CREST Tutorial

Moonzoo Kim

CS Dept. KAIST

CREST

Concolic test generation tool for C

CREST is an automatic test generation tool for C.

CREST works by inserting instrumentation code (using CIL) into a target program to perform symbolic execution concurrently with the concrete execution. The generated symbolic constraints are solved (using Yices) to generate input that drive the test execution down new, unexplored program paths.

CREST currently only reasons symbolically about linear, integer arithmetic. CREST should be usable on any modern Linux or Mac OS X system. For further building and usage information, see the README file. You may also want to check out the FAQ.

Further questions? Please e-mail the CREST-users mailing list (CREST-users at googlegroups.com, searchable at https://groups.google.com/forum/#!forum/crest-users).

A short paper and technical report about the search strategies in CREST are available at the homepage of Jacob Burnim.

News: CREST 0.1.2 is now available. It can be downloaded at https://github.com/jburnim/crest/releases/tag/v0.1.2. This is a bug fix release -- several build issues are fixed, as well as a bug in instrumenting unary expressions.

News: Heechul Yun has extended CREST to support non-linear arithmetic (using Z3). For more information, see: https://github.com/heechul/crest-z3 and https://github.com/heechul/crest-z3/blob/master/README-z3.

is maintained by jburnim.

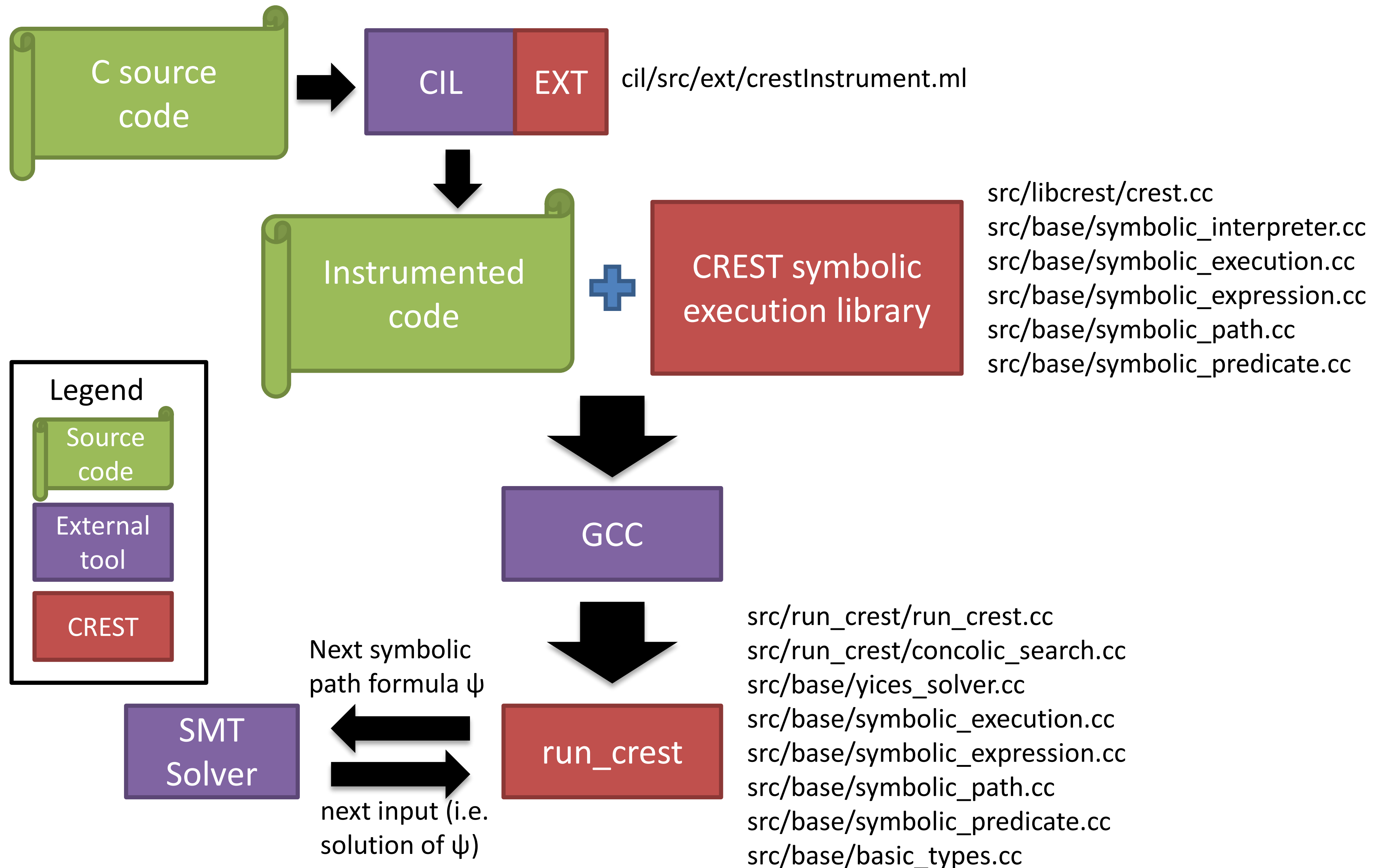
This page was generated by GitHub Pages using the Architect theme by Jason Long.

CREST

- CREST is a concolic testing tool for C programs
  - Generate test inputs automatically
  - Execute target under test on generated test inputs
  - Explore all possible execution paths of a target systematically

- CREST is a open-source re-implementation of CUTE
  - mainly written in C++
    - CREST's instrumentation is implemented as a module of CIL(C Intermetiate Language) written in Ocaml

Overview of CREST code



Moonzoo Kim
SWTV Group
KAIST

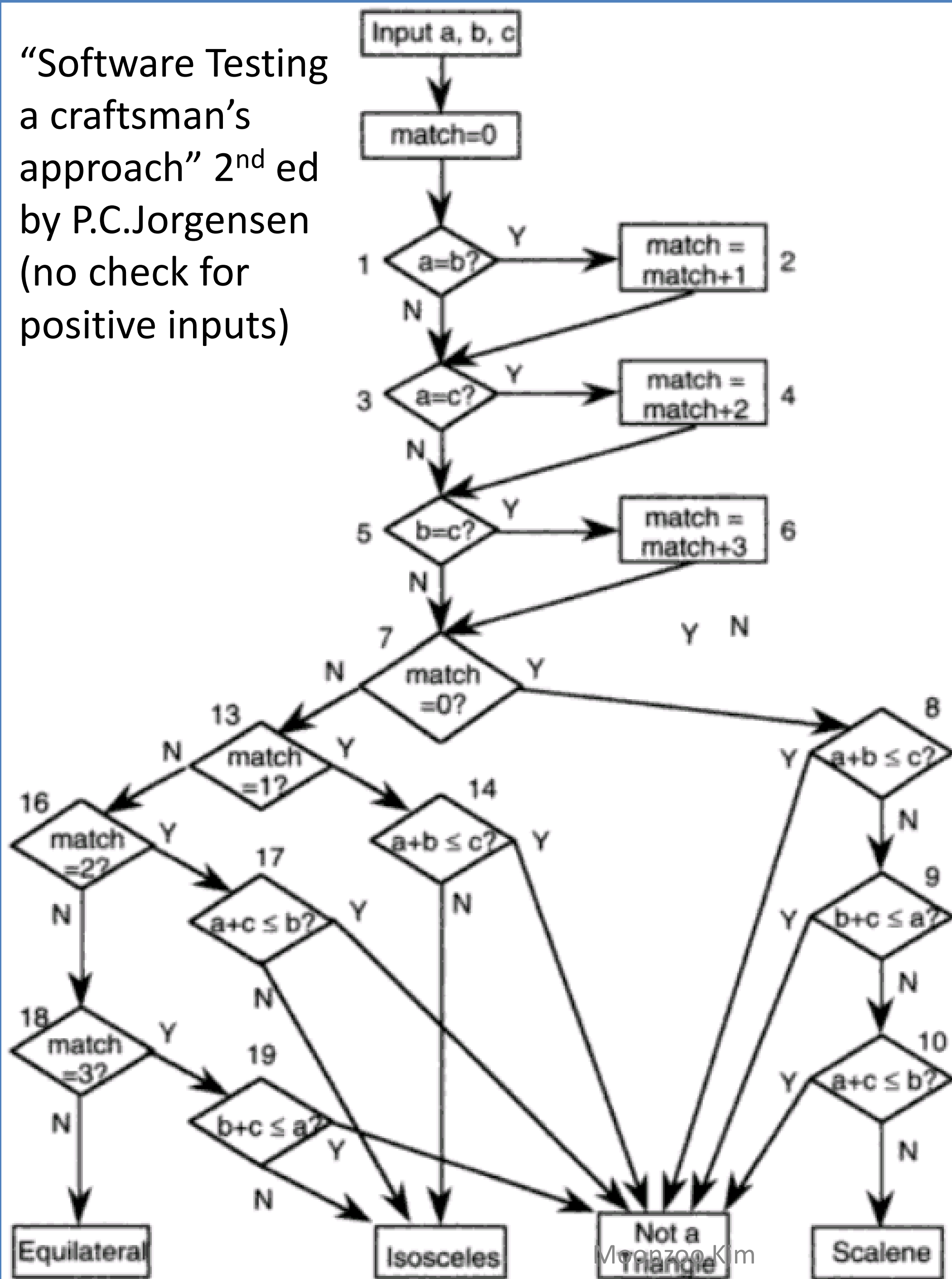
4 Main Steps of Concolic Testing

1. Instrumentation of a target program
   – To insert probes to build symbolic path formula
2. Transform a constructed symbolic path formula to SMT-compatible format
   – SMT solvers can solve simple formula only
3. Select one branch condition to negate
   – Core technique impacting both effectiveness and efficiency
4. Invoking SMT solvers on the SPF SMT formula
   – Selection of a SMT solver and proper configuration parameters

4 Main Tasks of Human Engineers

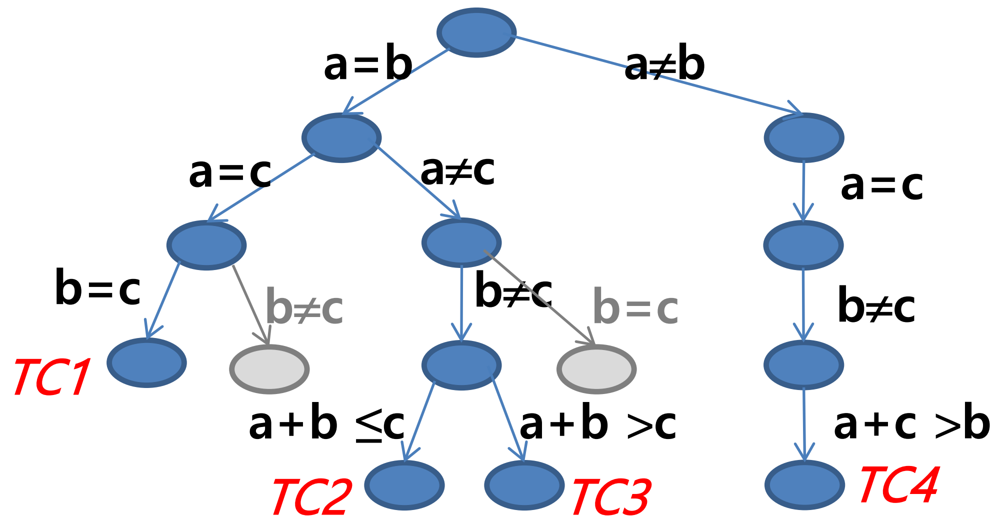
1. Adding proper assert() statements
   – W/o assert(), no test results obtained
2. Selection of symbolic variables in a target program
   – Identify which parts of a target program are most important
3. Construction of symbolic external environment
   – To detect real bugs
4. Performance tuning and debugging
   – To obtain better concolic testing results

```
1 #include <crest.h>
2 main() {
3    int a,b,c, match=0;
4    CREST_int(a); CREST_int(b); CREST_int(c);
     // filtering out invalid inputs
5    if(a <= 0 || b<= 0 || c<= 0) exit(0);
6    printf("a,b,c = %d,%d,%d:",a,b,c);
7
8    //0: Equilateral, 1:Isosceles,
     // 2: Not a traiangle, 3:Scalene
9    int result=-1;
10    if(a==b) match=match+1;
11    if(a==c) match=match+2;
12    if(b==c) match=match+3;
13    if(match==0) {
14       if( a+b <= c) result=2;
15       else if( b+c <= a) result=2;
16       else if(a+c <= b) result =2;
17       else result=3;
18    } else {
19       if(match == 1) {
20          if(a+b <= c) result =2;
21          else result=1;
22       } else {
23          if(match ==2) {
24             if(a+c <=b) result = 2;
25             else result=1;
26          } else {
27             if(match==3) {
28                if(b+c <= a) result=2;
29                else result=1;
30             } else result = 0;
31          } }}
32    printf("result=%d\n",result);
33 }
```



Concolic Testing the Triangle Program

Test case	Input (a,b,c)	Executed path conditions (PC)	Next PC	Solution for the next PC
1	1,1,1	a=b Æa=c Æb=c	a=b Æa=c Æb≠c	Unsat
			a=b Æa≠c	1,1,2
2	1,1,2	a=b Æa≠c Æb≠c Æa+b ≤c	a=b Æa≠c Æb≠c Æa+b > c	2,2,3
3	2,2,3	a=b Æa≠c Æb≠c Æa+b >c	a=b Æa≠c Æb=c	Unsat
			a≠b	2,1,2
4	2,1,2	a≠b Æa=c Æb≠c Æa+c>b	a≠bÆa=c Æb≠c Æa+c ≤b	2,5,2



CREST Commands

- `crestc <filename>.c`
  - Output
    - `<filename>.cil.c` // instrumented C file
    - `branches` // lists of paired branches
    - `<filename>` // executable file
- `run_crest ./filename <n> -[dfs|cfg|random|random_input|hybrid] [-TCDIR <tc_folder>]`
  - `<n>`: # of iterations/testings
  - `dfs`: depth first search (but in reverse order)
  - `cfg`: uncovered branch first
  - `random`: negated branch is randomly selected
  - `random_input`: pure random input
  - `hybrid`: combination of dfs and random
  - Output (updating at each iteration)
    - `input`: containing concrete values of symbolic variable
    - `szd_execution`: symbolic execution path
    - `coverage`: coverage achieved so far
    - A test case file in <tc_folder> if –TCDIR option is given

Instrumented C Code

```
#line 10
 { /* Creates symbolic expression a==b */
  __CrestLoad(36, (unsigned long )(& a), (long long )a);
  __CrestLoad(35, (unsigned long )(& b), (long long )b);
  __CrestApply2(34, 12, (long long )(a == b));
  if (a == b) {

    __CrestBranch(37, 11, 1); //extern void __CrestBranch(int id , int bid , unsigned char b )
    __CrestLoad(41, (unsigned long )(& match), (long long )match);
    __CrestLoad(40, (unsigned long )0, (long long )1);
    __CrestApply2(39, 0, (long long )(match + 1));
    __CrestStore(42, (unsigned long )(& match));
     match ++;

  } else {
    __CrestBranch(38, 12, 0);
  } }
```

Control dependency v.s. Data dependency

- match has control dependency on a and b
- match does not have data dependency on a and b

Execution Snapshot (1/2)

```
moonzoo@checker6:~/cs492d/crest/example$ run_crest ./triangle 100  -dfs -TCDIR testcases
------------------------
Iteration 1 (0s, 0.0s): covered 1 branches [1 reach funs, 32 reach branches].(1, 0)
------------------------
Iteration 2 (0s, 0.16s): covered 3 branches [1 reach funs, 32 reach branches].(3, 1)
------------------------
Iteration 3 (1s, 0.32s): covered 5 branches [1 reach funs, 32 reach branches].(5, 3)
------------------------
a,b,c = 1,1,1:result=0
Iteration 4 (1s, 0.48s): covered 13 branches [1 reach funs, 32 reach branches].(13, 5)
------------------------
a,b,c = 1610612736,536870912,1:result=2
Iteration 5 (1s, 0.64s): covered 18 branches [1 reach funs, 32 reach branches].(18, 13)
------------------------
a,b,c = 1610612736,536870912,1610612736:result=2
Iteration 6 (1s, 0.80s): covered 20 branches [1 reach funs, 32 reach branches].(20, 18)
------------------------
a,b,c = 2,1,2:result=1
Iteration 7 (1s, 0.111s): covered 21 branches [1 reach funs, 32 reach branches].(21, 20)
------------------------
a,b,c = 1610612736,536870912,536870912:result=2
Iteration 8 (2s, 0.126s): covered 23 branches [1 reach funs, 32 reach branches].(23, 21)
------------------------
a,b,c = 1,2,2:result=1
Iteration 9 (2s, 0.145s): covered 24 branches [1 reach funs, 32 reach branches].(24, 23)
------------------------
a,b,c = 272629760,1346371584,809500672:result=2
Iteration 10 (2s, 0.165s): covered 26 branches [1 reach funs, 32 reach branches].(26, 24)
------------------------
a,b,c = 1108719680,34977856,1108457536:result=2
Iteration 11 (2s, 0.188s): covered 28 branches [1 reach funs, 32 reach branches].(28, 26)
------------------------
a,b,c = 276823855,276823823,268435440:result=3
Iteration 12 (2s, 0.225s): covered 29 branches [1 reach funs, 32 reach branches].(29, 28)
------------------------
a,b,c = 1610612736,1610612736,536870912:result=2
Iteration 13 (3s, 0.242s): covered 31 branches [1 reach funs, 32 reach branches].(31, 29)
------------------------
a,b,c = 2,2,1:result=1
Iteration 14 (3s, 0.275s): covered 32 branches [1 reach funs, 32 reach branches].(32, 31)
```

KAIST

Execution Snapshot (2/2)

```
...$ cat branches
1 16 /* target branches */
3 4
5 6
7 8
11 12
14 15
17 18
20 27
21 22
23 24
25 26
28 31
29 30
32 35
33 34
36 39
37 38
```

```
...$ cat coverage
3 /* Covered branch id */
4
5
6
7
8
11
12
14
15
17
18
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
```

```
[moonzoo@checker6:~/cs492d/crest/example$ cat testcases/input.14
5 /* type of a symbolic input variable */
2 /* value of a symbolic input variable */
5
2
5
1

moonzoo@checker6:~/cs492d/crest/example$ print_execution
Symbolic variables & input values
(a_1 = 2)      [ Line: 4, File: triangle.c ]
(b_1 = 2)      [ Line: 4, File: triangle.c ]
(c_1 = 1)      [ Line: 4, File: triangle.c ]

Symbolic path for the input
! (a_1 <= 0)    [ Line: 6, File: triangle.c ]
! (b_1 <= 0)    [ Line: 6, File: triangle.c ]
! (c_1 <= 0)    [ Line: 6, File: triangle.c ]
(a_1 == b_1)    [ Line: 12, File: triangle.c ]
! (a_1 == c_1)  [ Line: 13, File: triangle.c ]
! (b_1 == c_1)  [ Line: 14, File: triangle.c ]
! ((a_1 + b_1) <= c_1)  [ Line: 22, File: triangle.c ]

Sequence of reached branch ids
-1      [Function enters]
4       [ Line: 6, File: triangle.c ]
6       [ Line: 6, File: triangle.c ]
8       [ Line: 6, File: triangle.c ]
11      [ Line: 12, File: triangle.c ]
15      [ Line: 13, File: triangle.c ]
18      [ Line: 14, File: triangle.c ]
27      [ Line: 15, File: triangle.c ]
28      [ Line: 21, File: triangle.c ]
30      [ Line: 22, File: triangle.c ]
-2      [Function exits]
```

KAIST

Supported Symbolic Datatypes

- #define CREST_unsigned_char(x) __CrestUChar(&x)
- #define CREST_unsigned_short(x) __CrestUShort(&x)
- #define CREST_unsigned_int(x) __CrestUInt(&x)
- #define CREST_char(x) __CrestChar(&x)
- #define CREST_short(x) __CrestShort(&x)
- #define CREST_int(x) __CrestInt(&x)
- #define CREST_float(x) __CrestFloat(&x)
- #define CREST_double(x) __CrestDouble(&x)

Decision/Condition Coverage Analysis by CREST

```
1 int main(){
2     int A, B, C, D;
3     if (A && B || C && D){
4         printf("Yes\n");
5     }else{
6         printf("No\n");
7     }
8 }
```

- CREST transforms a compound predicate into atomic ones with nested conditional statements
- CREST consider all possible cases with short-circuit
- Branch coverage reported by CREST might be lower than actual branch coverage

```
1    if (A != 0) {
 2      __CrestBranch(5, 2, 1);      A == T
 3      if (B != 0) {
 4        __CrestBranch(10, 3, 1);   A == T && B == T
 5        printf("Yes\n");
 6      } else {
 7        __CrestBranch(11, 4, 0);   A == T && B != T
 8        goto _L;
 9      }
10    } else {
11      __CrestBranch(6, 5, 0);      A != TRUE
12      _L: /* CIL Label */
13      if (C != 0) {               (A != T || A == T && B != T)
14        __CrestBranch(16, 6, 1);  && C == T
15        if (D != 0) {             (A != T || A == T && B != T)
16          __CrestBranch(21, 7, 1); && C == T && D == T
17          printf("Yes\n");
18        } else {
19          __CrestBranch(22, 8, 0); (A != T || A == T && B != T)
20          printf("No\n");          && C == T && D != T
21        }
22      } else {
23        __CrestBranch(17, 9, 0);   (A != T || A == T && B != T)
24        printf("No\n");            && C != T
25      }
26    }
```