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.

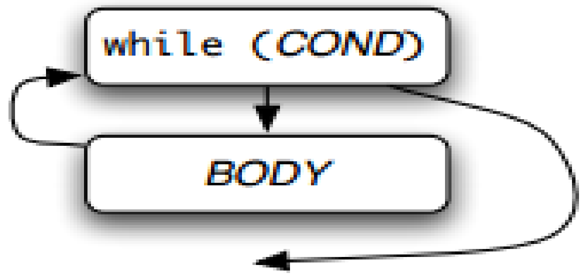
Control Flow Testing(contd..)

- ‘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.

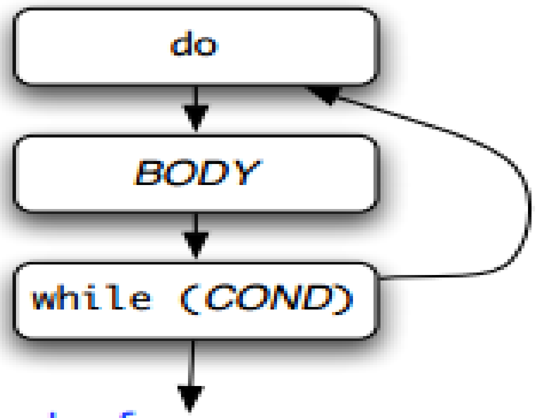
Types of CF testing

- Statement Coverage
- Branch coverage
- Condition coverage
- Path coverage

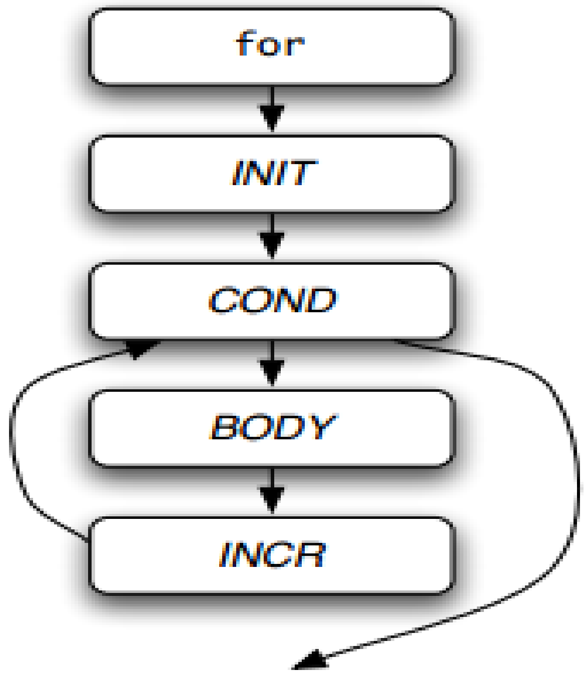
Control Flow Patterns



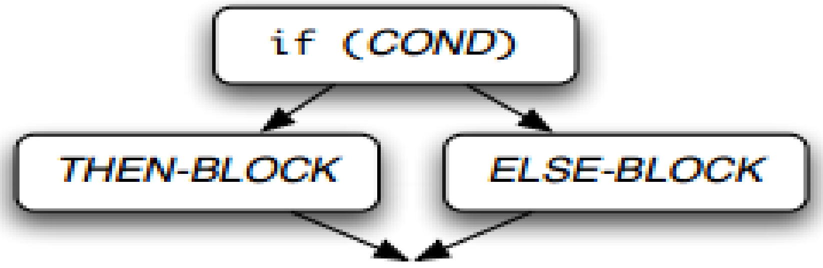
```
while (COND)
  BODY;
```



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



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

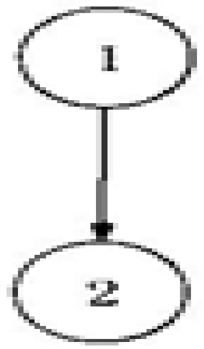


```
if (COND)
  THEN-BLOCK;
else
  ELSE-BLOCK;
```

Examples on CFG

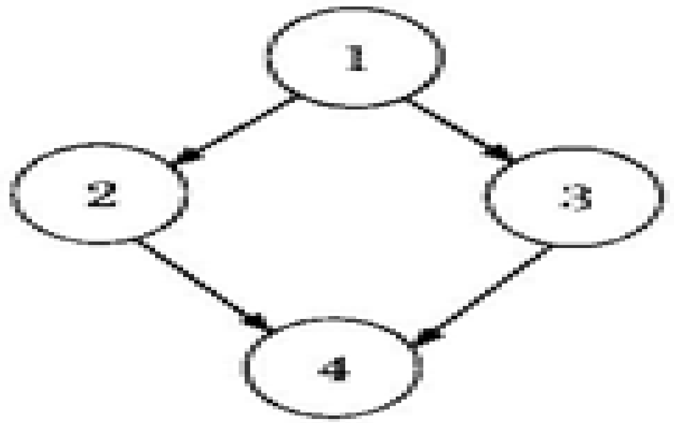
Sequence:

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



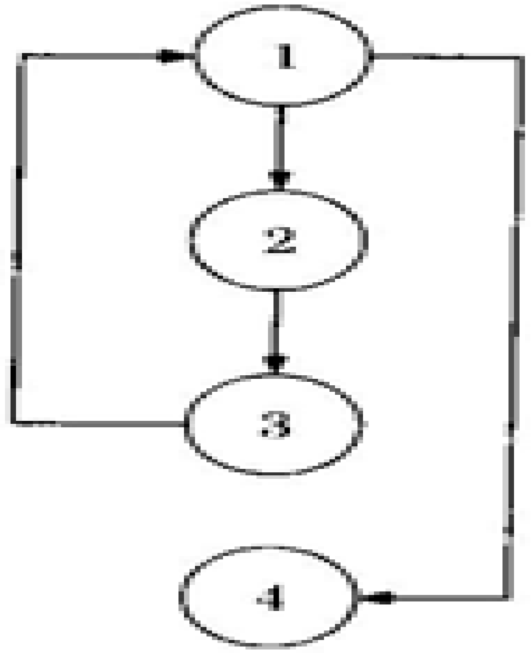
Selection:

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



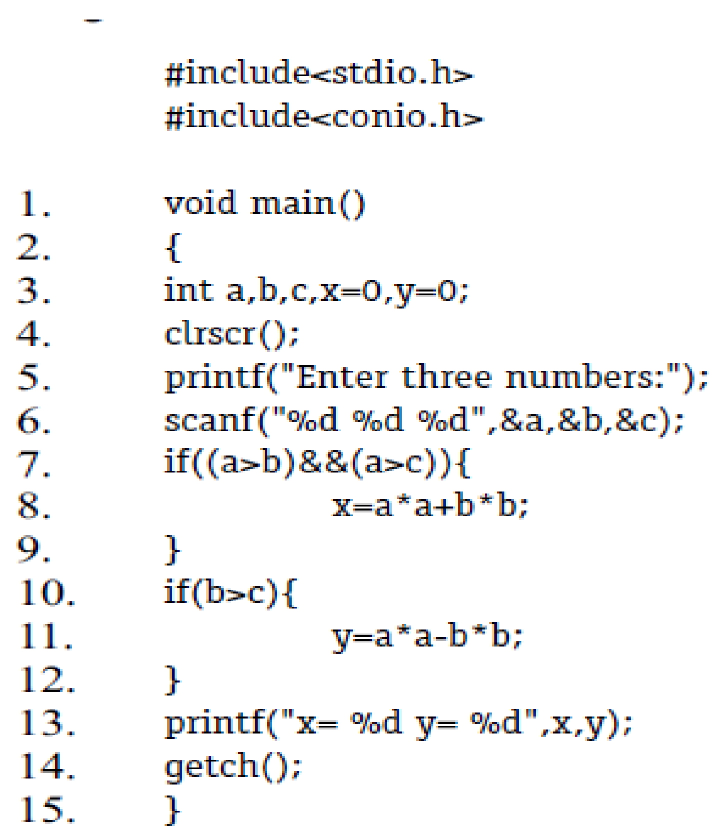
Iteration:

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

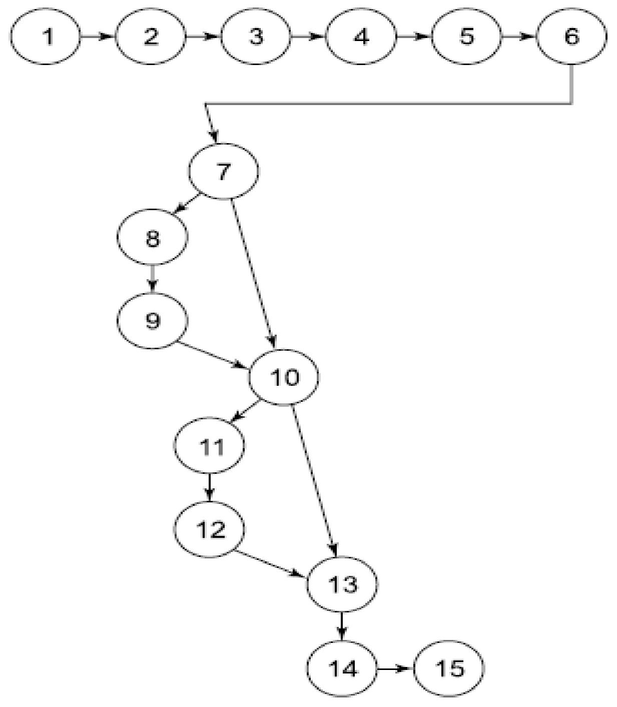


Statement Coverage

- 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.     }
```


$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[r] & 5 \arrow[r] & 6 \arrow[d] \\
 & & & & & 7 \arrow[d] \arrow[dd] \\
 & & & & & 8 \arrow[d] \\
 & & & & & 9 \arrow[d] \\
 & & & & & 10 \arrow[d] \arrow[dd] \\
 & & & & & 11 \arrow[d] \\
 & & & & & 12 \arrow[d] \\
 & & & & & 13 \arrow[d] \\
 & & & & & 14 \arrow[r] & 15
\end{tikzcd}$$

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

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.

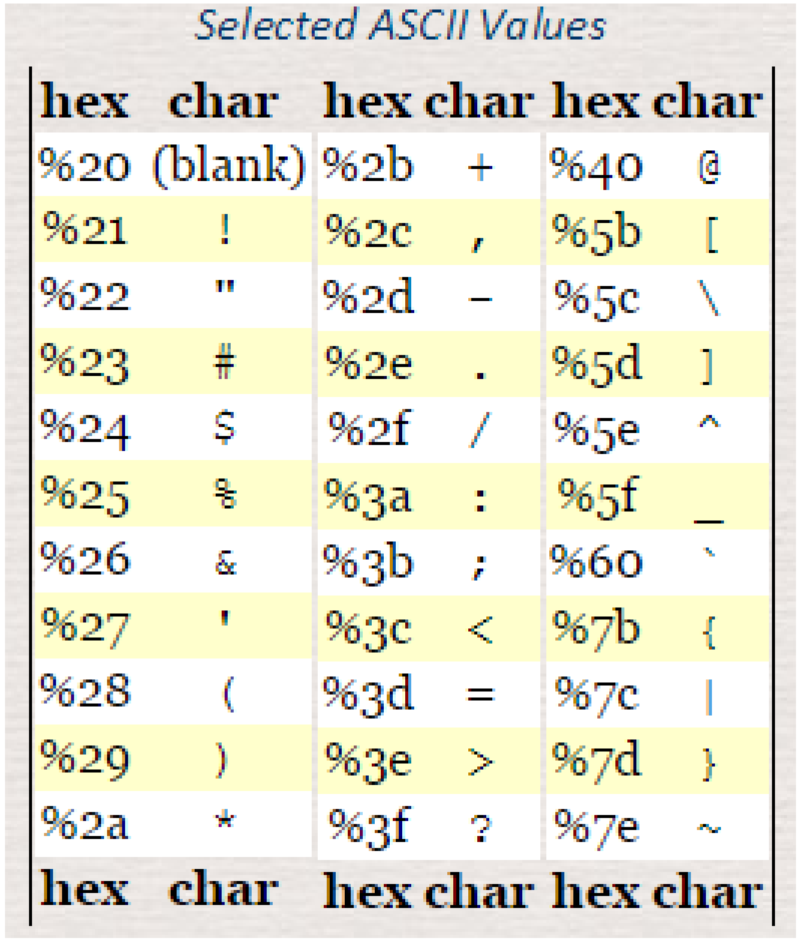
- Test case 1: a=9, b=8, c=7 (To test all the true conditions)
- Test case 2: a=7, b=8, c=9 (To test all the false conditions)

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)

Hex to char

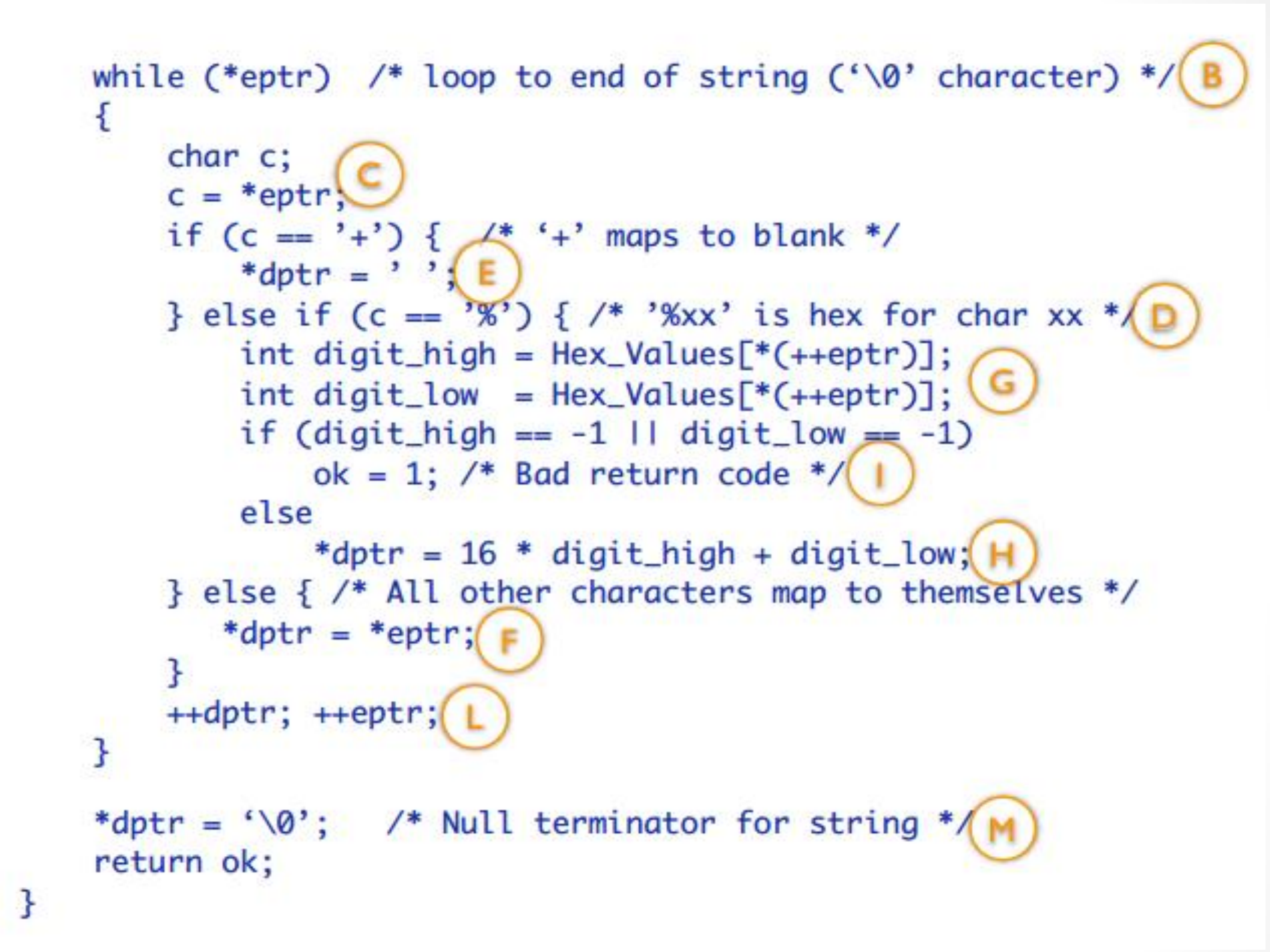


Selected ASCII Values

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

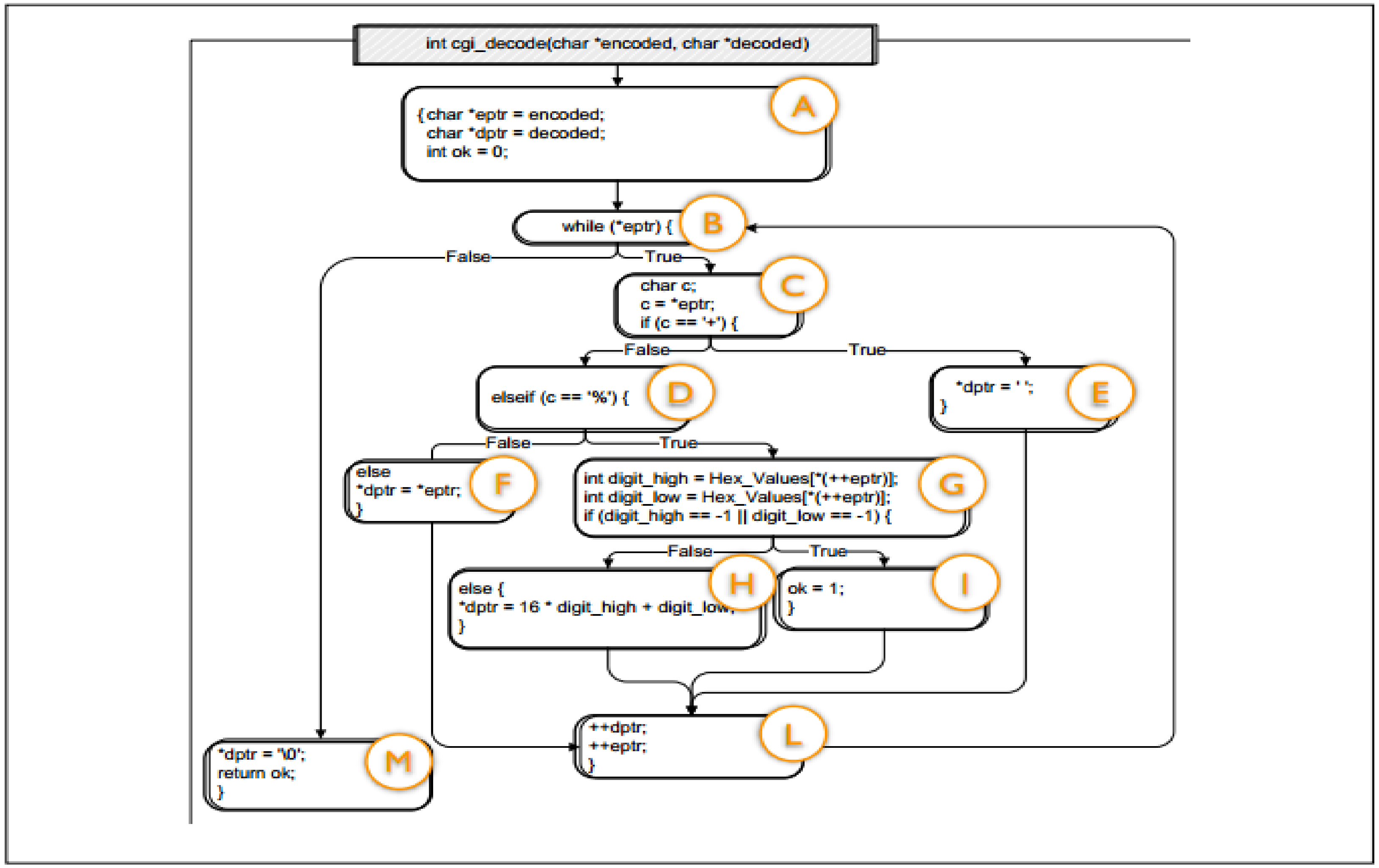
cgi_decode

```
/**
  * @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;
}
```

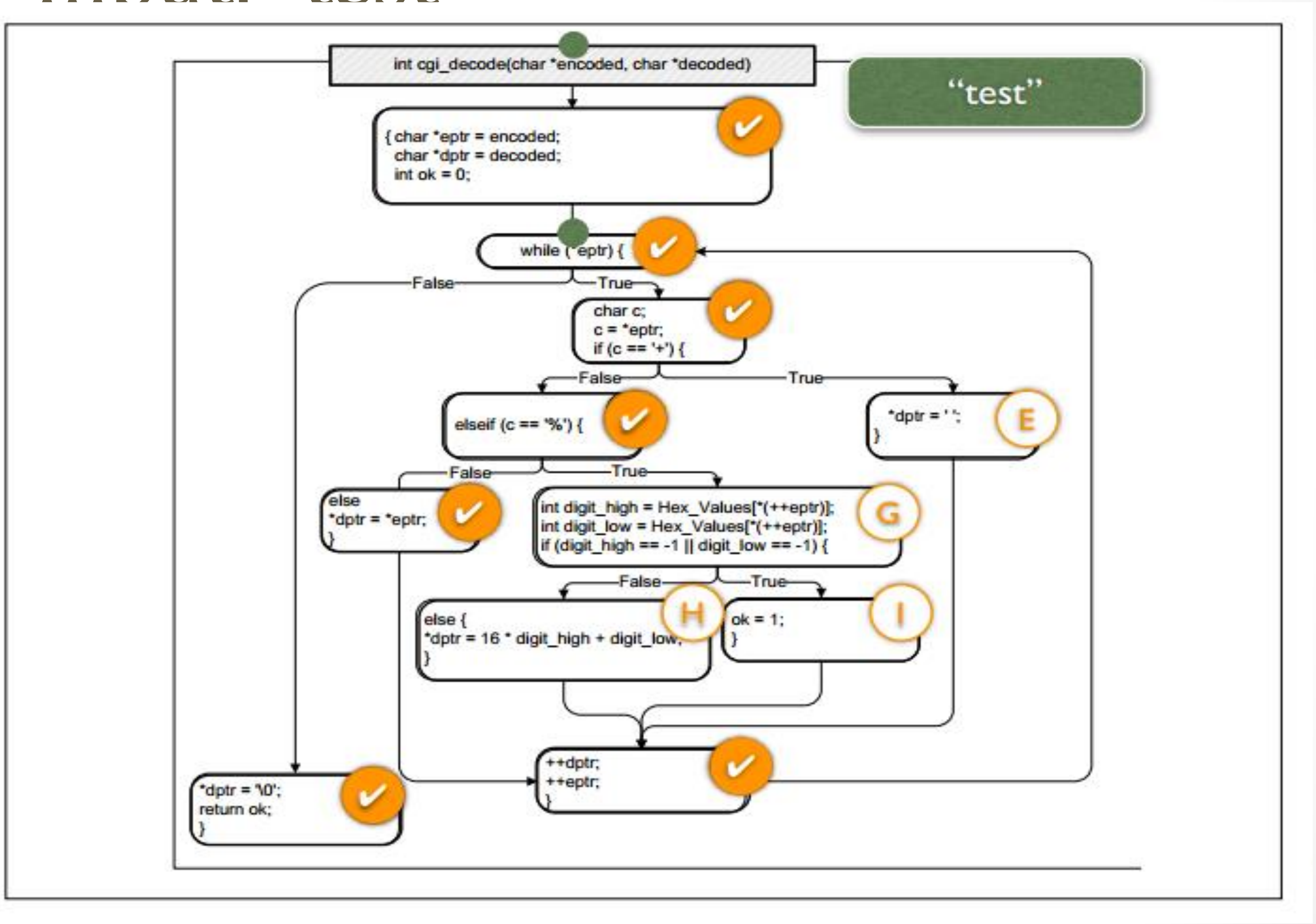
CFG for CGI program



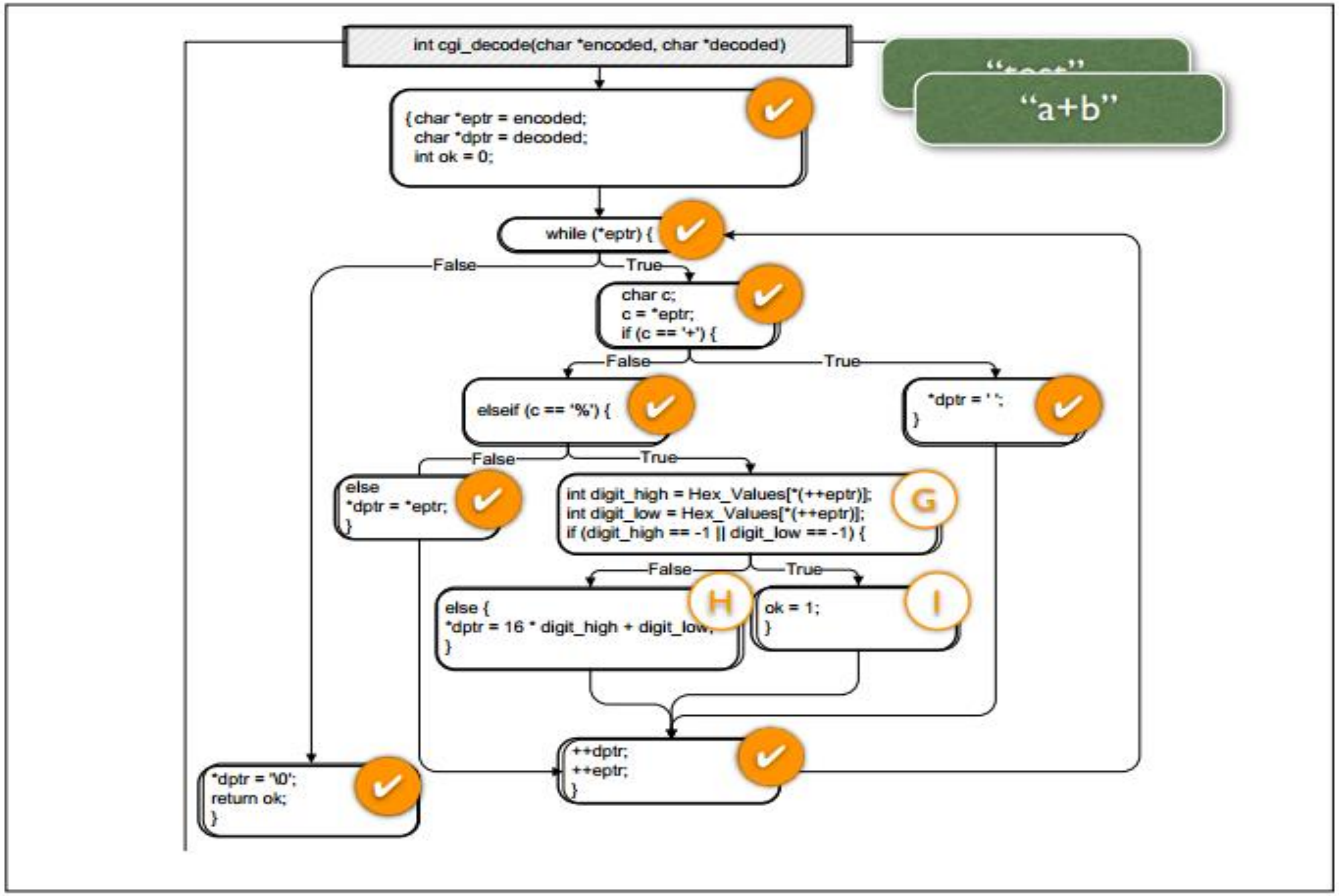
```
int cgi_decode(char *encoded, char *decoded)

{char *eptr = encoded;
 char *dptr = decoded;
 int ok = 0;

while (*eptr) {

char c;
c = *eptr;
if (c == '+') {

elseif (c == '%') {

*dptr = ' ';
}

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

else {
*dptr = 16 * digit_high + digit_low;
}

ok = 1;
}

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}
```

1

Input: “test”

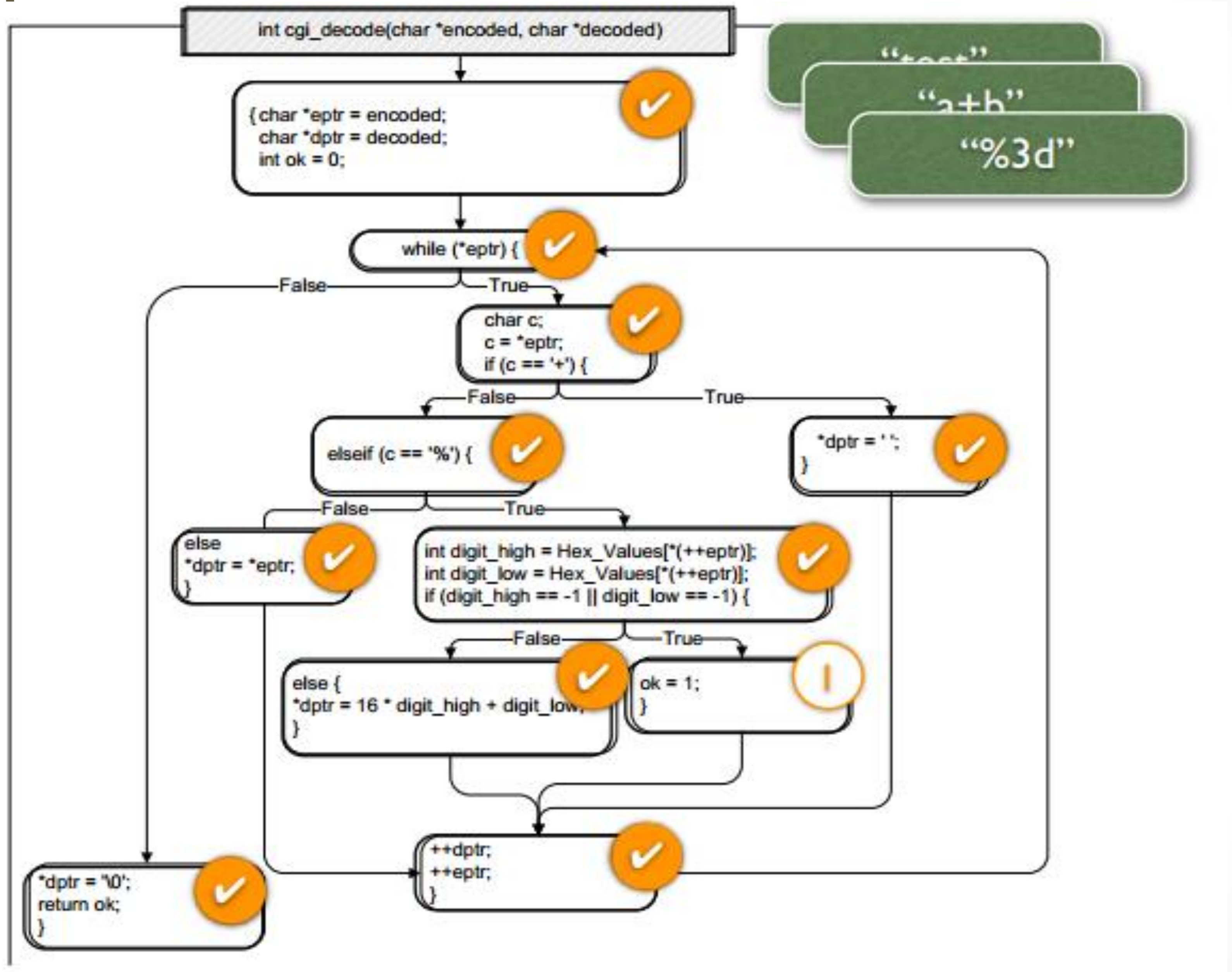

int cgi_decode(char *encoded, char *decoded)
“test”
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = 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"

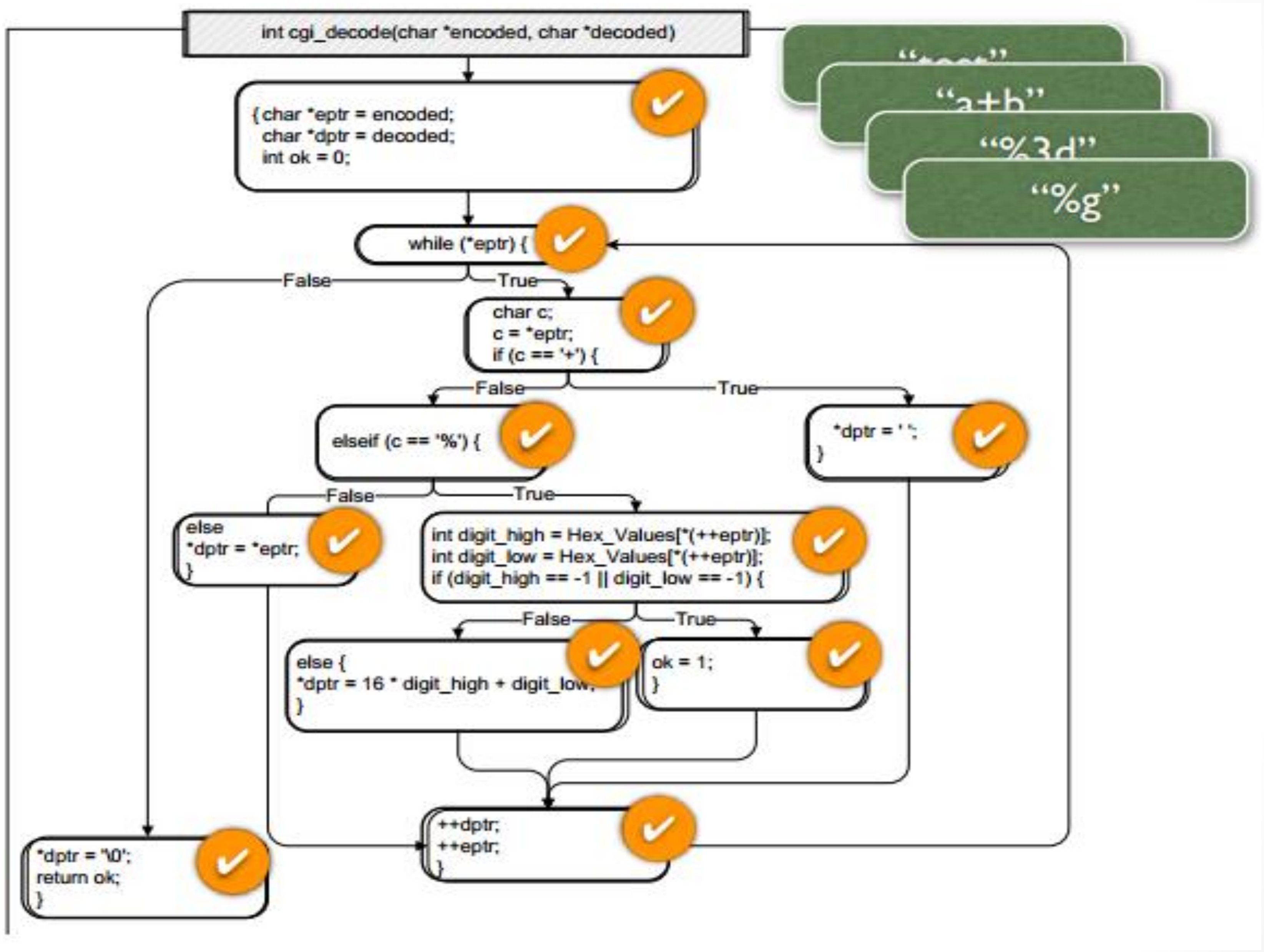


"test"
"a+b"
int cgi_decode(char *encoded, char *decoded)
{char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = 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;
}

Input: “%3d”


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

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;
}
“test”
“a+b”
“%3d”
“%g”