SENG 275

SOFTWARE TESTING

DR. NAVNEET KAUR POPLI

DEPT. OF ELECTRICAL AND COMPUTER ENGINEERING

University of Victoria

BLOCK COVERAGE

University of Victoria

Block coverage

- Program blocks do not depend on how the developer wrote the code.
- Thus, it does not suffer from having different coverage numbers due to different programming styles.

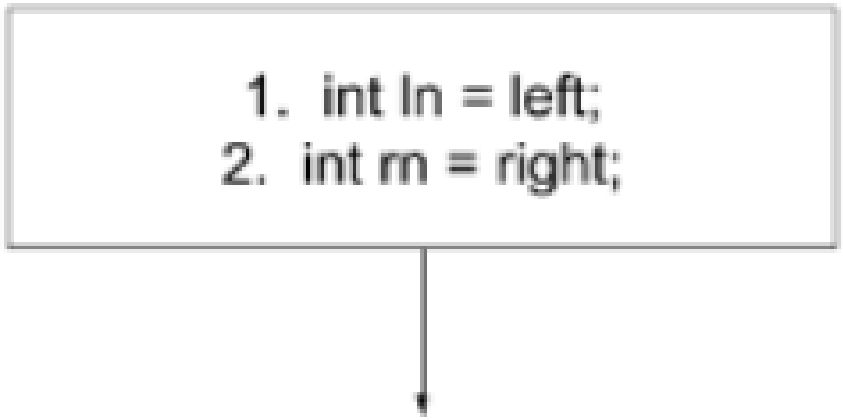
Blocks and Control-Flow Graph

- A **control-flow graph** (or CFG) is a representation of all paths that might be traversed during the execution of a piece of code. It consists of *basic blocks*, *decision blocks*, and *arrows/edges* that connect these blocks.
- Let us use the Black Jack implementation to illustrate the difference between them:

```
public class BlackJack {
  public int play(int left, int right) {
1.  int ln = left;
2.  int rn = right;
3.  if (ln > 21)
4.    ln = 0;
5.  if (rn > 21)
6.    rn = 0;
7.  if (ln > rn)
8.    return ln;
9.  else
10.   return rn;
  }
}
```

Basic Block

- A basic block is composed of **"the maximum number of statements that are executed together no matter what happens"**. In the code above, lines 1-2 are always executed together. Basic blocks are often represented by a square.
- At this moment, our control-flow graph looks like the following:

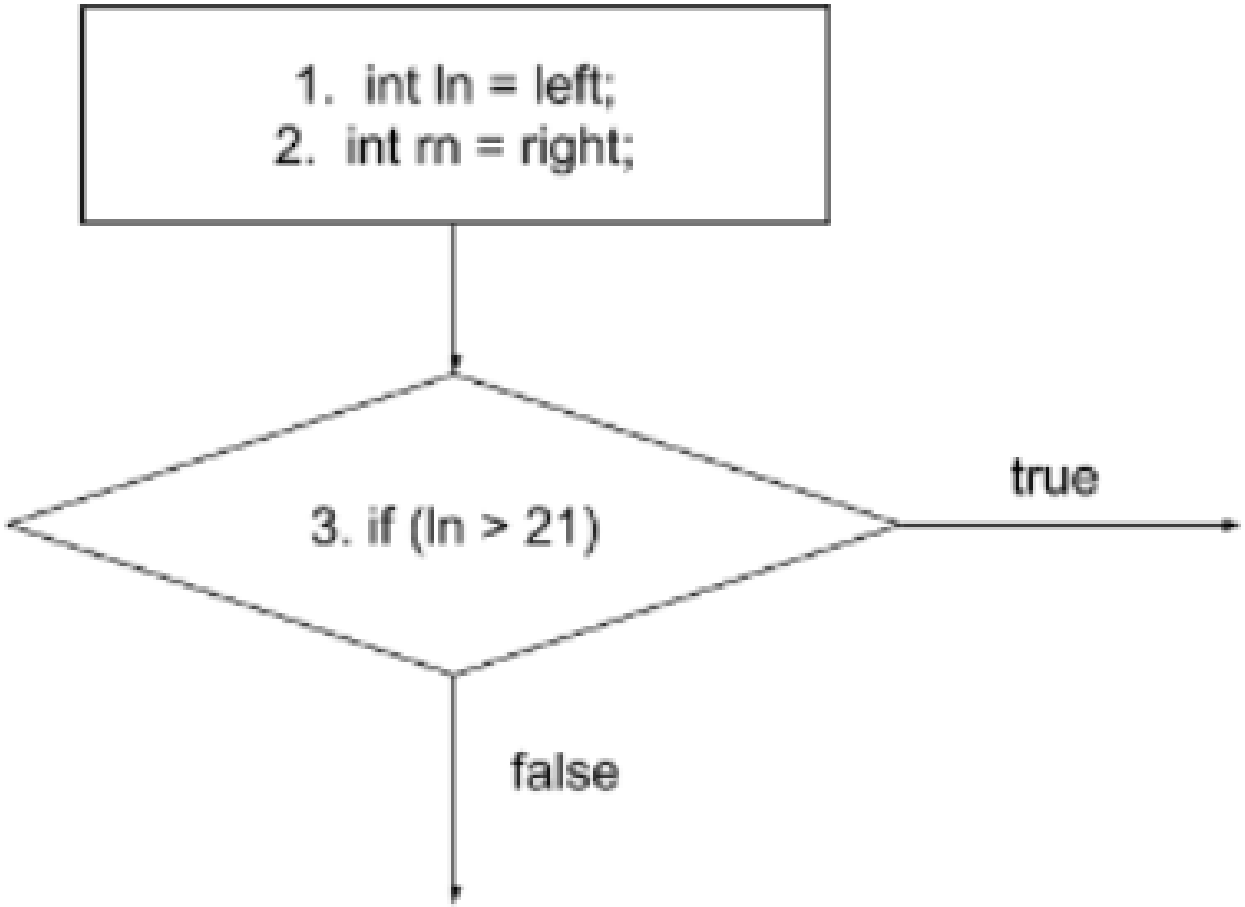


University of Victoria

Decision block

- A decision block, on the other hand, represents all the statements in the source code that **can create different branches**.
- See line 3: if (In > 21). This if statement creates a decision moment in the application: based on the condition, it is decided which code block will be executed next.
- Decision blocks are often represented by diamonds. This decision block happens right after the basic block we created above, and thus; they are connected by means of an edge.

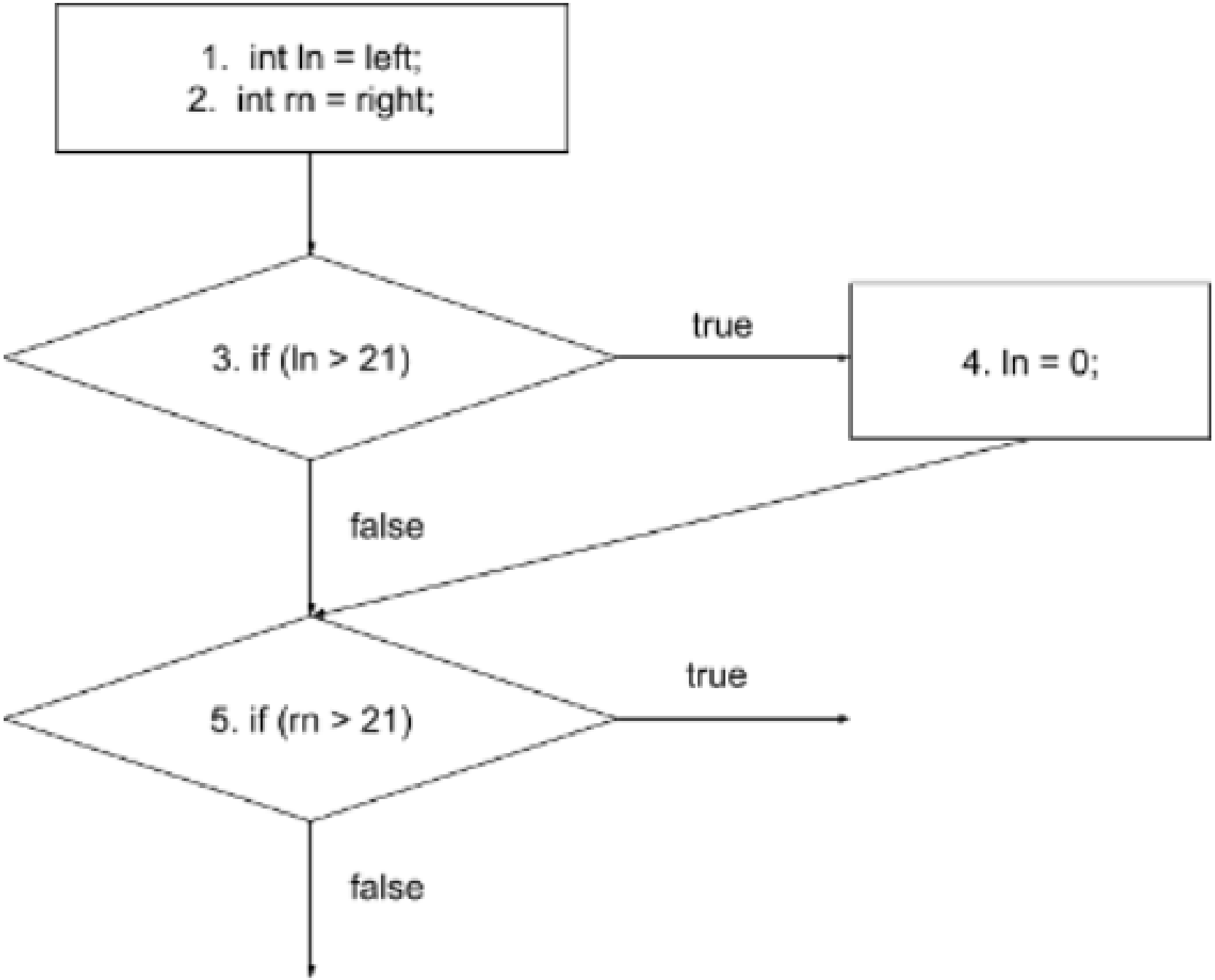
Decision block



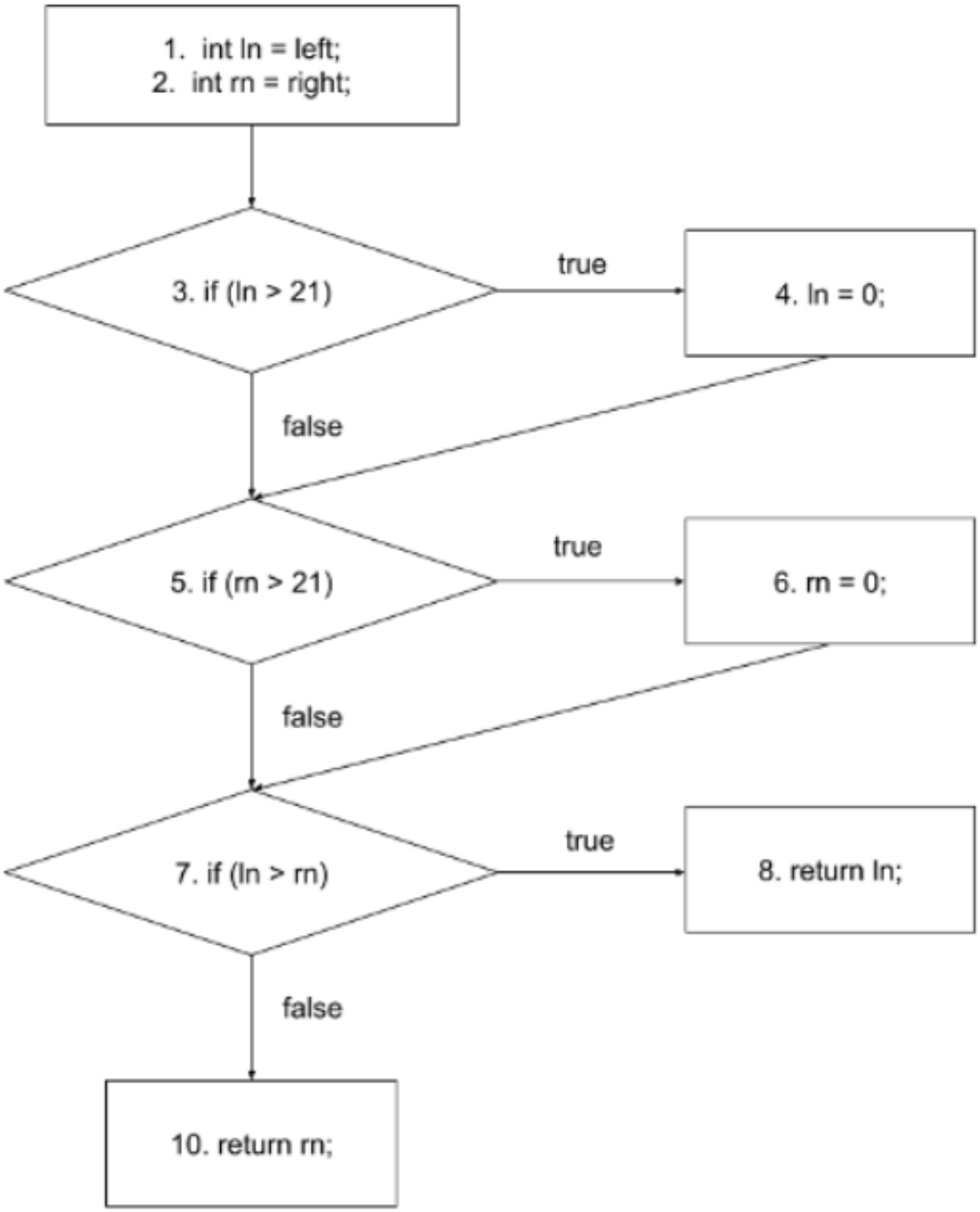
University of Victoria

Decision block

- A basic block has always a single outgoing edge.
- A decision block, on the other hand, always has two outgoing edges (indicating where you go in case of the decision being evaluated to true, and where you go in case the decision is evaluated to false).
- In case of the decision block being evaluated to true, line 4 is executed, and the program continues to line 5. Otherwise, it proceeds straight to line 5, which is another decision block:


1. int ln = left;
2. int rn = right;
3. if (ln > 21)
true
4. ln = 0;
false
5. if (rn > 21)
true
false

When you repeat the approach up to the end of the program, you end up with the following CFG:




```
public class BlackJack {
  public int play(int left, int right) {
1.  int ln = left;
2.  int rn = right;
3.  if (ln > 21)
4.    ln = 0;
5.  if (rn > 21)
6.    rn = 0;
7.  if (ln > rn)
8.    return ln;
9.  else
10.   return rn;
  }
}
```

University of Victoria

Block coverage

- We can use the control-flow graph to derive tests. A first idea would be to use blocks as a coverage criterion, in the same way we did with lines, but instead of aiming at covering 100% of the lines, we aim at covering 100% of the blocks.
- The formula that measures block coverage is similar to the line coverage formula:

$$\text{block coverage} = \frac{\text{blocks covered}}{\text{blocks total}} \cdot 100$$

- Note that blocks do not depend on how the developer wrote the code. Thus, it does not suffer from having different coverage numbers due to different programming styles.

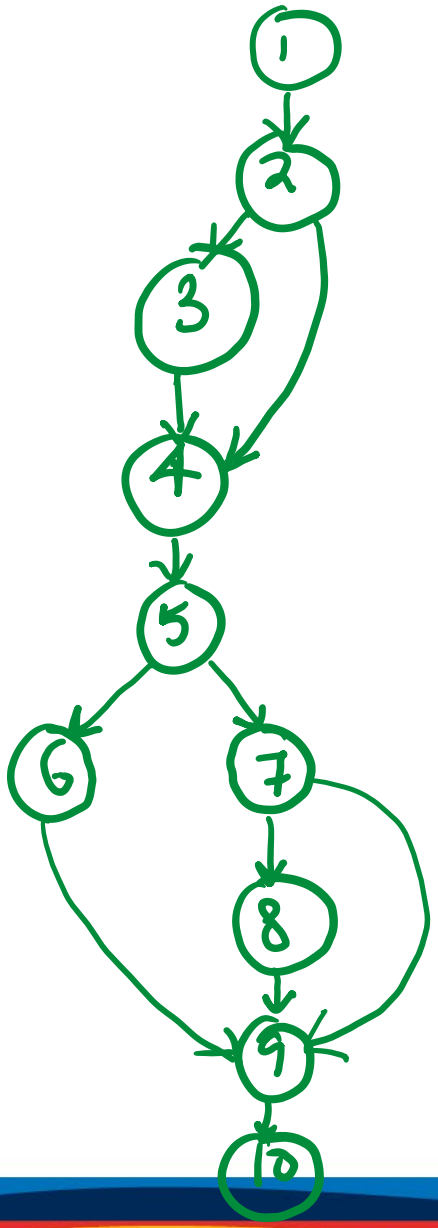
University of Victoria

Create a CFG for this pseudocode:

```
func Control(
  if (up clicked)
        Y-=Z
        DetectCollision
  endif
  if (left down)
        X-=Z
        DetectCollision
  else if (right down)
        X+=Z
        DetectCollision
  endif
End func
```

Create a CFG for this pseudocode:

```
1  func Control(
2    if (up clicked)
3    {   Y-=Z
        DetectCollision
4    endif
5    if (left down)
6    {   X-=Z
        DetectCollision
7    else if (right down)
8    {   X+=Z
        DetectCollision
9    endif
10 End func
```



University of Victoria

```
public class CountWords {
1 public int count(String str) {
2   int words = 0;
3   char last = ' ';

4   for (int i = 0; i < str.length(); i++) {

5     if (!isLetter(str.charAt(i)) &&
6      (last == 's' || last == 'r')) {
7       words++;
      }

8     last = str.charAt(i);
    }

9   if (last == 'r' || last == 's') {
10    words++;
    }

11  return words;
  }
}
```

Loops through each character in the string

If the current character is a non-letter and the previous character was "s" or "r", we have a word!

Stores the current character as the "last" one

Counts one more word if the string ends in "r" or "s"

index	last	w
0	d	0
1	o	0
2	g	0
3	s	0
4	' '	1
5	c	0
6	a	0
7	t	0
8	s	0

2