Fault-based testing

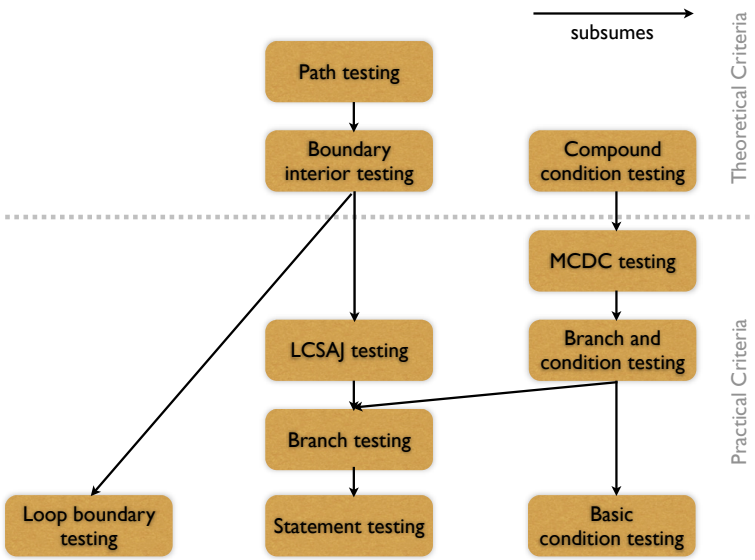
Automated testing and verification

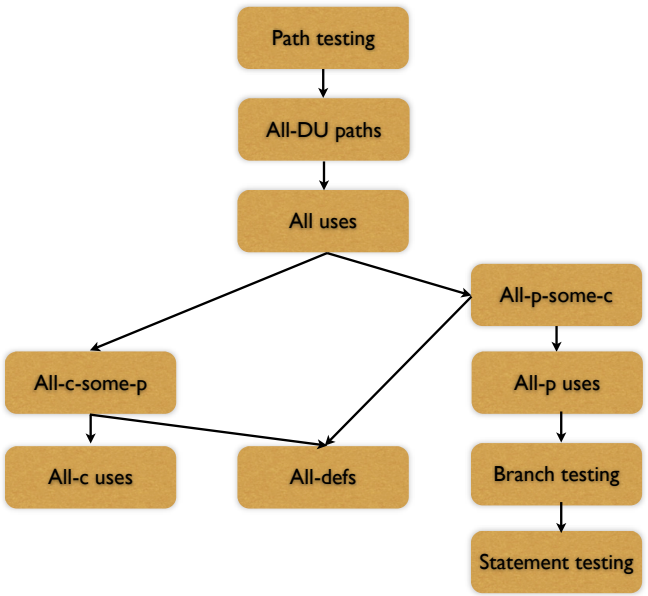
J.P. Galeotti - Alessandra Gorla

slides by Gordon Fraser

How good are my tests?

How good are my tests?

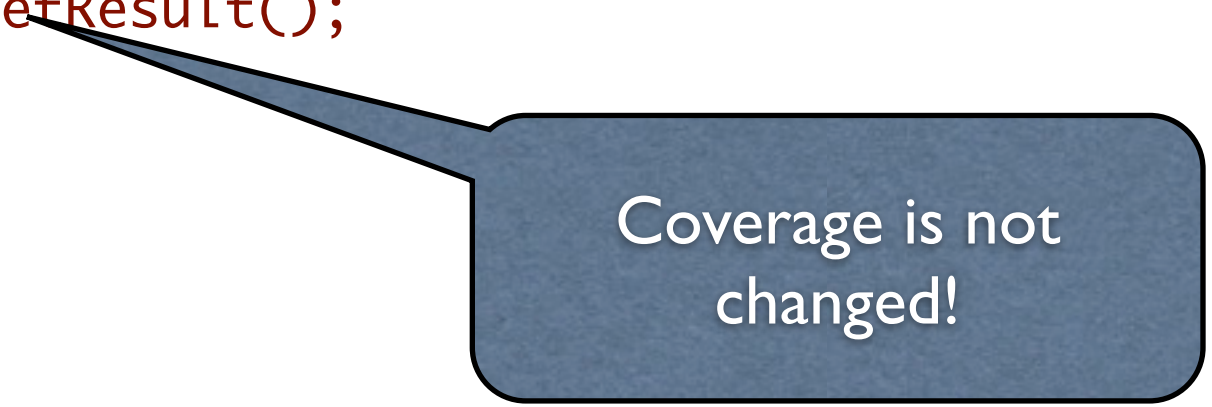




```
/**
 * Make sure Double.NaN is returned iff n = 0
 *
 */
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    assertTrue(Double.isNaN(std.getResult()));
    std.increment(1d);
    assertEquals(0d, std.getResult(), 0);
}
```

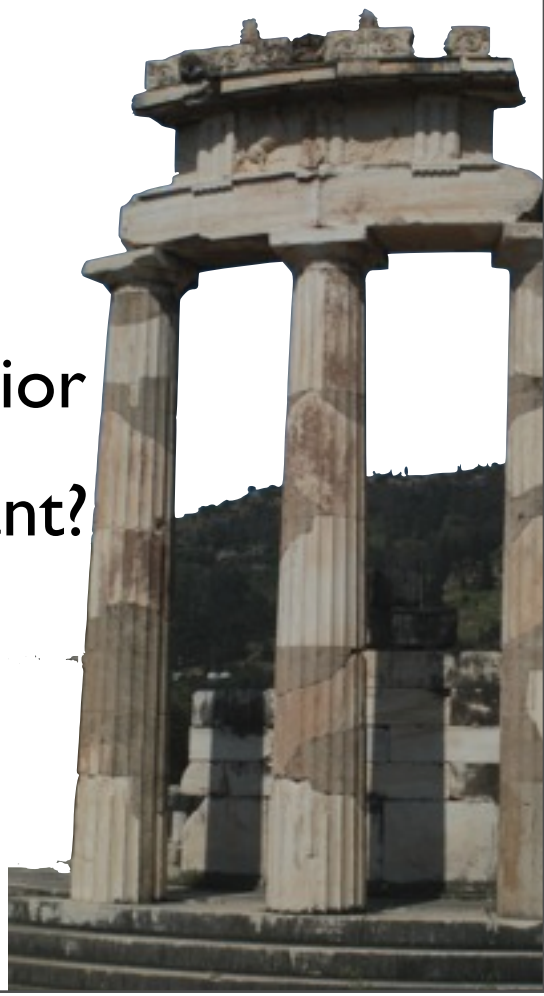
```
/**
 * Make sure Double.NaN is returned iff n = 0
 *
 */
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    Double.isNaN(std.getResult());
    std.increment(1d);
    std.getResult();
}
```

```
/**
 * Make sure Double.NaN is returned iff n = 0
 *
 */
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    Double.isNaN(std.getResult());
    std.increment(1d);
    std.getResult();
}
```



The Oracle Problem

- Executing all the code is not enough
- We need to check the functional behavior
- Does this thing actually do what we want?



Test Oracles

- Formal specifications
- Code assertions
- Test assertions
- Code contracts
- Manual oracles

How good are my tests?

- Coverage = how much of the code is executed
- But how much of the code is checked?
- We don't know where the bugs are
- But we know the bugs we have made in the past!

Learning from Mistakes

Learning from Mistakes

- **Key idea**: Learning from earlier mistakes to prevent them from happening again

Learning from Mistakes

- **Key idea**: Learning from earlier mistakes to prevent them from happening again
- **Key technique**: *Simulate earlier mistakes* and see whether the resulting defects are found

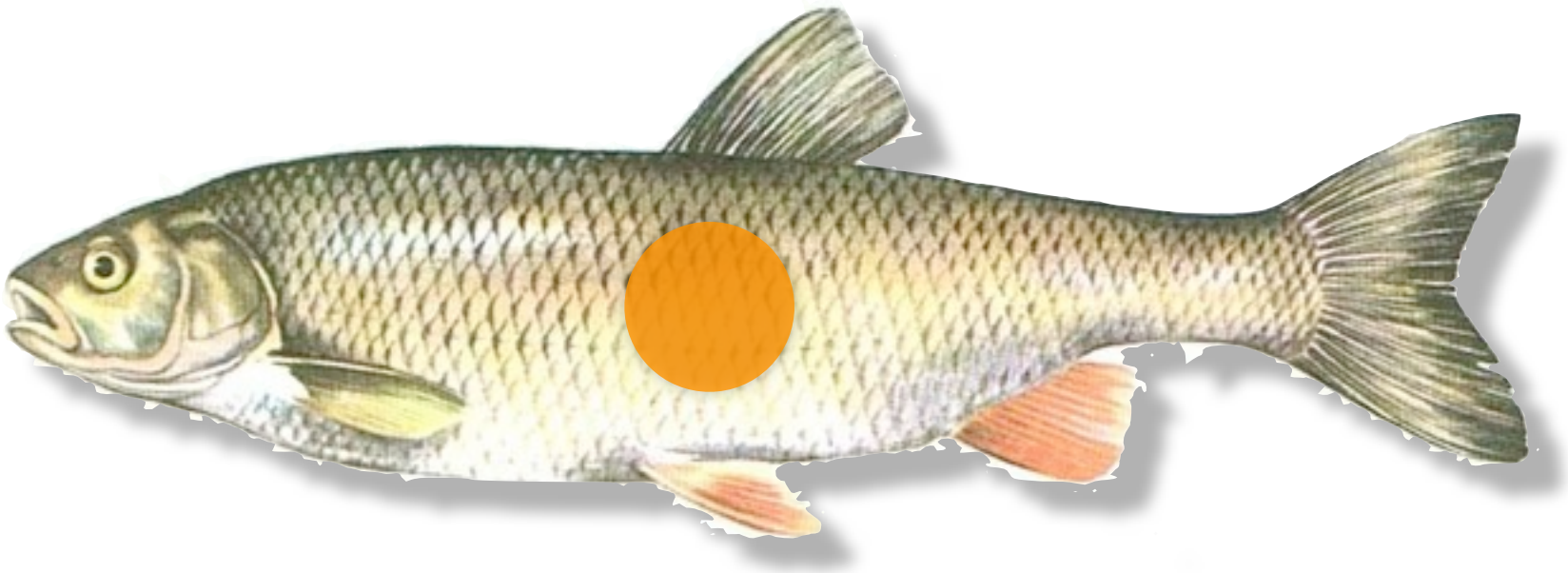
Learning from Mistakes

- **Key idea**: Learning from earlier mistakes to prevent them from happening again
- **Key technique**: *Simulate earlier mistakes* and see whether the resulting defects are found
- Known as *fault-based testing* or *mutation testing*

Fish Tag

- We catch 100 fish and tag them

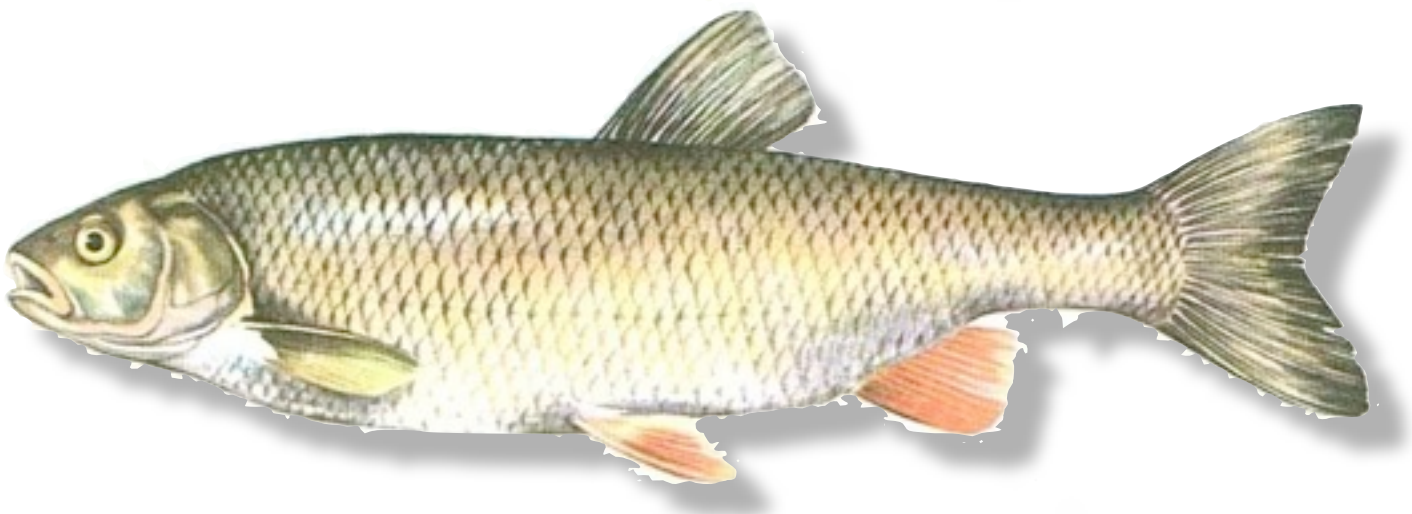
Fish Tag



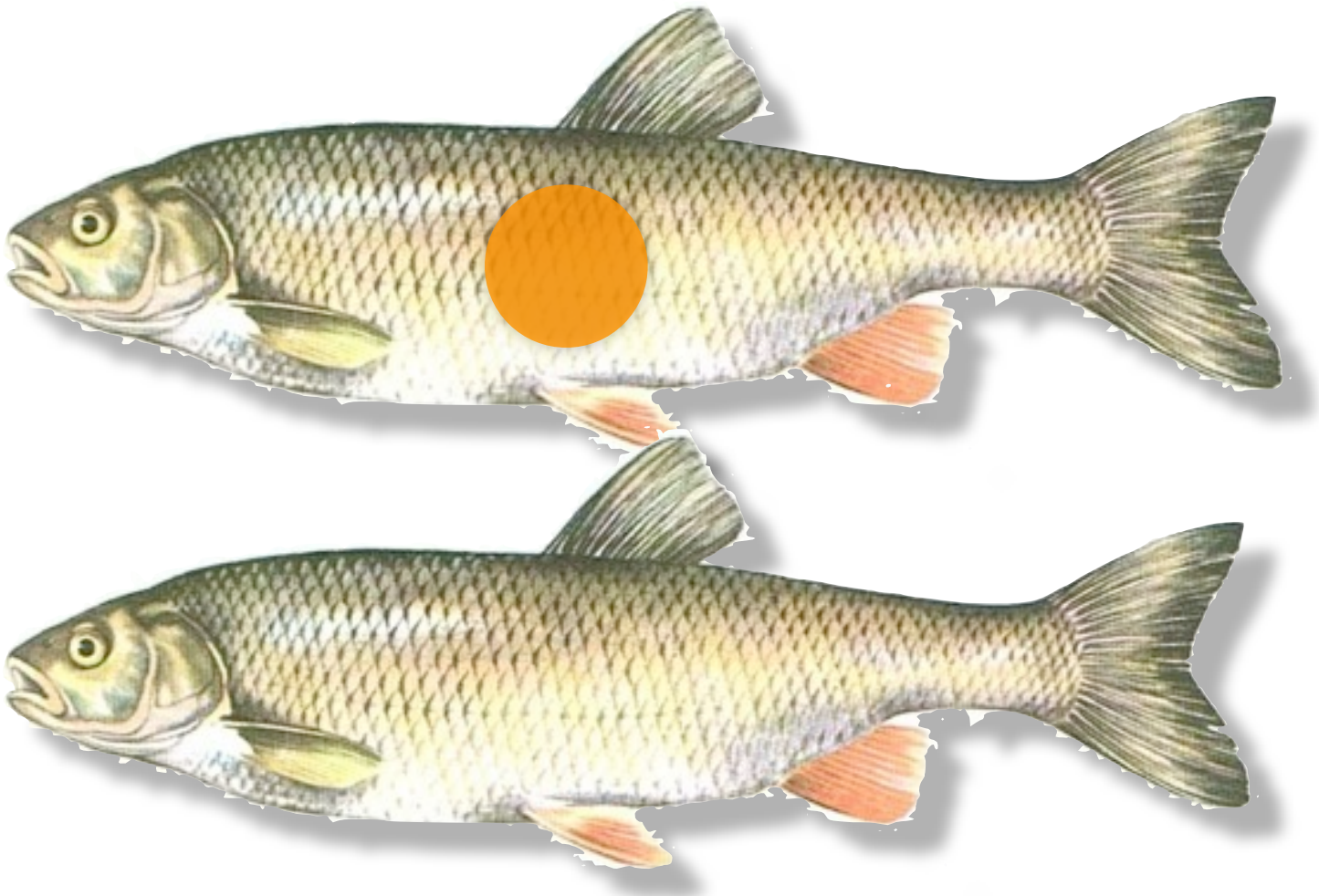
- We catch 100 fish and tag them

Counting Tags

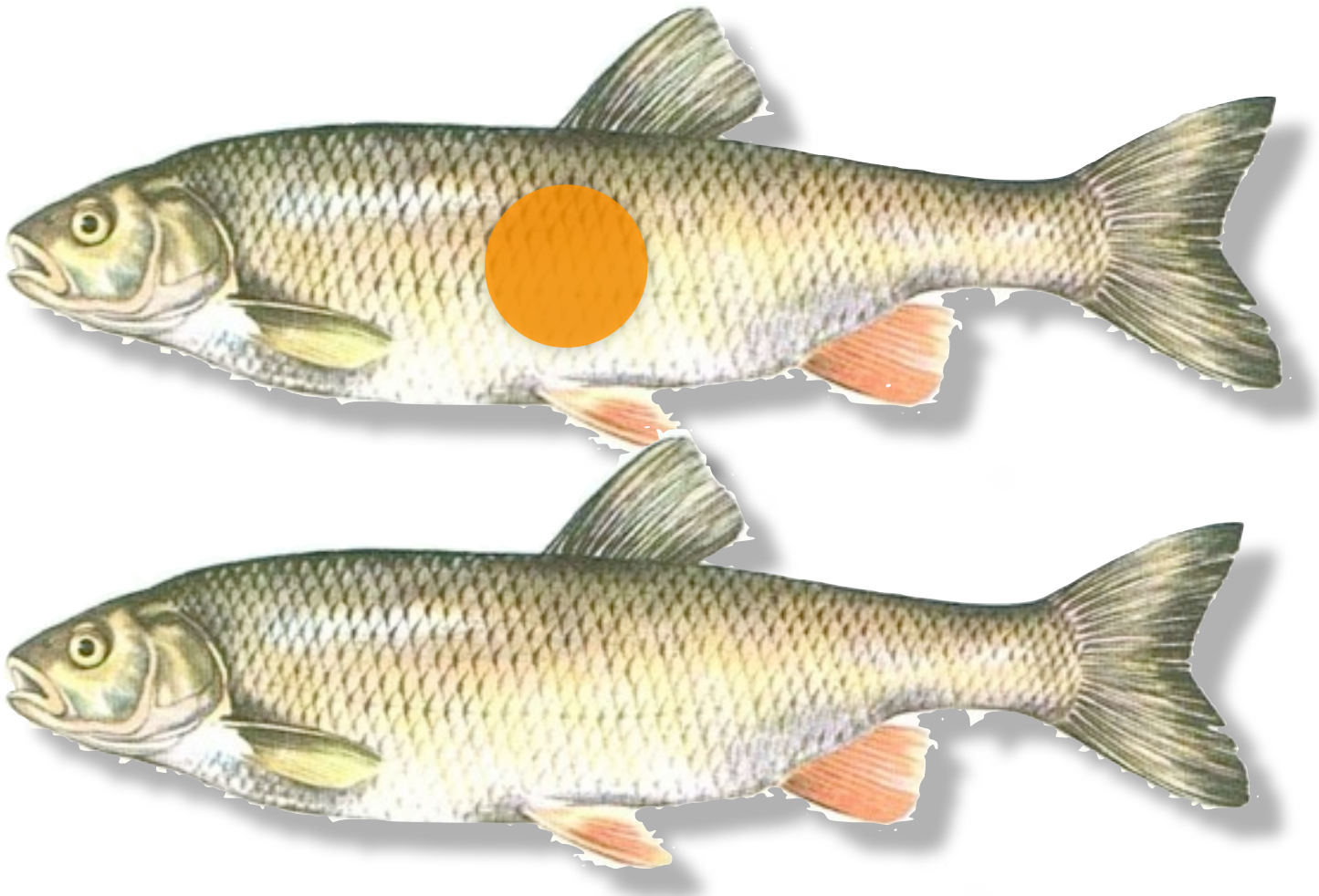
Counting Tags



Counting Tags



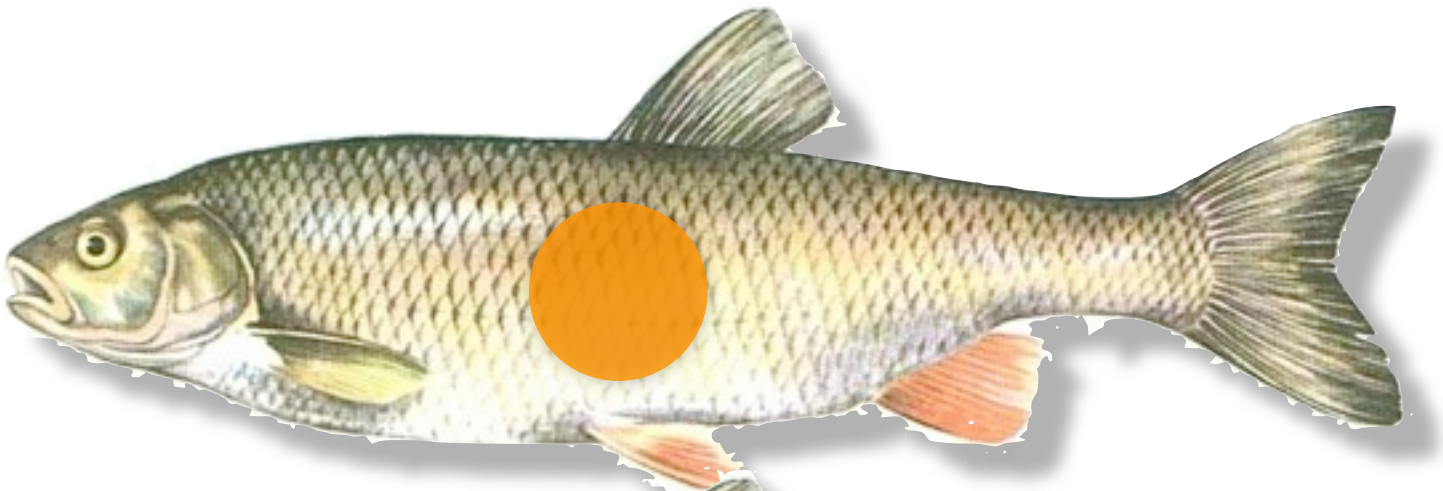
Counting Tags



80

Counting Tags

20



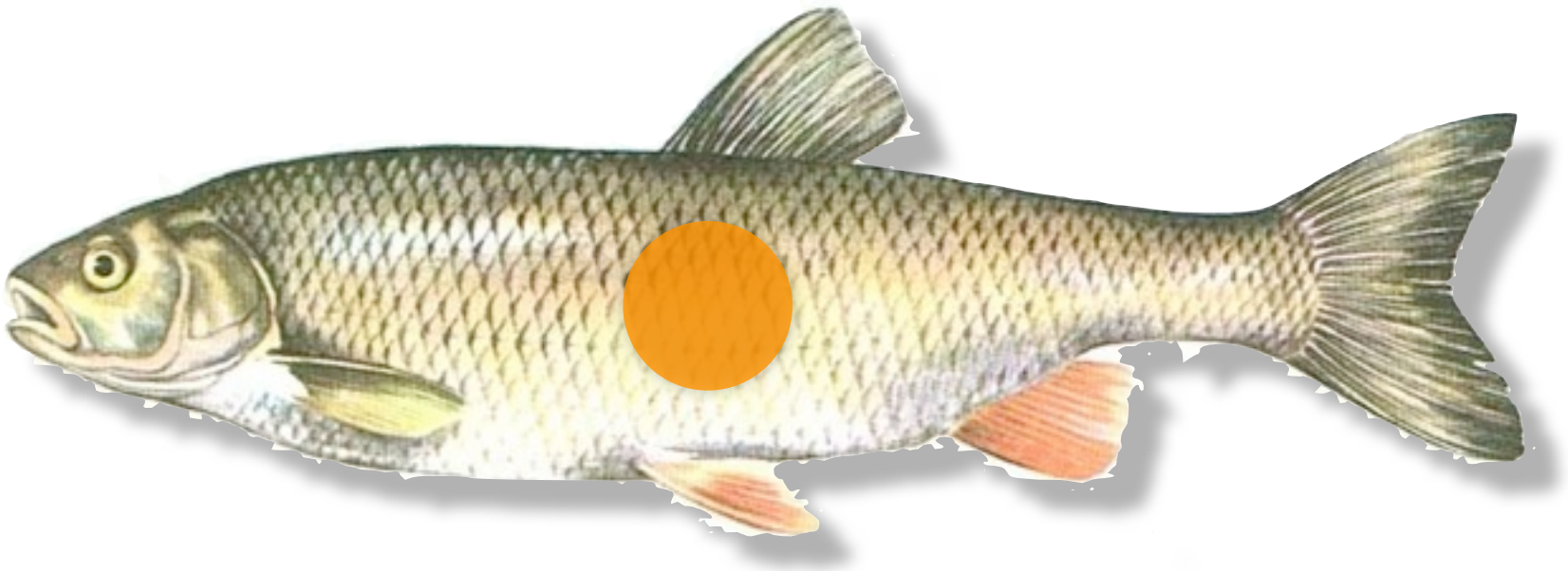
80

Estimate

$$\frac{100}{\text{untagged fish population}} = \frac{20}{80}$$


Program

A Mutant

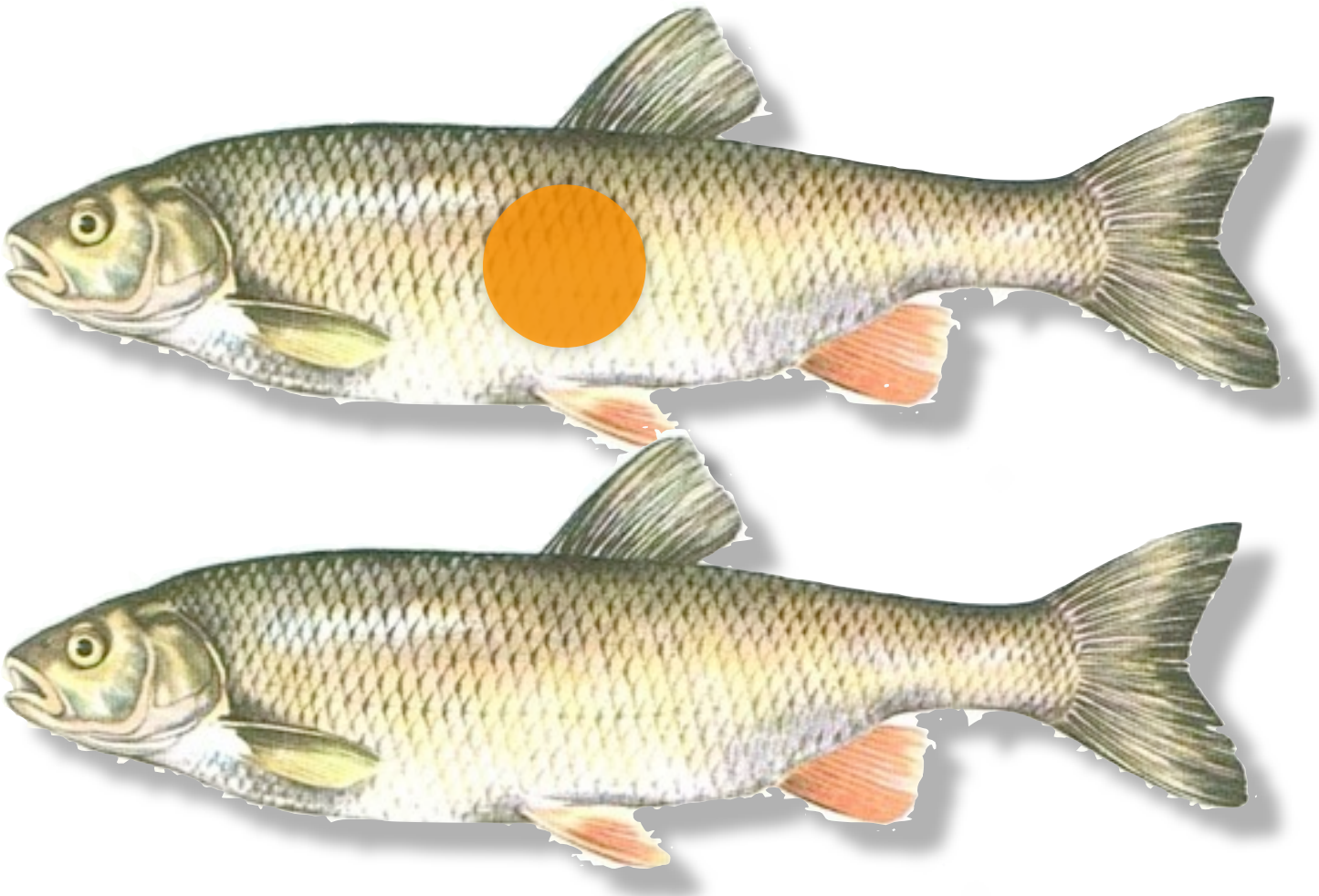


- We seed 100 mutations into the program

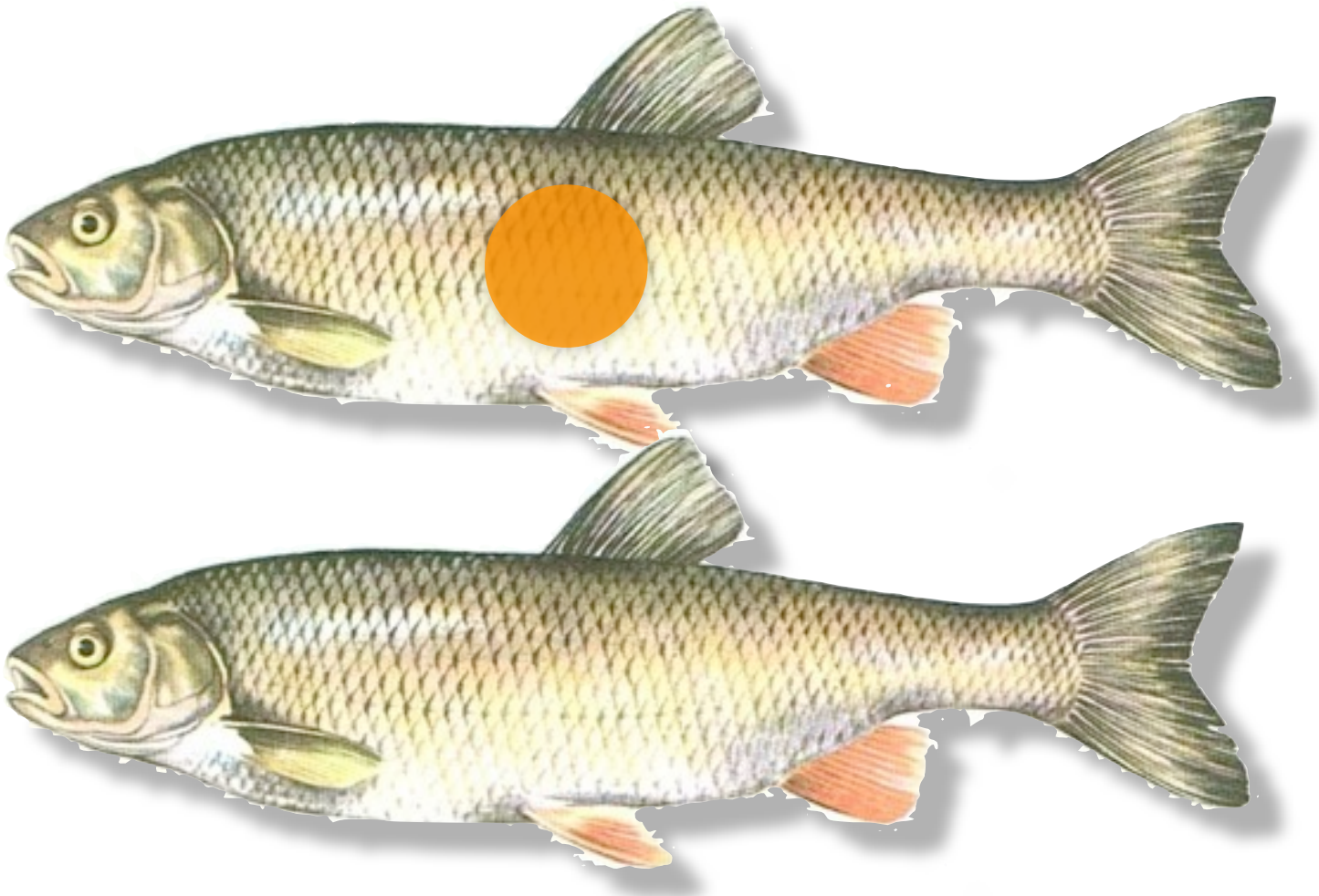
Counting Mutants

Counting Mutants

Counting Mutants



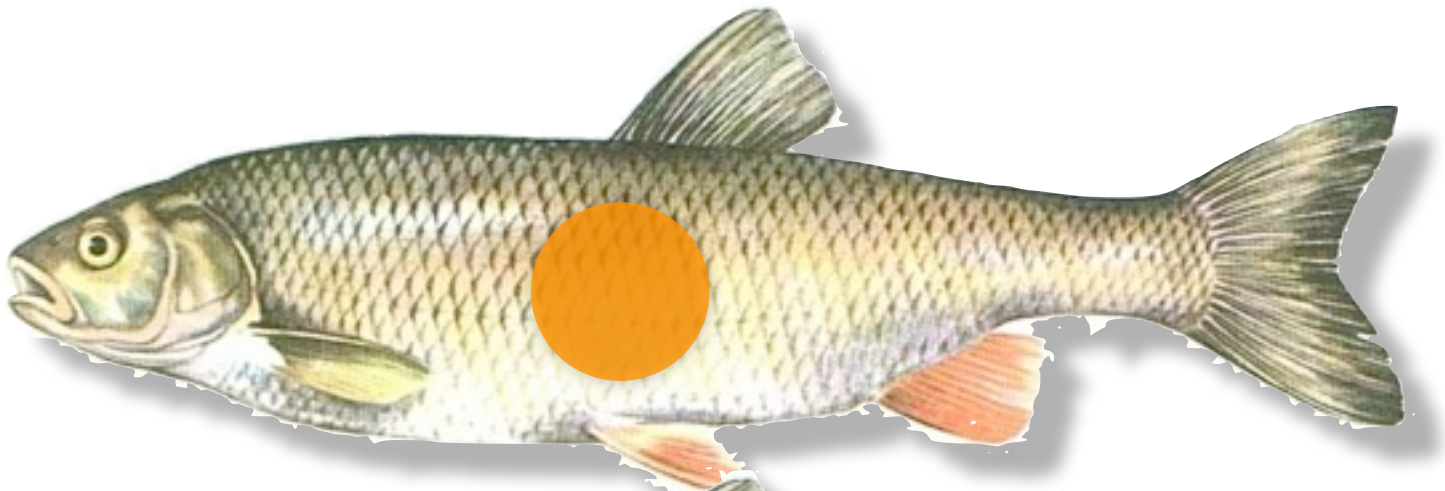
Counting Mutants



80

Counting Mutants

20



80

Estimate

$$\frac{100}{\text{remaining defects}} = \frac{20}{80}$$

Basic Assumptions

- Judge effectiveness of a test suite in finding real faults, by measuring how well it finds seeded fake faults.
- Valid to the extent that the seeded bugs are representative of real bugs
- Not necessarily identical (e.g., tagged fish not identical to non tagged fish); but the differences should not affect the selection

```
int do_something(int x, int
y) {
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int do_something(int x, int
y) {
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int a = do_something(5,
10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int
y) {
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int a = do_something(5,
10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int
y) {
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int do_something(int x, int
y) {
  if(x < y)
    return x-y;
  else
    return x*y;
}
```

Mutant

```
int a = do_something(5,
10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int
y) {
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int a = do_something(5,
10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int
y) {
  if(x < y)
    return x-y;
  else
    return x*y;
}
```

Mutant

```
int a = do_something(5,
10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int
y) {
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int do_something(int x, int
y) {
  if(x < y)
    return x-y;
  else
    return x*y;
}
```

Mutant

```
int a = do_something(5,
10);
assertEquals(a, 15);
```

Test

```
int a = do_something(5,
10);
assertEquals(a, 15);
```

Test

Mutants

- Mutant
  Slightly changed version of original program
- Syntactic change
  Valid (compilable code)
- Simple
  Programming “glitch”
- Based on faults
  Fault hierarchy

Generating Mutants

- Mutation operator
  Rule to derive mutants from a program
- Mutations based on real faults
  Mutation operators represent typical errors
- Dedicated mutation operators have been defined for most languages
- For example, > 100 operators for C language

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = abs(y);
    y = tmp;
  }

  return x;
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = abs(y);
    y = tmp;
  }

  return x;
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return -abs(x);
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return -abs(x);
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return 0;
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return 0;
}
```

AOR - Arithmetic Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

AOR - Arithmetic Operator Replacement

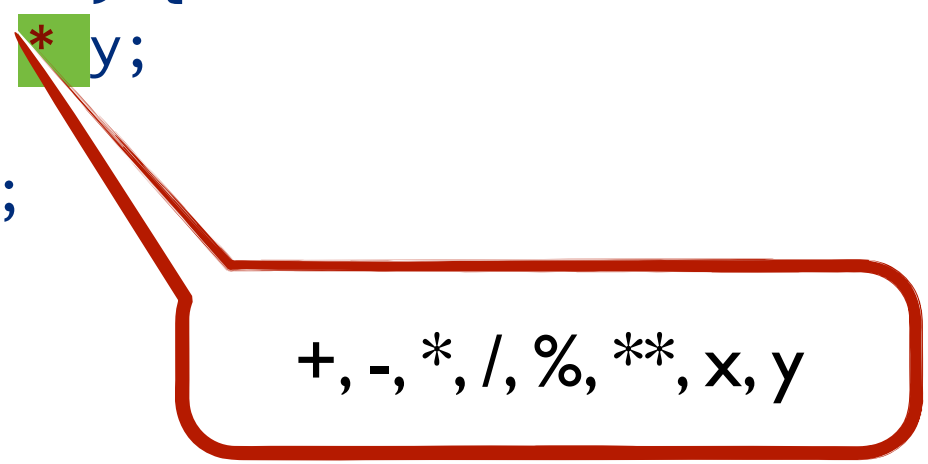
```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```

AOR - Arithmetic Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```

AOR - Arithmetic Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```



ROR - Relational Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

ROR - Relational Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y > 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

ROR - Relational Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y > 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

ROR - Relational Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y > 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

<, >, <=, >=, =,
!=, false, true

COR - Conditional Operator Replacement

```
if(a && b)
```

COR - Conditional Operator Replacement

```
if(a && b)
```

```
if(a || b)
```

COR - Conditional Operator Replacement

```
if(a && b)
```

```
if(a || b)
if(a & b)
```

COR - Conditional Operator Replacement

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
```

COR - Conditional Operator Replacement

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
```

COR - Conditional Operator Replacement

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
if(false)
```

COR - Conditional Operator Replacement

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
if(false)
if(true)
```

COR - Conditional Operator Replacement

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
if(false)
if(true)
if(a)
```

COR - Conditional Operator Replacement

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
if(false)
if(true)
if(a)
if(b)
```

SOR - Shift Operator Replacement

```
x = m << a
```

SOR - Shift Operator Replacement

```
x = m << a
```

```
x = m >> a
```

SOR - Shift Operator Replacement

```
x = m << a
```

```
x = m >> a
x = m >>> a
```

SOR - Shift Operator Replacement

```
x = m << a
```

```
x = m >> a
x = m >>> a
x = m
```

LOR - Logical Operator Replacement

```
x = m & n
```

LOR - Logical Operator Replacement

```
x = m & n
```

```
x = m | n
```

LOR - Logical Operator Replacement

```
x = m & n
```

```
x = m | n
x = m ^ n
```

LOR - Logical Operator Replacement

```
x = m & n
```

```
x = m | n
x = m ^ n
x = m
```

LOR - Logical Operator Replacement

```
x = m & n
```

```
x = m | n
x = m ^ n
x = m
x = n
```

ASR - Assignment Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

ASR - Assignment Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x -= y;
    y = tmp;
  }

  return x;
}
```

ASR - Assignment Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x -= y;
    y = tmp;
  }

  return x;
}
```

ASR - Assignment Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

+=, -=, *=, /=, %=,
&=,

UOI - Unary Operator Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

UOI - Unary Operator Insertion

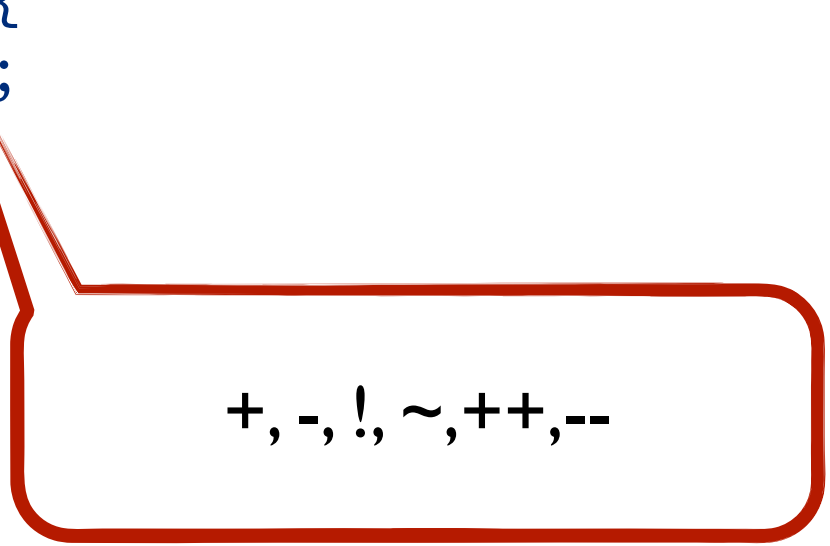
```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % +y;
    x = y;
    y = tmp;
  }

  return x;
}
```

UOI - Unary Operator Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % +y;
    x = y;
    y = tmp;
  }

  return x;
}
```

UOI - Unary Operator Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % +y;
    x = y;
    y = tmp;
  }

  return x;
}
```



UOD - Unary Operator Deletion

```
if !(a > -b)
```

UOD - Unary Operator Deletion

```
if !(a > -b)
```

```
if (a > -b)
```

UOD - Unary Operator Deletion

```
if !(a > -b)
```

```
if (a > -b)
if !(a > b)
```

SVR - Scalar Variable Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = y % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

SVR - Scalar Variable Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = y % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

SVR - Scalar Variable Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = y % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

tmp = x % y
tmp = x % x
tmp = y % y
x = x % y
y = y % x
tmp = tmp % y
tmp = x % tmp

OO Mutation

- So far, operators only considered method bodies
- Class level elements can be mutated as well:

OO Mutation

- So far, operators only considered method bodies
- Class level elements can be mutated as well:

```
public class test {
  // ..
  protected void do() {
    // ...
  }
}
```

OO Mutation

- So far, operators only considered method bodies
- Class level elements can be mutated as well:

```
public class test {
  // ..
  protected void do() {
    // ...
  }
}
```

```
public class test {
  // ..
  private void do() {
    // ...
  }
}
```

OO Mutation

- AMC - Access Modifier Change
- HVD - Hiding Variable Deletion
- HVI - Hiding Variable Insertion
- OMD - Overriding Method Deletion
- OMM - Overridden Method Moving
- OMR - Overridden Method Rename
- SKR - Super Keyword Deletion
- PCD - Parent Constructor Deletion
- ATC - Actual Type Change
- DTC - Declared Type Change
- PTC - Parameter Type Change
- RTC - Reference Type Change
- OMC - Overloading Method Change
- OMD - Overloading Method Deletion
- AOC - Argument Order Change
- ANC - Argument Number Change
- TKD - this Keyword Deletion
- SMV - Static Modifier Change
- VID - Variable Initialization Deletion
- DCD - Default Constructor 2

Interface Mutation

- Integration testing
- Change calling method by modifying the values that are sent to a called method
- Change a calling method by modifying the call
- Change a called method by modifying the values that enter/leave the method
- Change a called method by modifying statements that return from the method

Order of Mutants

- First order mutant (FOM)
  Exactly one mutation
- Each mutation operator yields a set of FOMs
- Number of FOMs
  ~ number of data references * number of data objects

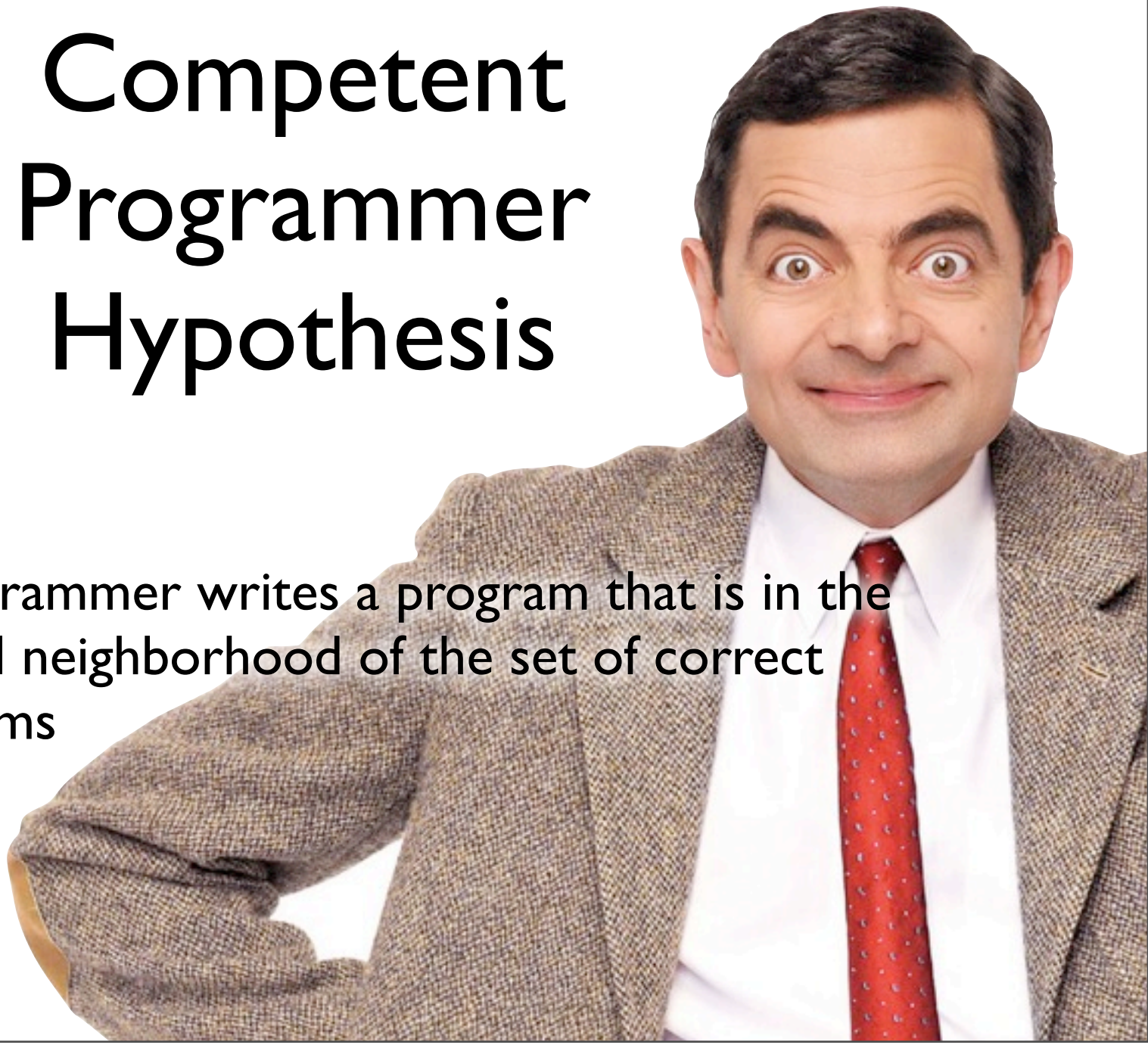
Order of Mutants

- First order mutant (FOM)
  Exactly one mutation
- Each mutation operator yields a set of FOMs
- Number of FOMs
  ~ number of data references * number of data objects
- Higher order mutant (HOM)
  Mutant of mutant

Order of Mutants

- First order mutant (FOM)
  Exactly one mutation
- Each mutation operator yields a set of FOMs
- Number of FOMs
  ~ number of data references * number of data objects
- Higher order mutant (HOM)
  Mutant of mutant
- $\#HOM = 2^{\#FOM} - 1$

Competent Programmer Hypothesis



A programmer writes a program that is in the general neighborhood of the set of correct programs

Coupling Effect

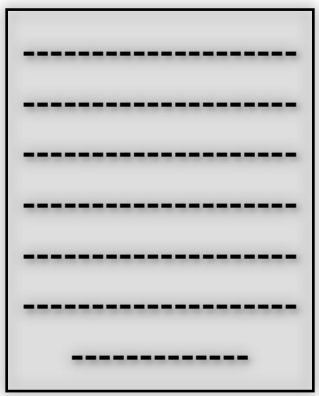
Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.

Mutation testing focuses on
First Order Mutants

Competent Programmer Hypothesis

Coupling Effect

Original Program

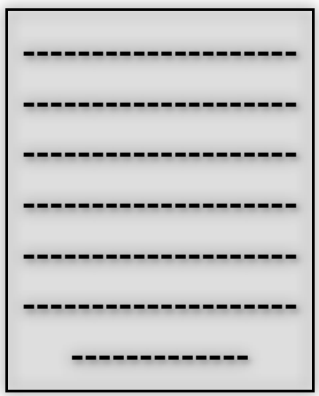


Operators

Test 1 | Test 2 | Test 3

Tests

Original Program

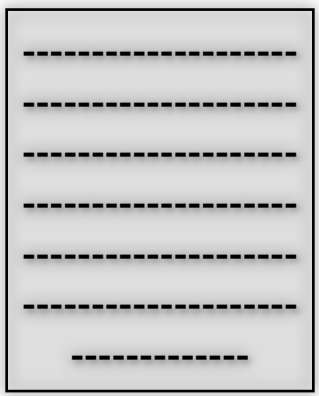


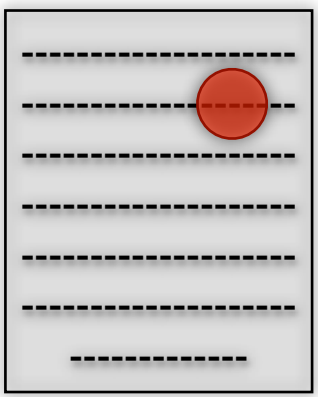
Operators

Test 1 | Test 2 | Test 3

Tests

Original Program



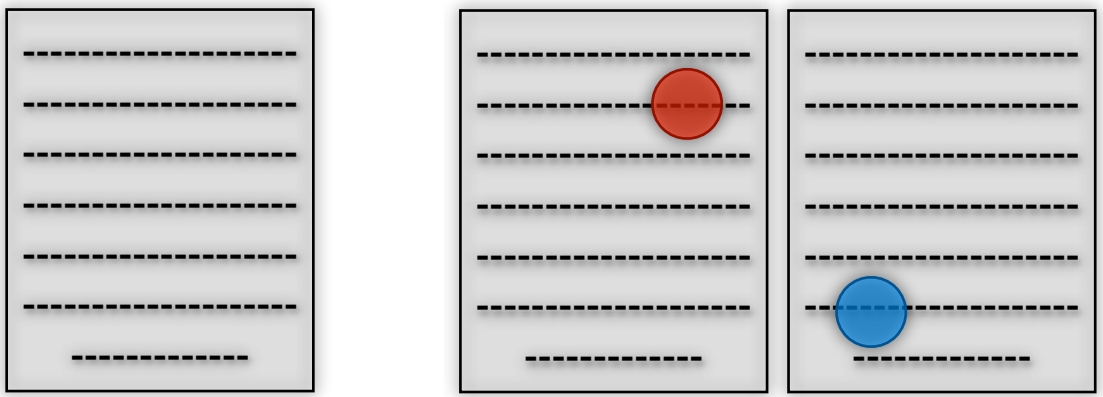


Operators

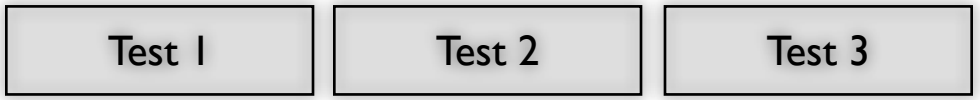
Test 1 | Test 2 | Test 3

Tests

Original Program

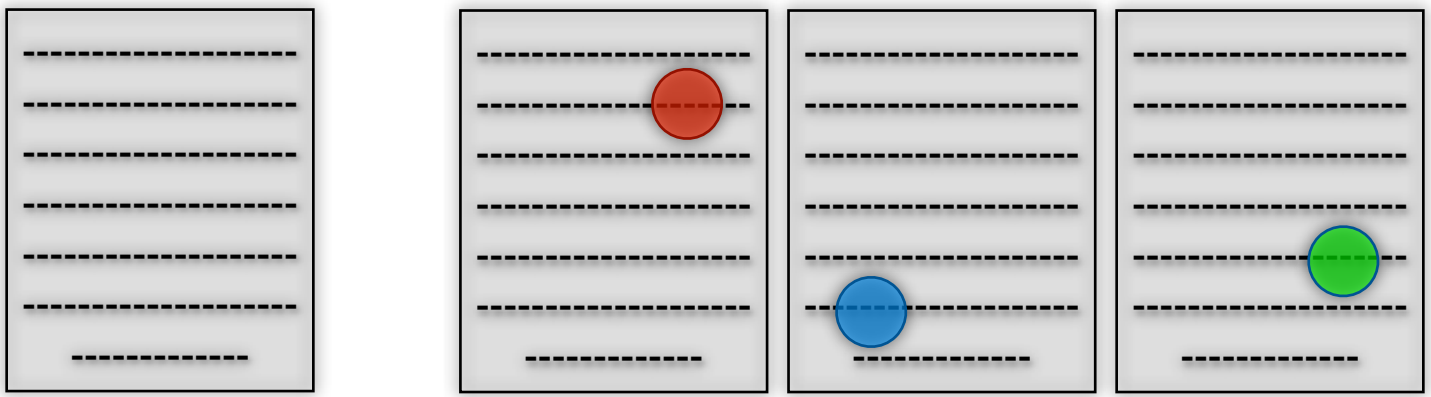


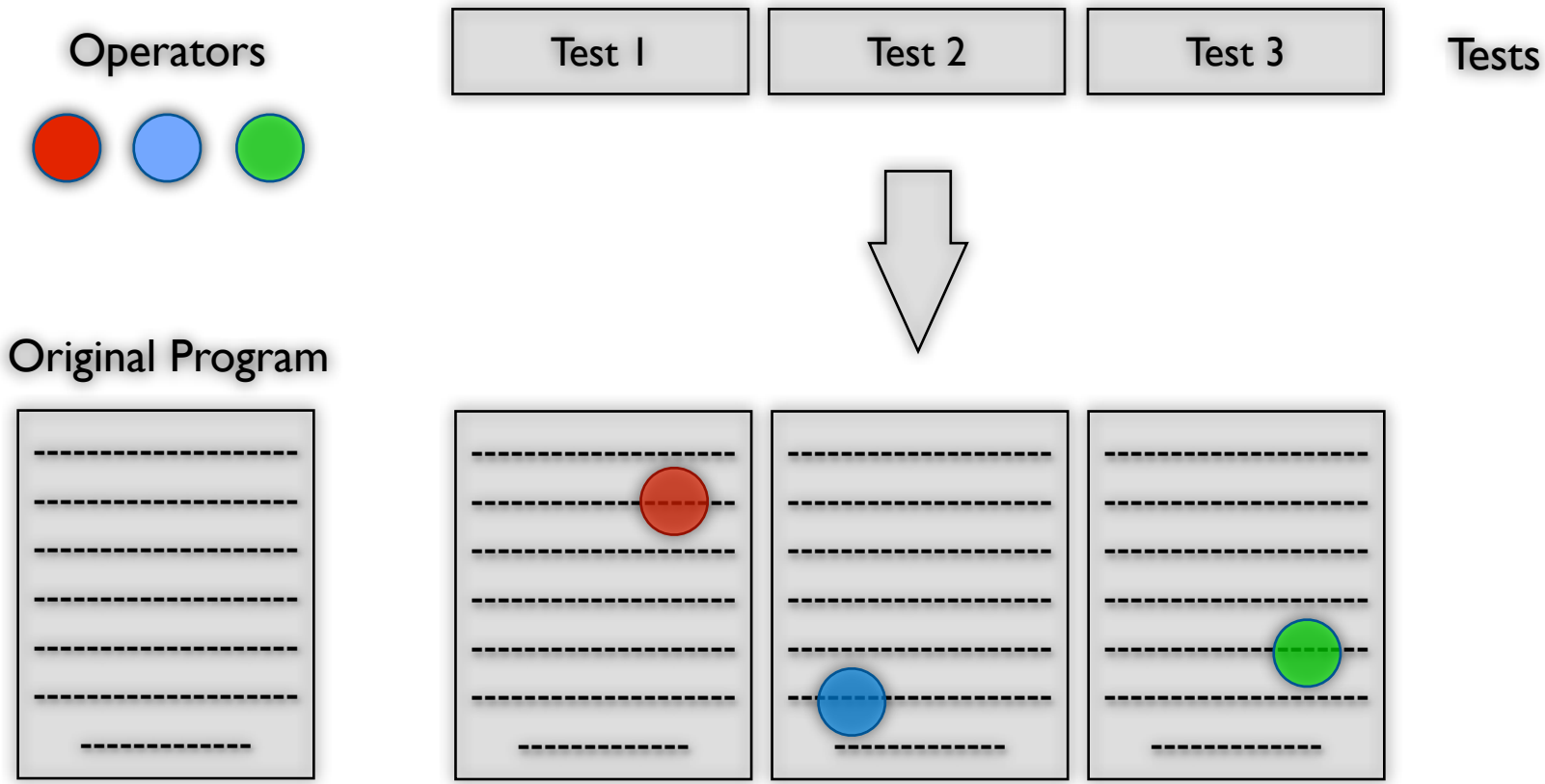
Operators

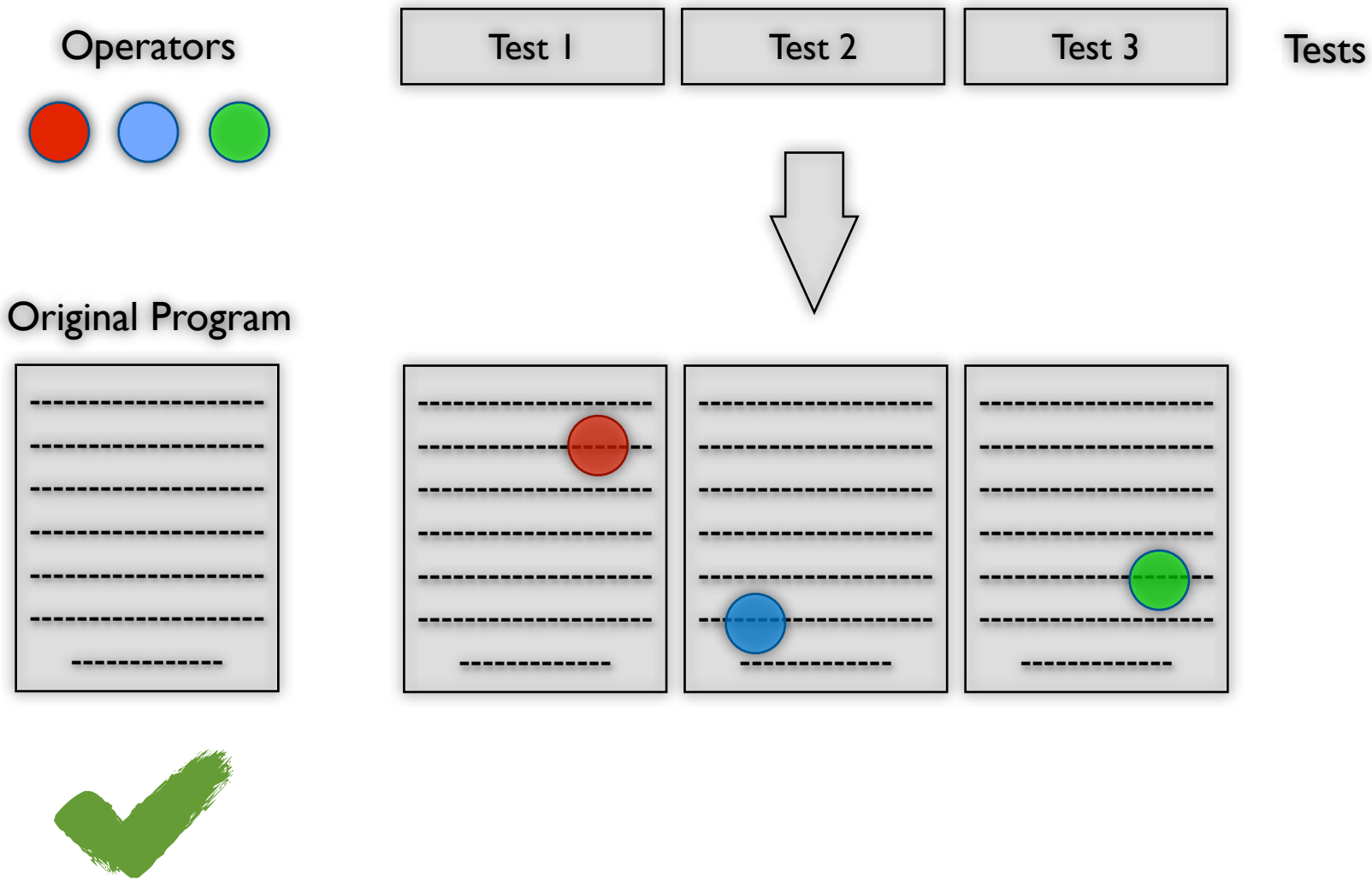


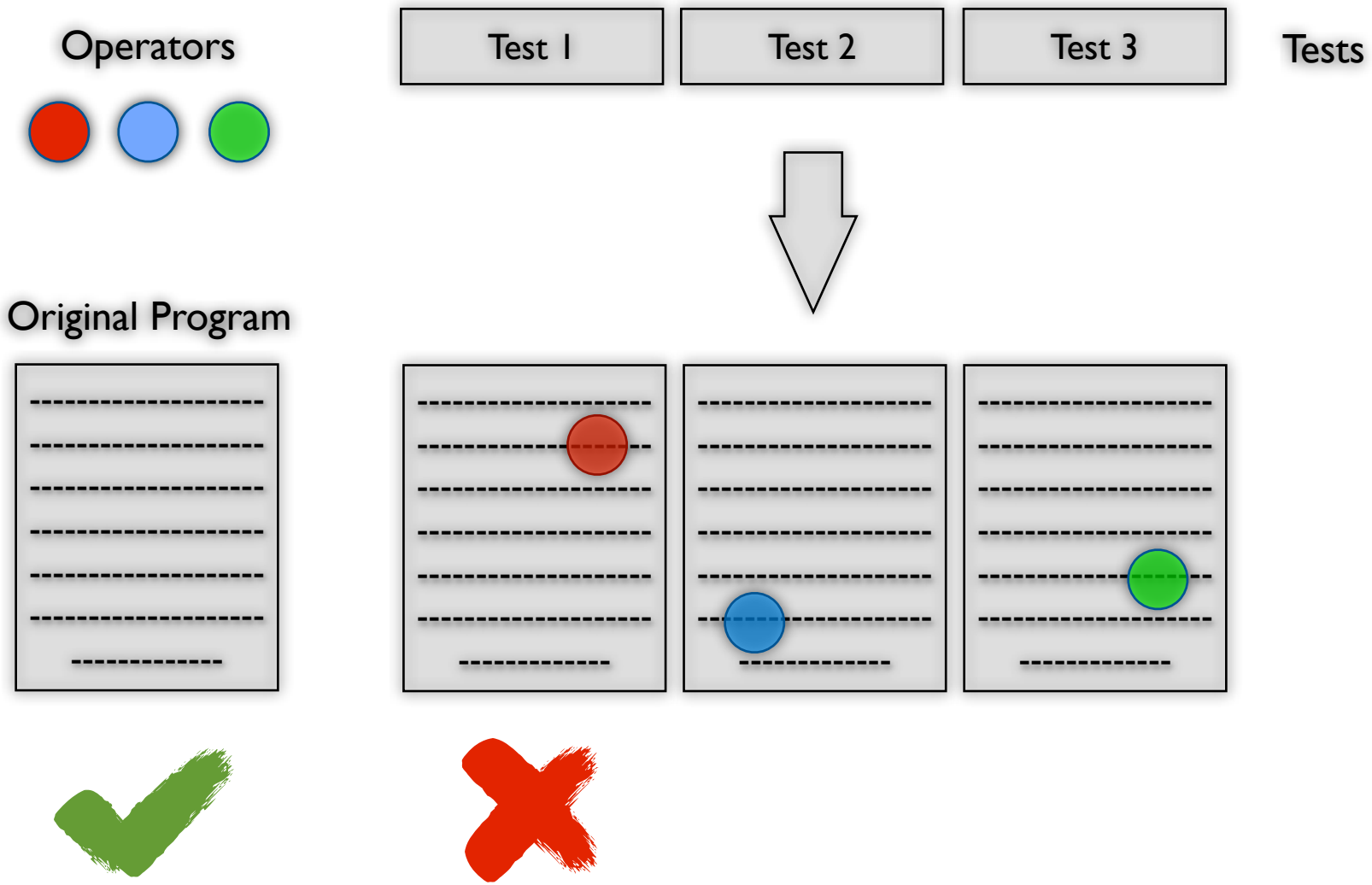
Tests

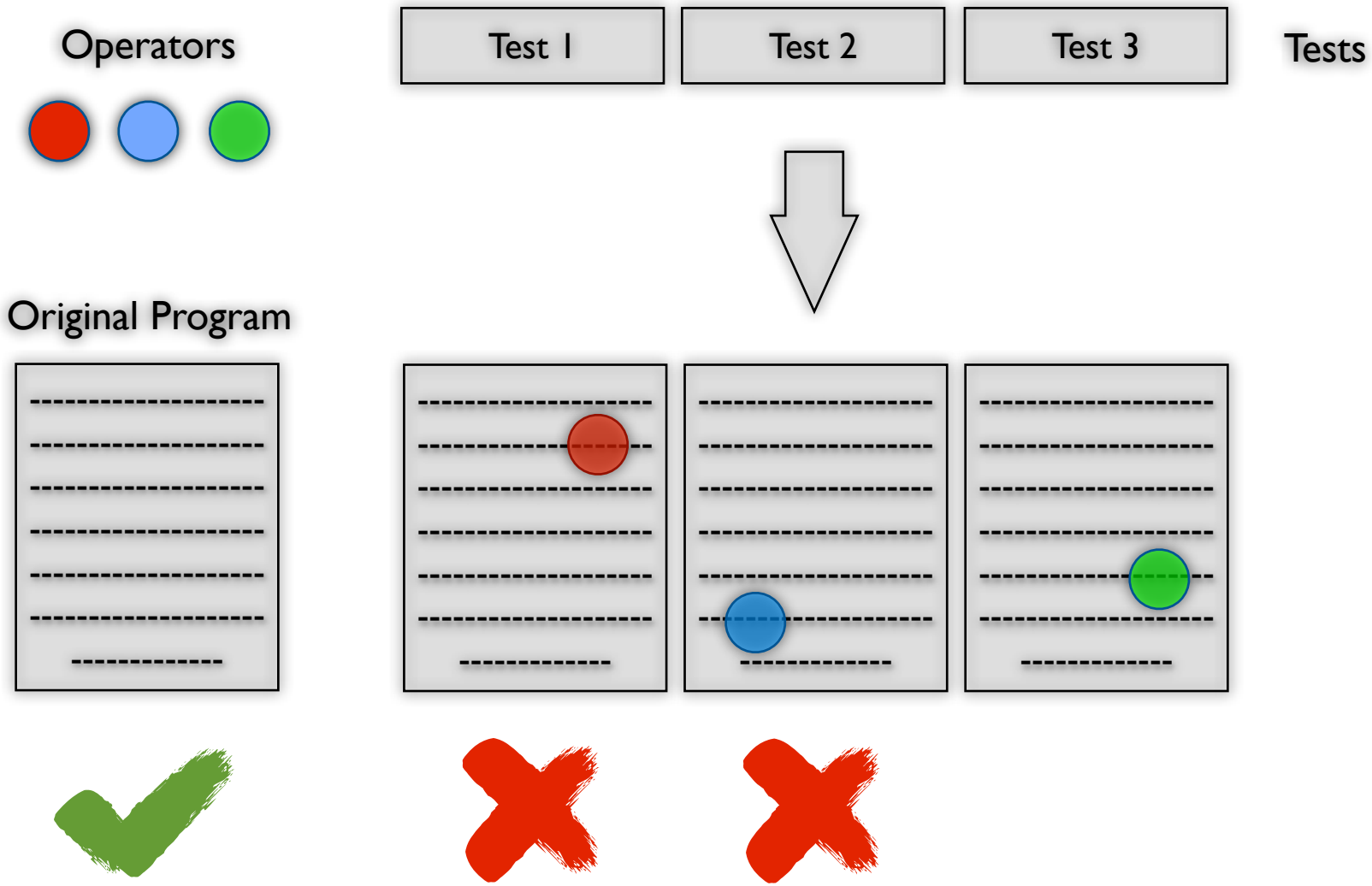
Original Program

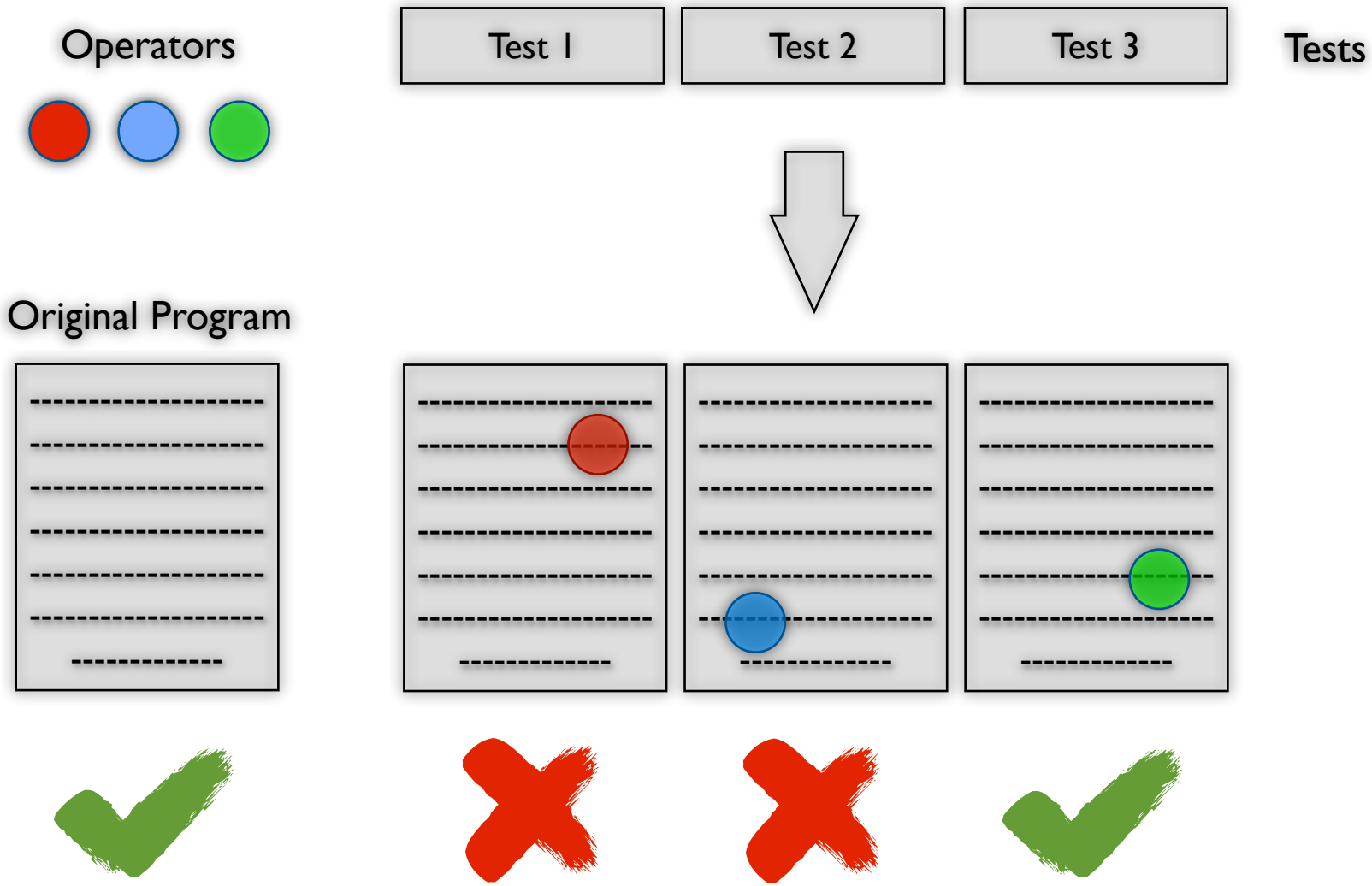


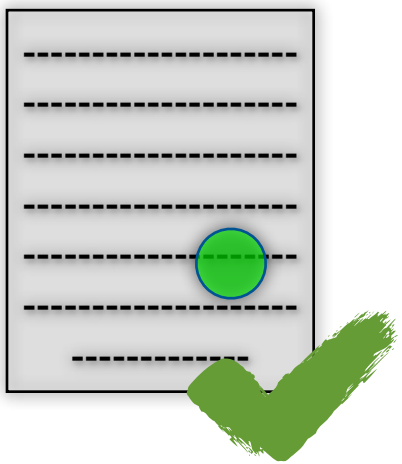

Operators
Test 1
Test 2
Test 3
Tests
Original Program


Operators
Test 1
Test 2
Test 3
Tests
Original Program

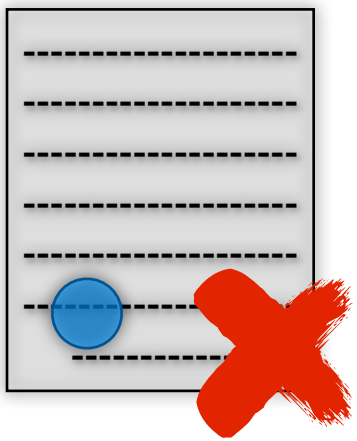

Operators
Test 1
Test 2
Test 3
Tests
Original Program


Operators
Test 1
Test 2
Test 3
Tests
Original Program

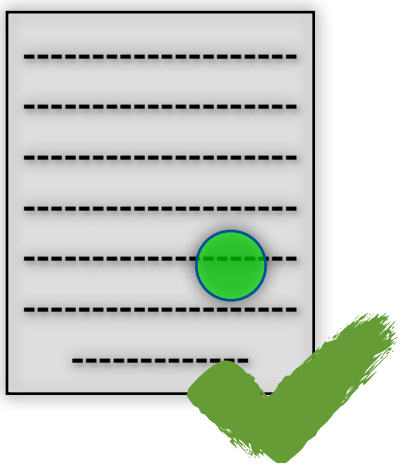

Operators
Test 1
Test 2
Test 3
Tests
Original Program

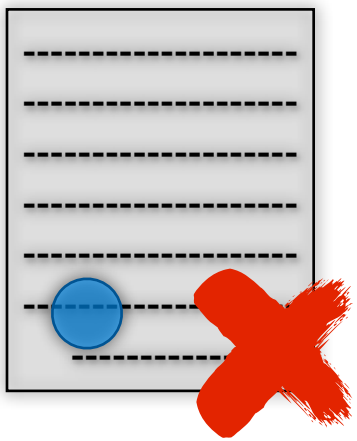


Live mutant - we need more tests



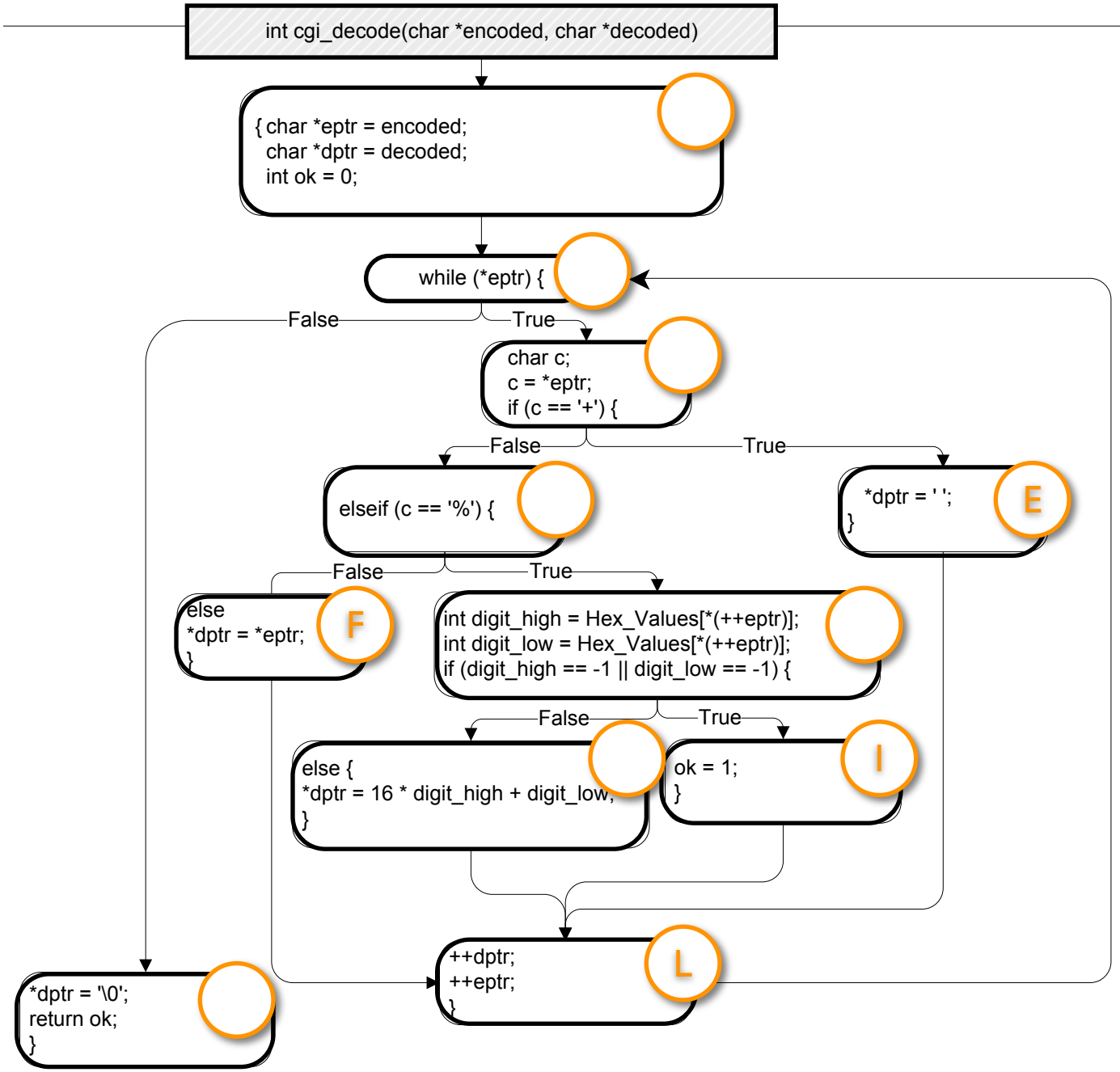
Dead mutant - of no further use

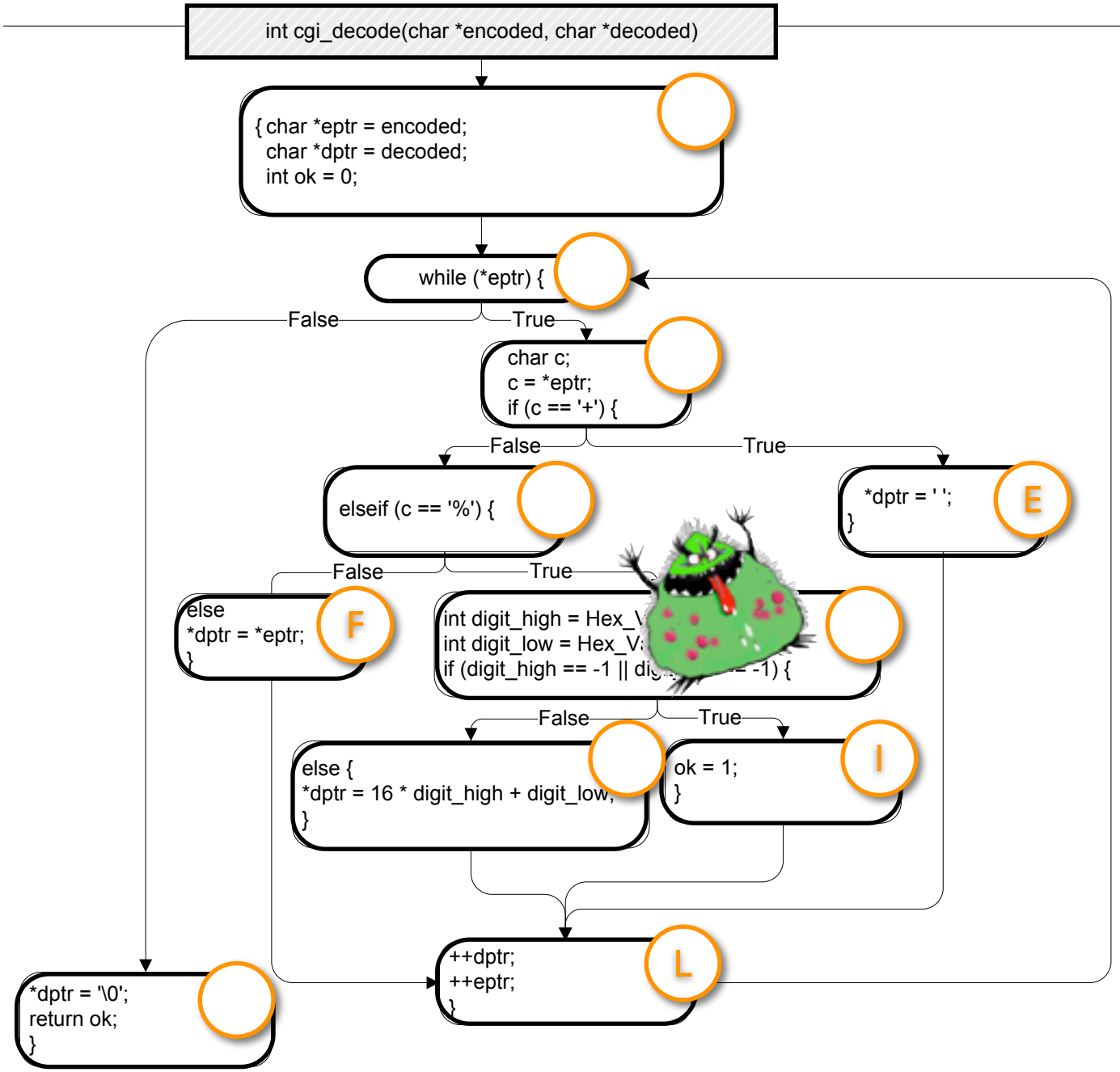


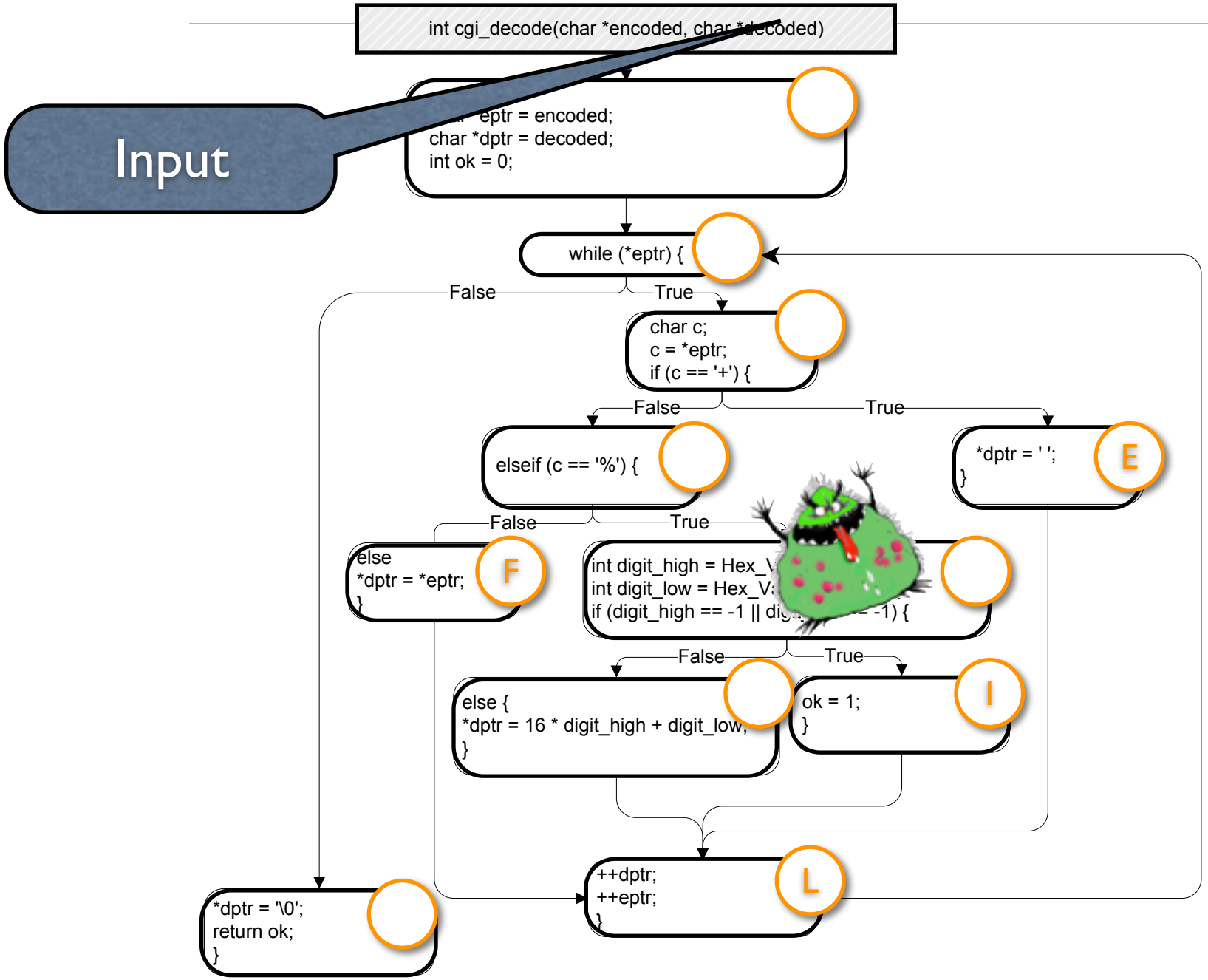


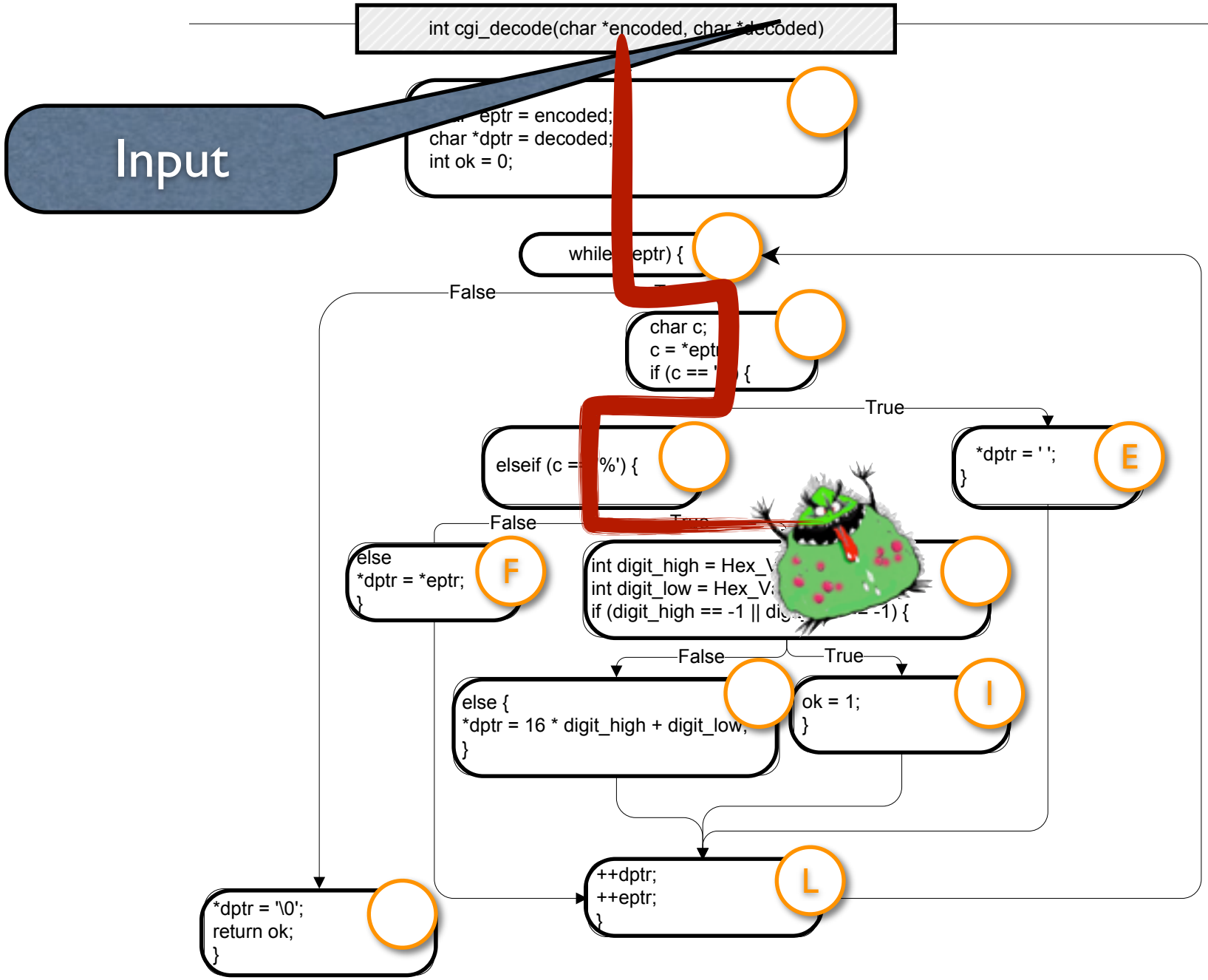
Mutation Score:

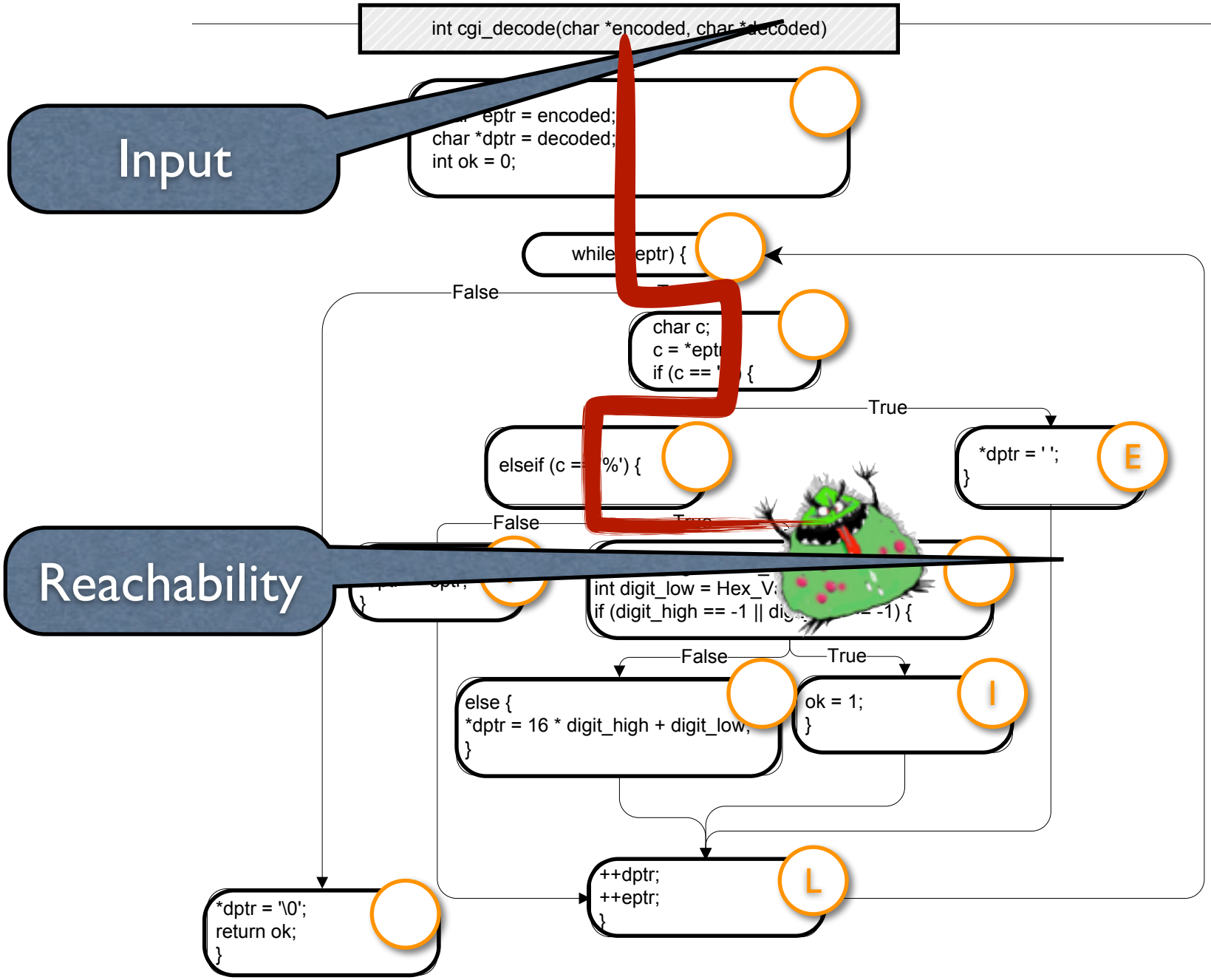
$$\frac{\text{Killed Mutants}}{\text{Total Mutants}}$$

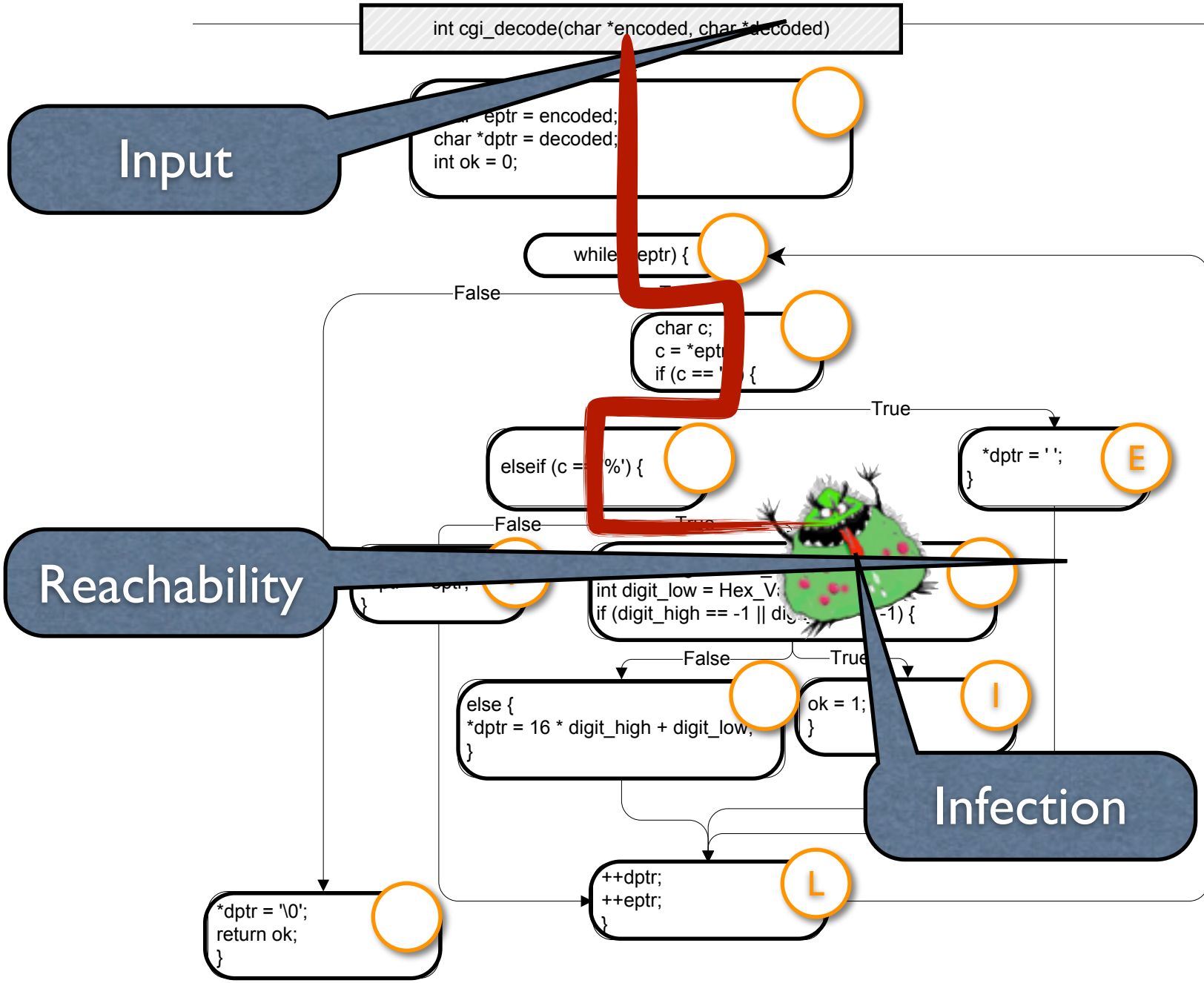

int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
F
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}

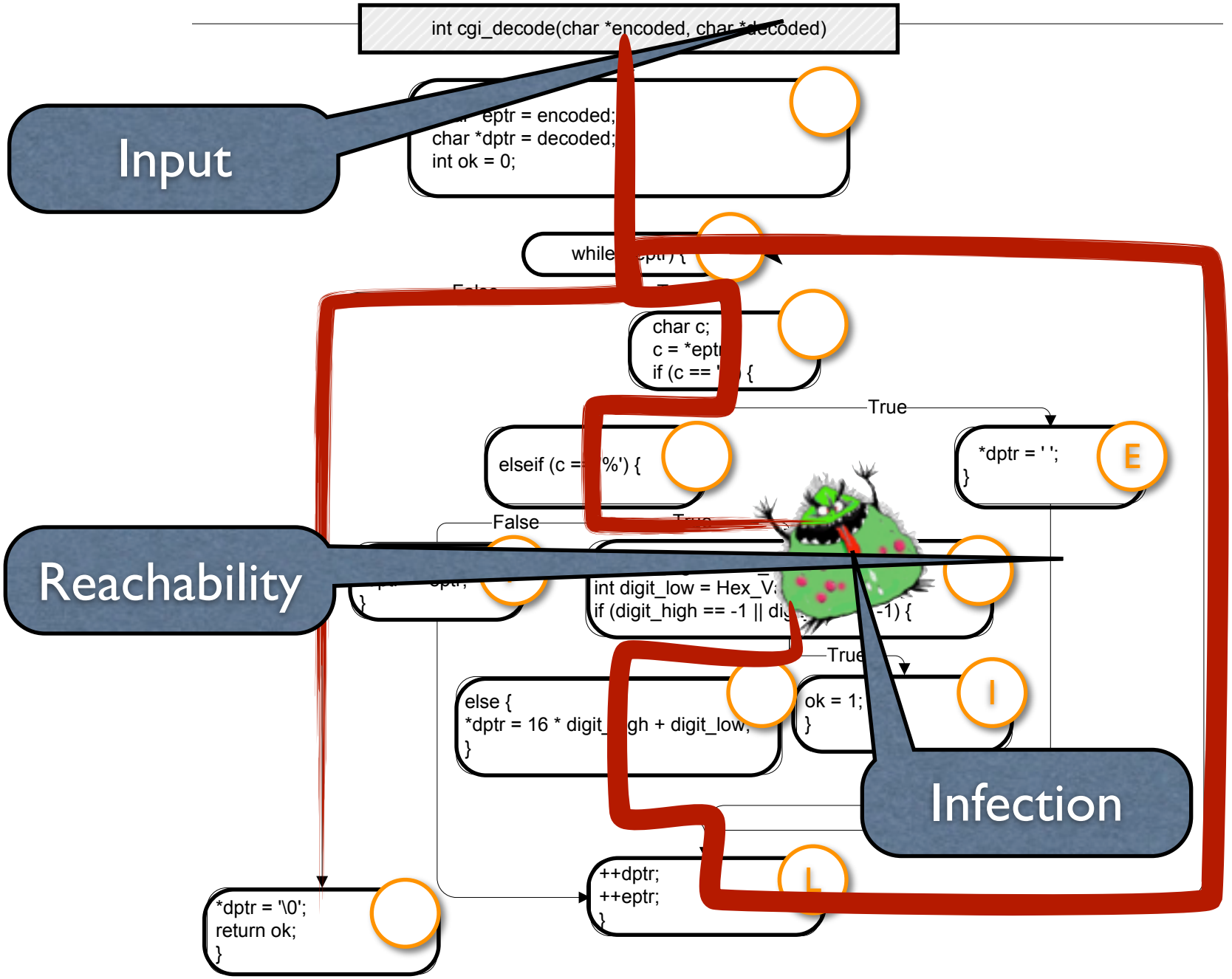

int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
F
int digit_high = Hex_V
int digit_low = Hex_V
if (digit_high == -1 || dig = -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}

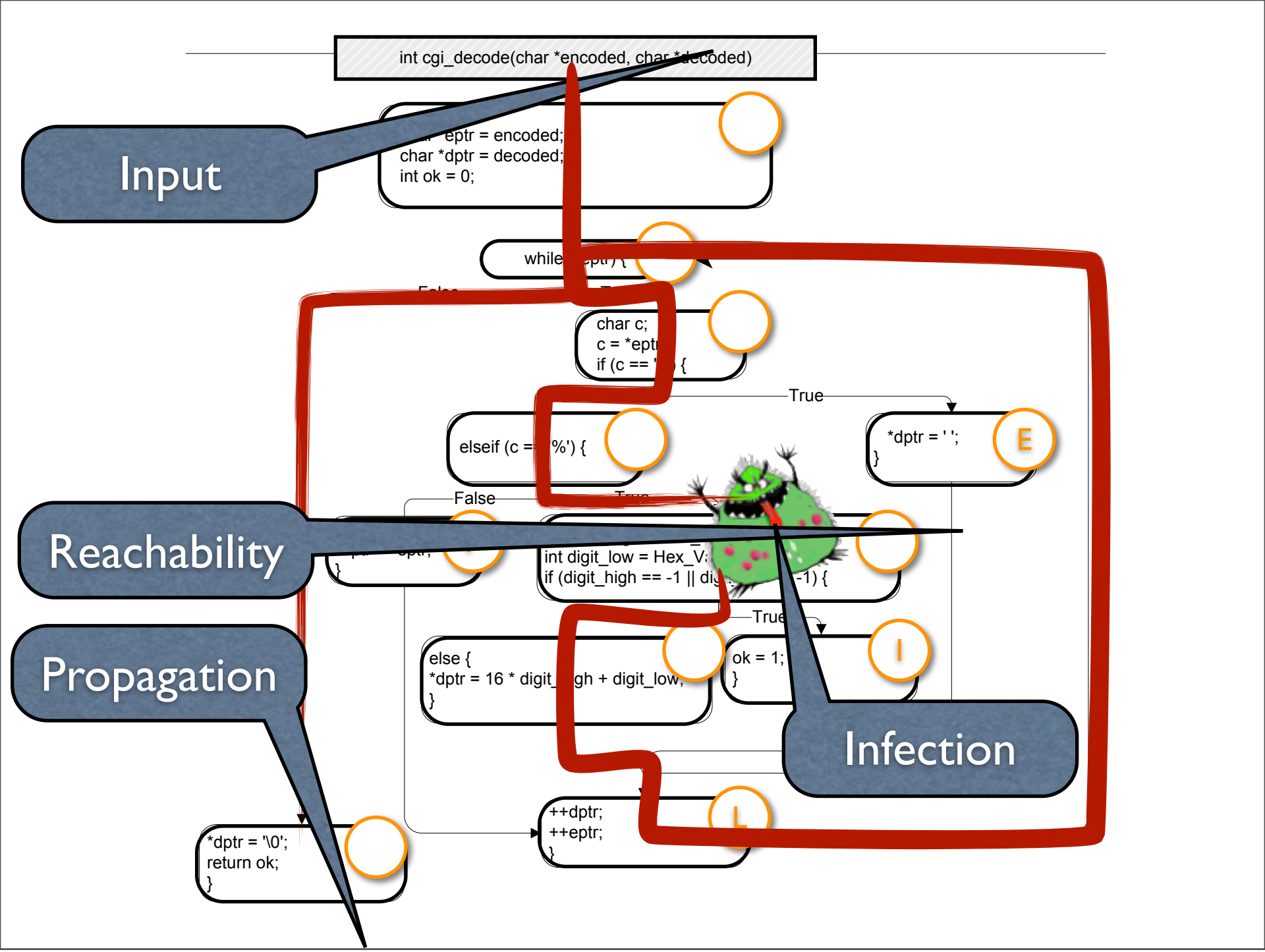

int cgi_decode(char *encoded, char *decoded)
Input
eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
F
int digit_high = Hex_V
int digit_low = Hex_V
if (digit_high == -1 || dig = -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}


int cgi_decode(char *encoded, char *decoded)
Input
eptr = encoded;
char *dptr = decoded;
int ok = 0;
while eptr) {
False
char c;
c = *eptr
if (c == ' ') {
True
elseif (c = %') {
*dptr = ' ';
}
E
False
else
*dptr = *eptr;
}
F
int digit_high = Hex_V
int digit_low = Hex_V
if (digit_high == -1 || dig = -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}


int cgi_decode(char *encoded, char *decoded)
Input
eptr = encoded;
char *dptr = decoded;
int ok = 0;
while eptr) {
False
char c;
c = *eptr
if (c == ' ') {
True
elseif (c = %') {
*dptr = ' ';
}
E
Reachability
False
int digit_low = Hex_V
if (digit_high == -1 || dig = -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}


int cgi_decode(char *encoded, char *decoded)
Input
eptr = encoded;
char *dptr = decoded;
int ok = 0;
while eptr) {
False
char c;
c = *eptr
if (c == ' ') {
True
elseif (c = %') {
*dptr = ' ';
E
False
int digit_low = Hex_V
if (digit_high == -1 || dig -1) {
Reachability
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
I
Infection
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}


int cgi_decode(char *encoded, char *decoded)
eptr = encoded;
char *dptr = decoded;
int ok = 0;
while
char c;
c = *eptr
if (c == '
True
elseif (c =
%') {
*dptr = ' ';
}
E
False
int digit_low = Hex_V
if (digit_high == -1 || dig
-1) {
True
else {
*dptr = 16 * digit_
gh + digit_low;
}
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}
Input
Reachability
Infection


int cgi_decode(char *encoded, char *decoded)
char *dptr = decoded;
int ok = 0;
while
char c;
True
*dptr = ' ';
}
E
elseif (c =
%') {
False
int digit_low = Hex_V
if (digit_high == -1 || dig
-1) {
True
I
else {
*dptr = 16 * digit_
gh + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}
Input
Reachability
Propagation
Infection

Equivalent Mutants

- Mutation = syntactic change
- The change might leave the semantics unchanged
- Equivalent mutants are hard to detect (undecidable problem)
- Might be reached, but no infection
- Might infect, but no propagation

Example 1

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[i] > values[r])
      r = i;
  }

  return values[r];
}
```

Example 1

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 0; i<values.length; i++) {
    if (values[i] > values[r])
      r = i;
  }

  return values[r];
}
```

Example 1

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 0; i<values.length; i++) {
    if (values[i] > values[r])
      r = i;
  }

  return values[r];
}
```

Example 1

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[i] >= values[r])
      r = i;
  }

  return values[r];
}
```

Example 1

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[i] >= values[r])
      r = i;
  }

  return values[r];
}
```

Example 1

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[r] > values[r])
      r = i;
  }

  return values[r];
}
```

Example 1

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[r] > values[r])
      r = i;
  }

  return values[r];
}
```

Example 2

```
if(x > 0) {
  if(y > x) {
    // ...
  }
}
```

Example 2

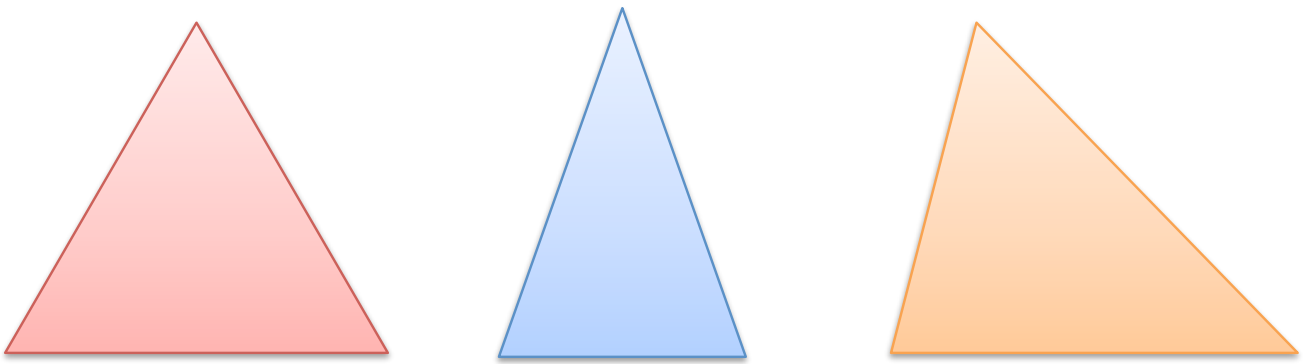
```
if(x > 0) {
  if(y > abs(x)) {
    // ...
  }
}
```

Example 2

```
if(x > 0) {
  if(y > abs(x)) {
    // ...
  }
}
```

Example

Classify triangle by the length of the sides



Equilateral Isosceles Scalene

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a - b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a - b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a - b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a - b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a - b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✗

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a - b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✗

(2, 2, 3) ✗

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a - b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✗

(2, 2, 3) ✗

(2, 3, 4) ✗

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 1, 1)

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 1, 1)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 1, 1)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(4, 3, 2)

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 1, 1)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(4, 3, 2)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 1, 1)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(4, 3, 2)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(1, 1, 1)

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 1, 1)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(4, 3, 2)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(1, 1, 1)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 1, 1)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(4, 3, 2)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(1, 1, 1)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(2, 3, 2)

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 1, 1)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(4, 3, 2)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(1, 1, 1)

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(2, 3, 2)

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b ==
        return 2; // is
    }

    return 3;  // scale
}
```

(1, 1, 1)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        celes
```

(4, 3, 2)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

```
int triangle(int a, int
    if (a <= 0 || b <=
        return 4; // in
    }
    if (! (a * b > c &&
        return 4; // in
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(2, 3, 2)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b ==
        return 2; // is
    }

    return 3;  // scale
}
```

(1, 1, 1)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        celes
```

(4, 3, 2)

```
int triangle(int a, int
    if (a <= 0 || b <=
        return 4; // in
    }
    if (! (a * b > c &&
        return 4; // in
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(2, 3, 2)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

equivalent!

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

Performance

```
int triangle(int a, int b, int c) {
  if (a <= 0 || b <= 0 || c <= 0) {
      return 4; // invalid
  }
  if (! ( a + b > c && a + c > b && b + c > a)) {
      return 4; // invalid
  }
  if (a == b && b == c) {
      return 1; // equilateral
  }
  if (a == b || b == c || a == c) {
      return 2; // isosceles
  }
  return 3;  // scalene
}
```

$$a + b > c$$

a + b > c

```
a - b > c
a * b > c
a / b > c
a % b > c
a > c
b > c
abs(a) - b > c
a - abs(b) > c
a - b > abs(c)
abs(a - b) > c
-abs(a) - b > c
a - -abs(b) > c
a - b > -abs(c)
-abs(a - b) > c
```

```
a - b >= c
a - b < c
a - b <= c
a - b = c
a - b != c
b - b > c
a - a > c
c - b > c
a - c > c
a - b > a
a - b > b
a - b > c
0 - b > c
a - 0 > c
```

```
a - b > 0
++a - b > c
a - ++b > c
a - b > ++c
--a - b > c
a - --b > c
a - b > --c
++(a - b) > c
--(a - b) > c
-a - b > c
a - -b > c
a - b > -c
-(a - b) > c
0 > c
```

Performance Problems

- Many mutation operators possible
  Proteum - 103 Mutation Operators for C
  MuJava - Adds 24 Class level Mutation Operators
- Each mutation operator results in many mutants
  Depending on program under test
- Each mutant needs to be compiled
- Each test case needs to be executed against every mutant

Improvements

Do fewer

- Mutant sampling
- Selective mutation

Do smarter

- Parallelize
- Weak mutation
- Use coverage
- Impact

Do faster

- Mutate bytecode
- Mutant schemata

Using Coverage

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

(0, 1, 1)

(4, 3, 2)

(1, 1, 1)

(2, 3, 2)

Using Coverage

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

(0, 1, 1)

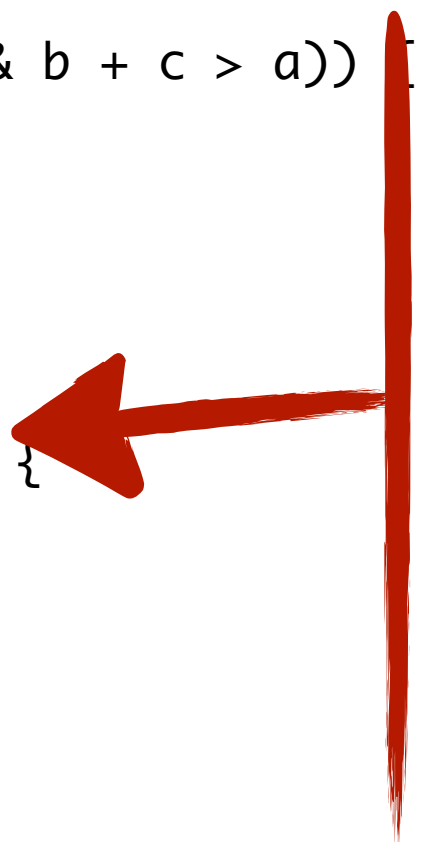
(4, 3, 2)

(1, 1, 1)

(2, 3, 2)

Using Coverage

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```



(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

(0, 1, 1)

(4, 3, 2)

(1, 1, 1)

(2, 3, 2)

Using Coverage

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // sc
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

(0, 1, 1)

(4, 3, 2)

(1, 1, 1)

(2, 3, 2)

Only these tests execute mutants in this line!

Strong vs. Weak Mutation

- Strong mutation
  Mutation has propagated to some observable behavior
- Weak mutation
  Mutation has affected state (infection)
- Compare internal state after mutation
- Does not guarantee propagation
- Reported to save 50% execution time

Strong vs. Weak Mutation

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```

Strong vs. Weak Mutation

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```


```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```

Strong vs. Weak Mutation

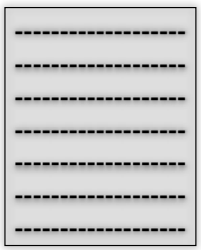
```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

Weak mutation

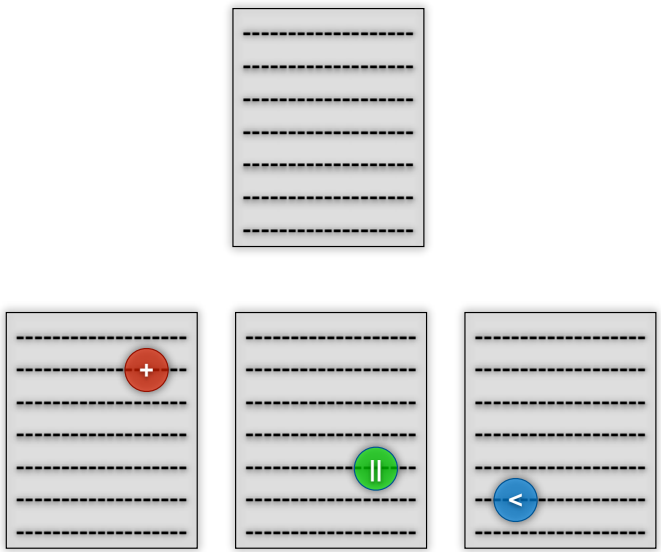
```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```

Strong mutation

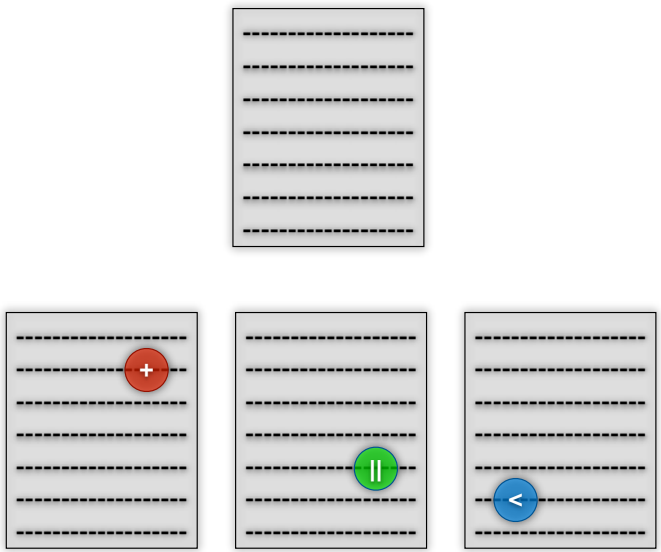
Mutant Schemata



Mutant Schemata

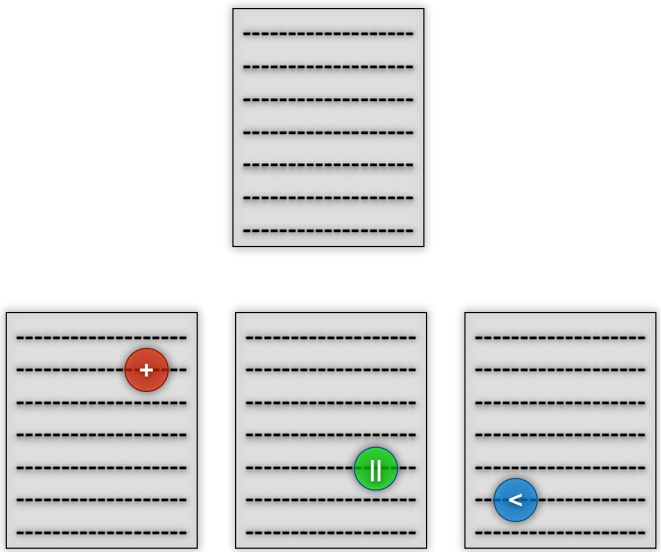


Mutant Schemata



Mutant 1:
Compile
Execute

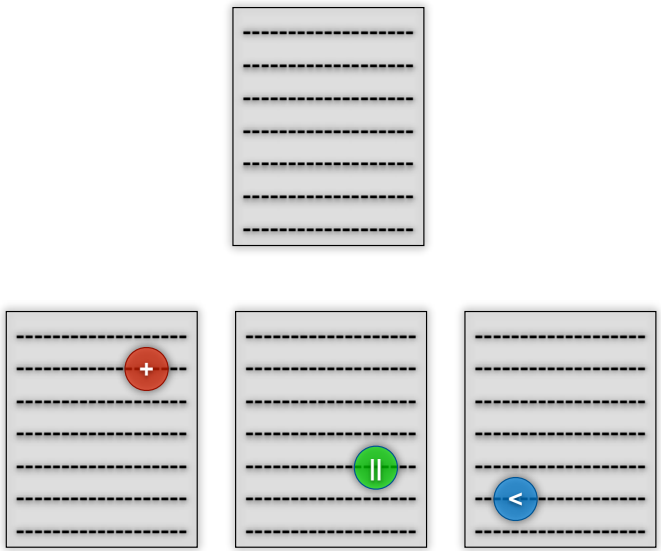
Mutant Schemata



Mutant 1:
Compile
Execute

Mutant 2:
Compile
Execute

Mutant Schemata

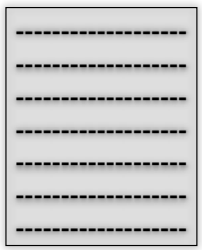


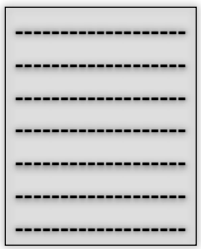
Mutant 1:
Compile
Execute

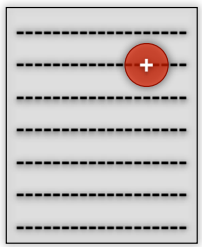
Mutant 2:
Compile
Execute

Mutant 3:
Compile
Execute

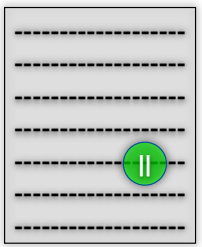
Mutant Schemata



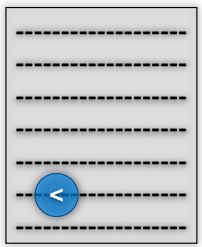




Mutant 1:
Compile
Execute

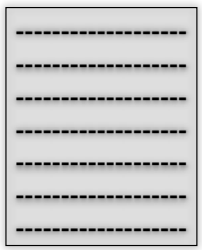


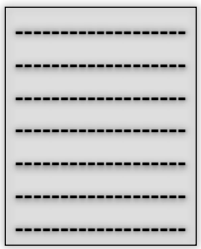
Mutant 2:
Compile
Execute

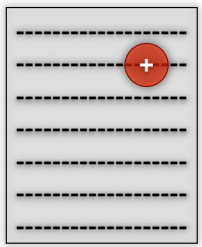


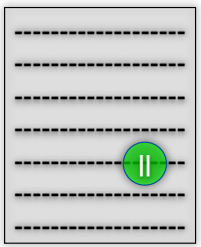
Mutant 3:
Compile
Execute

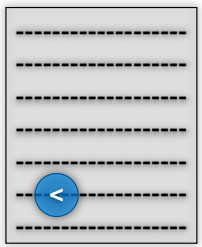
Mutant Schemata

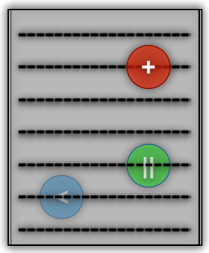










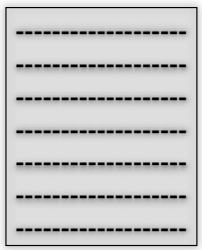


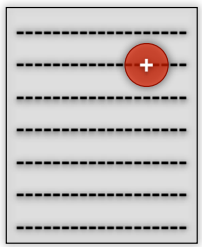
Mutant 1:
Compile
Execute

Mutant 2:
Compile
Execute

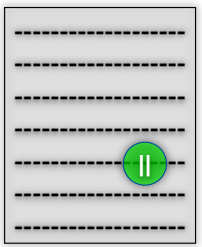
Mutant 3:
Compile
Execute

Mutant Schemata

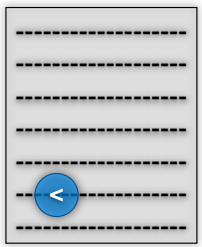




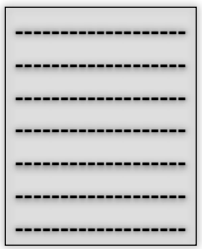
Mutant 1:
Compile
Execute

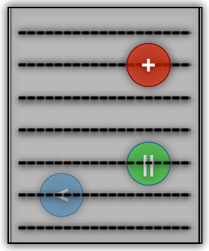


Mutant 2:
Compile
Execute



Mutant 3:
Compile
Execute





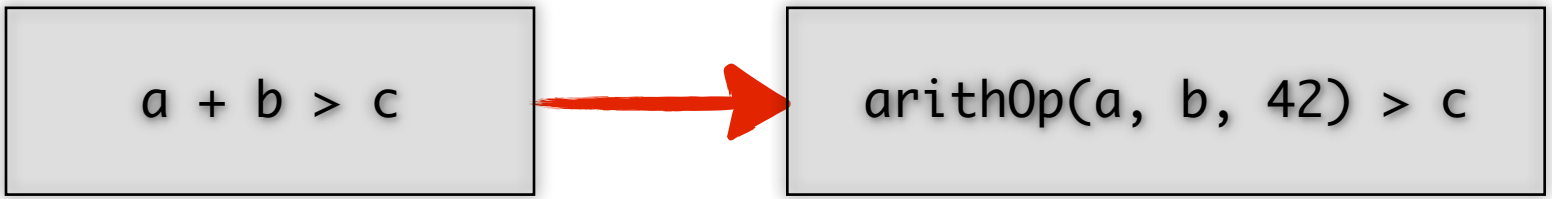
Meta-Mutant:
Compile
Switch on Mutant 1
Execute
Switch on Mutant 2
Execute
Switch on Mutant 3
Execute

Mutant Schemata

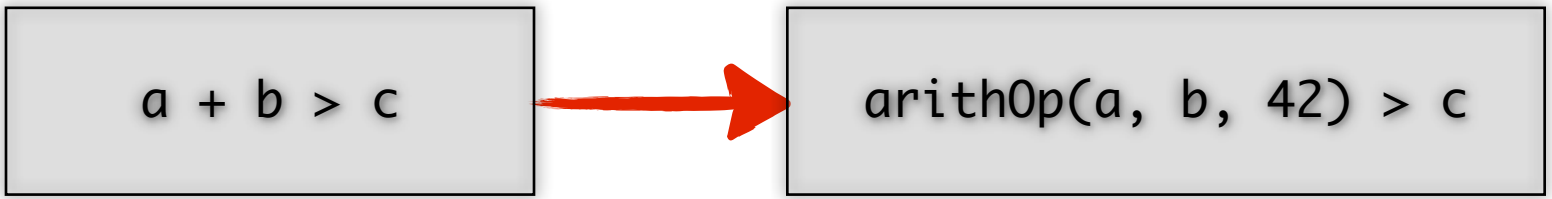
```
a + b > c
```

```
arithOp(a, b, 42) > c
```

Mutant Schemata



Mutant Schemata




```
int arithOp(int op1, int op2, int location) {
  switch(variant(location)) {
    case aoADD:   return op1 + op2;
    case aoSUB:   return op1 - op2;
    case aoMULT:  return op1 * op2;
    case aoDIV:   return op1 / op2;
    case aoMOD:   return op1 % op2;
    case aoLEFT:  return op1;
    case aoRIGHT: return op2;
  }
}
```

Mutant Schemata

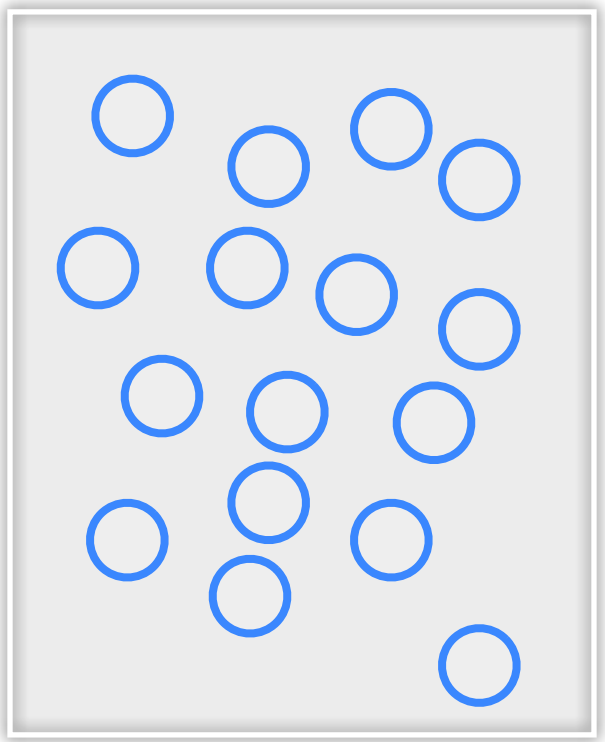
```
a + b > c
```

```
arithOp(a, b, 42) > c
```

```
int arithOp(int op1, int op2, int location) {
  switch(variant(location)) {
    case aoADD:   return op1 + op2;
    case aoSUB:   return op1 - op2;
    case aoMULT:  return op1 * op2;
    case aoDIV:   return op1 / op2;
    case aoMOD:   return op1 % op2;
    case aoLEFT:  return op1;
    case aoRIGHT: return op2;
  }
}
```

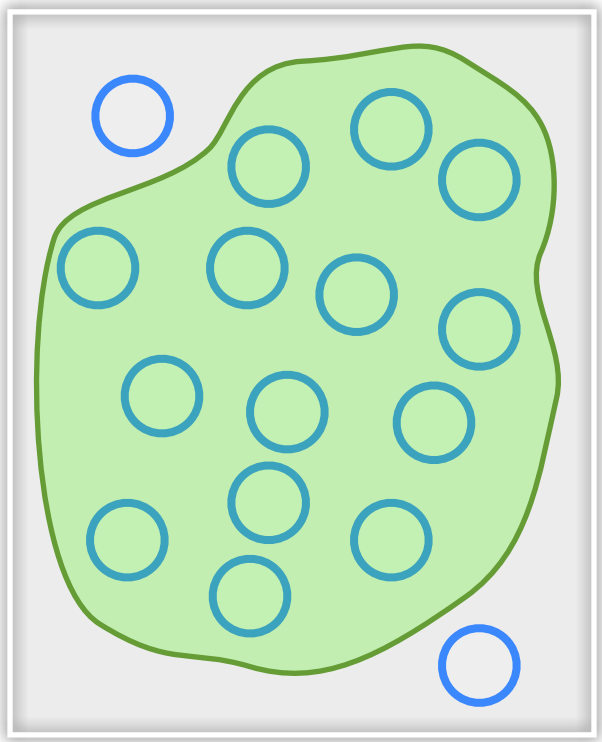
Select active

Selective Mutation



Selective Mutation

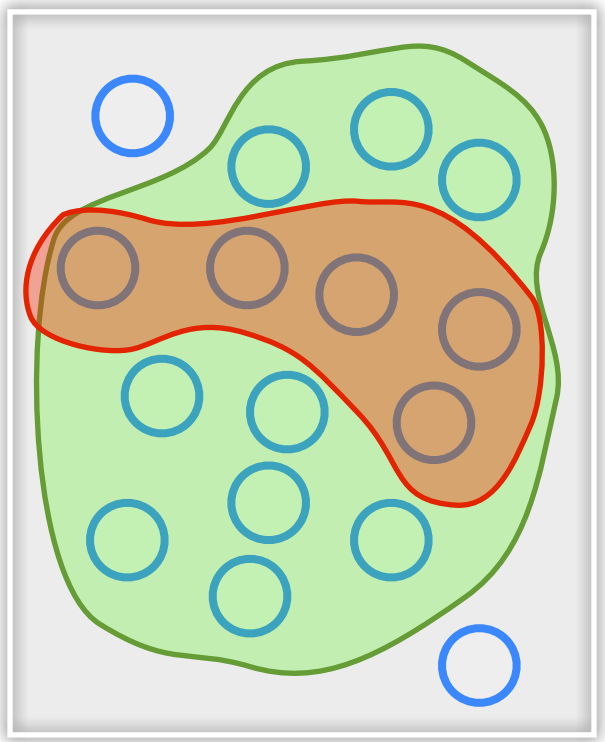
Full Set:
Test cases that
kill all mutants



Selective Mutation

Full Set:
Test cases that
kill all mutants

Sufficient Subset:
Test cases that kill
these mutants will
kill all mutants



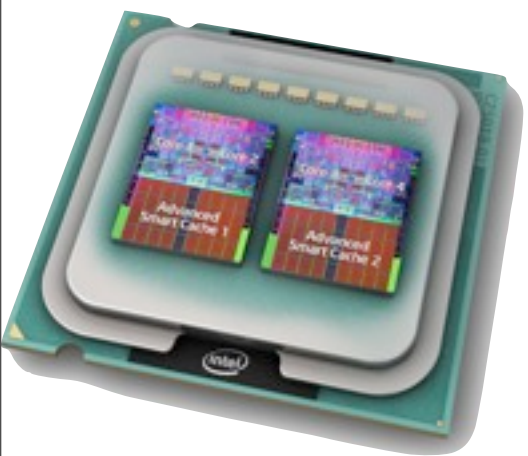
Selective Mutation

- Use only a subset of mutation operators instead of all operators
- Subset is sufficient
- Detecting mutants of sufficient subset will detect >99% of all mutants
- ABS, AOR, COR, ROR, UOI

Do Smarter/Faster

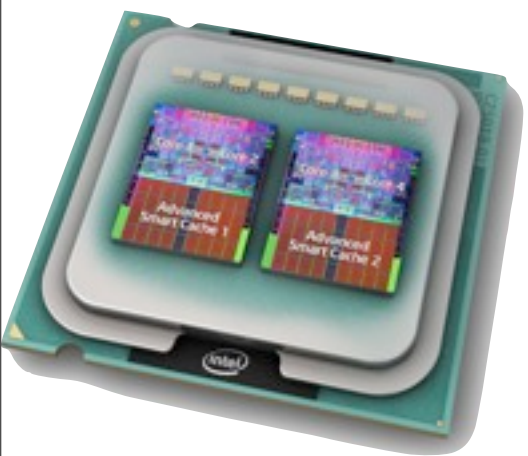
Do Smarter/Faster

Mutation testing is
inherently parallelizable



Do Smarter/Faster

Mutation testing is inherently parallelizable



```
6:  iload_1 // a
7:  ifeq 14
10: iload_2 // b
11: ifne 23
14: iload_3 // c
15: ifeq 34
// ...
23: invokevirtual  #4;
// ...
34: invokevirtual  #5;
```

Mutating bytecode avoids recompilation

Do Smarter/Faster

Mutation testing is inherently parallelizable

```
6:  iload_1 // a
7:  ifeq 14
10: iload_2 // b
11: ifne 23
14: iload_3 // c
15: ifeq 34
// ...
23: invokevirtual  #4;
// ...
34: invokevirtual  #5;
```

Mutating bytecode avoids recompilation

Sample subset of mutants