# Coverage-Guided Fuzzing

Security Testing

Andreas Zeller, Saarland University

Dynamic Coverage

Static Structure

Smart Algorithms

# Our Goal

- We want to *cause the program to fail*
- We have seen
  - *random* (unstructured) input
  - *structured* (grammar-based) input
  - generation based on *grammar coverage*

# A Challenge

```
class Roots {
    // Solve ax² + bx + c = 0
    public roots(double a, double b, double c)
    { … }

    // Result: values for x
    double root_one, root_two;
}
```

- Which values for *a, b, c* should we test?

  assuming a, b, c, were 32-bit integers, we'd have $(2^{32})^3 \approx 10^{28}$ legal inputs
  with 1.000.000.000.000 tests/s, we would still require 2.5 billion years

# The Code

```
// Solve ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // code for handling two roots
    }

    else if (q == 0) {
        // code for handling one root
    }

    else {
        // code for handling no roots
    }
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Test this case

and this

and this!

# The Test Cases

```
// Solve ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // code for handling two roots
    }

    else if (q == 0) {
        // code for handling one root
    }

    else {
        // code for handling no roots
    }
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

(a, b, c) = (3, 4, 1)

(a, b, c) = (0, 0, 1)

(a, b, c) = (3, 2, 1)

# A Defect

```
// Solve ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // code for handling two roots
    }

    else if (q == 0) {
        x = (-b) / (2 * a);
    }

    else {
        // code for handling no roots
    }
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

code must handle a = 0

(a, b, c) = (0, 0, 1)

# The Idea

**Use the *program***
**to guide test generation**

# The Ingredients

Dynamic Coverage

Static Structure

Smart Algorithms

# The Ingredients

Dynamic Coverage

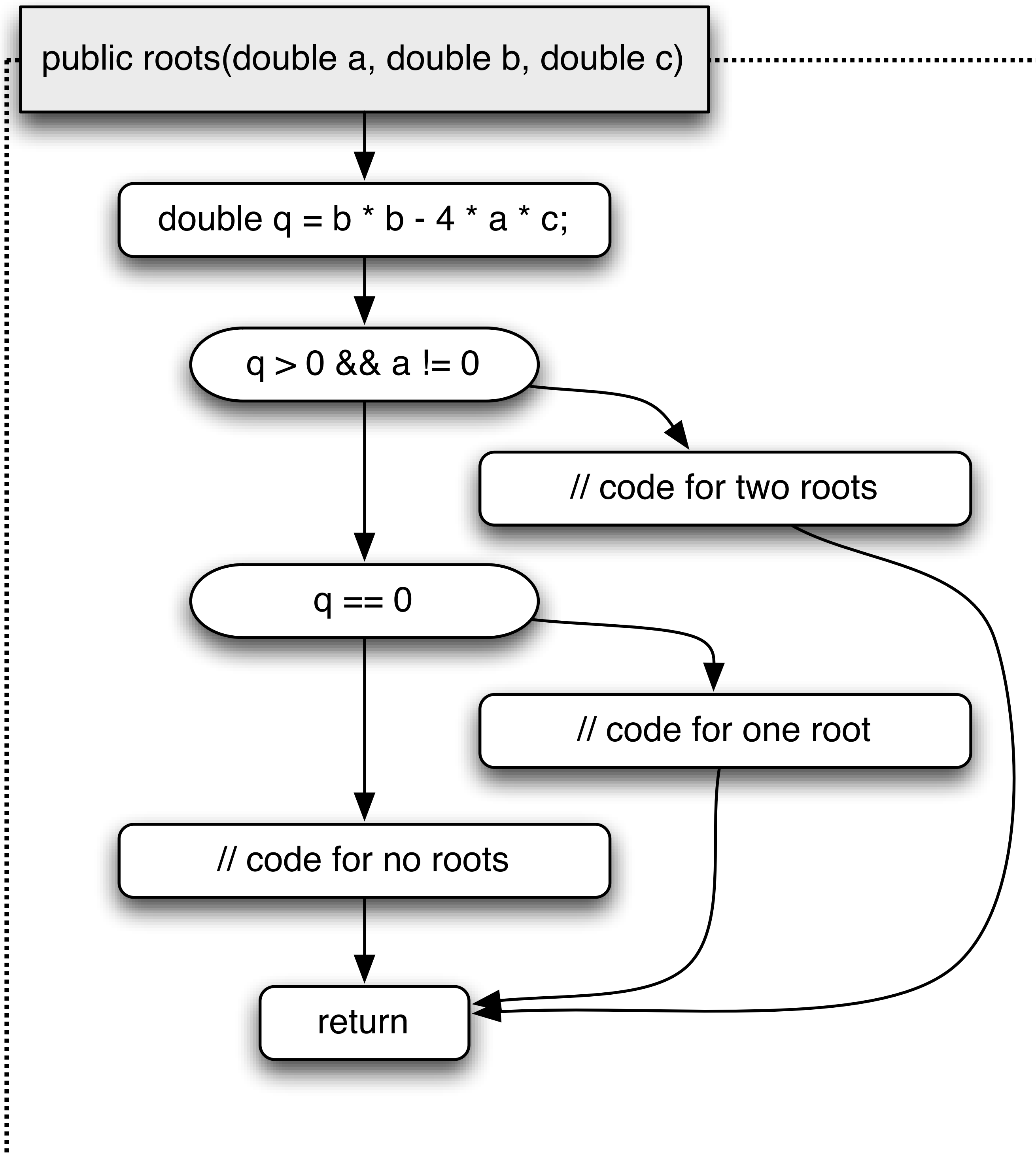
Static Structure

Smart Algorithms

# Expressing Structure

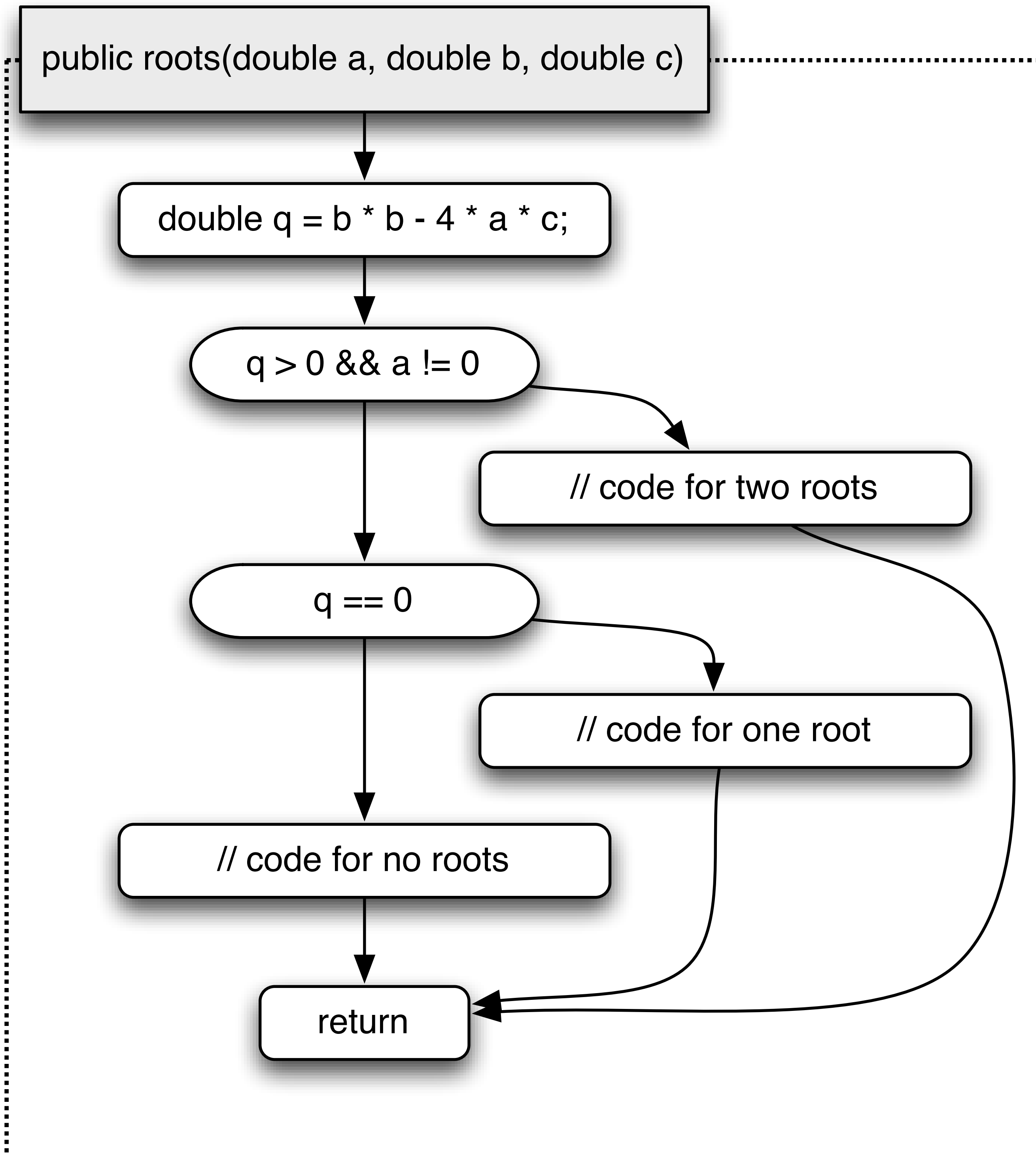
```
// Solve ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // code for handling two roots
    }

    else if (q == 0) {
        x = (-b) / (2 * a);
    }

    else {
        // code for handling no roots
    }
}
```

# Control Flow Graph



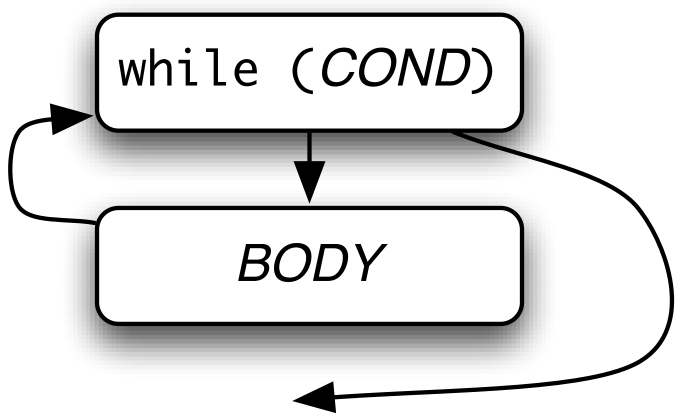

- A *control flow graph* expresses paths of program execution
- *Nodes* are *basic blocks* – sequences of statements with one entry and one exit point
- *Edges* represent *control flow* – the possibility that the program execution proceeds from the end of one basic block to the beginning of another

# Structural Testing

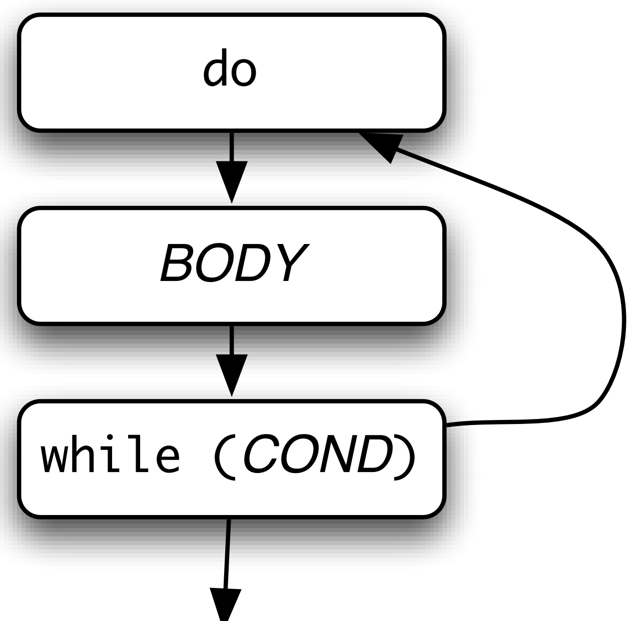



- The CFG can serve as an *adequacy criterion* for test cases
- The more parts are covered (executed), the higher the chance of a test to uncover a defect
- “parts” can be: nodes, edges, paths, conditions…

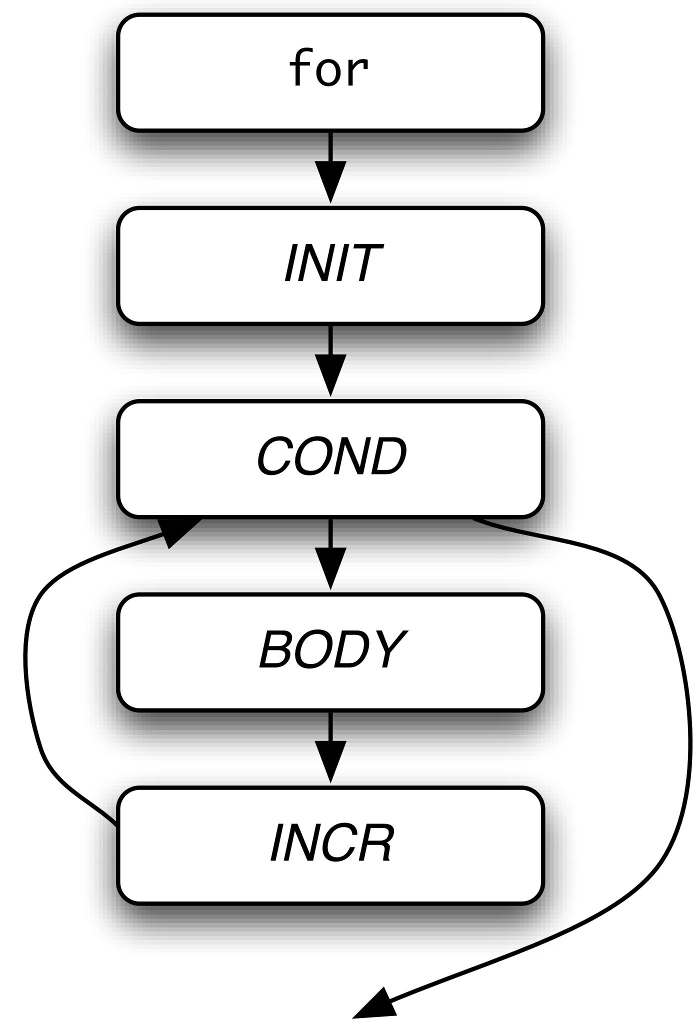
# Control Flow Patterns



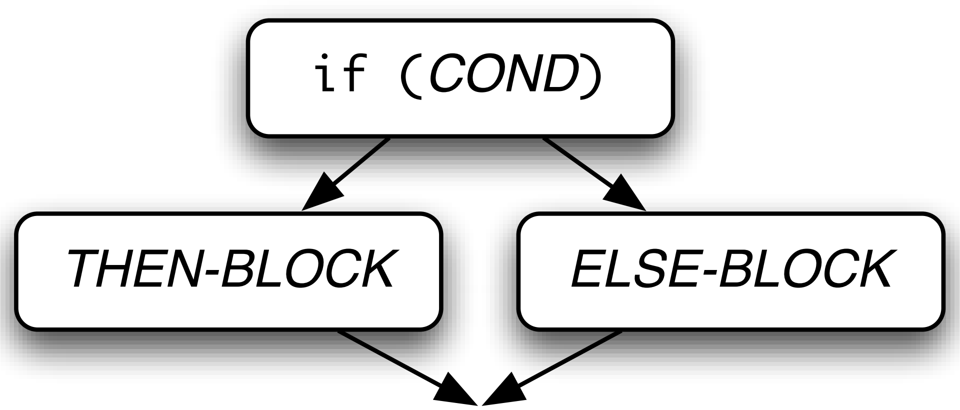

```
while (COND)
   BODY;
```




```
do {
      BODY
}while (COND);
```




```
for (INIT; COND; INCR)
   BODY;
```

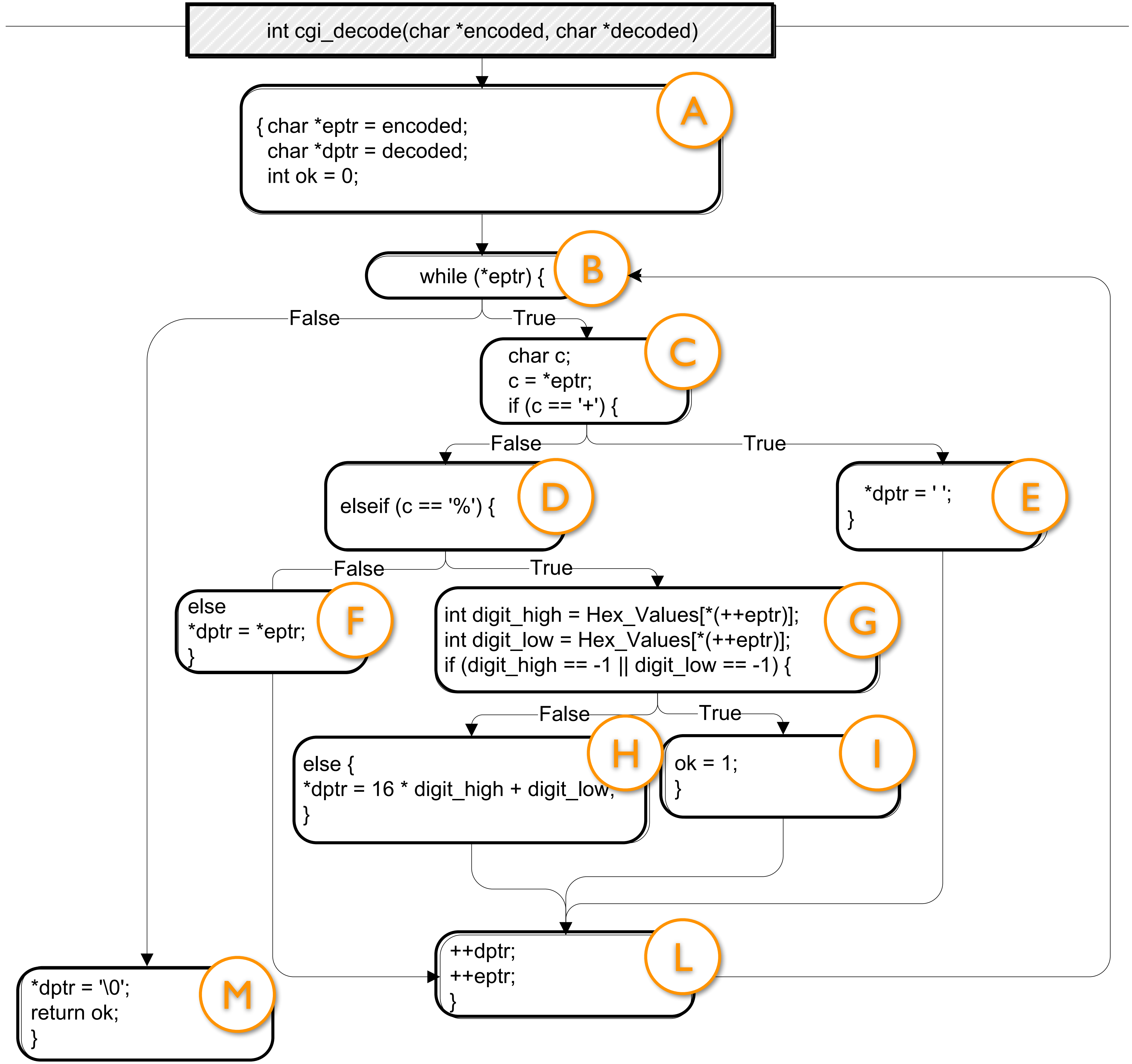



```
if (COND)
   THEN-BLOCK;
else
   ELSE-BLOCK;
```

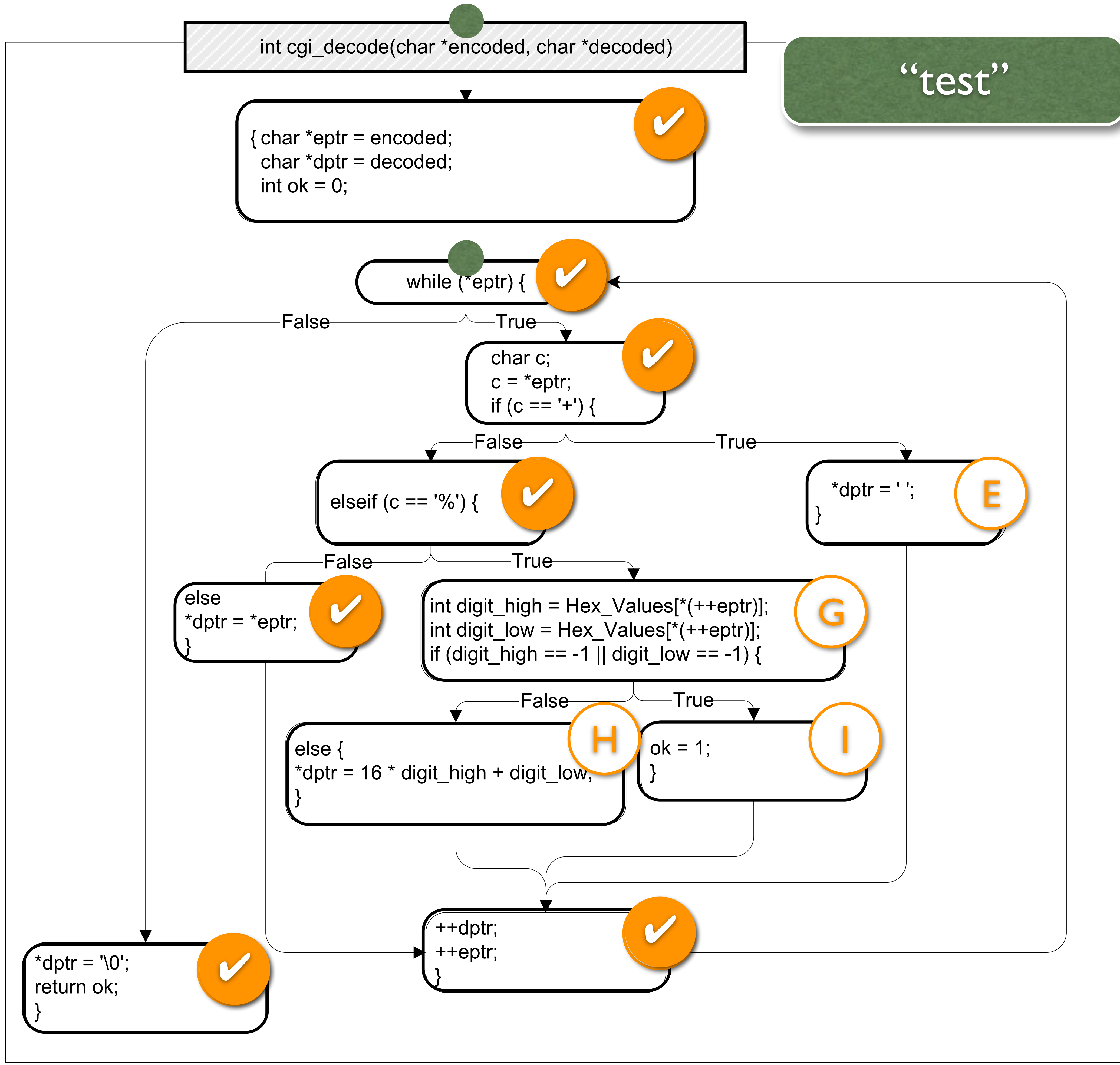
# cgi_decode

```
/**
  * @title cgi_decode
  * @desc
  * Translate a string from the CGI encoding to plain ascii text
  * '+' becomes space, %xx becomes byte with hex value xx,
  * other alphanumeric characters map to themselves
  *
  * returns 0 for success, positive for erroneous input
  * 1 = bad hexadecimal digit
  */

int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;  A
    int ok = 0;
```

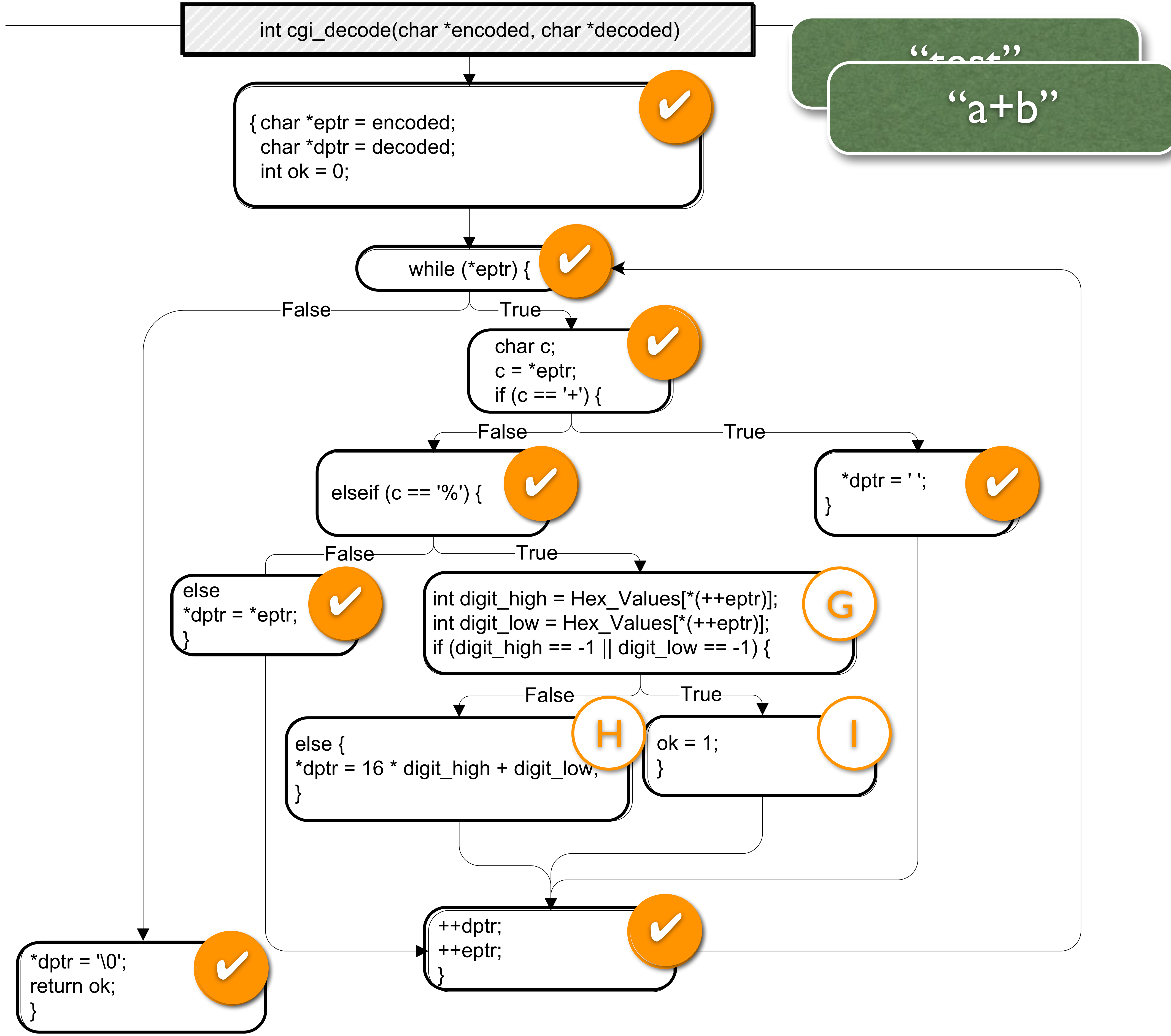
```
    while (*eptr)  /* loop to end of string (‘\0’ character) */ B
    {
        char c;
        c = *eptr; C
        if (c == ’+’) {  /* ‘+’ maps to blank */
            *dptr = ’ ’; E
        } else if (c == ’%’) { /* ’%xx’ is hex for char xx */ D
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low  = Hex_Values[*(++eptr)]; G
            if (digit_high == -1 || digit_low == -1)
                ok = 1; /* Bad return code */ I
            else
                *dptr = 16 * digit_high + digit_low; H
        } else { /* All other characters map to themselves */
           *dptr = *eptr; F
        }
        ++dptr; ++eptr; L
    }

    *dptr = ‘\0’;   /* Null terminator for string */ M
    return ok;
}
```

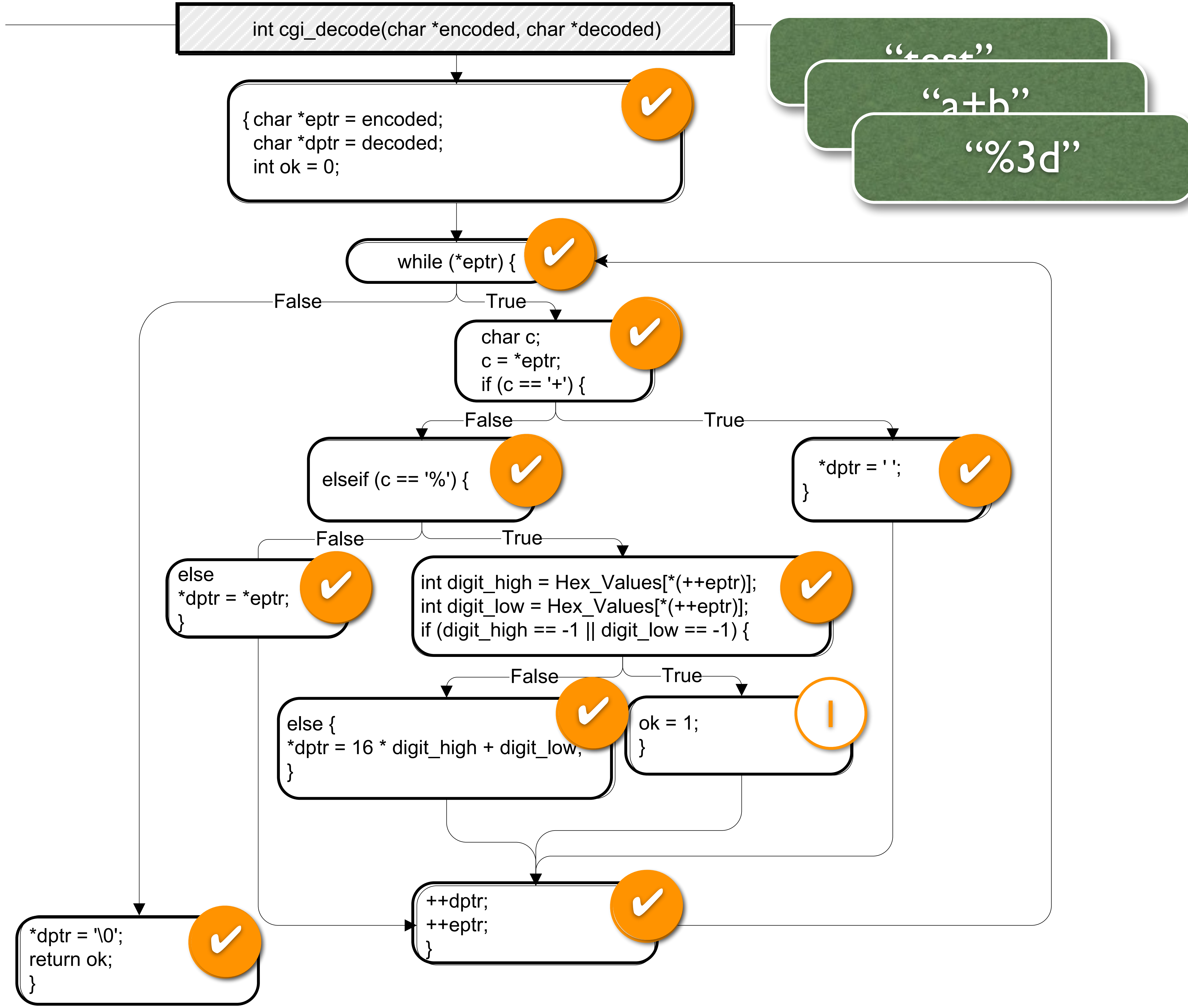


int cgi_decode(char *encoded, char *decoded)
A
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
B
while (*eptr) {
False
True
C
char c;
c = *eptr;
if (c == '+') {
False
True
D
elseif (c == '%') {
E
*dptr = ' ';
}
False
True
F
else
*dptr = *eptr;
}
G
int digit_high = Hex_Values[*(++eptr)];
int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

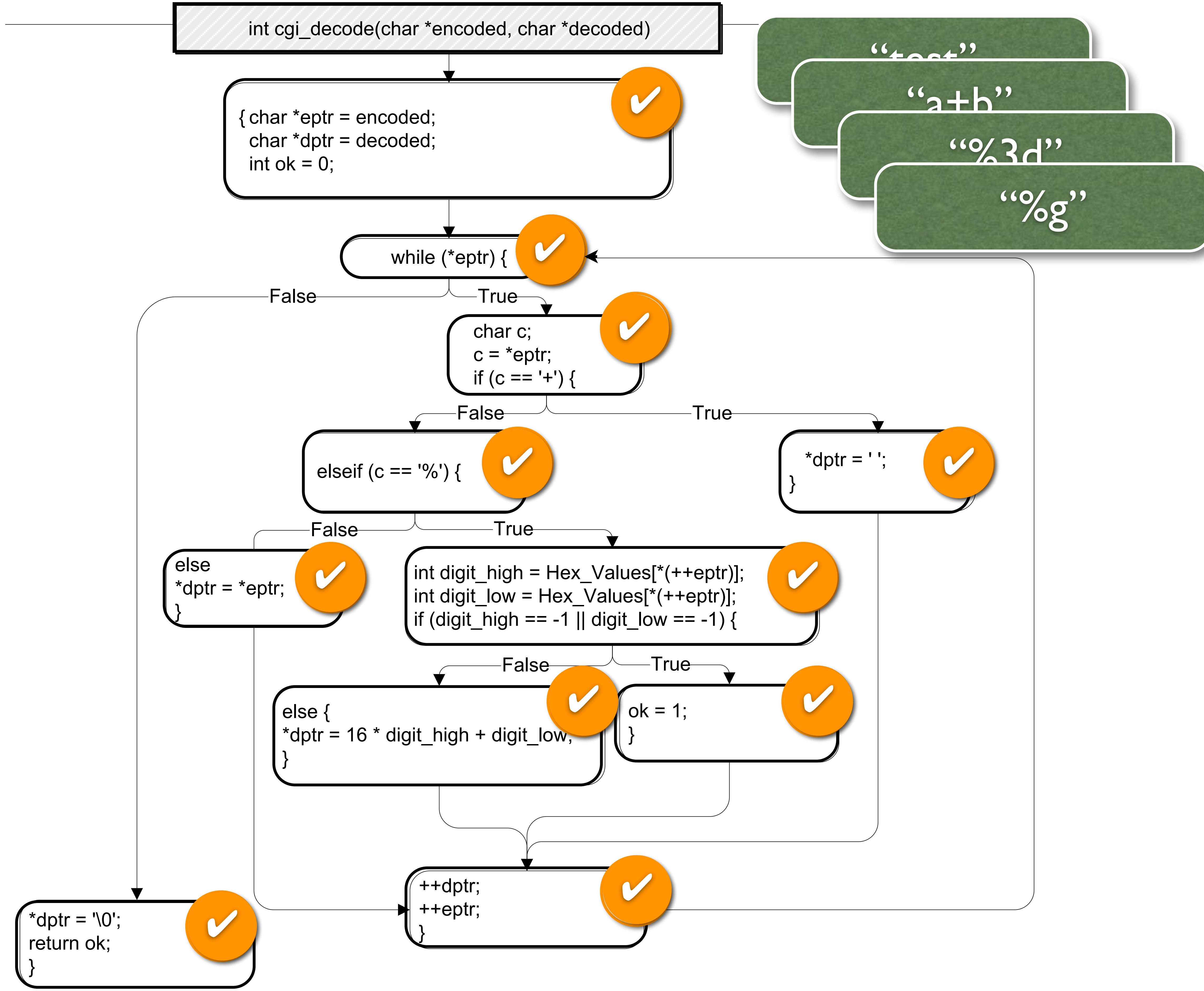


int cgi_decode(char *encoded, char *decoded)
"test"
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low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



"test"
"a+b"
"%3d"
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 = ' ';
}
False
True
else
*dptr = *eptr;
}
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;
}
*dptr = '\0';
return ok;
}
++dptr;
++eptr;
}



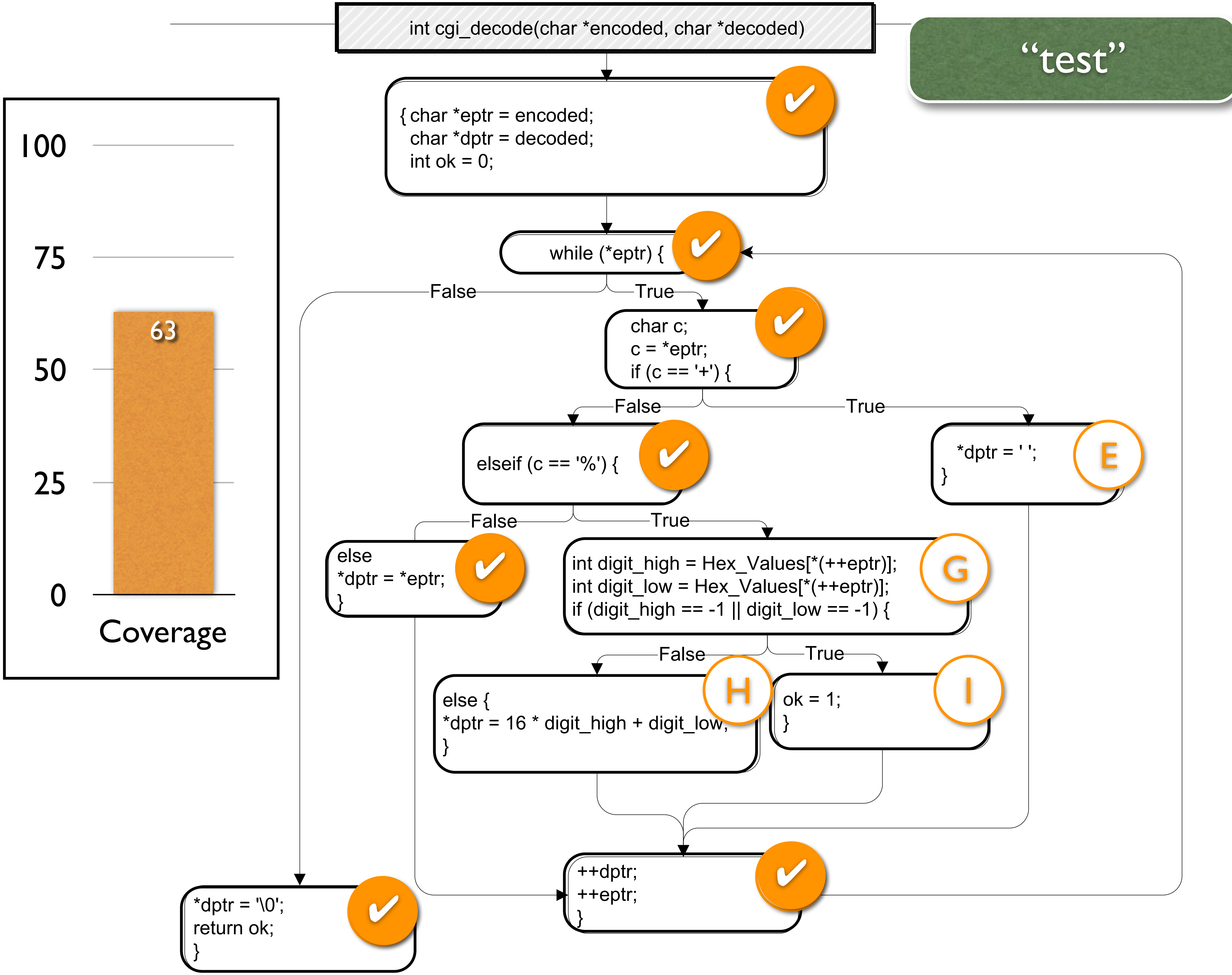
int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}
“test”
“a+b”
“%3d”
“%g”

# Test Adequacy Criteria

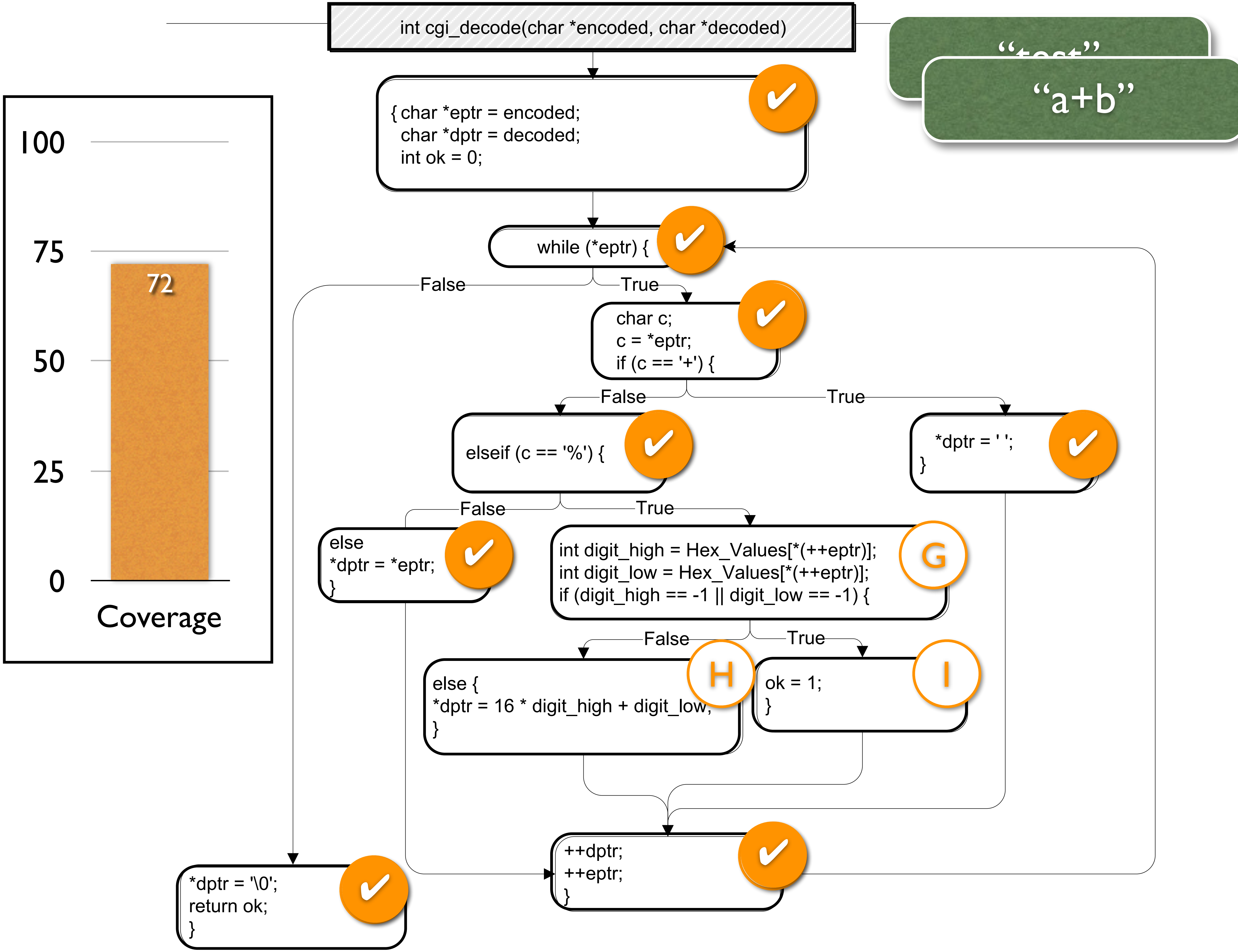
- How do we know a test suite is "good enough"?
- A *test adequacy criterion* is a predicate that is true or false for a pair ⟨*program, test suite*⟩
- Usually expressed in form of a *rule* – e.g., "all statements must be covered"

# Statement Testing

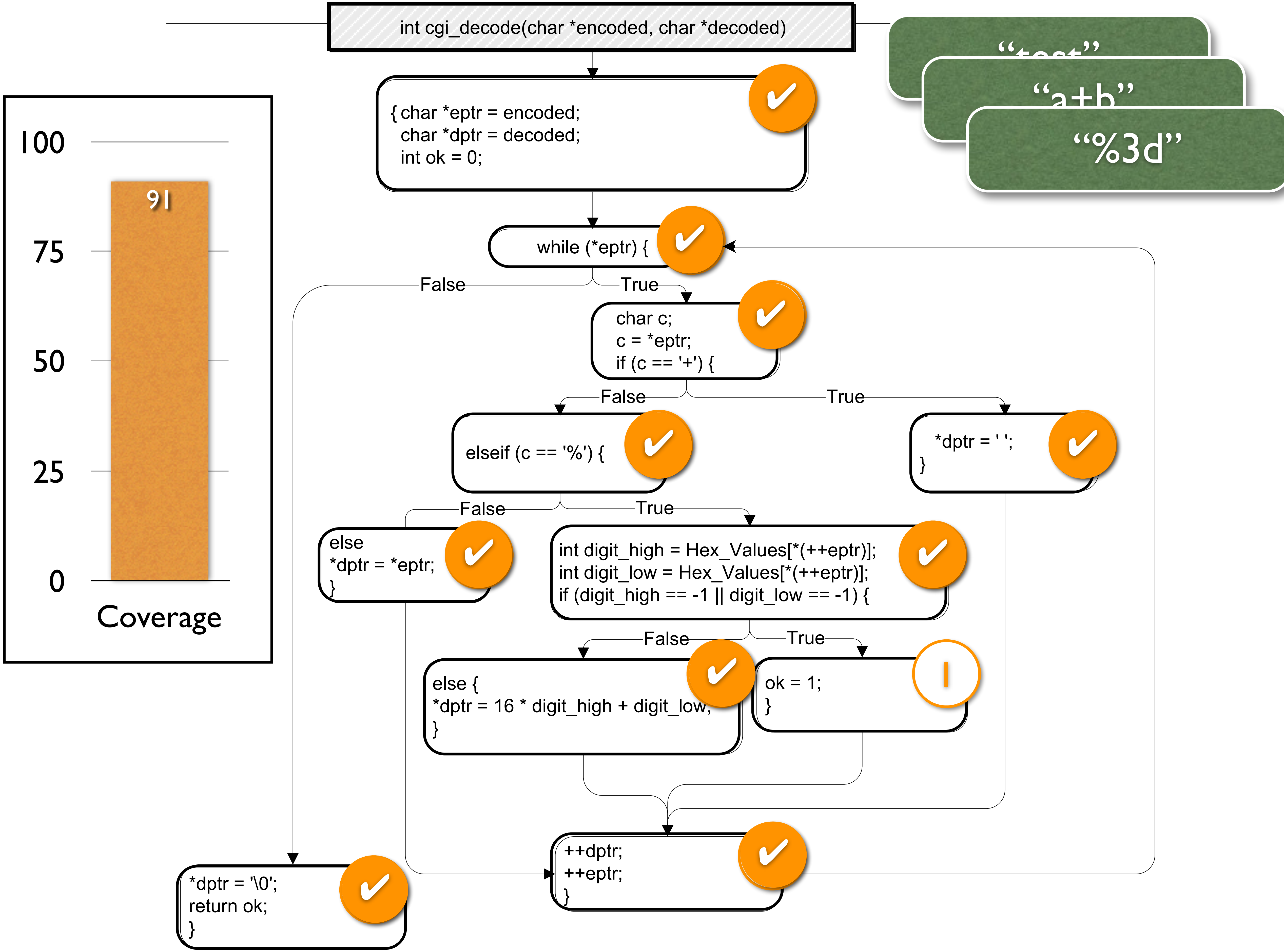
- Adequacy criterion: each statement (or node in the CFG) must be *executed at least once*
- Rationale: a defect in a statement can only be revealed by *executing* the defect
- Coverage: $\dfrac{\text{\# executed statements}}{\text{\# statements}}$



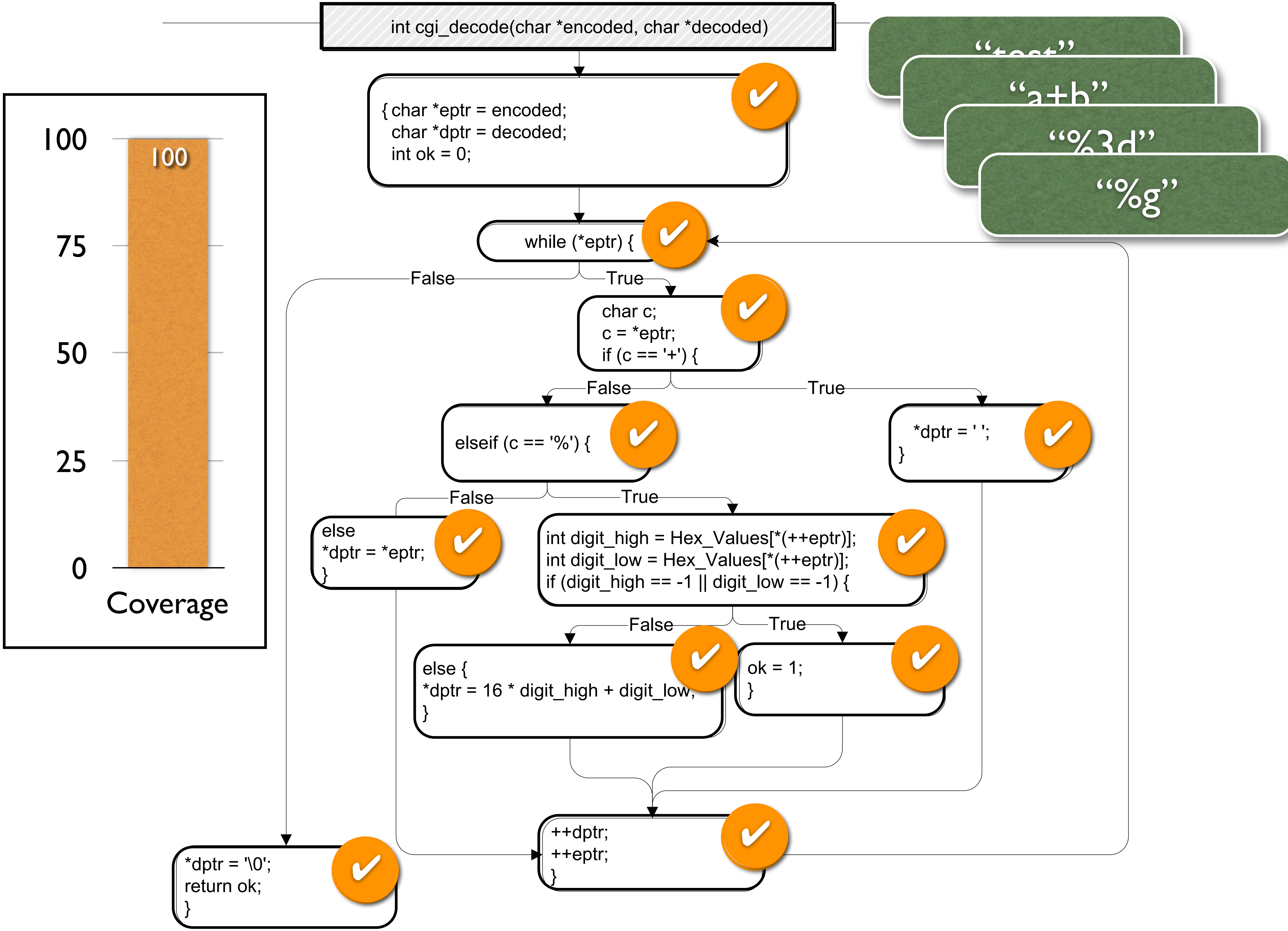
int cgi_decode(char *encoded, char *decoded)
"test"
100
75
50
25
0
63
Coverage
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}



100
75
50
25
0
72
Coverage
int cgi_decode(char *encoded, char *decoded)
“a+b”
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}



100
75
50
25
0
91
Coverage
"test"
"a+b"
"%3d"
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 = ' ';
}
False
True
else
*dptr = *eptr;
}
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;
}
1
++dptr;
++eptr;
}
*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 = ' ';
}
False
True
else
*dptr = *eptr;
}
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;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}
100
75
50
25
0
100
Coverage
“a+b”
“%3d”
“%g”

# Computing Coverage

- Coverage is computed automatically while the program executes
- Requires *instrumentation* at compile time
  With GCC, for instance, use options -ftest-coverage -fprofile-arcs
- After execution, *coverage tool* assesses and summarizes results
  With GCC, use "gcov *source-file*" to obtain readable .gcov file

Pippin: cgi_encode — less — 80×24

```
        4:   18:    int ok = 0;
        -:   19:
       38:   20:    while (*eptr)  /* loop to end of string ('\0' character) */
        -:   21:    {
        -:   22:        char c;
       30:   23:        c = *eptr;
       30:   24:        if (c == '+') {  /* '+' maps to blank */
        1:   25:            *dptr = ' ';
       29:   26:        } else if (c == '%') { /* '%xx' is hex for char xx */
        3:   27:            int digit_high = Hex_Values[*(++eptr)];
        3:   28:            int digit_low  = Hex_Values[*(++eptr)];
        5:   29:            if (digit_high == -1 || digit_low == -1)
        2:   30:                ok = 1; /* Bad return code */
        -:   31:            else
        1:   32:                *dptr = 16 * digit_high + digit_low;
        -:   33:        } else { /* All other characters map to themselves */
       26:   34:           *dptr = *eptr;
        -:   35:        }
       30:   36:        ++dptr; ++eptr;
        -:   37:    }
        4:   38:    *dptr = '\0';   /* Null terminator for string */
        4:   39:    return ok;
        -:   40:}
(END)
```

# And now…

Let's build our own coverage tools!

# cgi_decode.py

```
def cgi_decode(s):
    t = ""                 A
    i = 0
    while i < len(s):      B
        c = s[i]           C
        if c == '+':
            t = t + ' '    E
        elif c == '%':     D
            digit_high = s[i + 1]
            digit_low  = s[i + 2]      G
            i = i + 2
            if (digit_high in hex_values and
                digit_low in hex_values):
                v = (hex_values[digit_high] * 16 +
                     hex_values[digit_low])        H
                t = t + chr(v)
            else:
                raise Exception      I
        else:
            t = t + c      F

        i = i + 1          L

    return t               M
```

# Python Tracing

- In Python, tracing executions is much simpler than in compiled languages.
- The function sys.settrace(*f*) defines *f*() as a *tracing function* that is invoked for every line executed
- *f*() has access to the *entire interpreter state*

# Python Tracing

current frame (PC + variables)

“line”, “call”, “return”, …

```
import sys

def traceit(frame, event, arg):
    if event == "line":
        lineno = frame.f_lineno
        print("Line", lineno, frame.f_locals)
    return traceit

sys.settrace(traceit)
```

tracer to be used
in this scope (this one)

https://docs.python.org/2/library/sys.html?highlight=settrace#sys.settrace

Demo

# The Ingredients

Dynamic Coverage

Static Structure

Smart Algorithms

# The Ingredients

Dynamic Coverage

Static Structure

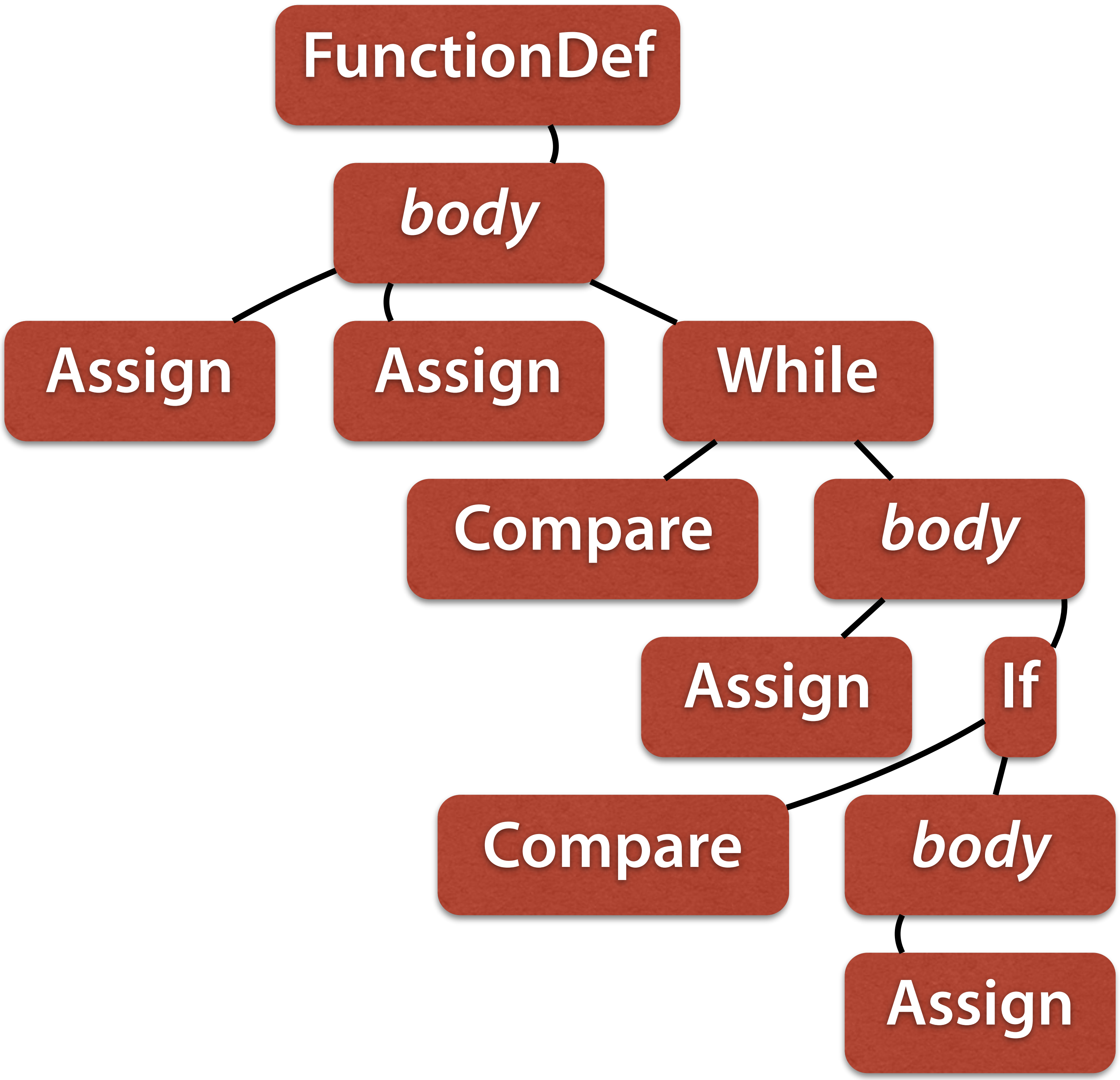
Smart Algorithms

# Coverage Goals

- With dynamic coverage, we can find out all statements executed
  (also *branches* and *paths*, if we track pairs or lists of lines)
- But how do we know the set of *possible statements?*
- Need to analyze program *statically*

# Abstract Syntax Trees

```
def cgi_decode(s):
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+':
            t = t + ' '
        elif c == '%':
            ...
        else:
            t = t + c

        i = i + 1

    return t
```



# Python AST

- The Python AST module converts a Python source file into an *abstract syntax tree*
- The tree can be traversed using a *visitor* pattern

# Python AST

AST root

Python input

```
import ast

root = ast.parse('x = 1')
print(ast.dump(root))
```

AST as string
(for debugging)

https://docs.python.org/2/library/ast.html#ast.AST

# Python AST

```
import ast

root = ast.parse('x = 1')
print ast.dump(root)


→
Module(
  body = [
    Assign(
      targets = [
        Name(id = 'x', ctx = Store())
      ],
      value = Num(n=1)
    )
  ]
)
```

Demo

# AST Visitor

- The *ast.NodeVisitor* class provides a visit(*n*) method which traverses all subnodes of *n*
- Should be subclassed to be extended
- On each node *n* of type *TYPE*, the method visit_*TYPE*(*n*) is called if it exists
- If there is no visit_*TYPE*(*n*), the method generic_visit() traverses all children

# AST Visitor

line number

```
class IfVisitor(ast.NodeVisitor):

    def visit_If(self, node):
        print("if", node.lineno, ":")
        for n in node.body:
            print("    ", n.lineno)
        print "else:"
        for n in node.orelse:
            print("    ", n.lineno)

        self.generic_visit(node)
```

show body
and “else” part

traverse children

# AST Visitor

Read Python source

```
root = ast.parse(open('cgi_decode.py').read())

v = IfVisitor()
v.visit(root)
```

Visit all IF nodes

```
→
if 34 :
        35
else:
        36
if 36 :
        37
        38
        39
        40
else:
        47
if 40 :
        42
        43
else:
        45
if 81 :
        82
        83
else:
```

# AST Visitor

```
   def cgi_decode(s):
       t = ""
       i = 0
       while i < len(s):
           c = s[i]
34         if c == '+':
35             t = t + ' '
36         elif c == '%':
37             digit_high = s[i + 1]
38             digit_low  = s[i + 2]
39             i = i + 2
40             if (digit_high in hex_values and
41                 digit_low in hex_values):
42                 v = (hex_values[digit_high] * 16 +
43                      hex_values[digit_low])
44                 t = t + chr(v)
45             else:
46                 raise Exception
47         else:
               t = t + c

           i = i + 1

       return t
```

```
→
if 34 :
    35
else:
    36
if 36 :
    37
    38
    39
    40
else:
    47
if 40 :
    42
    43
else:
    45
if 81 :
    82
    83
else:
```

Demo

# The Ingredients

Dynamic Coverage

Static Structure

Smart Algorithms

# The Ingredients

Dynamic Coverage

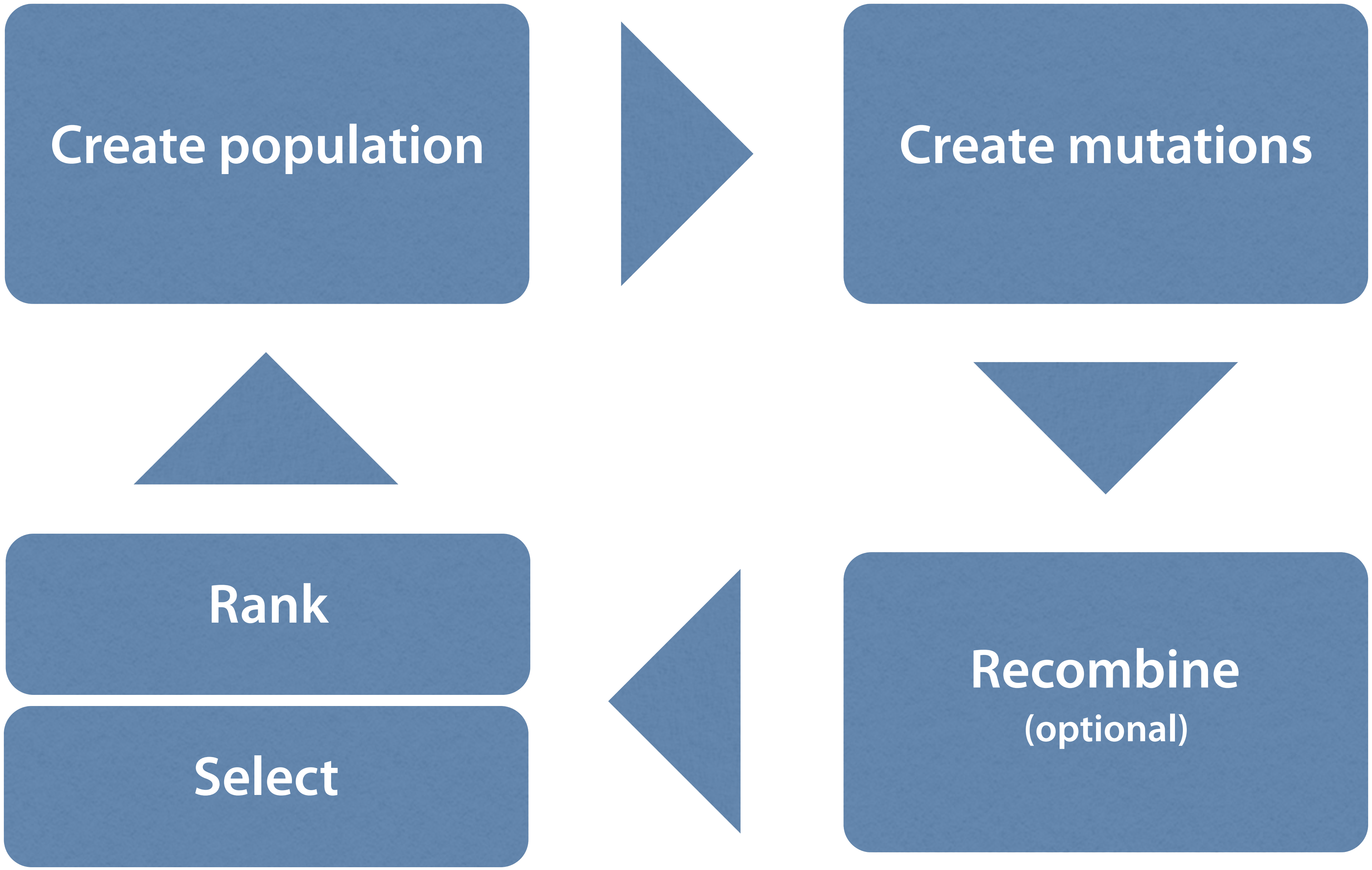
Static Structure

Smart Algorithms

# Approaches

- **Random Testing**: *ignore* program structure
- **Symbolic Testing**: *solve* path conditions leading to uncovered statements
- **Search-Based Testing**: still random, but have structure *guide* test generation

# Evolutionary Algorithms



# Evolutionary Algorithms

**Create population**

“fdsakfh+ew%3gfhdi4f”

“fwe8^ru786234jä”

**Mutate**

“fdsakfh+br%3gfhdi%4f”

“fdsakfh+ew%4gfhdi%4f”

“fwe8^ru&26234jä”

“xb3#ru786234jä”

# Evolutionary Algorithms

**Mutate**

“fdsakfh+br%3gfhdi%4f”

“fdsakfh+ew%4gfhdi%4f”

“fwe8^ru&26234jä”

“xb3#ru786234jä”

**Recombine**

“fdsakfh+ew%4gfhdi%4f”

“xb3#ru786234jä”

# Evolutionary Algorithms

## Mutate

“fdsakfh+br%3gfhdi%4f”

“fdsakfh+ew%4gfhdi%4f”

“fwe8^ru&26234jä”

“xb3#ru786234jä”

## Recombine

“fdsakfh+ew%4gfhdi%4f”

“xb3#ru786234jä”

“xb3#akfh+ew%4gfhdi%4f”

# Selection and Ranking

```
if (angle = 47 ∧ force = 532) { … }
```

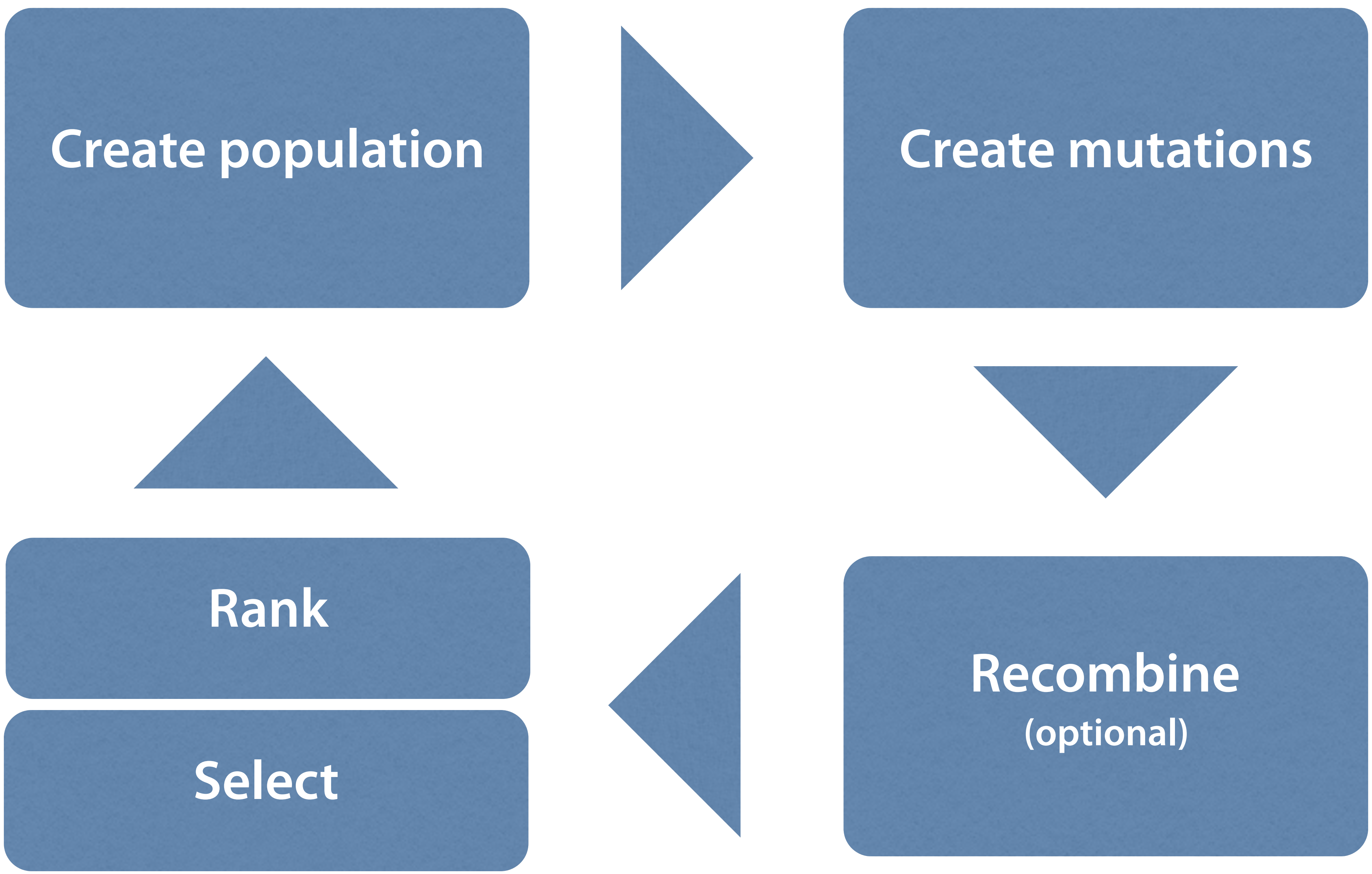
angle = 51

“xb3#ru786234jä”

angle = 48

angle = 47

# Evolutionary Algorithms



# And now…

Let’s implement this in Python!

# The General Plan

- Create a *population* of random inputs
- Obtain their *coverage*
- Higher coverage = higher fitness
  (a bit simplistic, but will do the job)
- *Select* individuals with high fitness
  (say, the 25% fittest individuals)
- *Mutate* them to obtain offspring

# The Mutation Plan

- For each input, keep a history of the *grammar productions* that lead to it

```
$START → $EXPR → $TERM →
$FACTOR → $INTEGER → $DIGIT → 2
```

- To mutate, *truncate* that history and apply different productions from there on

```
$START → $EXPR → $TERM →
~~$FACTOR → $INTEGER → $DIGIT → 2~~
$FACTOR → $INTEGER → $DIGIT → 4
```

# CGI Grammar

```
cgi_grammar = {
    "$START": ["$STRING"],

    "$STRING": ["$CHARACTER", "$STRING$CHARACTER"],

    "$CHARACTER": ["$REGULAR_CHARACTER", "$PLUS", "$PERCENT"],

    "$REGULAR_CHARACTER": ["a", "b", "c", ".", ":", "!"],
              # actually more

    "$PLUS": ["+"],

    "$PERCENT": ["%$HEX_DIGIT$HEX_DIGIT"],

    "$HEX_DIGIT": ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
                   "a", "b", "c", "d", "e", "f"]
}
```

Demo

# Evolution Cycle

```
pop = population(grammar)

for i in range(EVOLUTION_CYCLES):
    # Evolve the population

    print("Evolved:")
    next_pop = evolve(pop, grammar)
    print_population(next_pop)
    pop = next_pop
```

# Initial Population

```
# Create a random population
def population(grammar):
    pop = []
    while len(pop) < POPULATION_SIZE:
        try:
            # Create a random individual
            term, productions = produce(cgi_grammar)
        except AssertionError:
            # Try again
            continue

        # Determine its fitness (by running the test, actually)
        fitness = coverage_fitness(term)

        # Add it to the population
        pop.append((term, productions, fitness))

    return pop
```

# Fitness

```
# Where we store the coverage
coverage = {}

# Now, some dynamic analysis
def traceit(frame, event, arg):
    global coverage
    if event == "line":
        lineno = frame.f_lineno
        # print("Line", lineno, frame.f_locals)
        coverage[lineno] = True
    return traceit

# Define the fitness of an individual term - by actually testing it
def coverage_fitness(term):
    # Set up the tracer
    global coverage
    coverage = {}
    sys.settrace(traceit)

    # Run the function under test
    result = cgi_decode(term)

    # Turn off the tracer
    sys.settrace(None)

    # Simple approach:
    # The term with the highest coverage gets the highest fitness
    return len(coverage.keys())
```

# Evolution

```
def by_fitness(individual):
    (term, production, fitness) = individual
    return fitness

# Evolve the set
def evolve(pop, grammar):
    # Sort population by fitness (highest first)
    best_pop = sorted(pop, key=by_fitness, reverse=True)

    # Select the fittest individuals
    best_pop = best_pop[:SELECTION_SIZE]

    # Breed
    offspring = []
    while len(offspring) + len(best_pop) < POPULATION_SIZE:
        parent = random.choice(best_pop)
        child = mutate(parent, grammar)

        (parent_term, parent_productions, parent_fitness) = parent
        (child_term, child_productions, child_fitness) = child
```

```python
# Evolve the set
def evolve(pop, grammar):
    # Sort population by fitness (highest first)
    best_pop = sorted(pop, key=by_fitness, reverse=True)

    # Select the fittest individuals
    best_pop = best_pop[:SELECTION_SIZE]

    # Breed
    offspring = []
    while len(offspring) + len(best_pop) < POPULATION_SIZE:
        parent = random.choice(best_pop)
        child = mutate(parent, grammar)

        (parent_term, parent_productions, parent_fitness) = parent
        (child_term, child_productions, child_fitness) = child

        if child_fitness >= parent_fitness:
            offspring.append(child)

    next_pop = best_pop + offspring

    # Keep it sorted
    next_pop = sorted(next_pop, key=by_fitness, reverse=True)

    return next_pop
```

# Mutation

```
# Create a mutation from PARENT, generating one offspring
def mutate(parent, grammar):
    (parent_term, parent_productions, parent_fitness) = parent

    # Truncation cutoff: only keep CUTOFF productions
    cutoff = random.randint(0, len(parent_productions) - 1)

    # Repeat the first CUTOFF production steps of parent
    child_term = "$START"
    child_productions = []
    for i in range(cutoff):
        rule = parent_productions[i]
        child_term = apply_rule(child_term, rule)
        child_productions.append(rule)

    # From here on, proceed in random direction
    extra_productions = None
    while extra_productions is None:
        try:
            child_term, extra_productions = produce(grammar, child_term)
        except AssertionError:
            pass # Just try again
```

```
# Repeat the first CUTOFF production steps of parent
child_term = "$START"
child_productions = []
for i in range(cutoff):
    rule = parent_productions[i]
    child_term = apply_rule(child_term, rule)
    child_productions.append(rule)

# From here on, proceed in random direction
extra_productions = None
while extra_productions is None:
    try:
        child_term, extra_productions = produce(grammar, child_term)
    except AssertionError:
        pass # Just try again

child_productions += extra_productions

# Compute its fitness
child_fitness = coverage_fitness(child_term)

print("Mutated " + repr(parent_term) + " to " + repr(child_term))

# And we're done
return child_term, child_productions, child_fitness
```

# Populations

### (with fitness)

| Initial | After 20 Cycles |
|---|---|
| '+%60c!a%08' 16 | '+%60c!a%08' 16 |
| '%8fc+%8da.+' 16 | '%8fc+%8da.+' 16 |
| '++%f2!' 16 | '++%f2!' 16 |
| 'b%26%d2%60' 15 | '+++%80a.+' 16 |
| '%f6b' 15 | '%9fc+%8da.+' 16 |
| 'b%f2' 15 | '%61+%75a.+' 16 |
| ':%c5' 15 | '++%21b.+' 16 |
| '%60+' 15 | '%1c%04+%a3+.+' 16 |
| '%87' 14 | '%7b++c.+' 16 |
| '%1a' 14 | '+!%fa+%21a.+' 16 |
| '%53' 14 | '+%ca+%71!.+' 16 |
| '%77' 14 | '+%60c!a%08' 16 |
| '+.' 10 | '%e0b+a' 16 |
| '+!+' 10 | '%99c+%8da.+' 16 |
| '!' 9 | '%d4++%8ca.+' 16 |
| ':' 9 | '%20a+%f7b' 16 |
| '+' 8 | '++%f2a' 16 |
| '+' 8 | '%95c+' 16 |
| '++' 8 | '%7fc+%8da.+' 16 |
| '++' 8 | '++%f2!' 16 |

# Things to do

- Use a *derivation tree* to represent both inputs and histories (much more efficient)
- Use a *genetic algorithm* with *recombination* rather than only mutation
- Base fitness function on *approach level* – how close are we to a yet uncovered line?
- Integrate code and grammar coverage…