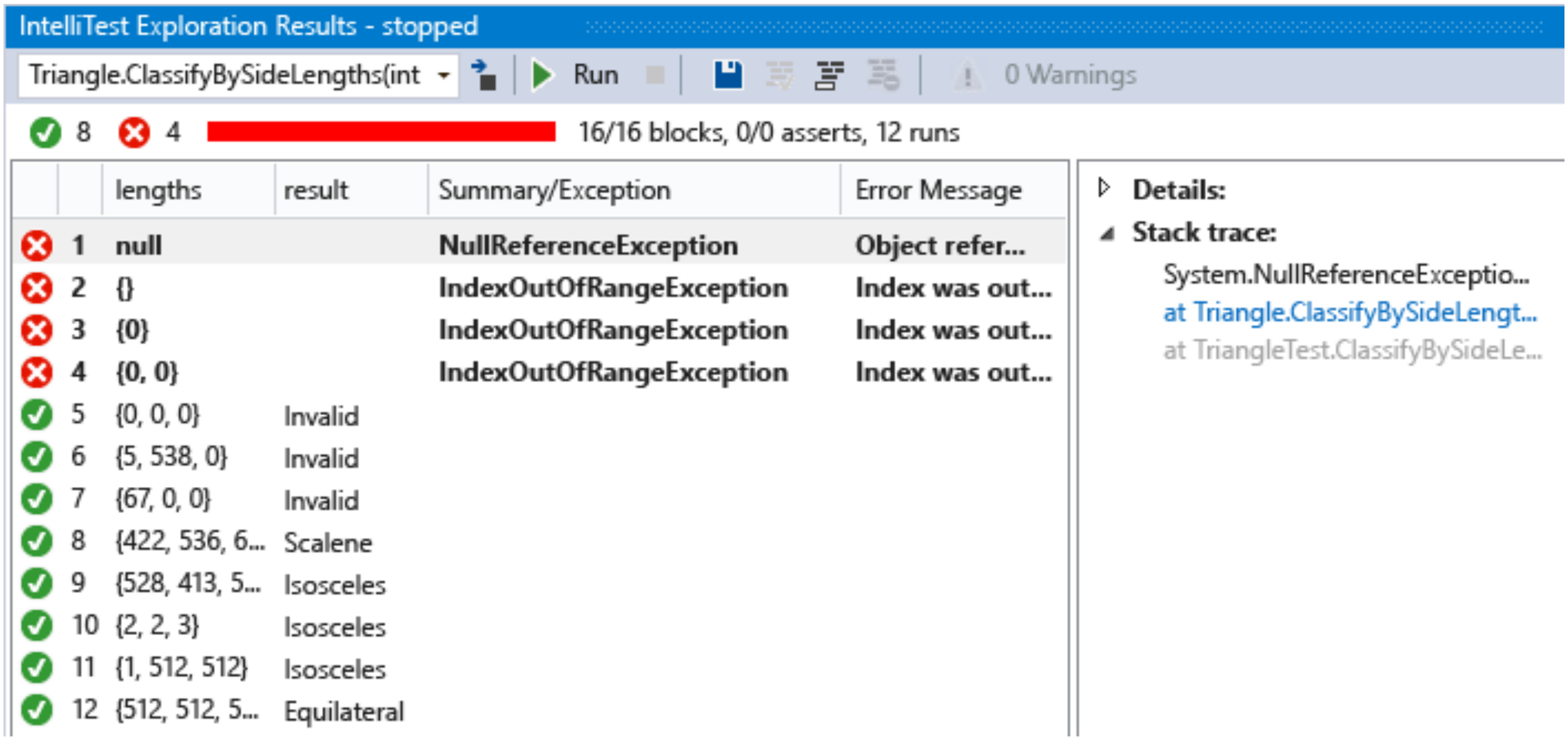
Microsoft Visual Studio 2015 IntelliTest





https://msdn.microsoft.com/en-us/library/dn823749.aspx

Microsoft Visual Studio 2015
IntelliTest



https://msdn.microsoft.com/en-us/library/dn823749.aspx

http://www.codehunt.com

SELECT SECTOR

CHANGE ZONE

00 TRAINING 00/03

01 ARITHMETIC

02 LOOPS

03 LOOPS 2

04 CONDITIONALS

05 CONDITIONALS 2

06 STRINGS

07 STRINGS 2

08 NESTED LOOPS

09 1D ARRAYS

10 JAGGED ARRAYS

11 ARRAYS 2

12 SEARCH SORT

13 CYPHERS

14 PUZZLES

HUNTER STATS

01/15 SECTORS UNLOCKED

00/15 SECTORS COMPLETED

Discover the arithmetic operation applied to 'x'.

CAPTURE CODE

RESET LEVEL

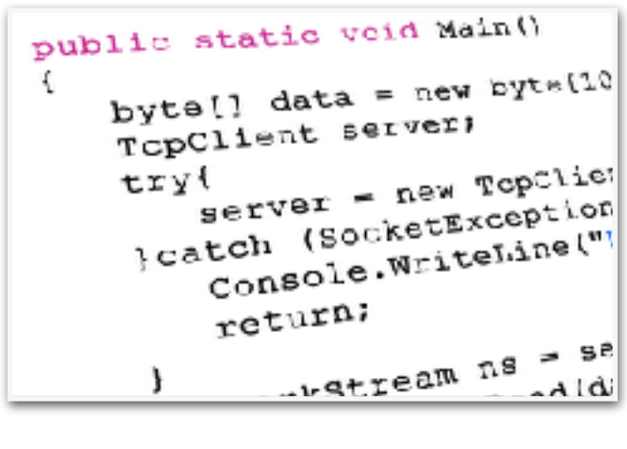
Java

```
1
2 public class Program {
3     public static int Puzzle(int x) {
4         return 0;
5     }
6 }
```

	X	EXPECTED RESULT	YOUR RESULT	DESCRIPTION
✗	0	1	0	Mismatch
✓	-1	0	0	
@	Well done so far. Look at numbers on line 4 to capture the code.			

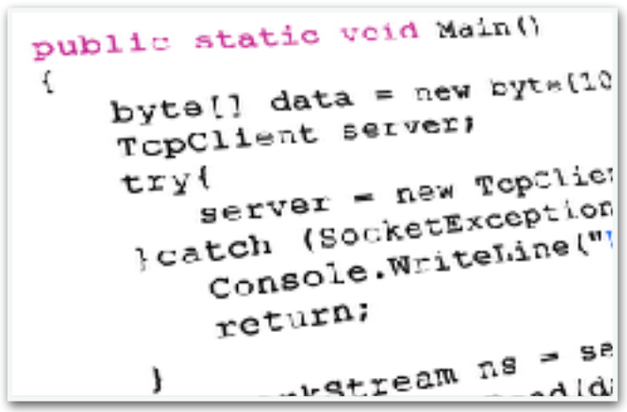
Code Hunt

Code Hunt

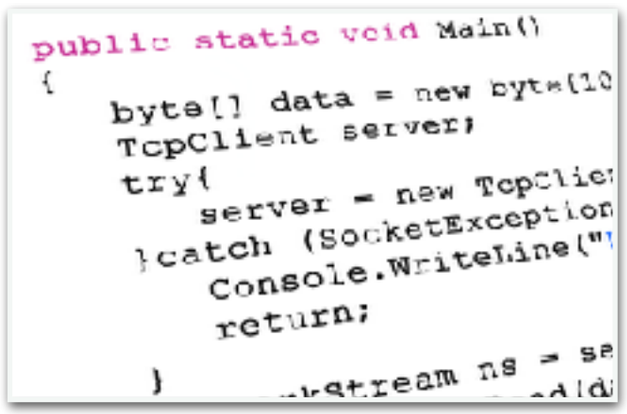


secret_Foo(x)

Code Hunt

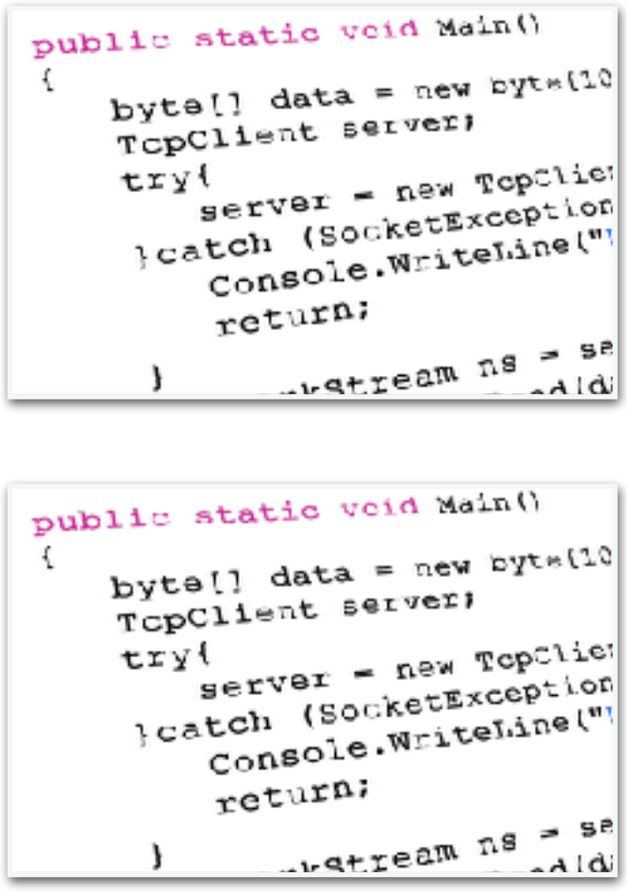


secret_Foo(x)



user_Foo(x)

Code Hunt



secret_Foo(x)

user_Foo(x)

```
p_test(x) {
  r0 = secret_Foo(x)
  r1 = user_Foo(x)
  assert r0==r1
}
```

Code Hunt

```
public static void Main()
{
    byte[] data = new byte[10
    TcpClient server;
    try{
        server = new TcpClie
    }catch (SocketException
        Console.WriteLine("
        return;
    }
        NetworkStream ns = se
```

```
public static void Main()
{
    byte[] data = new byte[10
    TcpClient server;
    try{
        server = new TcpClie
    }catch (SocketException
        Console.WriteLine("
        return;
    }
        NetworkStream ns = se
```

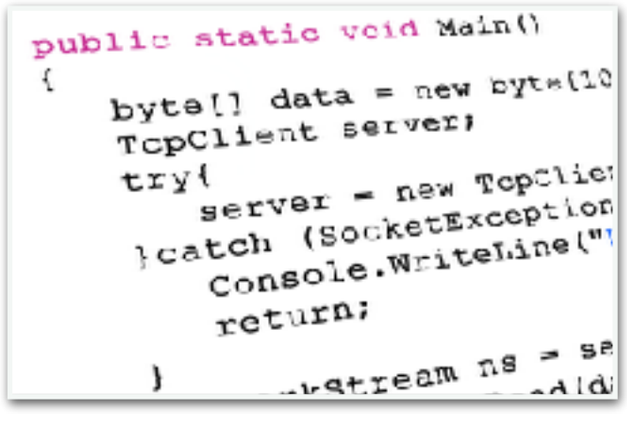
secret_Foo(x)

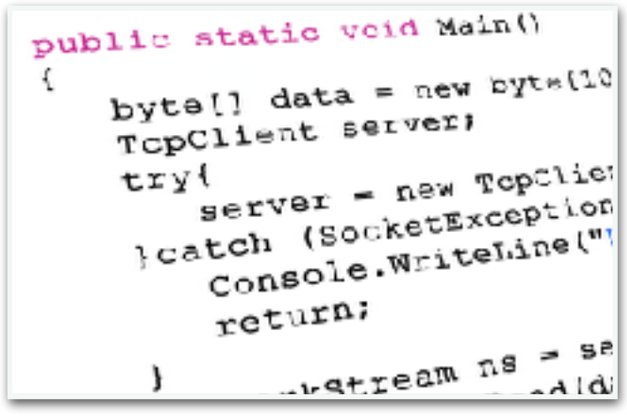
user_Foo(x)

```
p_test(x) {
  r0 = secret_Foo(x)
  r1 = user_Foo(x)
  assert r0==r1
}
```

PEX

Code Hunt





secret_Foo(x)

user_Foo(x)

```
p_test(x) {
  r0 = secret_Foo(x)
  r1 = user_Foo(x)
  assert r0==r1
}
```

PEX

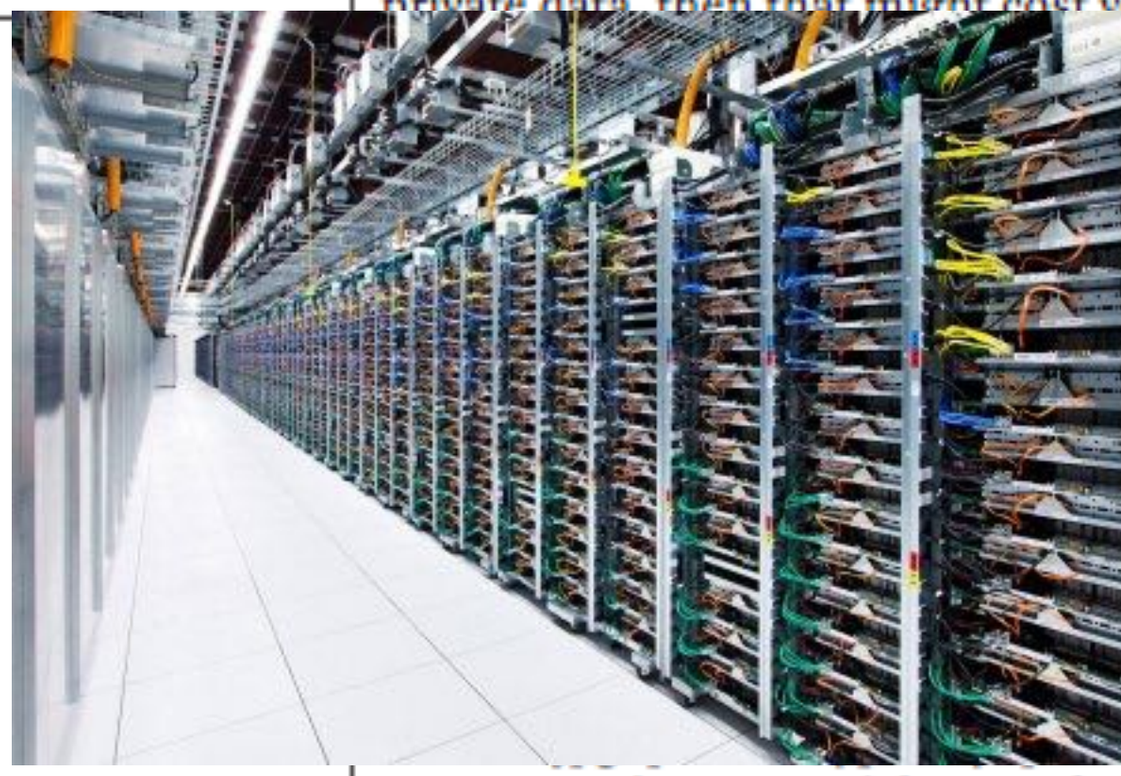
DOI:10.1145/2093548.2093564

Article development led by acmqueue
queue.acm.org

SAGE has had a remarkable impact at Microsoft.

BY PATRICE GODEFROID, MICHAEL Y. LEVIN, AND DAVID MOLNAR

SAGE: Whitebox Fuzzing for Security Testing



and its users millions of dollars. I
monthly security update costs y
$0.001 (one tenth of one cent) in ju
electricity or loss of productivity, th
this number multiplied by one b
lion people is $1 million. Of course
malware were spreading on your m
chine, possibly leaking some of yo
private data, then that might cost y

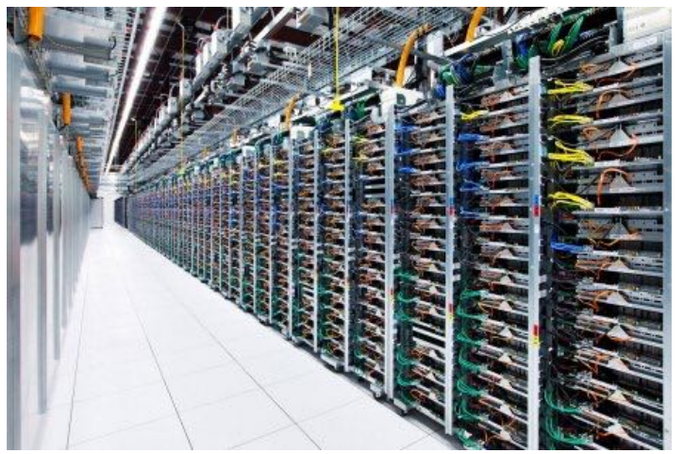
your operating system) has read t
image data, decoded it, created n
data structures with the decoded da
and passed those to the graphics ca
in your computer. If the code imp
menting that jpg parser contains
bug such as a buffer overflow that c
be triggered by a corrupted jpg imag
then the execution of this jpg pars
on your computer could potentially

Scalable Automated Guided Execution

- Primer herramienta en realizar **concolic execution** en binario x86
  - *"what you fuzz is what you ship,"* ya que los compiladores pueden realizar transformaciones que afectan la seguridad.
- Security bugs: los programadores pueden fallar en alocar memoria o manipular buffers apropiadamente, resultando en **vulnerabilidades**.

White-Box Fuzzing con SAGE

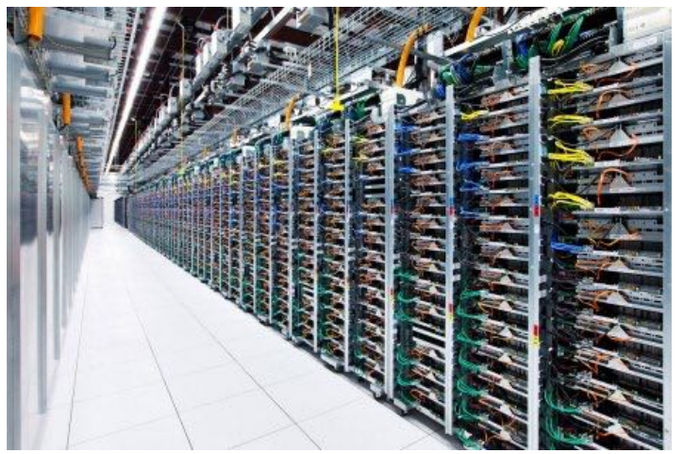
- La exploración de program paths se realiza paralelamente
- Cuanto más diverso y rico el conjunto inicial de inputs (ie seeds o semillas), mayor profundidad se logrará durante el fuzzing
- Utiliza heurísticas para maximizar el cubrimiento de código tan rápido como sea posible, con el objeto de encontrar bugs más rápidamente.
- Las **optimizaciones** son cruciales para manejar el **ENORME** tamaño de las path conditions.



SAGE

Microsoft

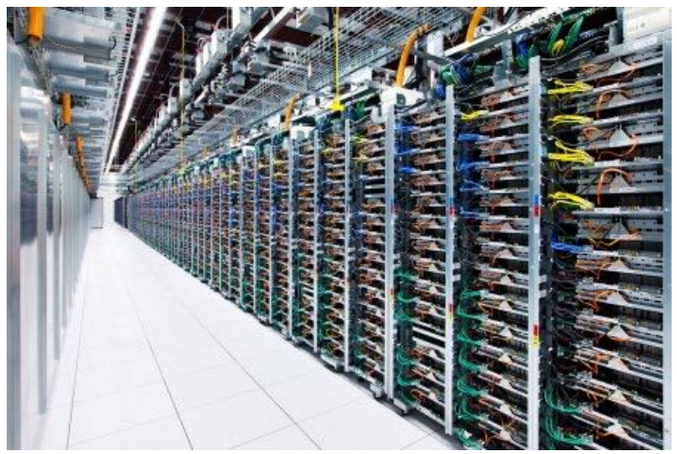
- *"Since 2008, SAGE has been running 24/7 on approximately 100-plus machines/cores automatically fuzzing hundreds of applications in Microsoft security testing labs"*



SAGE

Microsoft

- *"Since 2008, SAGE has been running 24/7 on approximately 100-plus machines/cores automatically fuzzing hundreds of applications in Microsoft security testing labs"*
- *"…has saved Microsoft millions of dollars as well as saved world time and energy, by avoiding expensive security patches to more than one billion PCs."*



SAGE

Microsoft

- *"Since 2008, SAGE has been running 24/7 on approximately 100-plus machines/cores automatically fuzzing hundreds of applications in Microsoft security testing labs"*
- *"…has saved Microsoft millions of dollars as well as saved world time and energy, by avoiding expensive security patches to more than one billion PCs."*
- *"The software running on your PC has been affected by SAGE."*

Ejecución Simbólica
Al Rescate!

Necesitamos…

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

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 descriptas en un lenguaje
  - SAT
  - UNSAT
  - UNKNOWN/TIMEOUT
- Si es SAT, da un valor para cada variable

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 descriptas 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)

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 descriptas 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!