Structural Testing

Structural Testing

- Also called White-Box testing.
- More technical.
- Designs test cases from source code not from specifications.
- Considers internal structure of the code.

Structural Testing Techniques:

- Control Flow testing
- Data Flow testing

Control Flow Testing

- This technique is very popular due to its simplicity and effectiveness.
- We identify paths of the program and write test cases to execute those paths.
- path is a sequence of statements that begins at an entry and ends at an exit.
- There may be too many paths in a program and it may not be feasible to execute all of them.
- As the number of decisions increase in the program, the number of paths also increase accordingly.

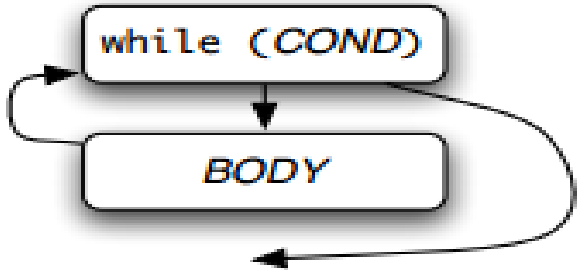
Control Flow Testing(contd..)

- ‘Coverage’ is defined as a ‘percentage of source code that has been tested with respect to the total source code available for testing’.
- The most reasonable level may be to test every statement of a program at least once before the completion of testing.
- Write test cases that ensure the execution of every statement.

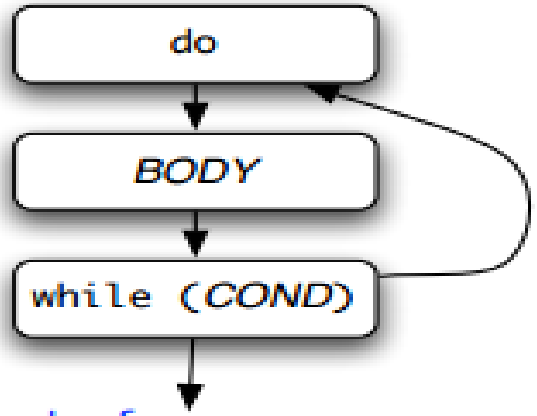
Types of CF testing

- Statement Testing
- Branch coverage
- Condition coverage
- Path coverage

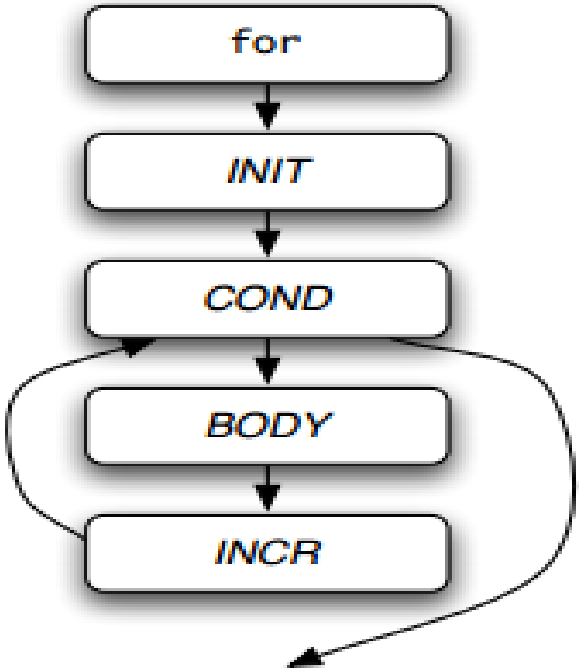
Control Flow Patterns



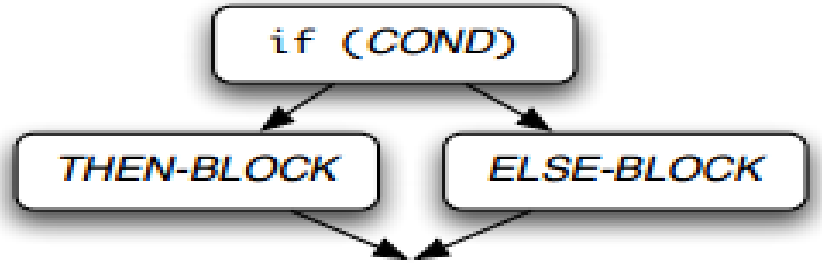

```
while (COND)
  BODY;
```




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




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

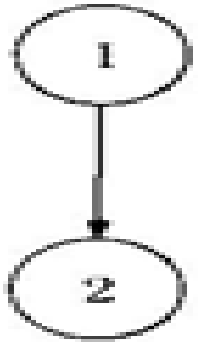



```
if (COND)
  THEN-BLOCK;
else
  ELSE-BLOCK;
```

Examples on CFG

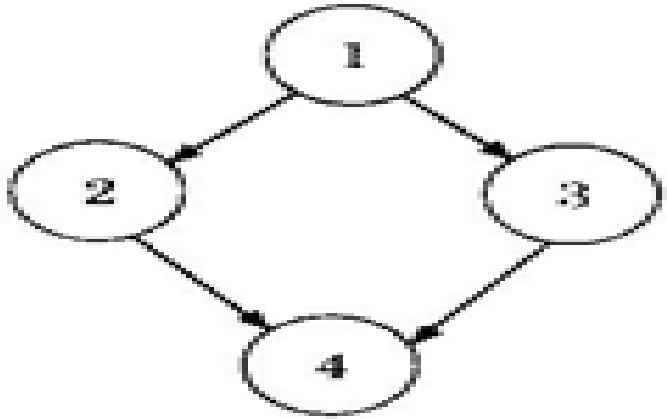
Sequence:

```
1. a=5;
2. b=a*2-1
```



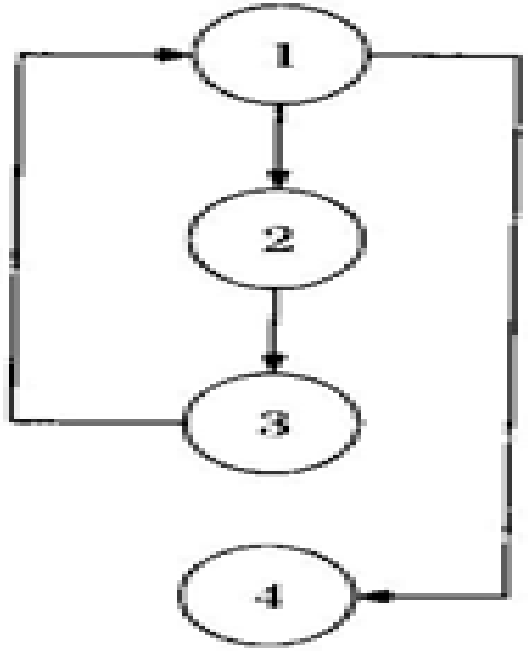
Selection:

```
1. if(a>b)
2.    c=3;
3. else  c=5;
4. c=c*c;
```



Iteration:

```
1. while(a>b){
2.    b=b-1;
3.    b=b*a;}
4. c=a+b;
```



No guarantees

- Executing all control flow elements does not guarantee finding all faults
  - Execution of a faulty statement may not always result in a failure
    - The state may not be corrupted when the statement is executed with some data values
    - Corrupt state may not propagate through execution to eventually lead to failure
- What is the value of structural coverage?
  - Increases confidence in thoroughness of testing
    - Removes some obvious *inadequacies*

Statement testing

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once
- Coverage:

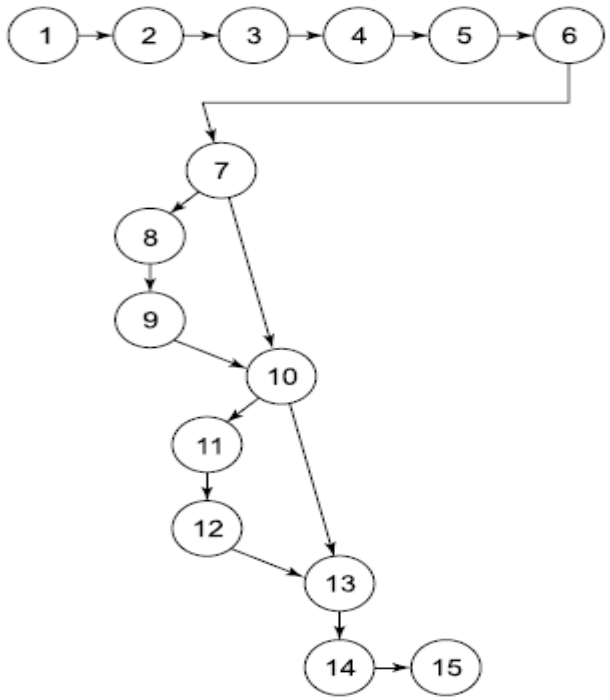
$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

- Rationale: a fault in a statement can only be revealed by executing the faulty statement

Statement Testing

- Execute every statement of the program in order to achieve 100% statement coverage/node coverage.
- Granularity of a node in CFG can be one or more statements.

```
#include<stdio.h>
#include<conio.h>

1.    void main()
2.    {
3.    int a,b,c,x=0,y=0;
4.    clrscr();
5.    printf("Enter three numbers:");
6.    scanf("%d %d %d",&a,&b,&c);
7.    if((a>b)&&(a>c)){
8.          x=a*a+b*b;
9.    }
10.   if(b>c){
11.         y=a*a-b*b;
12.   }
13.   printf("x= %d y= %d",x,y);
14.   getch();
15.   }
```



Test case 1: a=9, b=8, c=7

Statements or blocks?

- Nodes in a control flow graph often represent basic blocks of multiple statements
  - Some standards refer to *basic block* coverage or *node coverage*
  - Difference in granularity, not in concept
- No essential difference
  - 100% node coverage <-> 100% statement coverage
    - but levels will differ below 100%
  - A test case that improves one will improve the other
    - though not by the same amount, in general

Branch coverage

- Test every branch of the program. Hence, we wish to test every ‘True’ and ‘False’ condition of the program.
- The branch coverage guarantees 100% statement coverage.

- Test case 1: a=9, b=8, c=7 (To test all the true conditions)
- Test case 2: a=7, b=8, c=9 (To test all the false conditions)

Condition Coverage

- Condition coverage is better than branch coverage because we want to test every condition at least once.
- However, branch coverage can be achieved without testing every condition.
- In the previous example there are two conditions (a>b) and (a>c). Hence we have four possibilities namely:
  - ✓ Both are true
  - ✓ First is true, second is false
  - ✓ First is false, second is true
  - ✓ Both are false

- (i) a = 9, b = 8, c = 7 (first possibility when both are true)
- (ii) a = 9, b = 8, c = 10 (second possibility – first is true, second is false)
- (iii) a = 7, b = 8, c = 9 (third and fourth possibilities- first is false, statement number 7 is false)

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
  */
```

CGI

Selected ASCII Values

hex	char	hex	char	hex	char
%20	(blank)	%2b	+	%40	@
%21	!	%2c	,	%5b	[
%22	"	%2d	-	%5c	\
%23	#	%2e	.	%5d	]
%24	$	%2f	/	%5e	^
%25	%	%3a	:	%5f	_
%26	&	%3b	;	%60	`
%27	'	%3c	<	%7b	{
%28	(	%3d	=	%7c	|
%29	)	%3e	>	%7d	}
%2a	*	%3f	?	%7e	~
hex	char	hex	char	hex	char

ASCII & HEX

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0	NUL	16	10	DLE	32	20	(space)	48	30	0
1	1	SOH	17	11	DC1	33	21	!	49	31	1
2	2	STX	18	12	DC2	34	22	"	50	32	2
3	3	ETX	19	13	DC3	35	23	#	51	33	3
4	4	EOT	20	14	DC4	36	24	$	52	34	4
5	5	ENQ	21	15	NAK	37	25	%	53	35	5
6	6	ACK	22	16	SYN	38	26	&	54	36	6
7	7	BEL	23	17	ETB	39	27	'	55	37	7
8	8	BS	24	18	CAN	40	28	(	56	38	8
9	9	TAB	25	19	EM	41	29	)	57	39	9
10	A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	C	FF	28	1C	FS	44	2C	,	60	3C	<
13	D	CR	29	1D	GS	45	2D	-	61	3D	=
14	E	SO	30	1E	RS	46	2E	.	62	3E	>
15	F	SI	31	1F	US	47	2F	/	63	3F	?

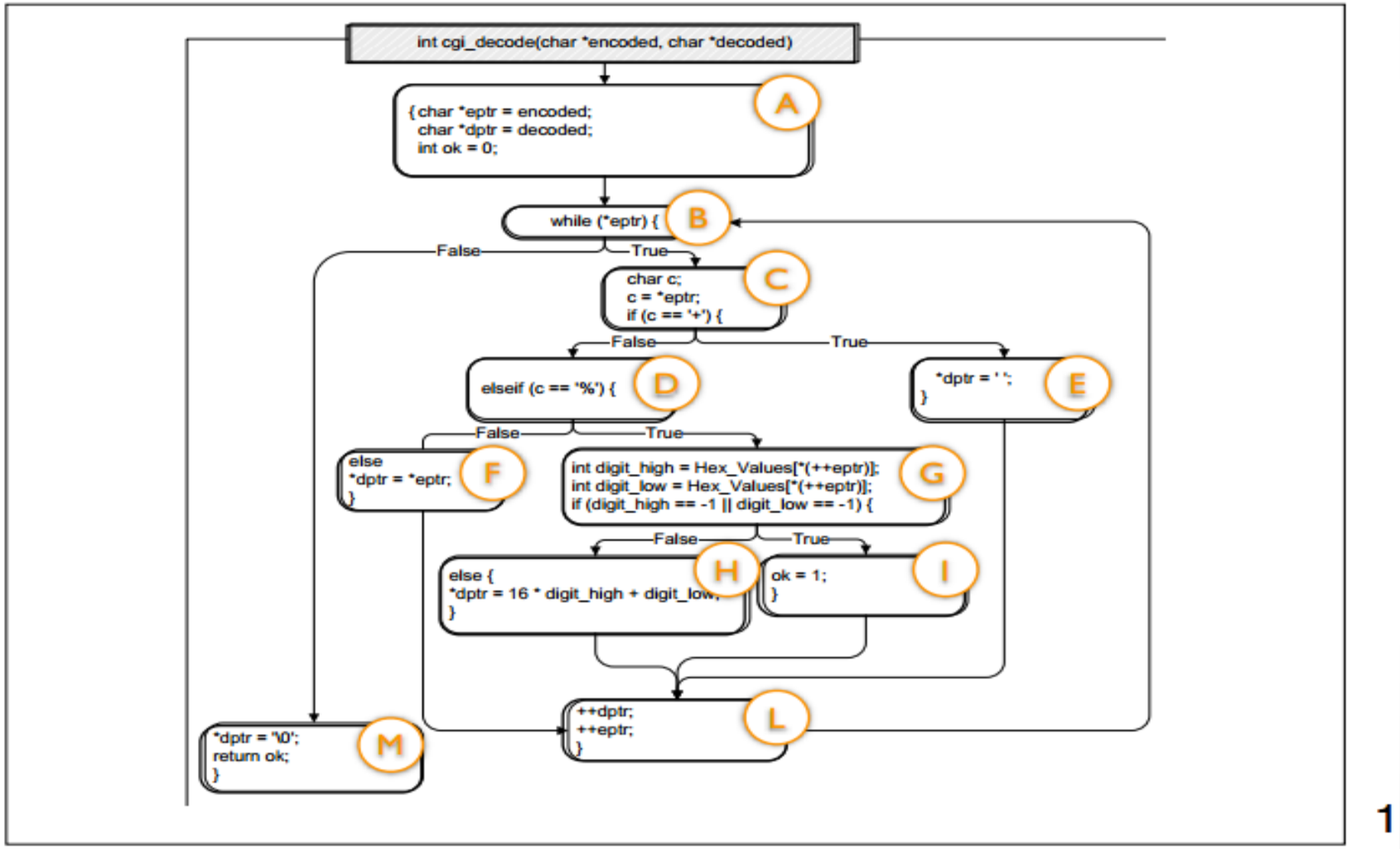
ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[	107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	|
77	4D	M	93	5D	]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	

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;
    int ok = 0;
```

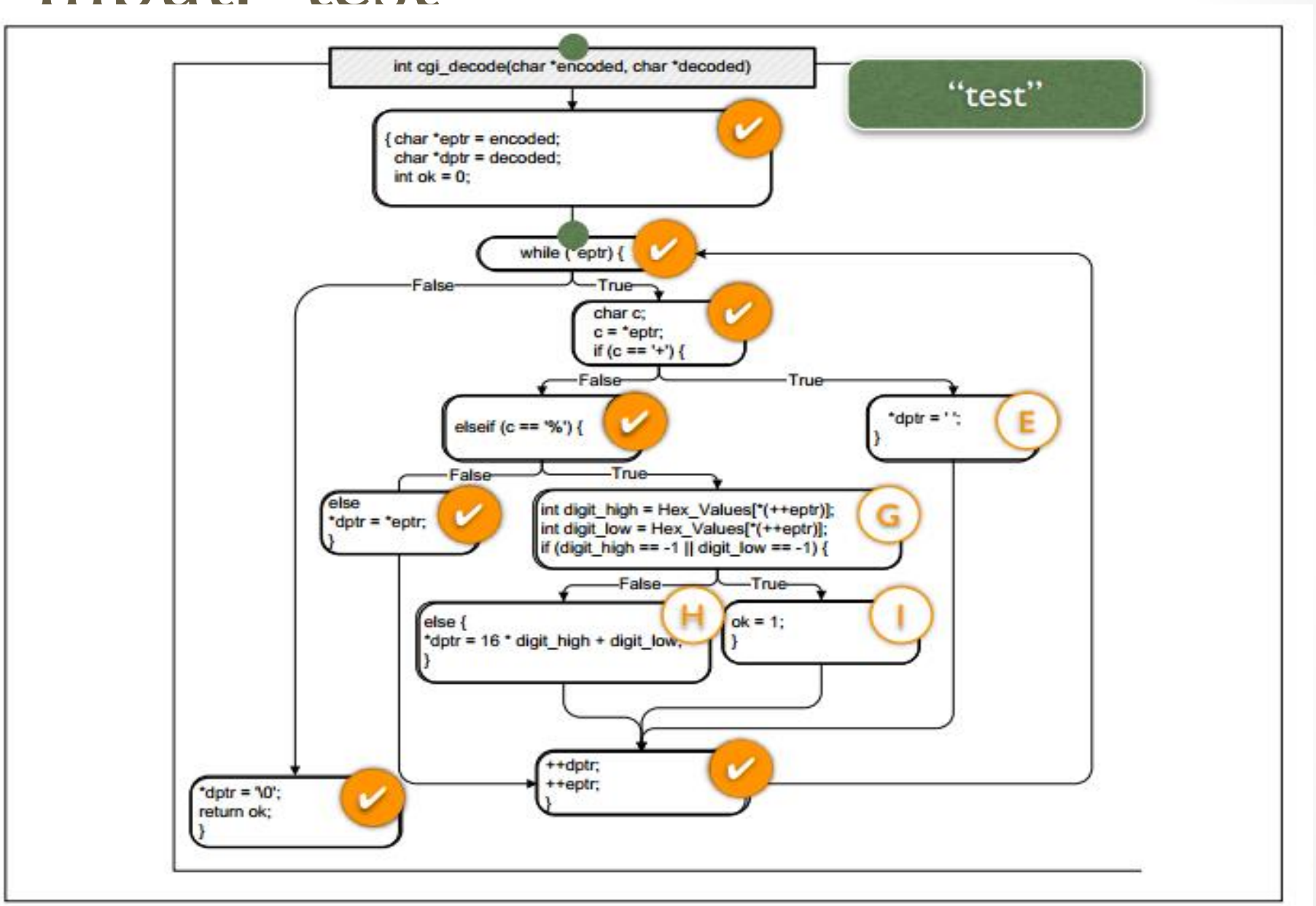
```
    while (*eptr)  /* loop to end of string ('\0' character) */ B
    {
        char c;
        c = *eptr; C
        if (c == '+') {  /* '+' maps to blank */
            *dptr = ' '; E
        } else if (c == '%') { /* '%xx' is hex for char xx */ D
            int digit_high = Hex_Values[*(++eptr)];
            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;
}
```

CFG for CGI program

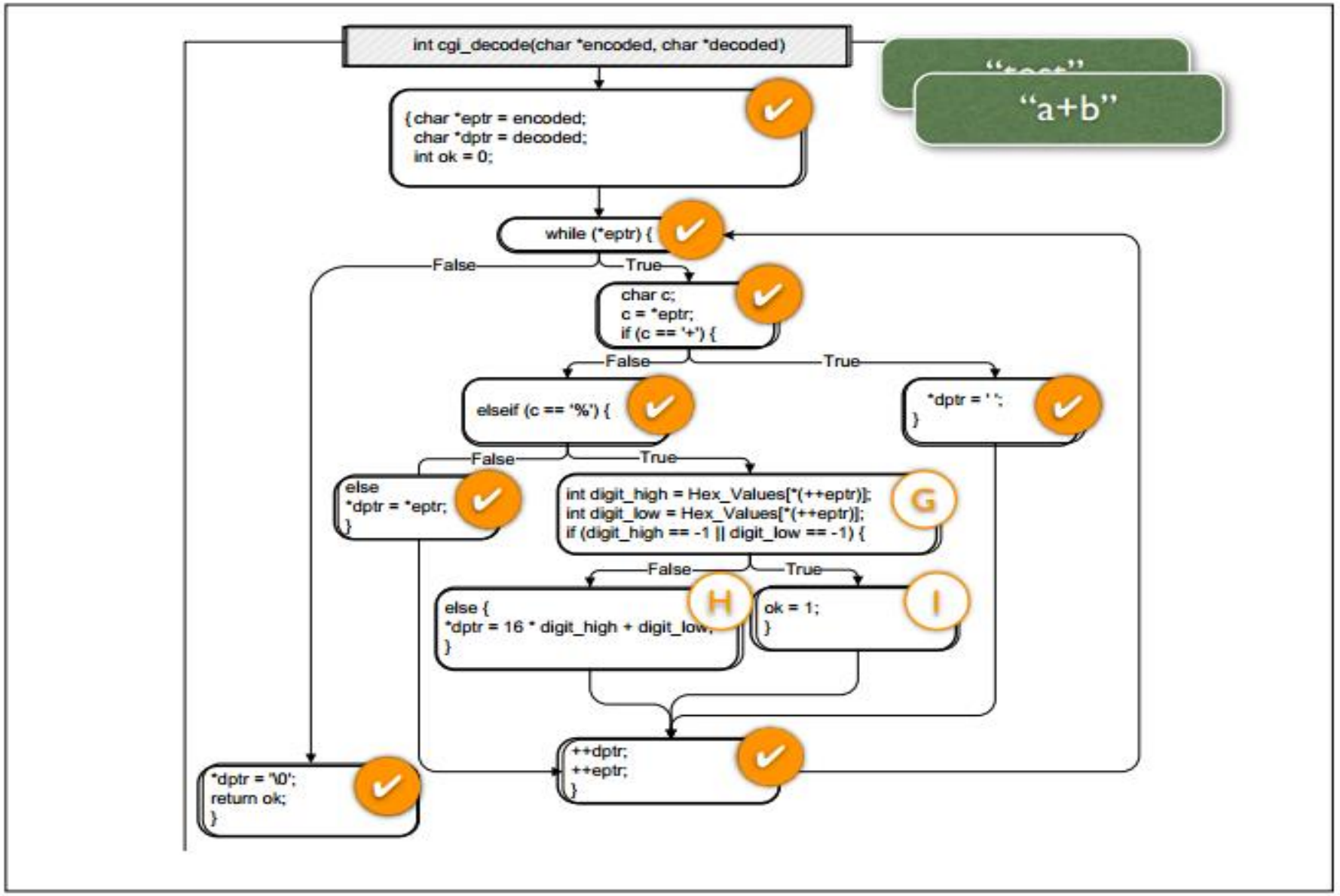


1

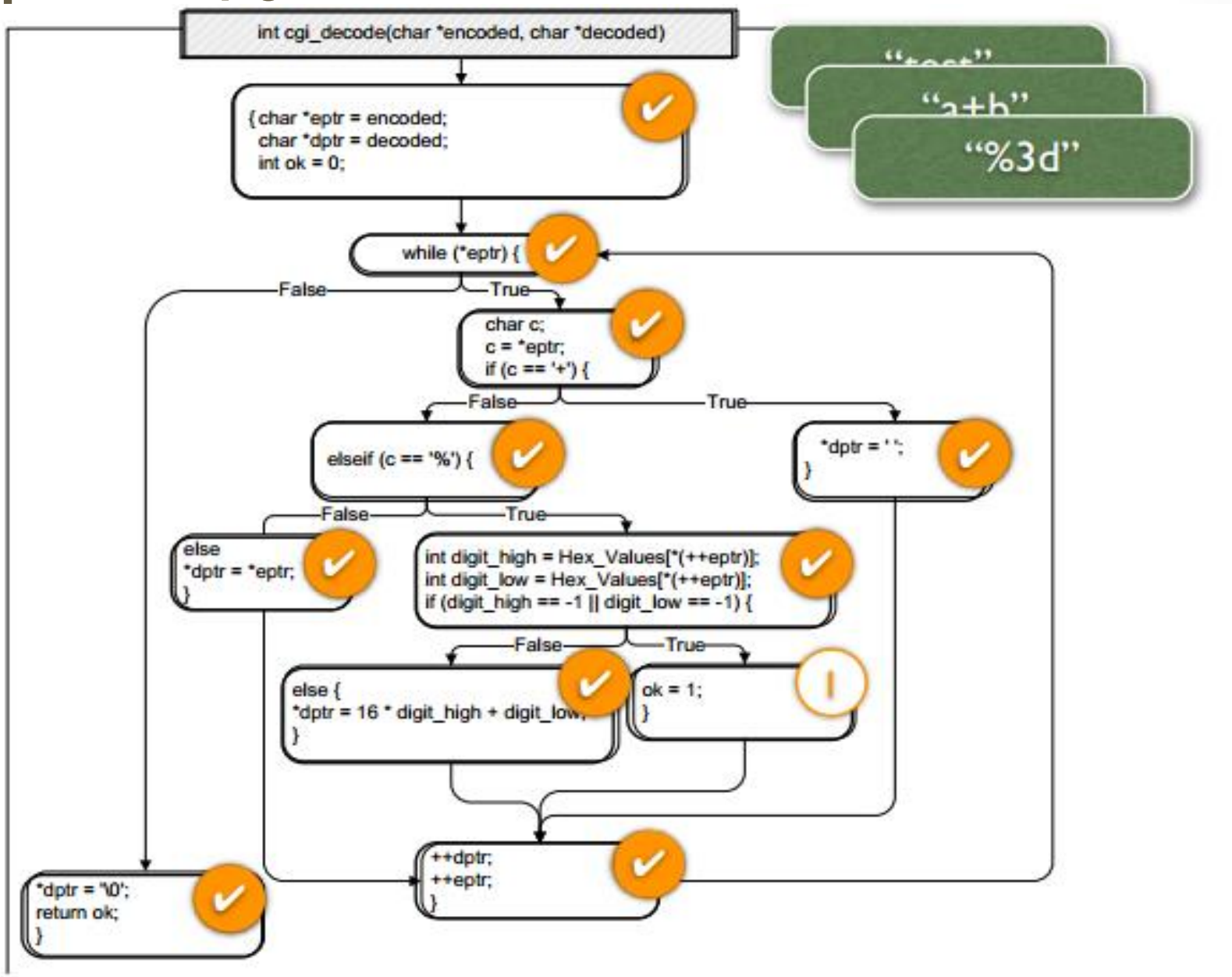
Input: “test”


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;
}

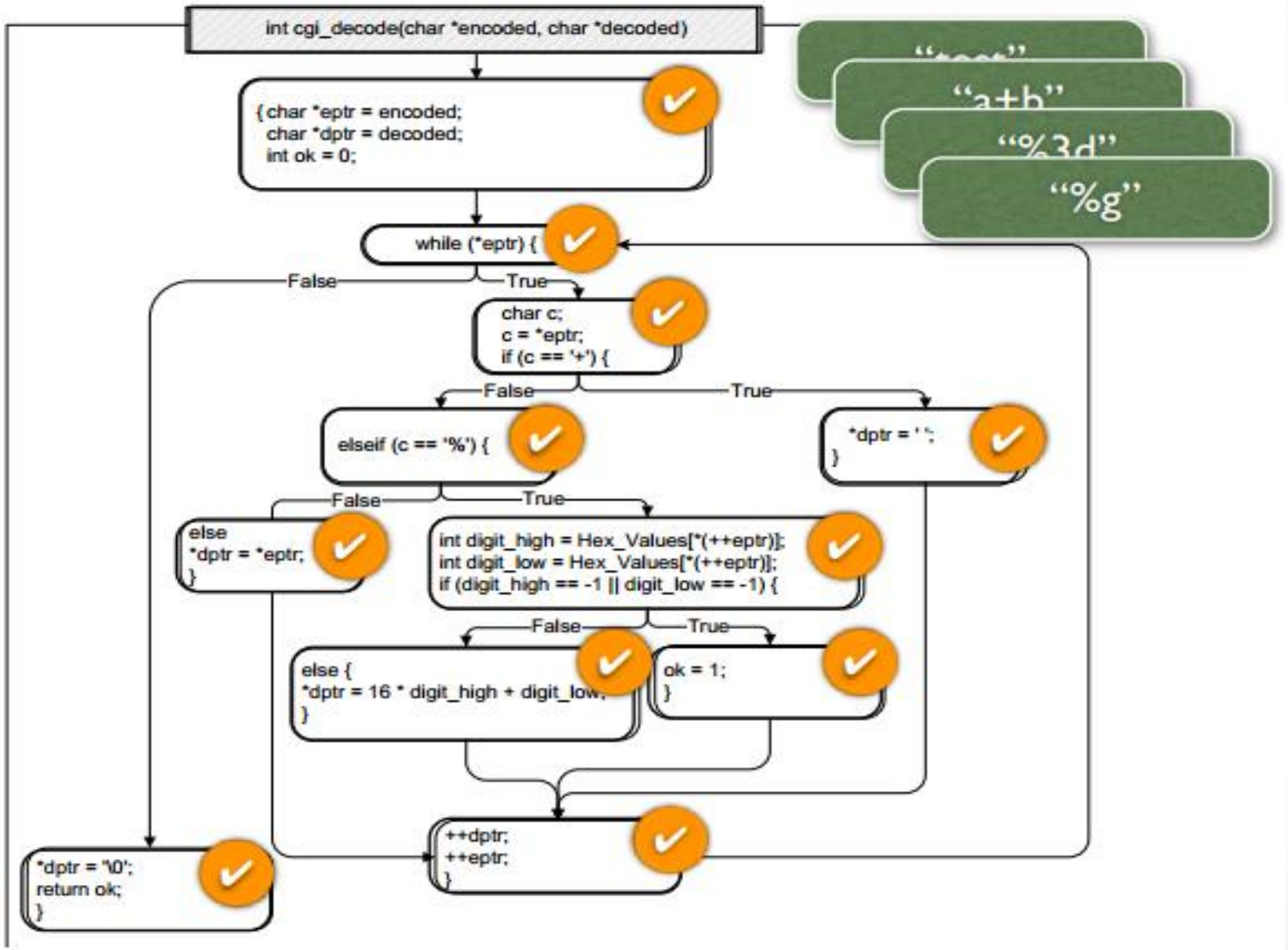
Input: “a+b”


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;
}

Input: “%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;
++eptr;
}
*dptr = '\0';
return ok;
}
“%3d”

Input: “%g”


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;
}
“a+b”
“%3d”
“%g”

THE END