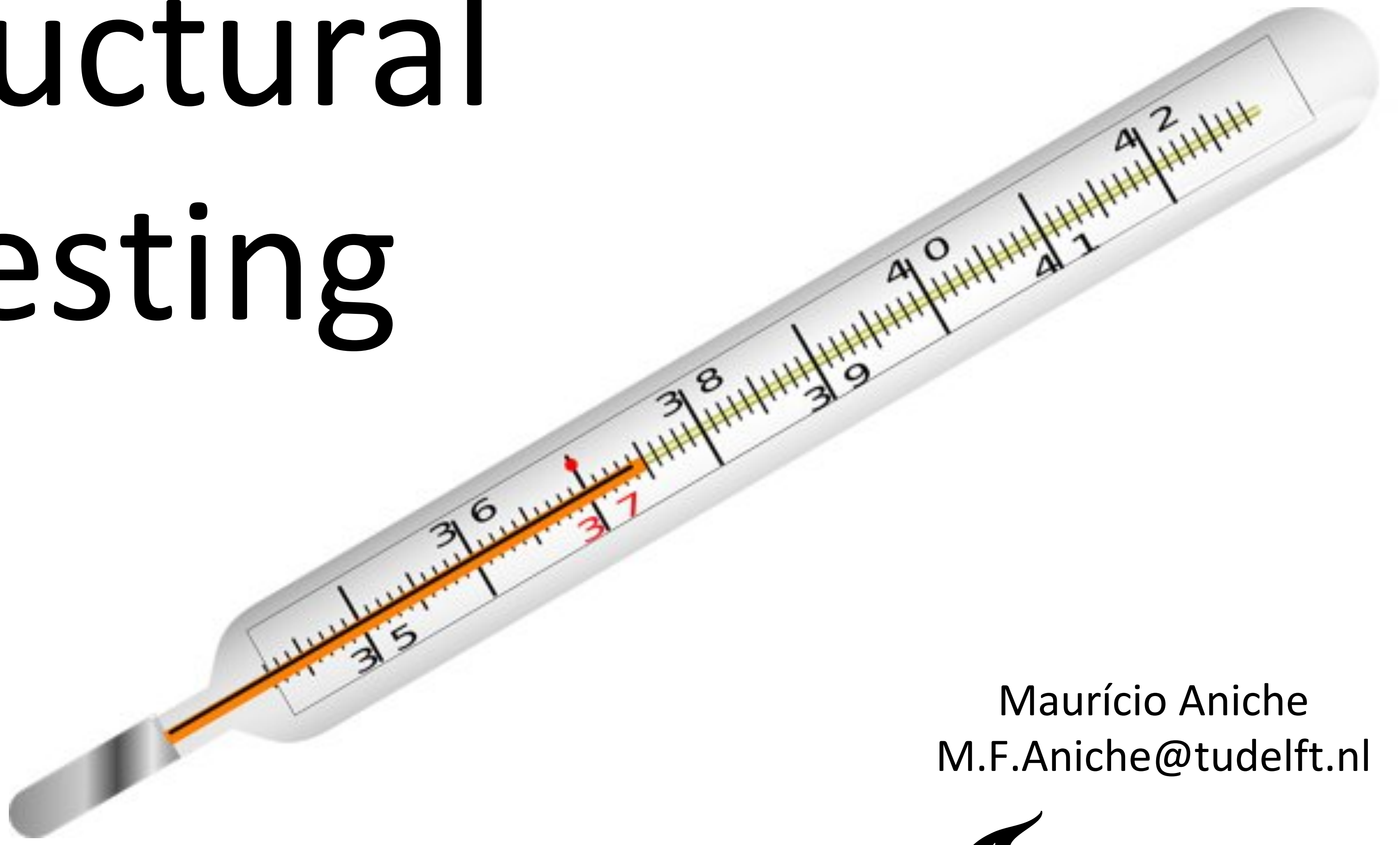
Structural Testing



Maurício Aniche
M.F.Aniche@tudelft.nl

TU Delft

SPECIFICATION

Requirements
Models

STRUCTURAL

Structure
(e.g., source code)

SPECIFICATION

Requirements
Models

STRUCTURAL

Structure
(e.g., source code)

```java
public int play(int left,
 int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return rn;
    else
        return ln;
}
```

Given the points of two different players, the program must return the number of points the one who wins has!

```java
public int play(int left,
 int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return rn;
    else
        return ln;
}
```

What would you test?

(now, only looking to
the source code)

```
public int play(int left,
 int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return rn;
    else
        return ln;
}
```

First idea: **“going through all the lines”**

If our test suite exercises all the lines, we are happy.

```java
public int play(int left,
 int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return rn;
    else
        return ln;
}
```

First idea: **"going through all the lines"**

If our test suite exercises all the lines, we are happy.

T1 = (30, 30)

How many lines does it cover?

```
public int play(int left,
 int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return rn;
    else
        return ln;
}
```

First idea: **“going through all the lines”**

If our test suite exercises all the lines, we are happy.

T1 = (30, 30)

```java
public int play(int left,
 int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return rn;
    else
        return ln;
}
```

First idea: **"going through all the lines"**

If our test suite exercises all the lines, we are happy.

T1 = (30, 30)

9 / 10 = 90% line coverage

```
public int play(int left,
 int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return rn;
    else
        return ln;
}
```

Make it true

First criteria: **“going through all the lines”**

If our test suite exercises all the lines, we are happy.

T1 = (30, 30)

T2 = (10,9) <-- left player wins

```java
public int play(int left,
 int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return rn;
    else
        return ln;
}
```

First criteria: **“going through all the lines”**

If our test suite exercises all the lines, we are happy.

T1 = (30, 30)

T2 = (10,9) <-- left player wins

10 / 10 = 100% line coverage

```
public int play(int left,
 int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return rn;
    else
        return ln;
}
```

Is this useful?

Yes, it is. We actually **just found a bug**!

```
public int play(int left,
 int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return rn;
    else
        return ln;
}
```

Is this useful?

Yes, it is. We actually **just found a bug**!

```java
public int play(int left,
 int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return ln;
    else
        return rn;
}
```

Is this useful?

Yes, it is. We actually **just found a bug**!


Great! We found a bug after some **structural testing**!

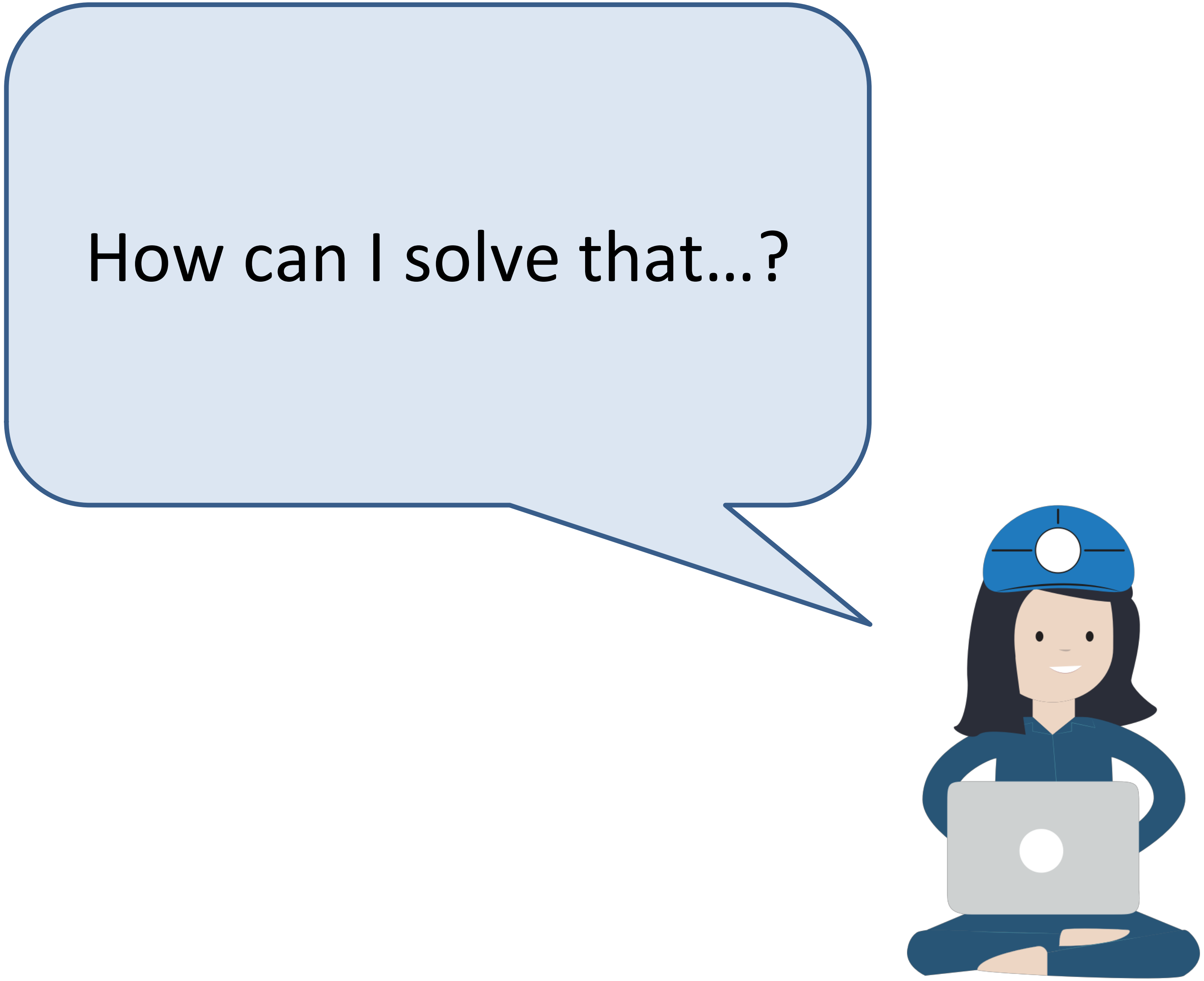
10 lines!

```
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;
}
```

6 lines!

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

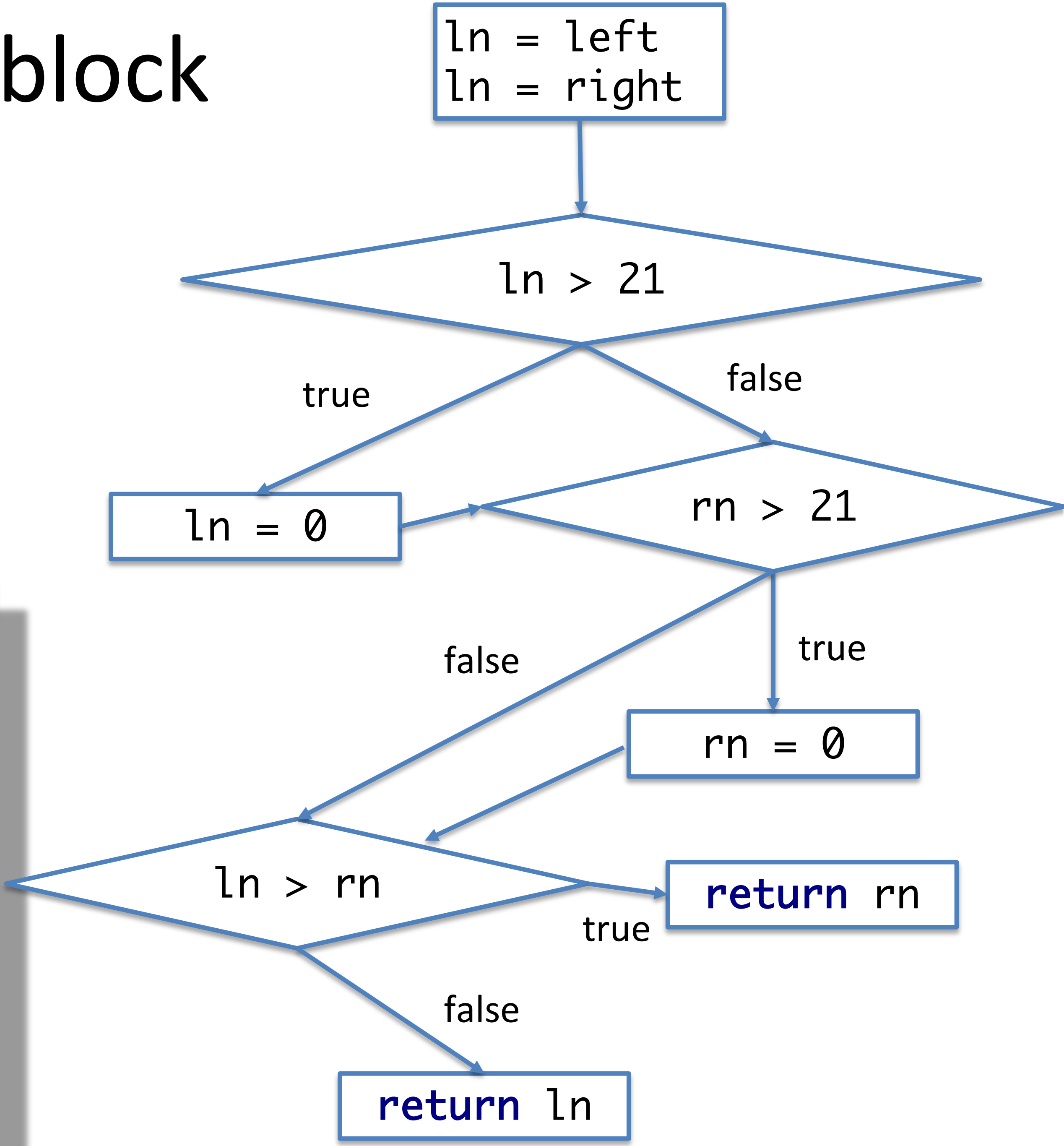
0 = 9
5/ 8 ...
From no ll write as
many nes a can!!


How can I solve that...?

Basic block

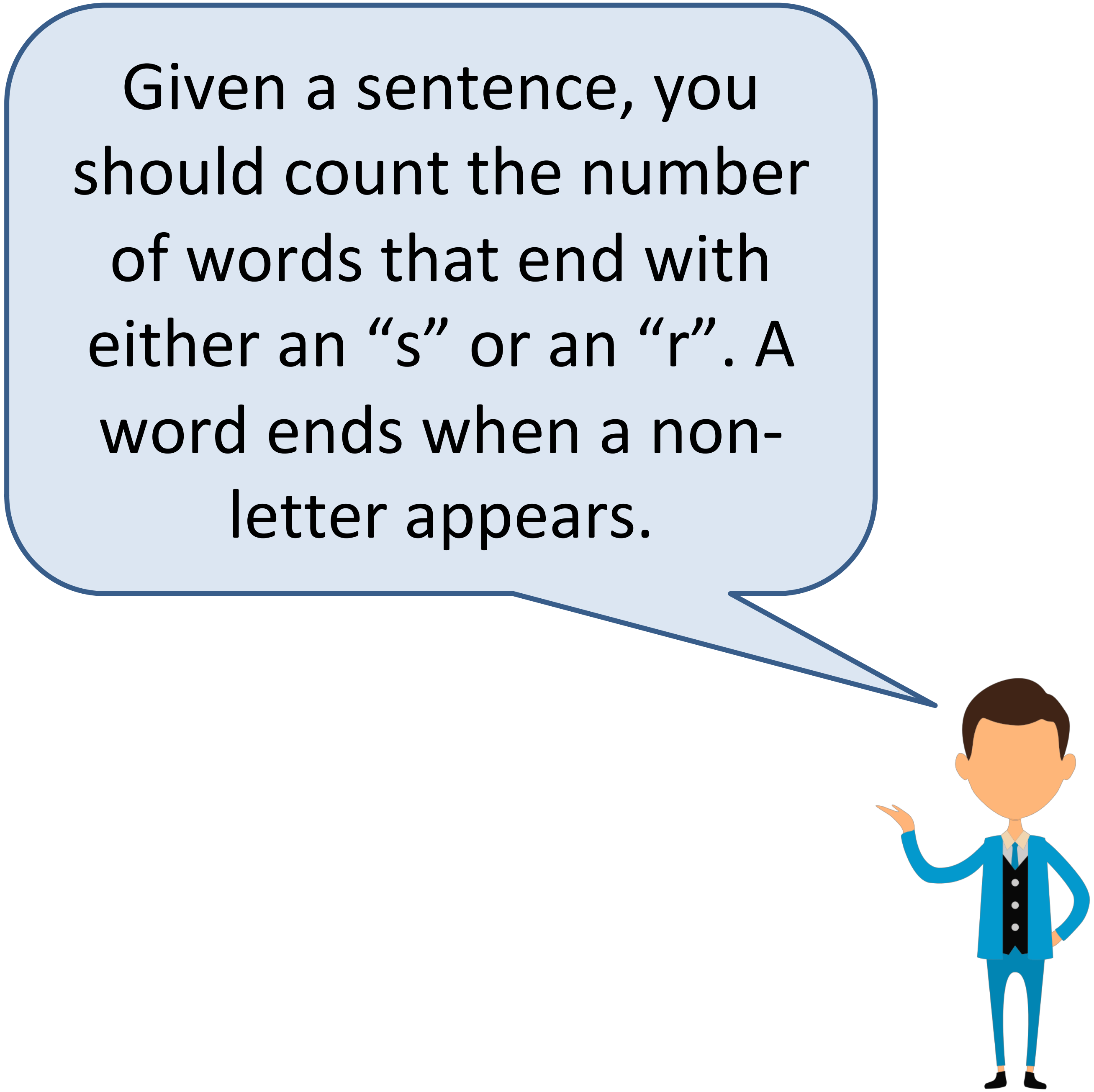
- A basic block is a straight-line code sequence with no branches.
- In other words, whenever you have a decision point, you start a new block.

```
int play(int left, int right) {
    int ln = left;
    int rn = right;
    if (ln > 21)
        ln = 0;
    if (rn > 21)
        rn = 0;
    if (ln > rn)
        return rn;
    else
        return ln; }
```



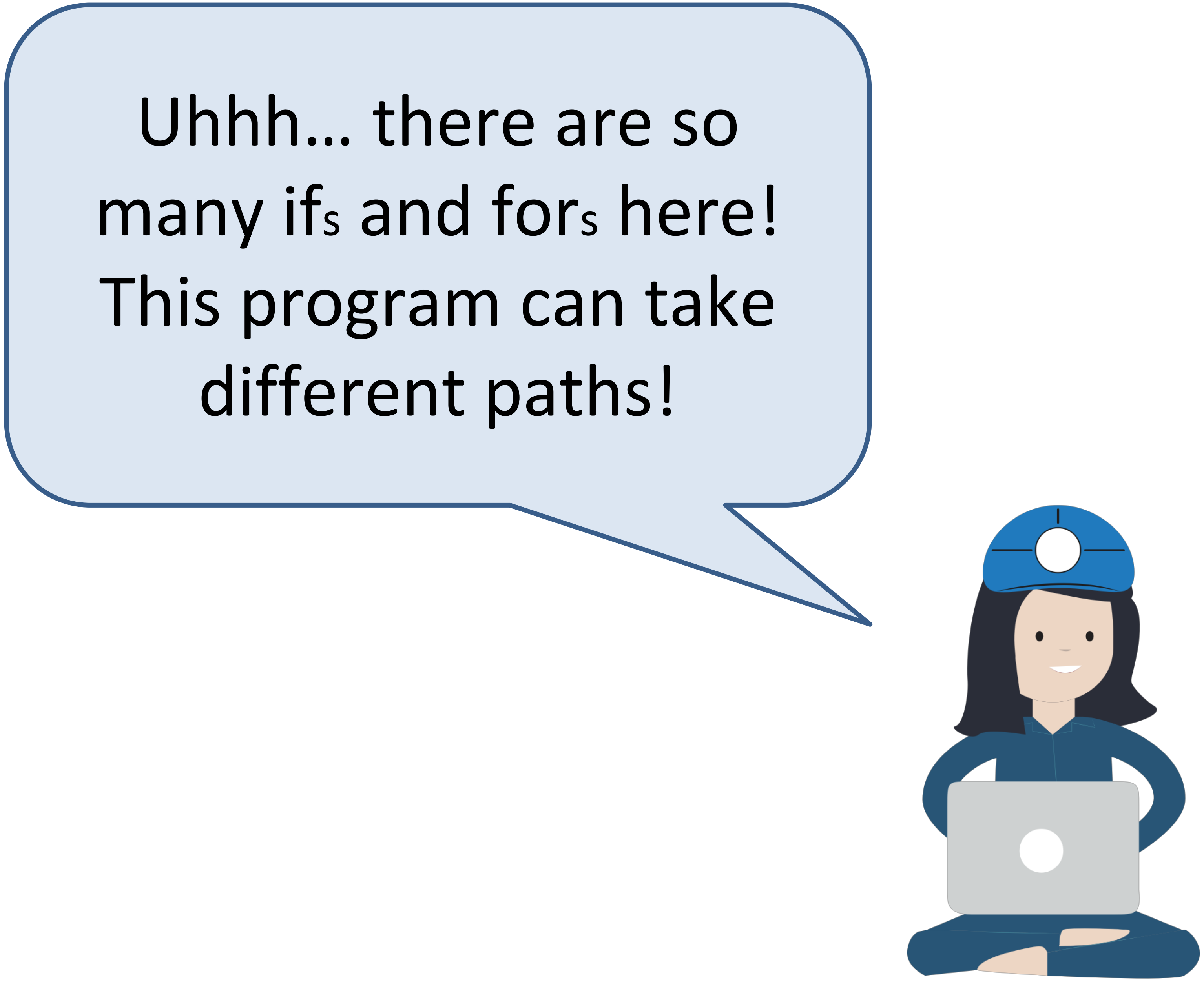
What’s the difference between **line** and **statement** coverage?

- Line coverage looks at the lines of your program (as in the source code).
- A line can contain more than one statement:
  - E.g., “a = 10; b=20;”


Given a sentence, you should count the number of words that end with either an “s” or an “r”. A word ends when a non-letter appears.

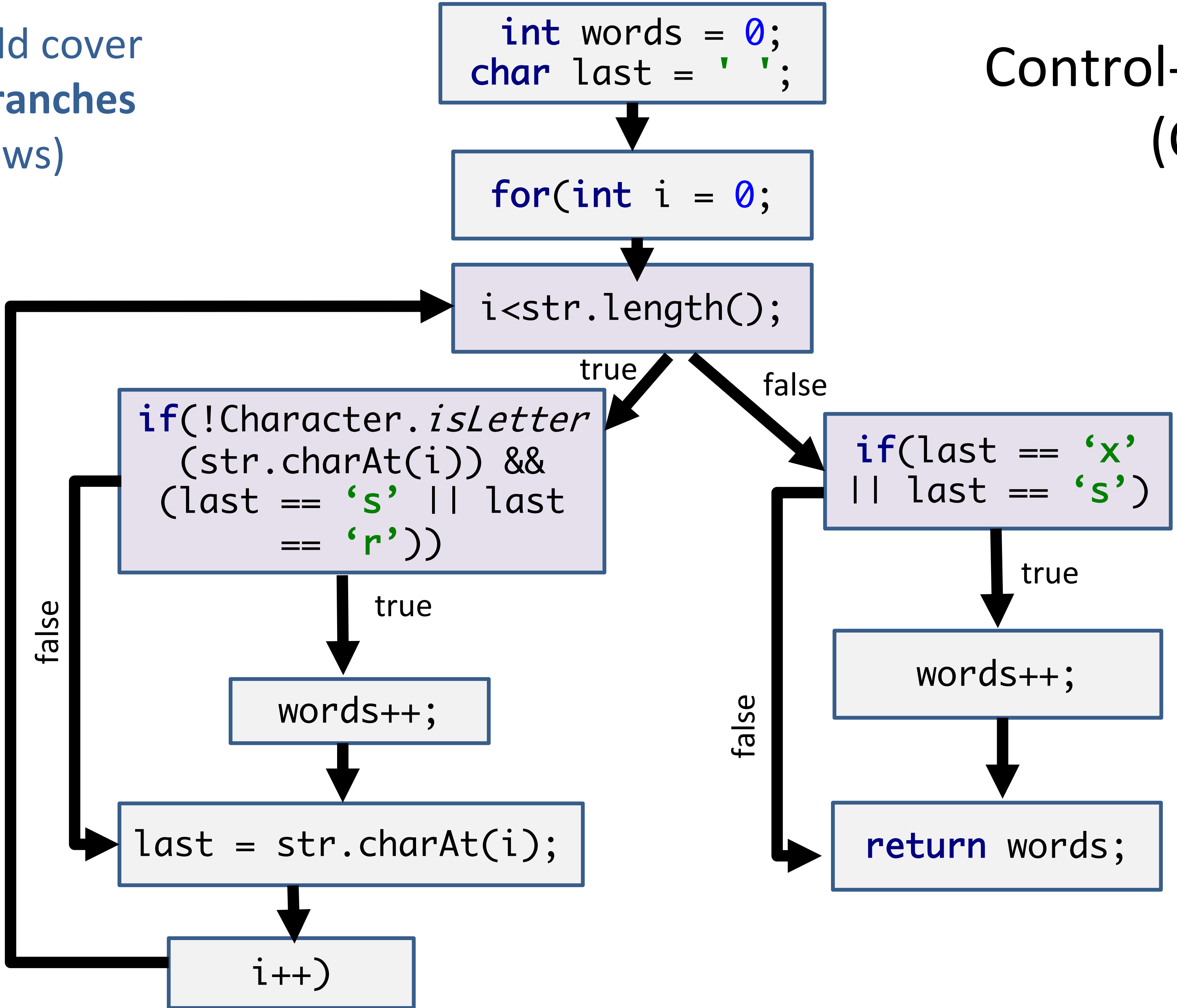
```java
public int count(String str) {
  int words = 0; char last = ' ';
  for(int i = 0;i<str.length(); i++) {
 if(!Character.isLetter(str.charAt(i))
  && (last == 'r' || last == 's’)) {
      words++;
    }
    last = str.charAt(i);
  }
  if(last == 'x' || last == 's’)
    words++;
  return words;
}
```

What’s the difference between this program and the other one (when it comes to testing)?

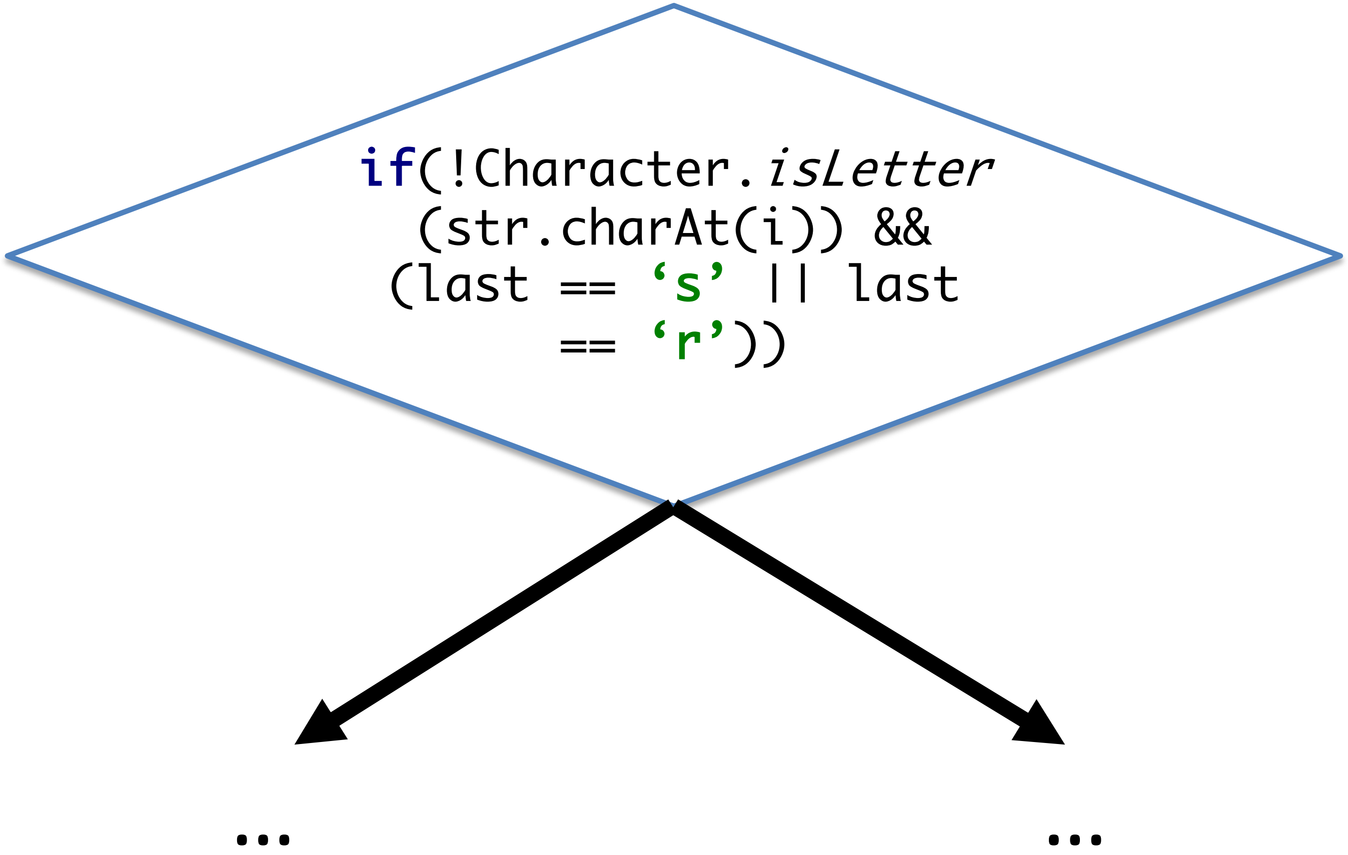

Uhhh… there are so many ifs and fors here! This program can take different paths!

We should cover all the **branches** (arrows)

Control-flow graph (CFG)



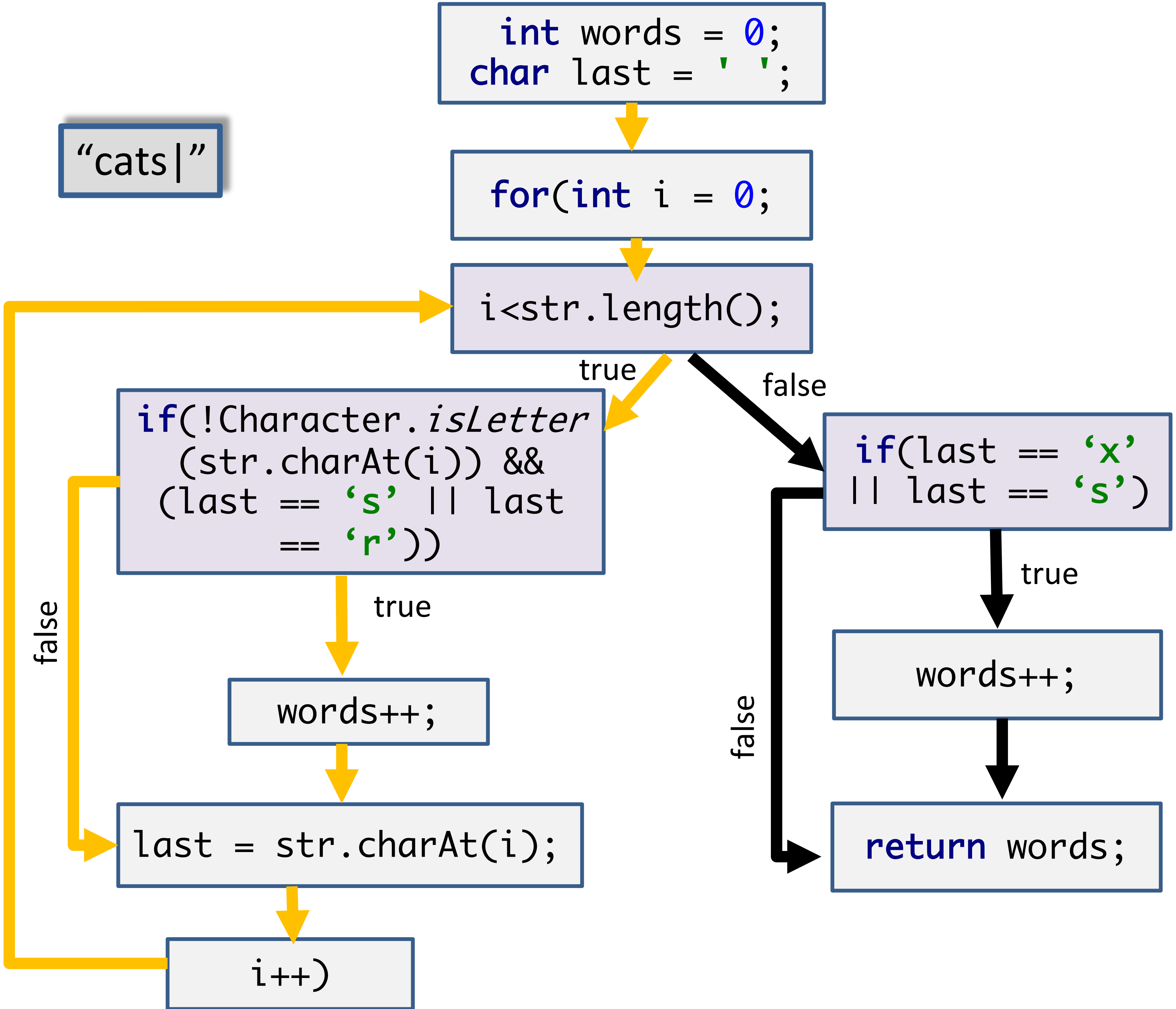
Note on notation



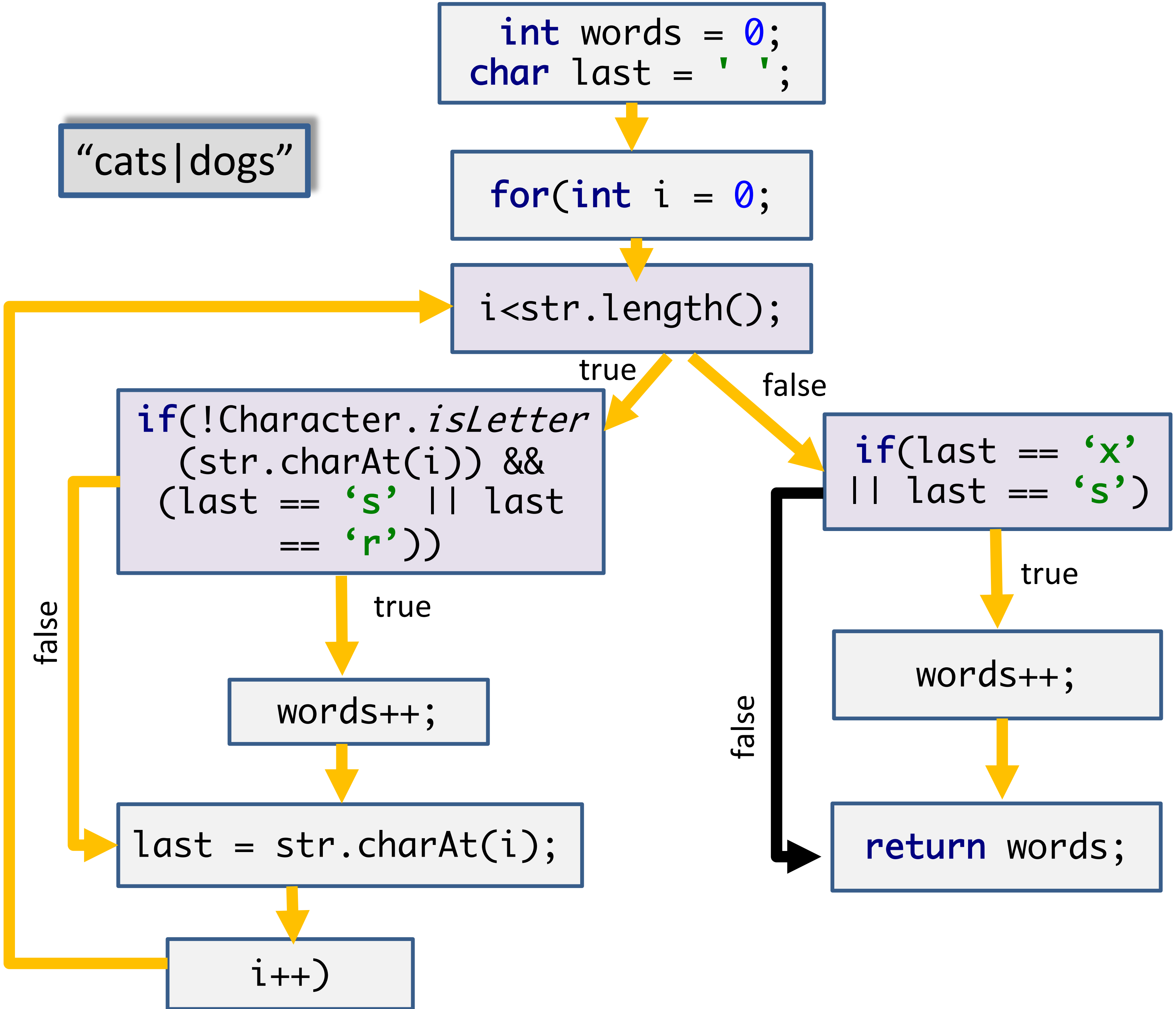
Decision blocks are often represented with **diamonds**.

(In here, I do not use it, because they get too big and don’t fit an slide…)

```java
@Test
public void multipleMatchingWords() {

  int words = new CountLetters()
       .count("cats|dogs");

  Assertions.assertEquals(2, words);
}
```

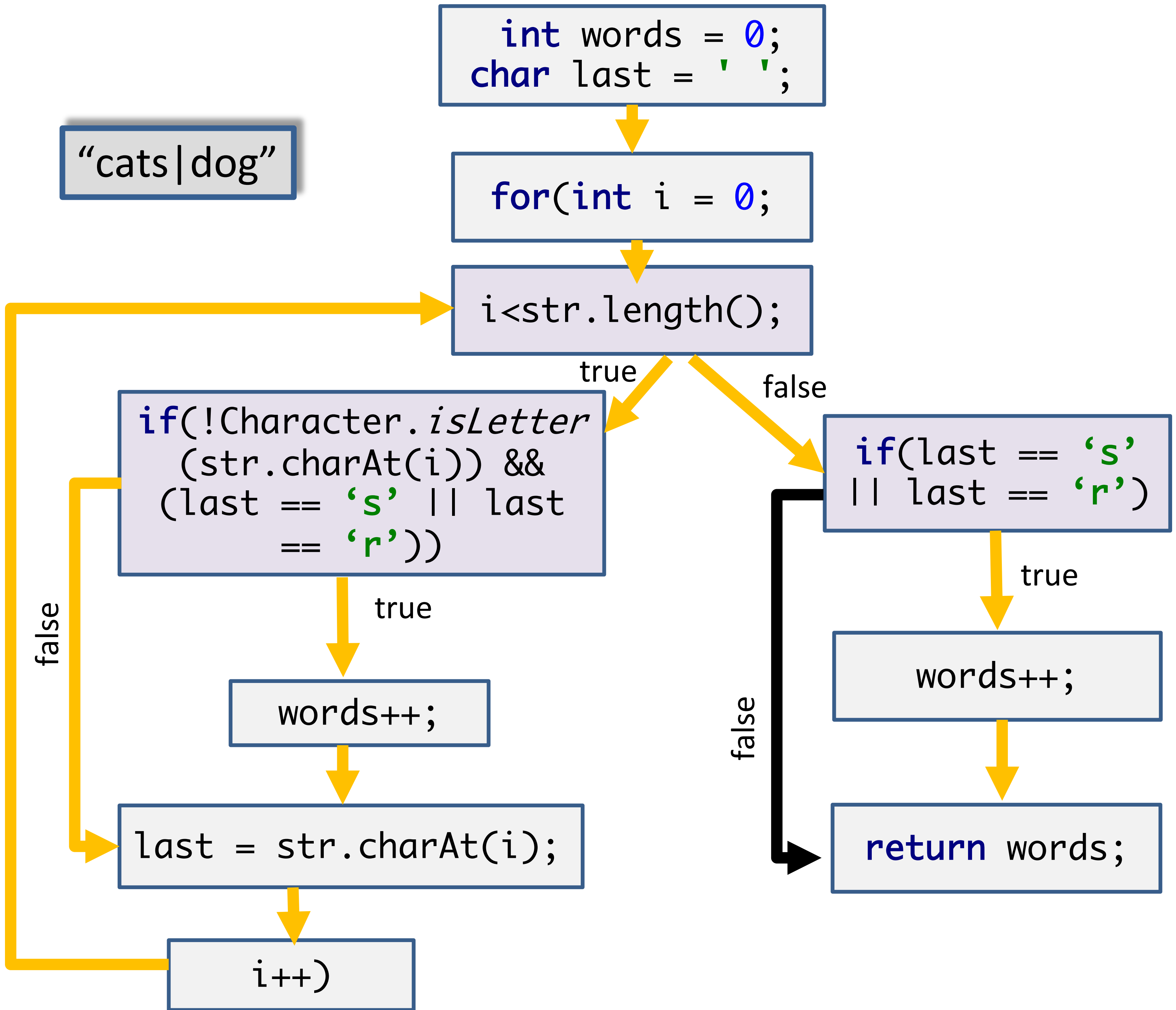


"cats|"
int words = 0;
char last = ' ';
for(int i = 0;
i<str.length();
true
false
if(!Character.isLetter(str.charAt(i)) && (last == 's' || last == 'r'))
true
false
words++;
last = str.charAt(i);
i++)
if(last == 'x' || last == 's')
true
false
words++;
return words;

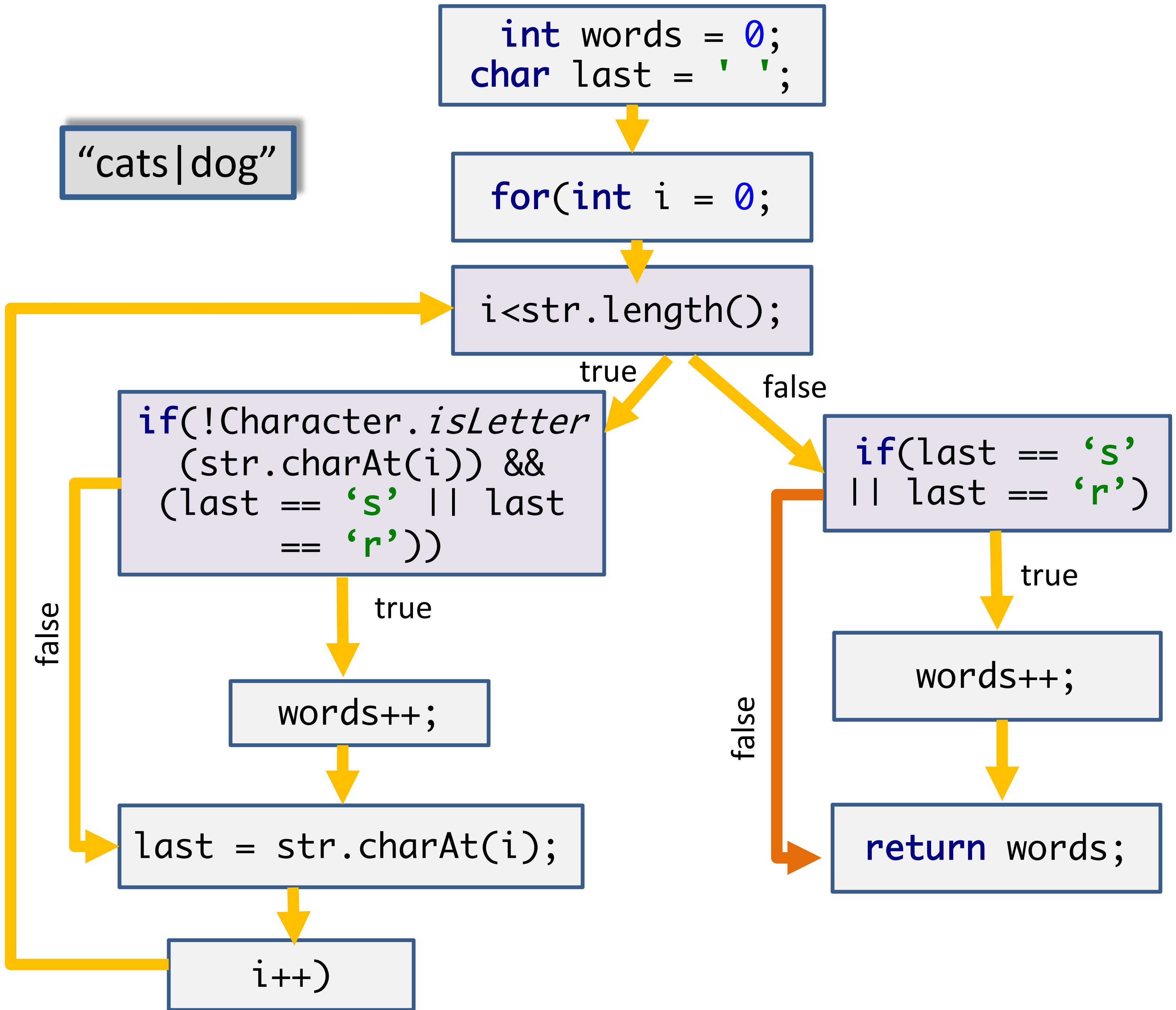


int words = 0;
char last = ' ';
"cats|dogs"
for(int i = 0;
i<str.length();
true
false
if(!Character.isLetter(str.charAt(i)) && (last == 's' || last == 'r'))
true
false
words++;
last = str.charAt(i);
i++)
if(last == 'x' || last == 's')
true
false
words++;
return words;

```java
@Test
public void lastWordDoesntMatch() {

    int words = new CountLetters()
            .count("cats|dog");

    Assertions.assertEquals(1, words);
}
```



int words = 0;
char last = ' ';
"cats|dog"
for(int i = 0;
i<str.length();
true
false
if(!Character.isLetter (str.charAt(i)) && (last == 's' || last == 'r'))
true
false
words++;
last = str.charAt(i);
i++)
if(last == 's' || last == 'r')
true
false
words++;
return words;

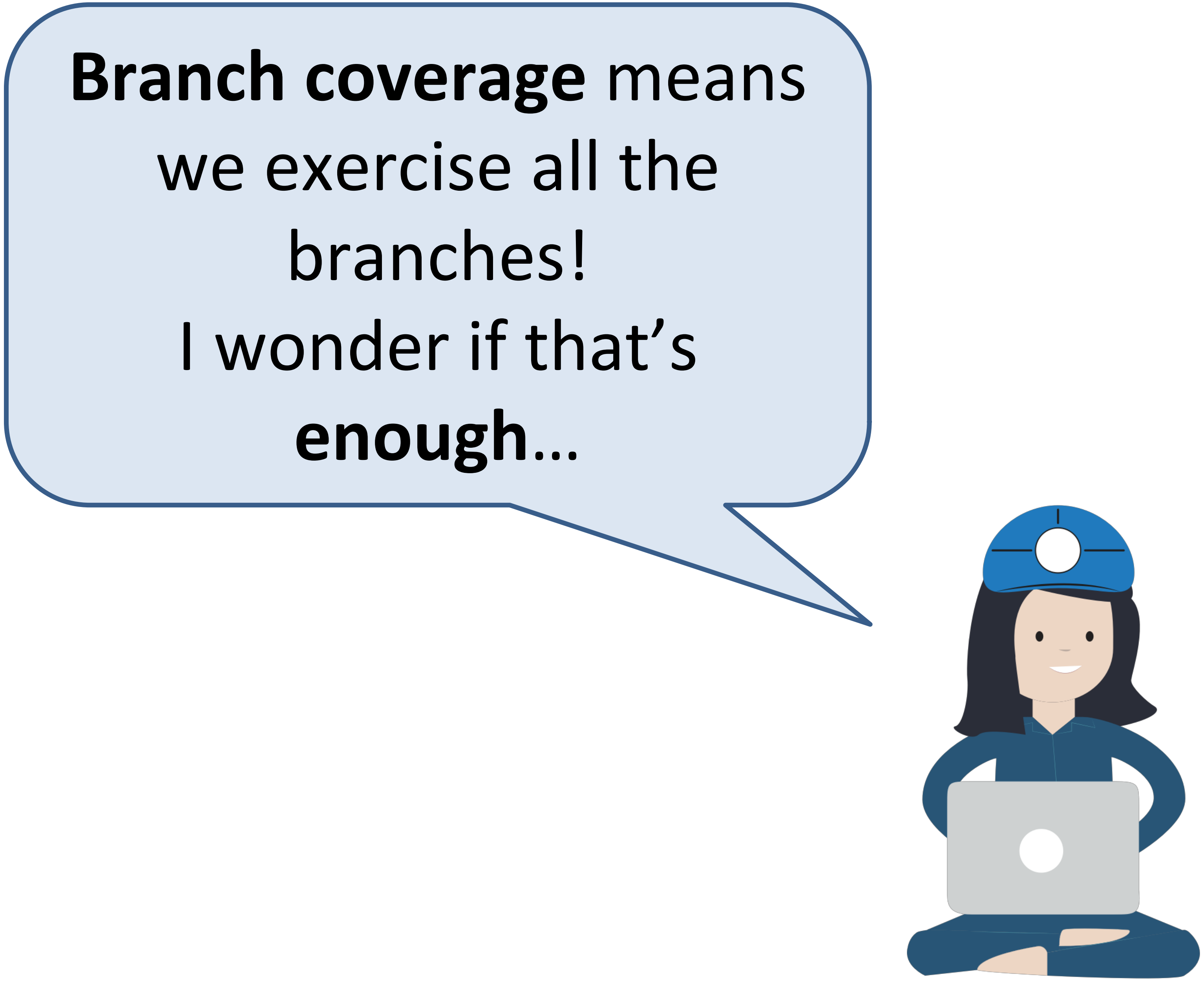


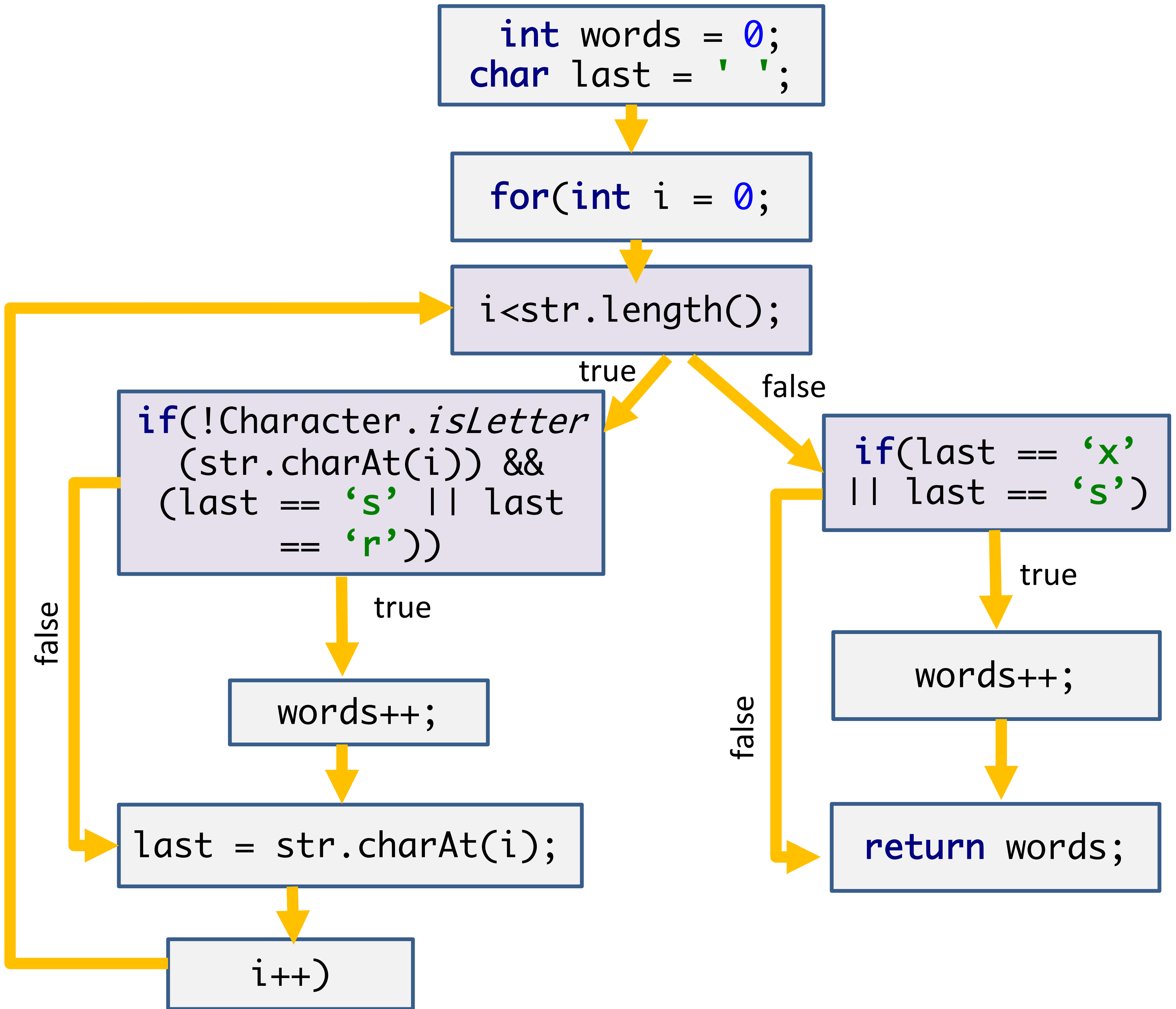
"cats|dog"
int words = 0;
char last = ' ';
for(int i = 0;
i<str.length();
true
false
if(!Character.isLetter (str.charAt(i)) && (last == 's' || last == 'r'))
true
false
words++;
last = str.charAt(i);
i++)
if(last == 's' || last == 'r')
true
false
words++;
return words;

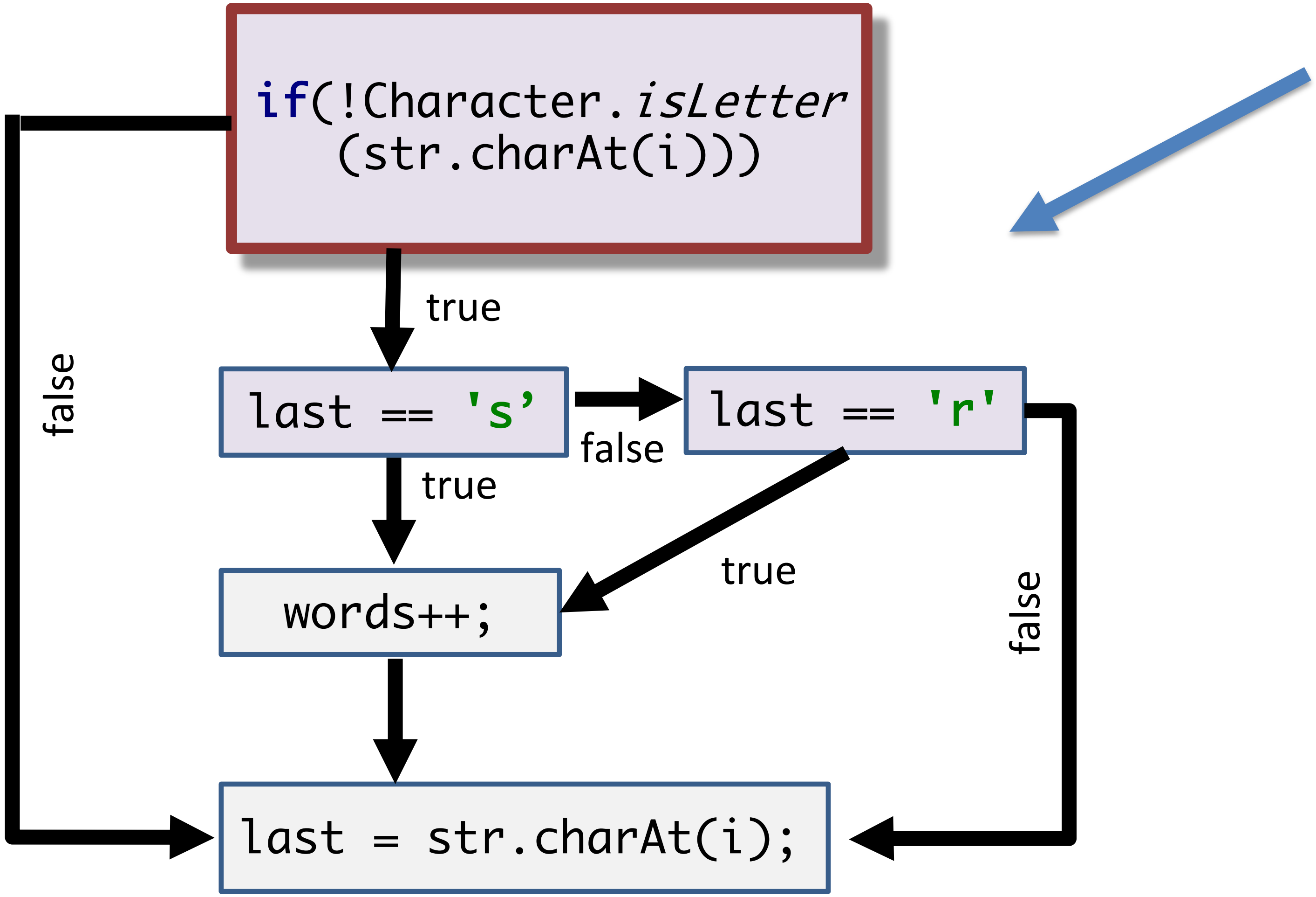
Calculating decision (branch) coverage

- $Branch\ coverage = 100\% \times \frac{Number\ of\ decision\ outcomes\ exercised}{Total\ number\ of\ decision\ outcomes}$

- Each decision (“if”) has two outcomes (true and false).
- In the prior example, there were a total of 6 decisions outcomes.
  - i<str.length();
  - if(!Character.isLetter(str.charAt(i)) && (last == ‘s’ || last == ‘r’))
  - if(last == 's' || last == 'r’)
- Thus, branch coverage: *decision outcomes exercised / 6*

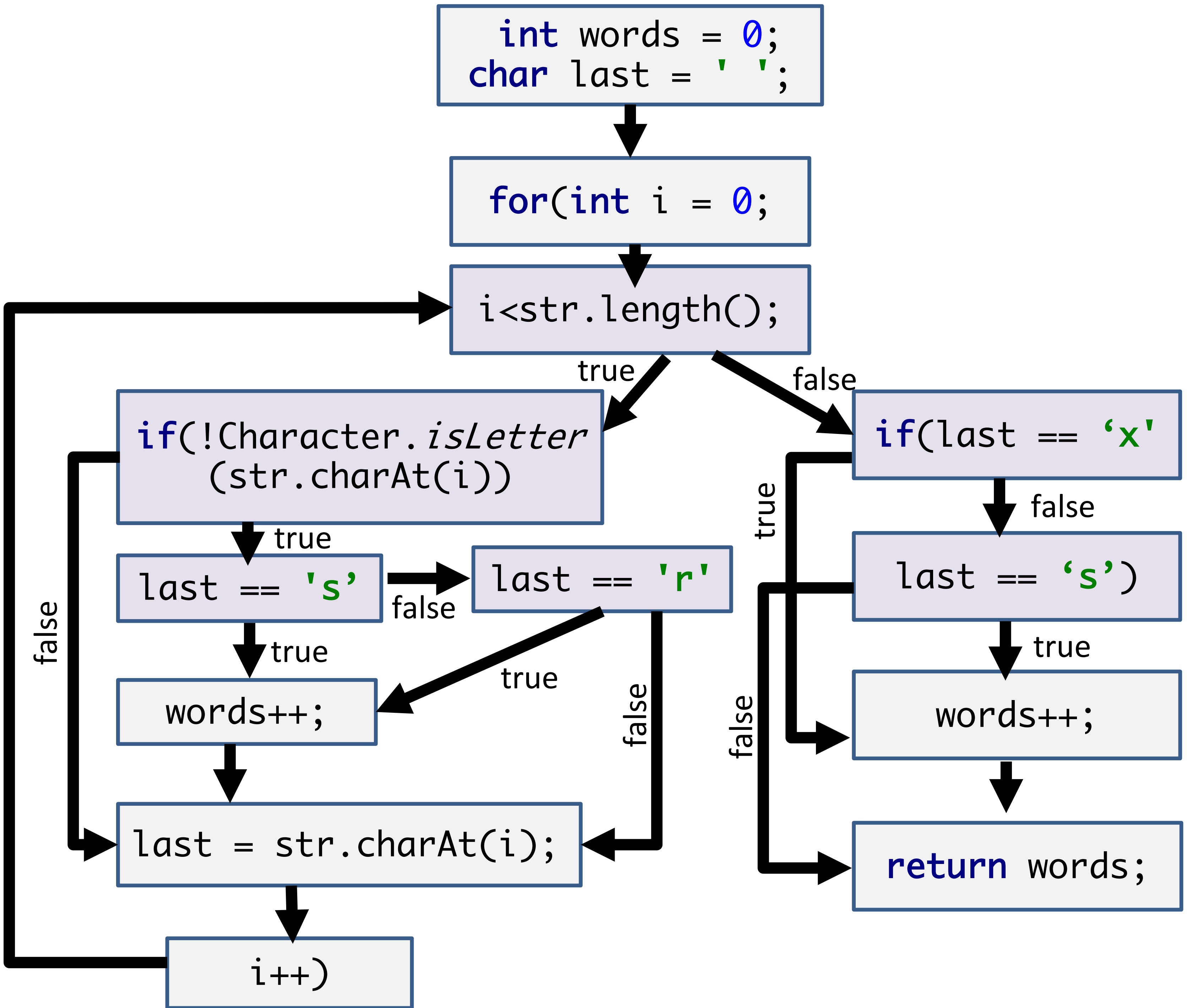

Branch coverage means we exercise all the branches!
I wonder if that's **enough**...

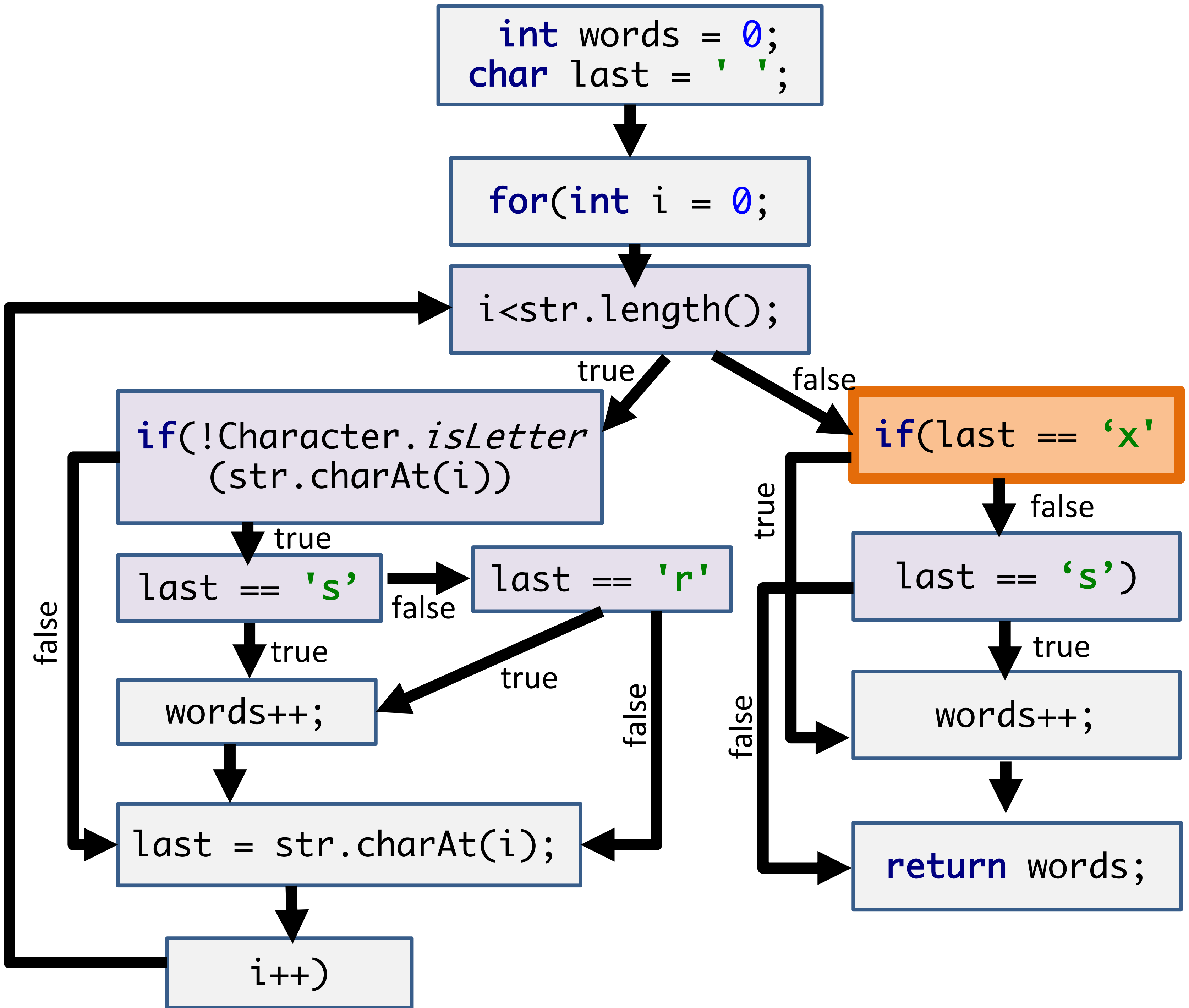

int words = 0;
char last = ' ';
for(int i = 0;
i<str.length();
true
false
if(!Character.isLetter
(str.charAt(i)) &&
(last == ‘s’ || last
== ‘r’))
true
false
words++;
last = str.charAt(i);
i++)
if(last == ‘x’
|| last == ‘s’)
true
false
words++;
return words;



If we “explode” the if into its several conditions, we have more paths to explore!

A basic block contains just a single condition now.


int words = 0;
char last = ' ';
for(int i = 0;
i<str.length();
true
false
if(!Character.isLetter
(str.charAt(i))
if(last == ‘x'
true
false
last == 's’
false
last == 'r'
last == ‘s’)
true
words++;
true
false
false
words++;
last = str.charAt(i);
return words;
i++)



We’d find this bug!

Ok, **condition coverage** seems to cover more than **branch coverage**!

It's your time!

The squirrels in Palo Alto spend most of the day playing. In particular, they play if the temperature is between 60 and 90 (inclusive). Unless it is summer, then the upper limit is 100 instead of 90. Given an int temperature and a boolean is_summer, return True if the squirrels play and False otherwise.

```
def squirrel_play(temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60 and temp <= up)
  return result
```

What's the **minimum amount of tests** you need to achieve:

- 100% Line coverage
- 100% Branch coverage
- 100% Condition coverage

Inspiration: https://codingbat.com/prob/p135815

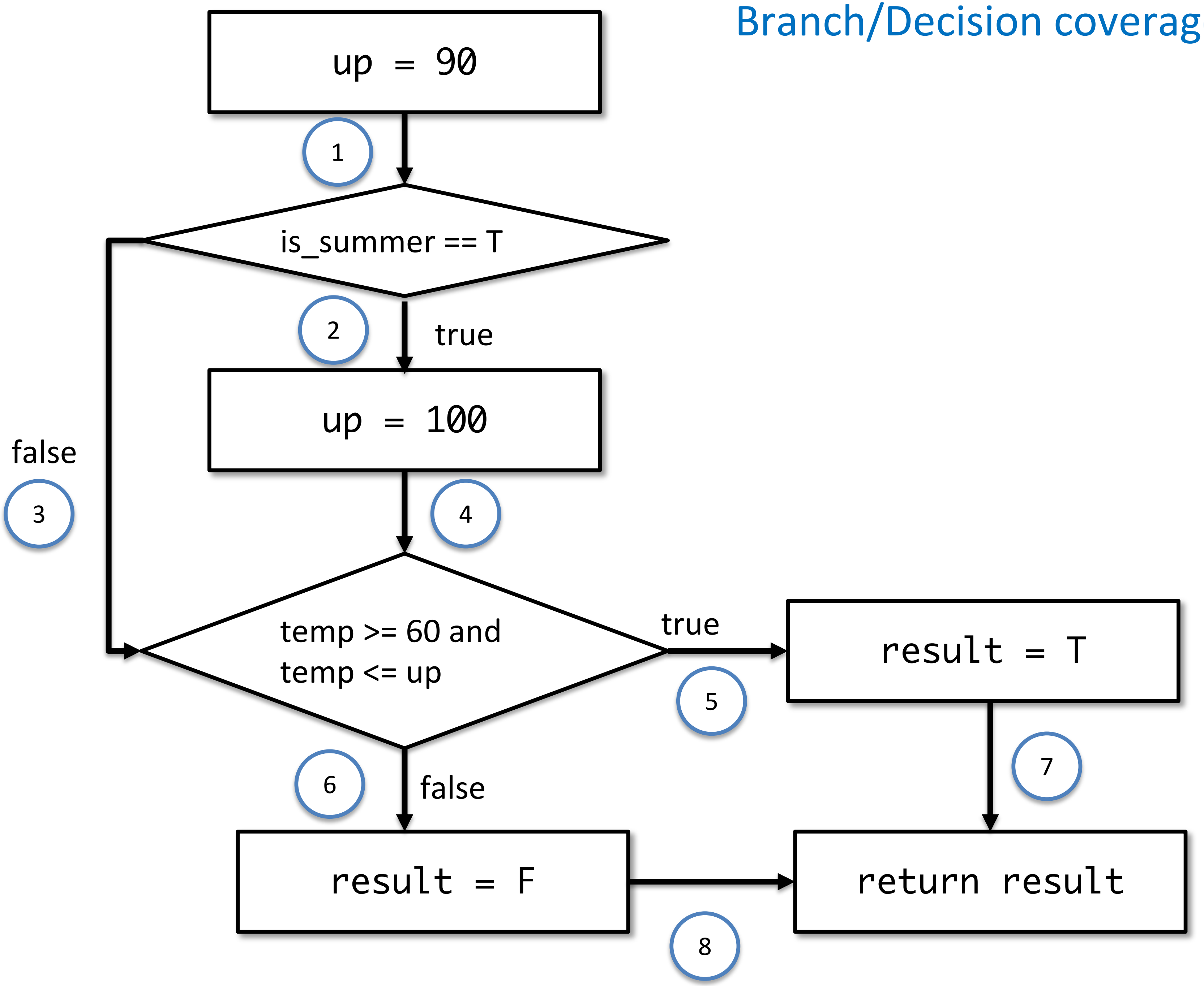
```python
def squirrel_play(temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60 and temp <= up)
  return result
```

T1: <80, true>

1 test = 100% line coverage!

Branch/Decision coverage

```
def squirrel_play
  (temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60
   and temp <= up)
  return result
```

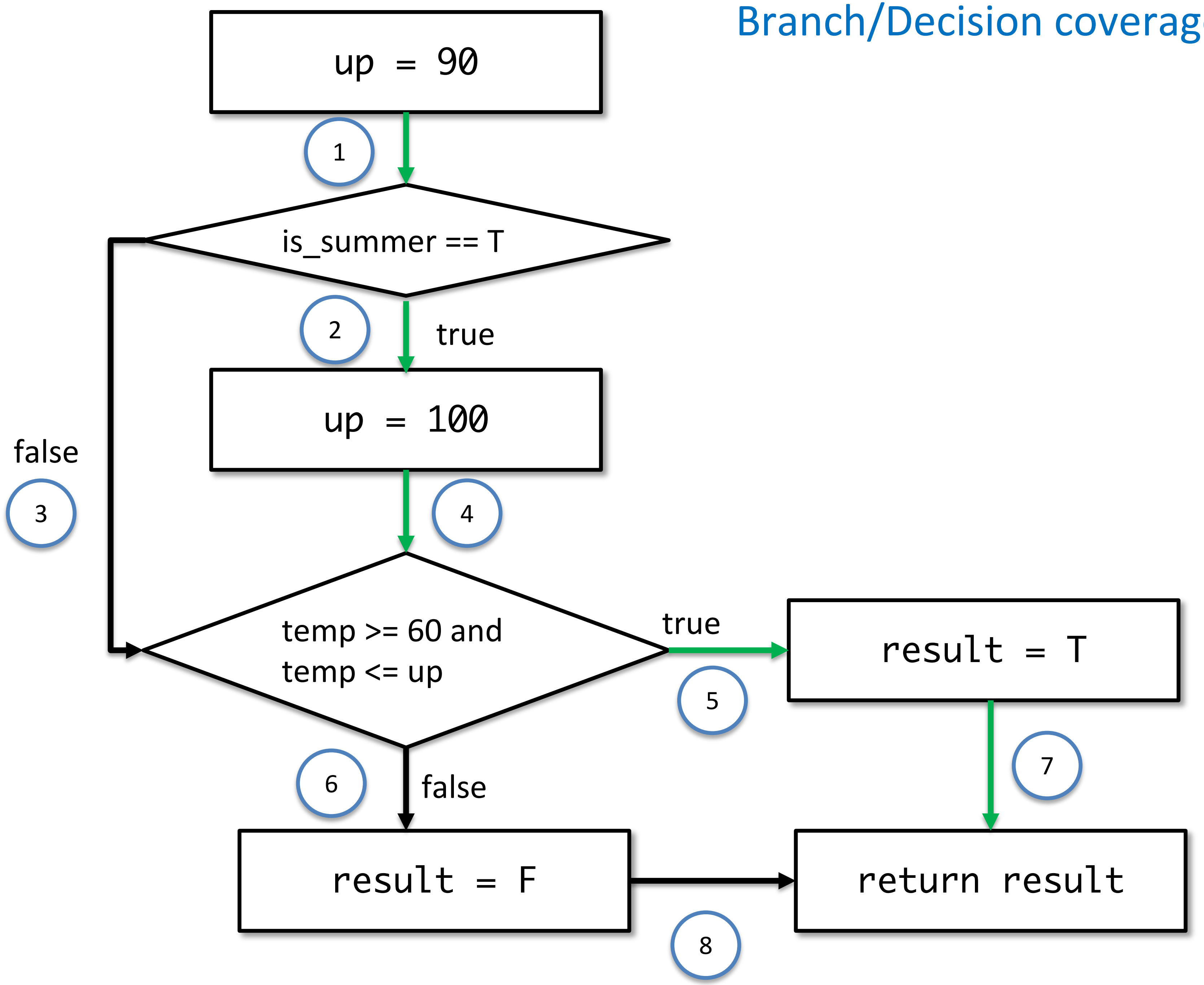


Branch/Decision coverage

```python
def squirrel_play
  (temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60
   and temp <= up)
  return result
```

T1: <80, true>

1, 2, 4, 5, 7

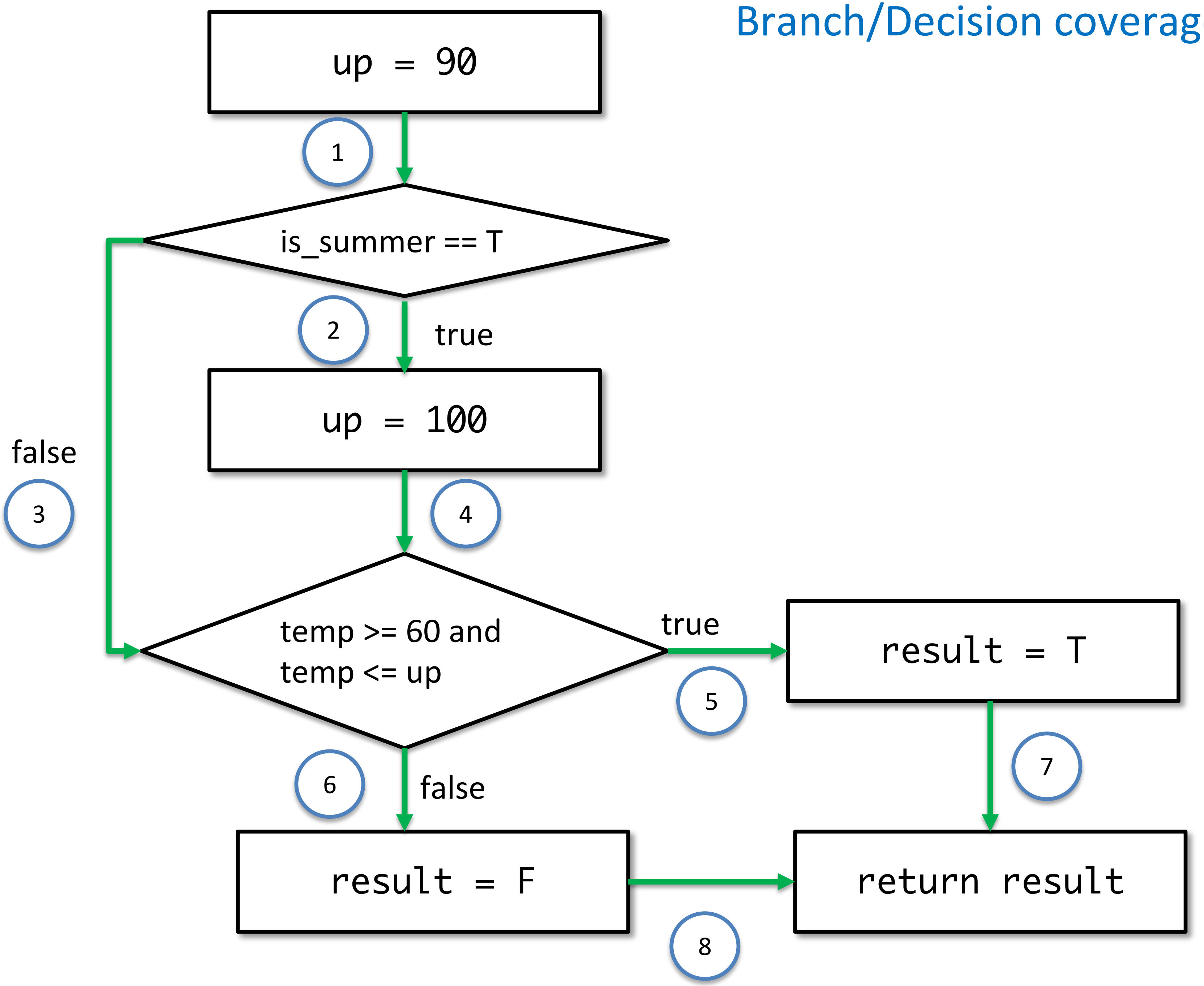


Branch/Decision coverage

```python
def squirrel_play
  (temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60
   and temp <= up)
  return result
```

T1: <80, true>
1, 2, 4, 5, 7

T2: <40, false>
1, 3, 6, 8



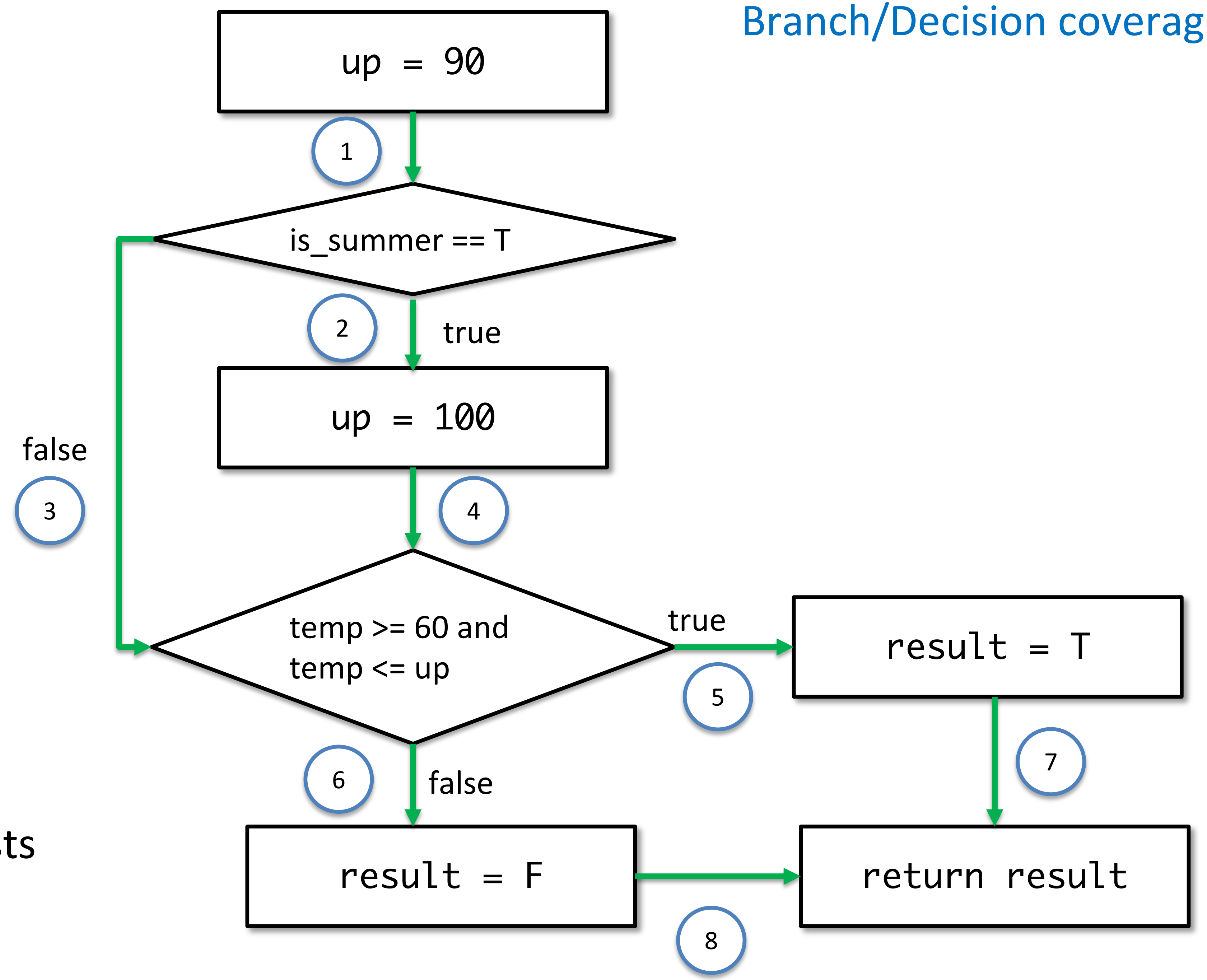
Branch/Decision coverage

```
def squirrel_play
  (temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60
   and temp <= up)
  return result
```

T1: <80, true>
1, 2, 4, 5, 7

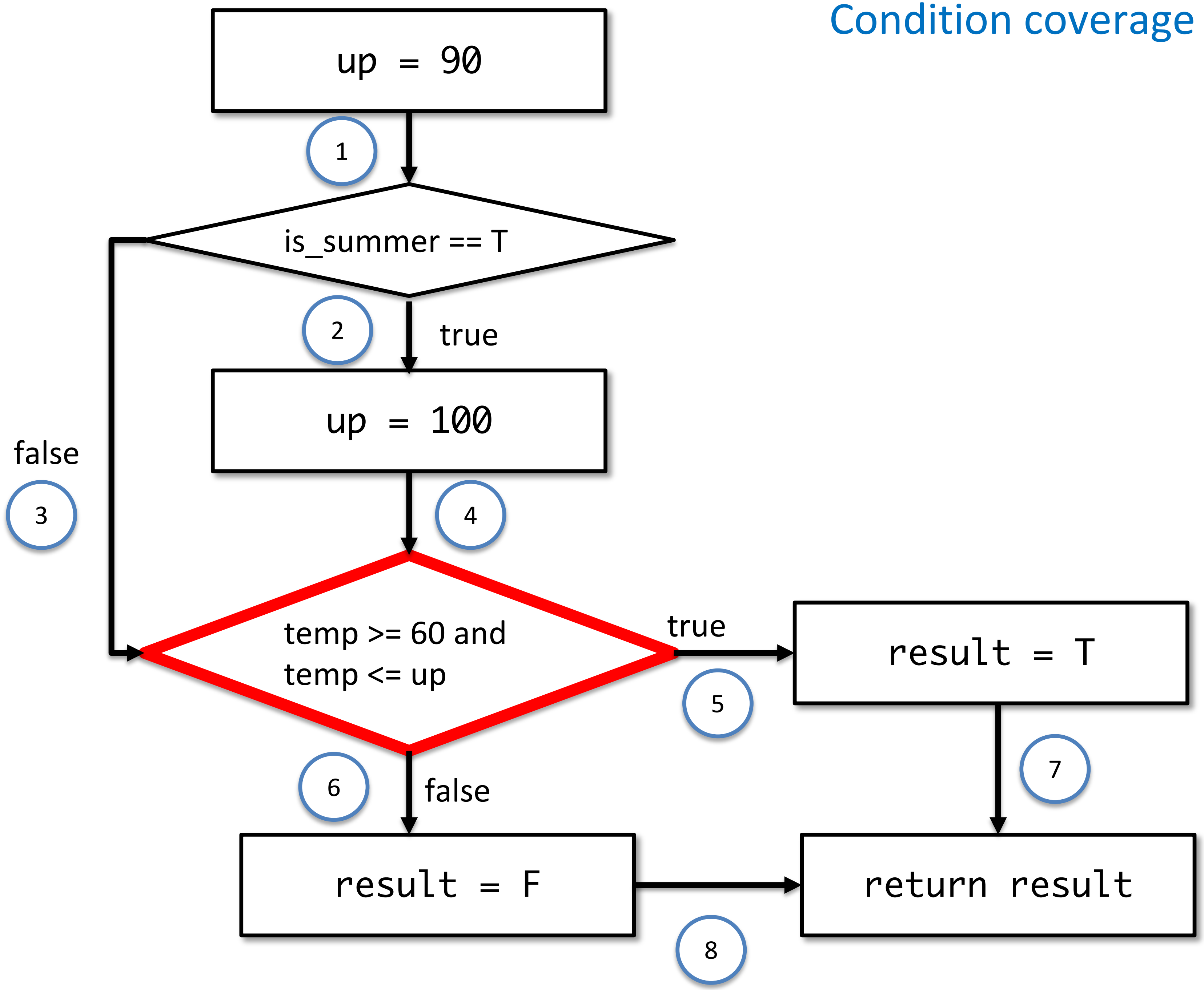
T2: <40, false>
1, 3, 6, 8

100% branch coverage: 2 tests



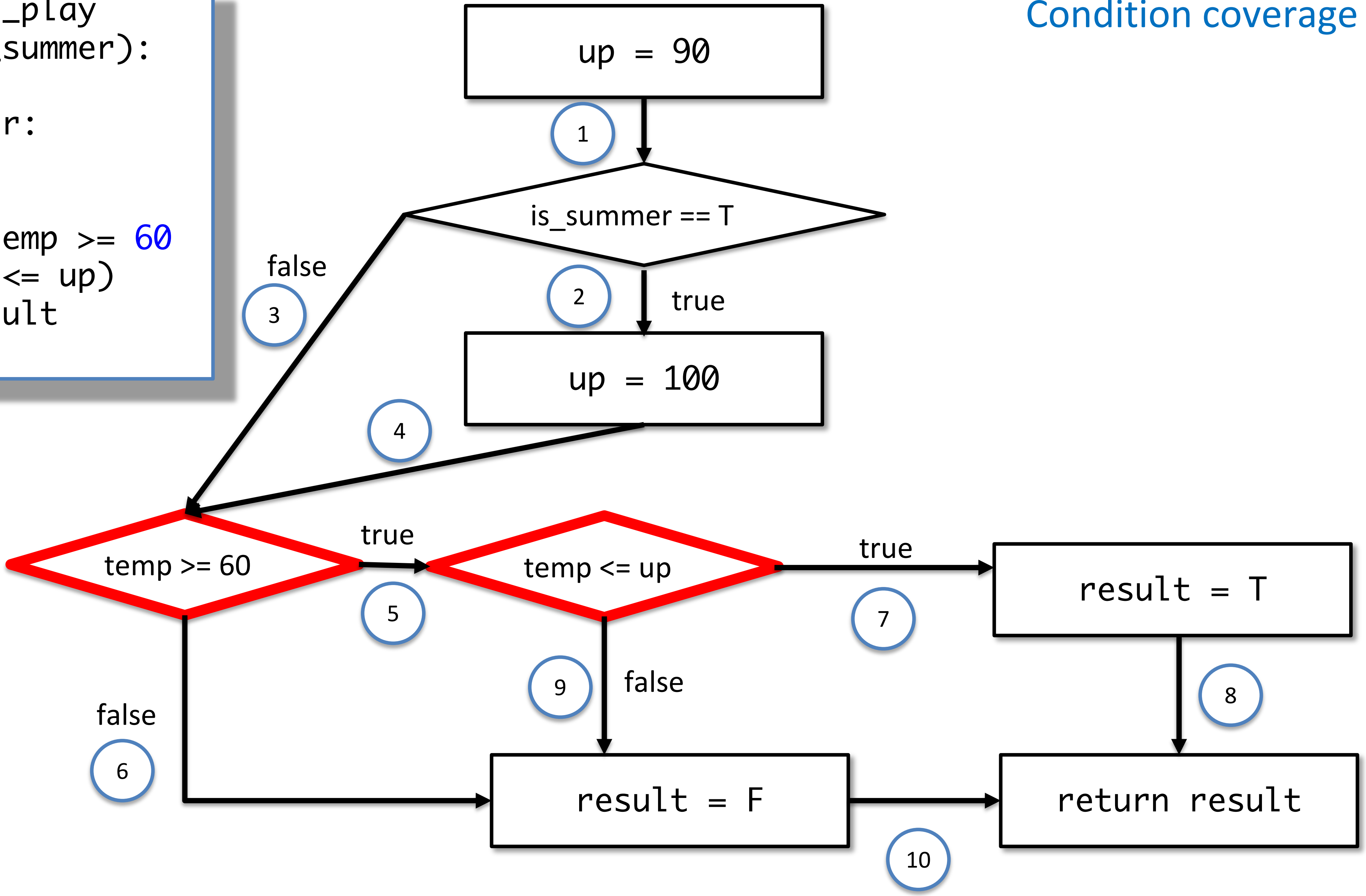
Condition coverage

```python
def squirrel_play
  (temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60
   and temp <= up)
  return result
```

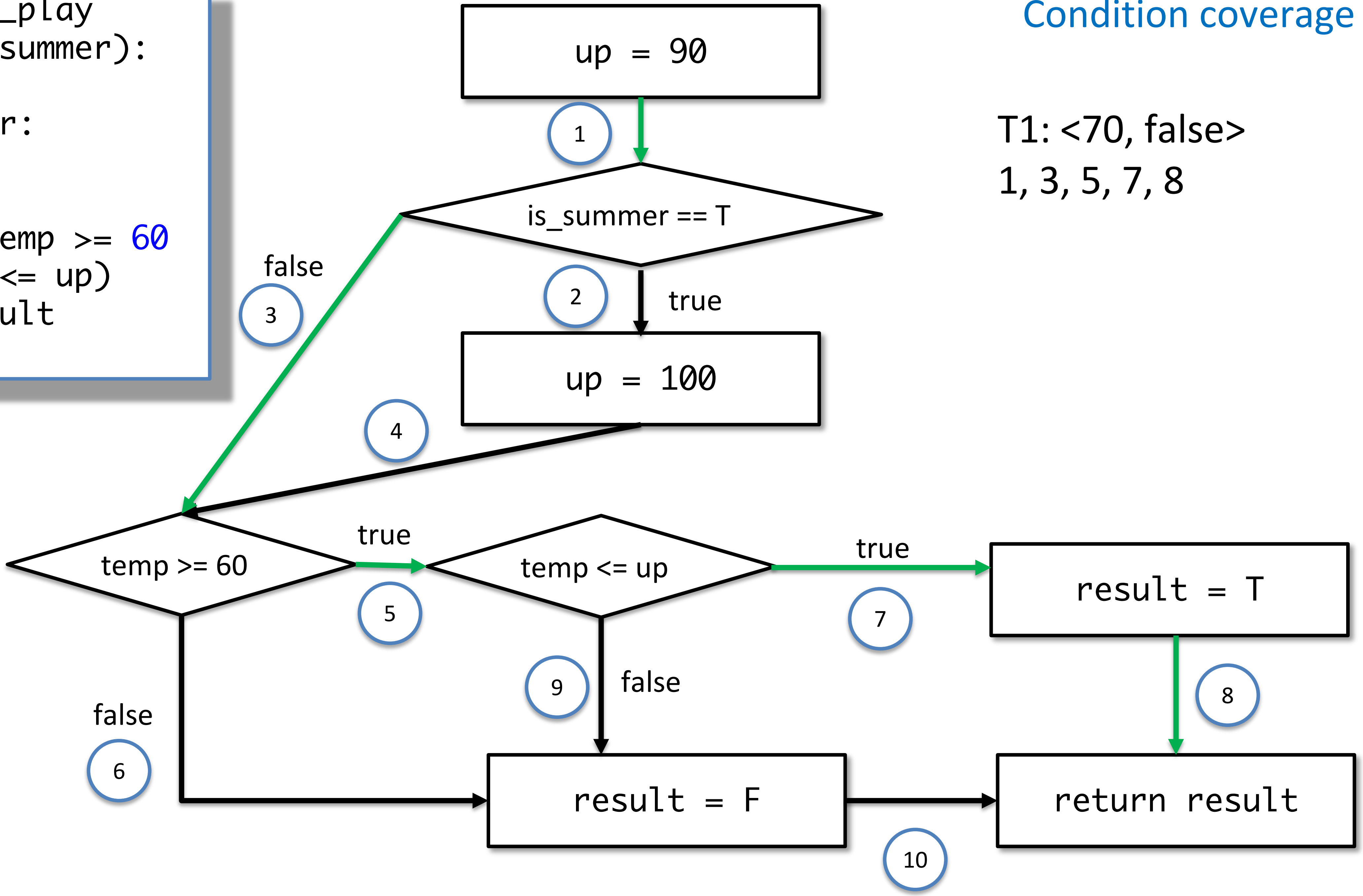


Condition coverage

```
def squirrel_play
  (temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60
   and temp <= up)
  return result
```



```python
def squirrel_play
  (temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60
   and temp <= up)
  return result
```



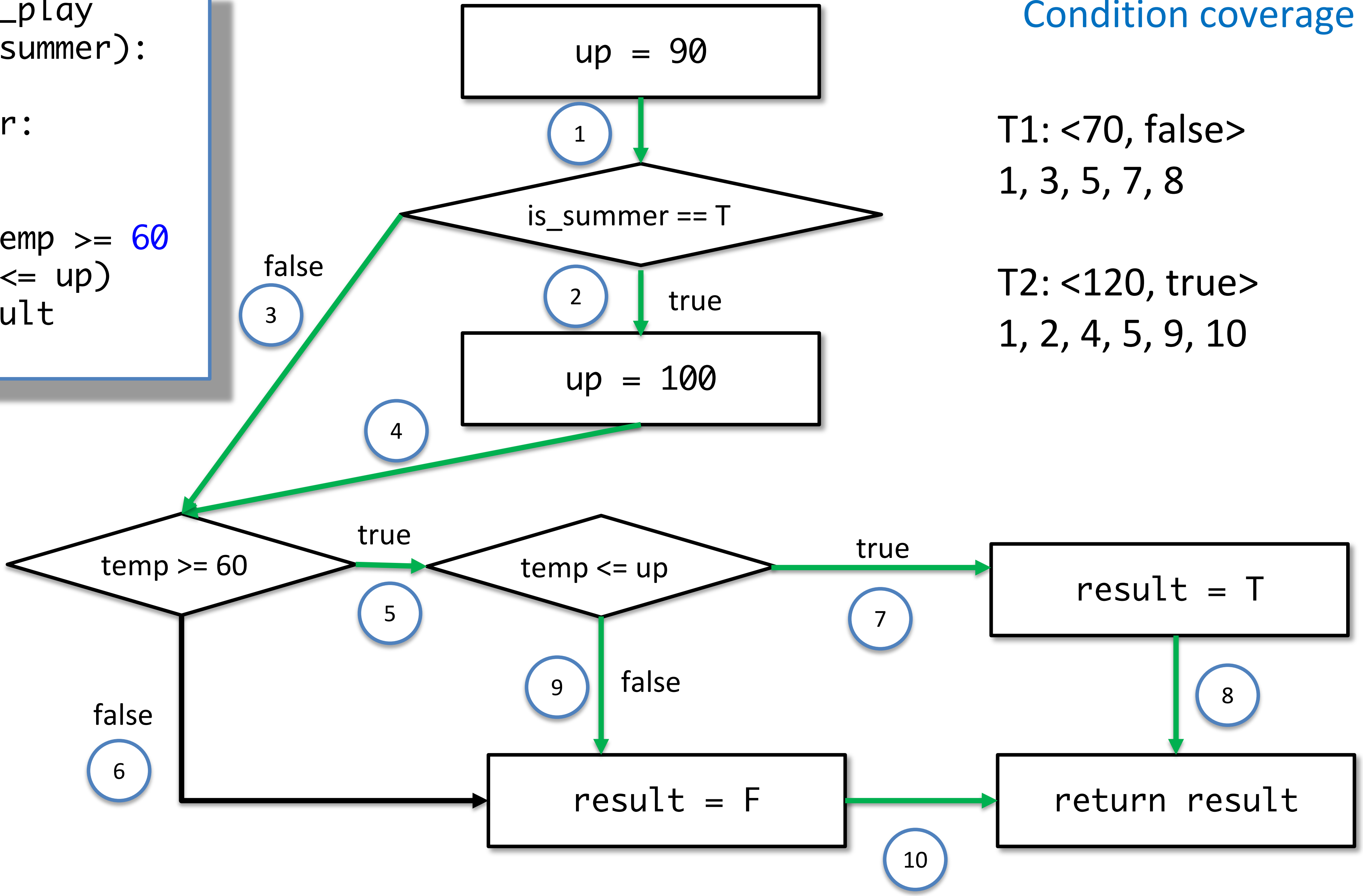
Condition coverage

T1: <70, false>

1, 3, 5, 7, 8

Condition coverage

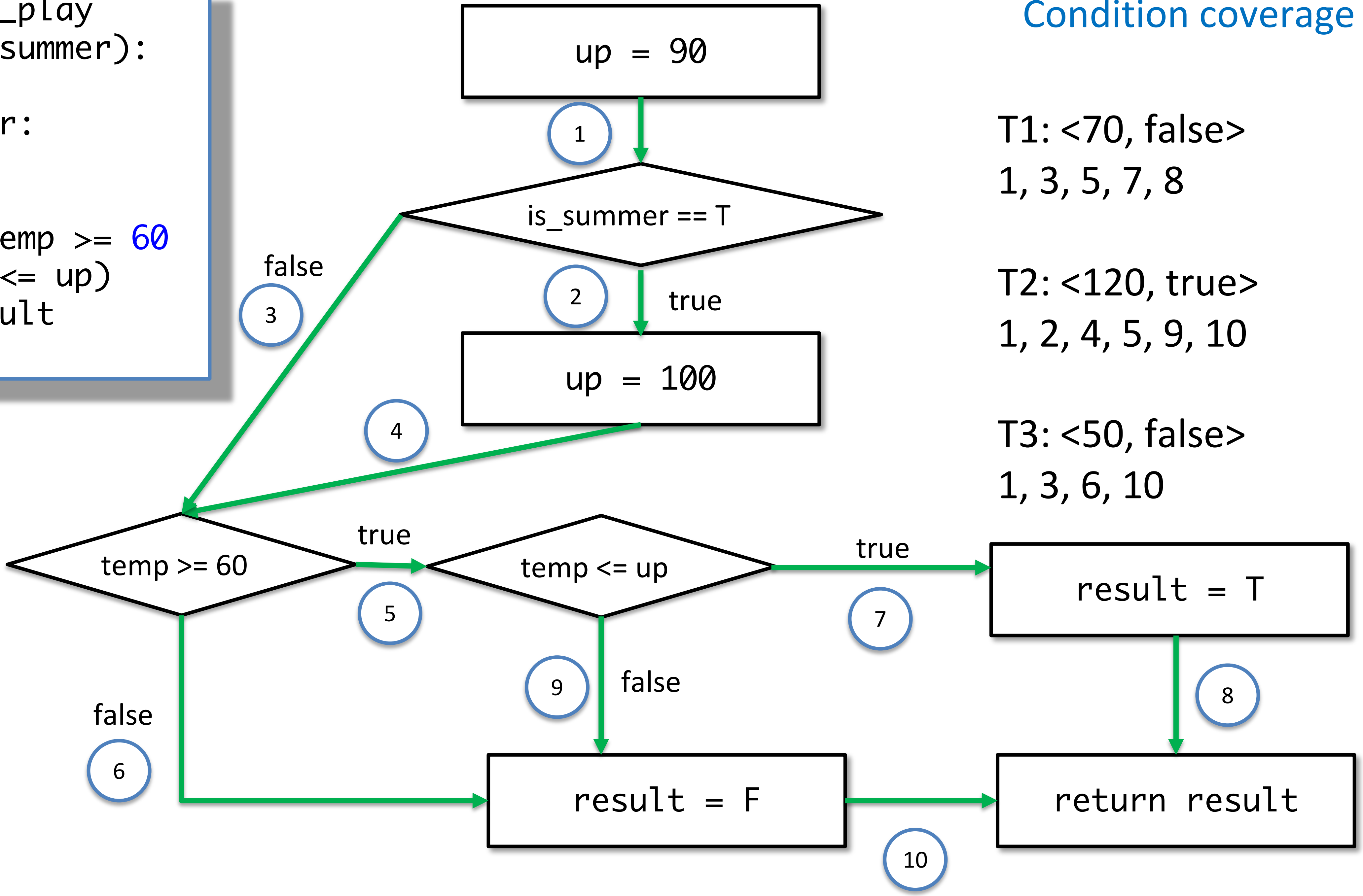
```python
def squirrel_play
  (temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60
   and temp <= up)
  return result
```



T1: <70, false>
1, 3, 5, 7, 8

T2: <120, true>
1, 2, 4, 5, 9, 10

```python
def squirrel_play
  (temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60
   and temp <= up)
  return result
```



Condition coverage

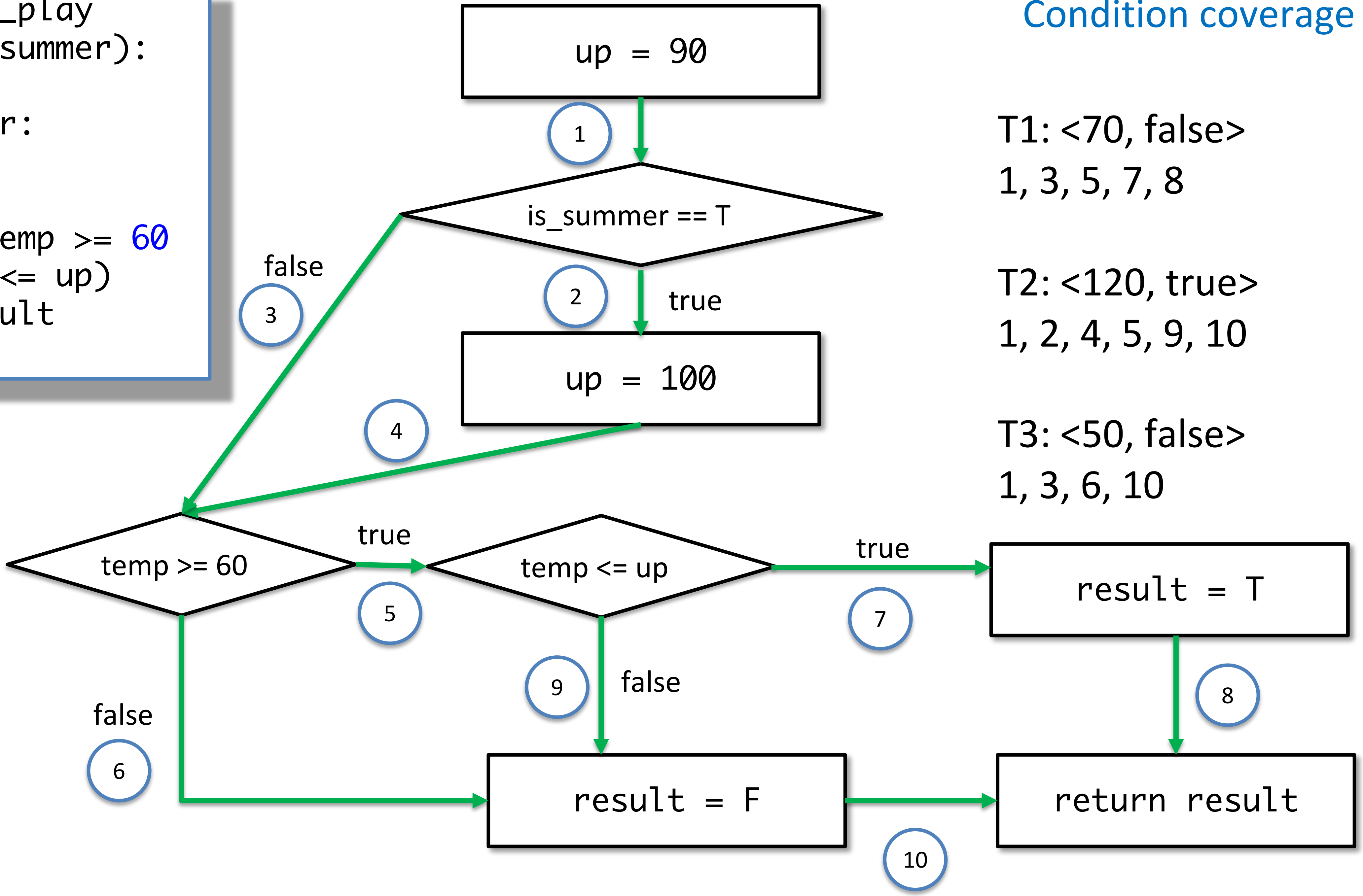
T1: <70, false>
1, 3, 5, 7, 8

T2: <120, true>
1, 2, 4, 5, 9, 10

T3: <50, false>
1, 3, 6, 10

Condition coverage

```python
def squirrel_play
  (temp, is_summer):
 up = 90
 if is_summer:
  up = 100

 result = (temp >= 60
   and temp <= up)
  return result
```



T1: <70, false>
1, 3, 5, 7, 8

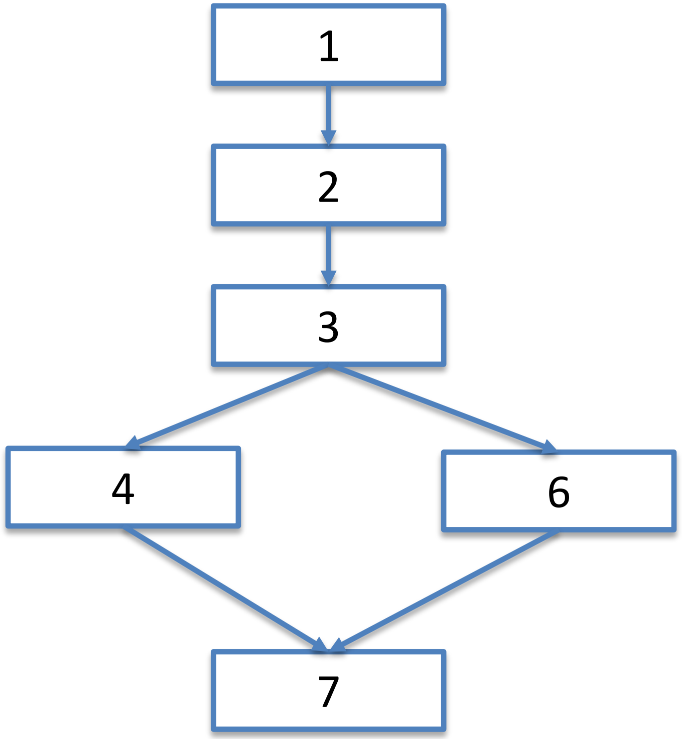
T2: <120, true>
1, 2, 4, 5, 9, 10

T3: <50, false>
1, 3, 6, 10

3 tests!

Does 100% condition coverage imply in 100% branch coverage?

```
1. read x
2. read y
3. if(x == 0 || y > 0)
4.   y = y / x;
5. else
6.   x = y + 2;
7. print x + y
```



100% condition coverage!

Test cases:

X = 0, Y = -5 X is true/false

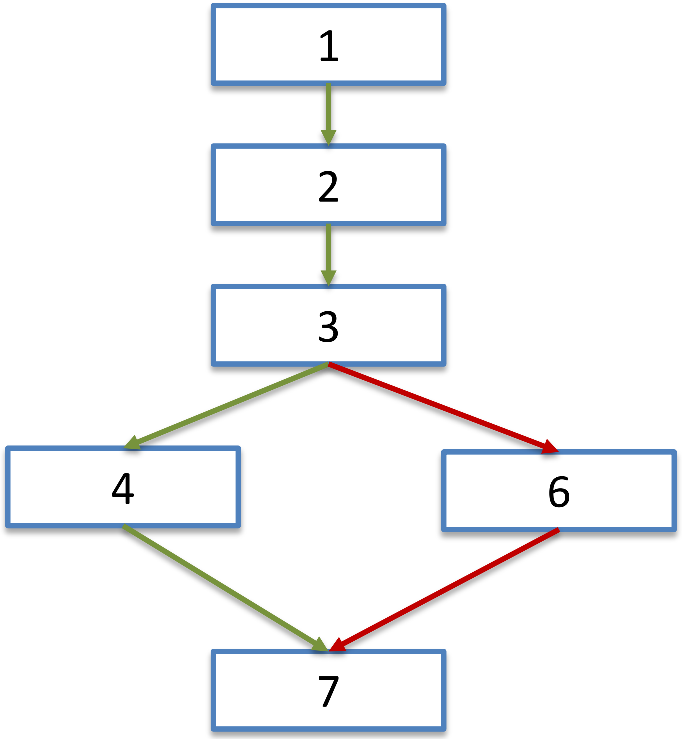
X = 5, Y = 5 Y is true/false

Does 100% condition coverage imply in 100% branch coverage?

```
read x
read y
if(x == 0 || y > 0)
  y = y / x;
else
  x = y + 2;
print x + y
```

Test cases:

X = 0, Y = -5	X is true/false
X = 5, Y = 5	Y is true/false



100% condition coverage!
50% decision/branch coverage!

Thus, 100% **(BASIC)** condition coverage does not necessarily mean 100% branch coverage.

Condition + Branch coverage does imply in 100% branch coverage.


If we aim for condition coverage, are we testing all the paths?

(A && (B | C))

Tests	a	b	c	O
1	T	T		
2				
				T
			F	F
		T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Path Coverage

Can we actually achieve 100% path coverage?

The number of paths can still grow exponentially

```
if (a) {
    S1;
}
if (b) {
    S2;
}
if (C) {
    S3;
}
...
if (x) {
    Sn;
}
```

- The subpaths through this control flow can include or exclude each of the statements **Si**, so that in total N branches result in 2^N paths that must be traversed
- Choosing input data to force execution of one particular path may be very difficult, or even impossible if the conditions are not independent

Modified Condition/Decision Coverage (MC/DC)

- Each entry and exit point is invoked
- Each decision takes every possible outcome (decision/branch coverage)
- Each condition in a decision takes every possible outcome (condition coverage)
- Each condition in a decision is shown to independently affect the outcome of the decision.

- When decisions are binary, with N conditions, I always have only **N+1** tests. That's definitely better than 2^n!

(A & (B | C))

Imagine this being a complex if condition in your system.

We saw how to:
1. Cover lines
2. Cover branches
3. Cover conditions
4. Cover all paths

(3) and (4) might be too expensive when number of combinations is big. **MC/DC is going to give us something in between condition and path coverage.**

In this example, 4 tests will give us good (MC/DC) coverage.

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(A & (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

We start with the first condition

(**A** && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(**A** && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(**A** && (B | C))

The one where "a" is flipped, and the rest is the same!

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

The result is different!

(**A** && (B | C))

Tests = {1, 5}

Let's keep track of this pair!

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(**A** && (B | C))

Tests = {1, 5}

We move to the next row

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

The result is also different!

$(A \ \&\& \ (B \ | \ C))$

Tests = {1, 5}, {2, 6}

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(**A** && (B | C))

Tests = {1, 5}, {2, 6}

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(**A** && (B | C))

Tests = {1, 5}, {2, 6}, {3,7}

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(**A** && (B | C))

Tests = {1, 5}, {2, 6}, {3,7}

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

The result is the same. So, “not interesting for us”

We now go to the next condition

(A && (**B** | C))

A = {1, 5}, {2, 6}, {3,7}

B =

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(A && (**B** | C))

A = {1, 5}, {2, 6}, {3,7}

B =

The result is the same. So, “not interesting for us”

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(A && (**B** | C))

A = {1, 5}, {2, 6}, {3,7}
B =

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(A && (**B** | C))

A = {1, 5}, {2, 6}, {3,7}
B = {2, 4}

Different results, so we keep it!

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(we continue doing the same, but there are no other interesting ones)

(A && (B | **C**))

A = {1, 5}, {2, 6}, {3,7}

B = {2, 4}

C =

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(A && (B | **C**))

A = {1, 5}, {2, 6}, {3,7}
B = {2, 4}
C = {3, 4}

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(A && (B | C))

A = {1, 5}, {2, 6}, {3,7}
B = {2, 4}
C = {3, 4}

Set of tests we need!

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

But it's almost like testing them all…

(A && (B | C))

A = {1, 5}, {2, 6}, {3,7}
B = {2, 4}
C = {3, 4}

Final = {2, 3, 4, 6}

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

They are the same!
We don't need them all

(A && (B | C))

A = {1, 5}, **{2, 6}**, {3,7}
B = **{2, 4}**
C = **{3, 4}**

Final = {2, 3, 4, 6}

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

It's your turn!

(a II b) && c

Truth Table

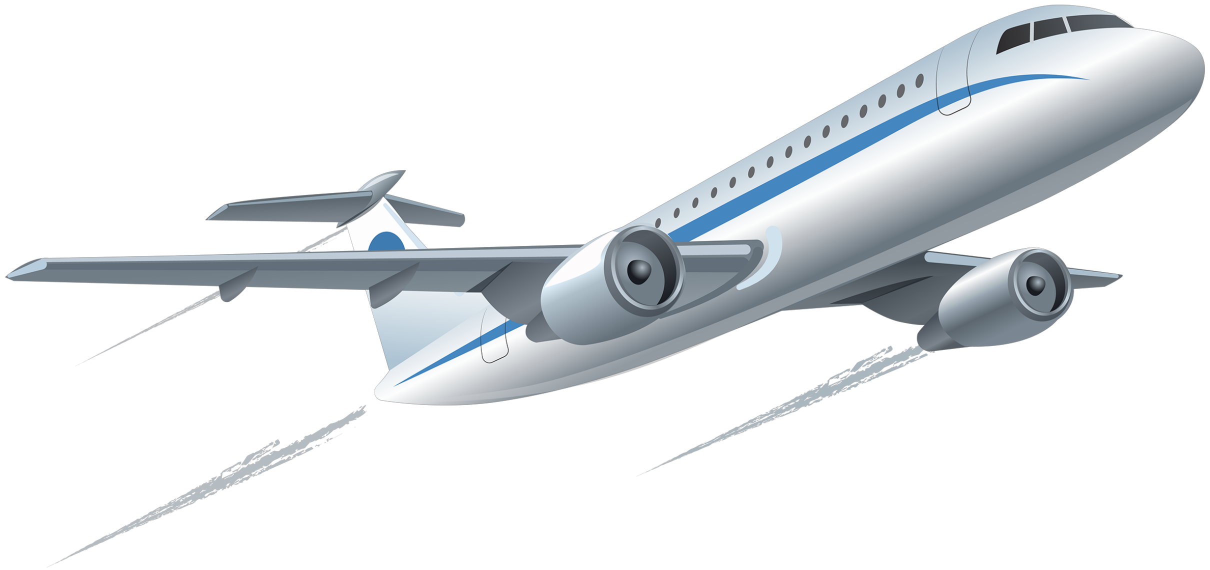
Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

MC/DC

Tests	a	b	c	Outcome
3	T	F	T	T
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F

(3 conditions + 1) = 4 tests

Federal Aviation Administration (FAA) requires that all softwares running on commercial airplane must be tested using MC/DC!

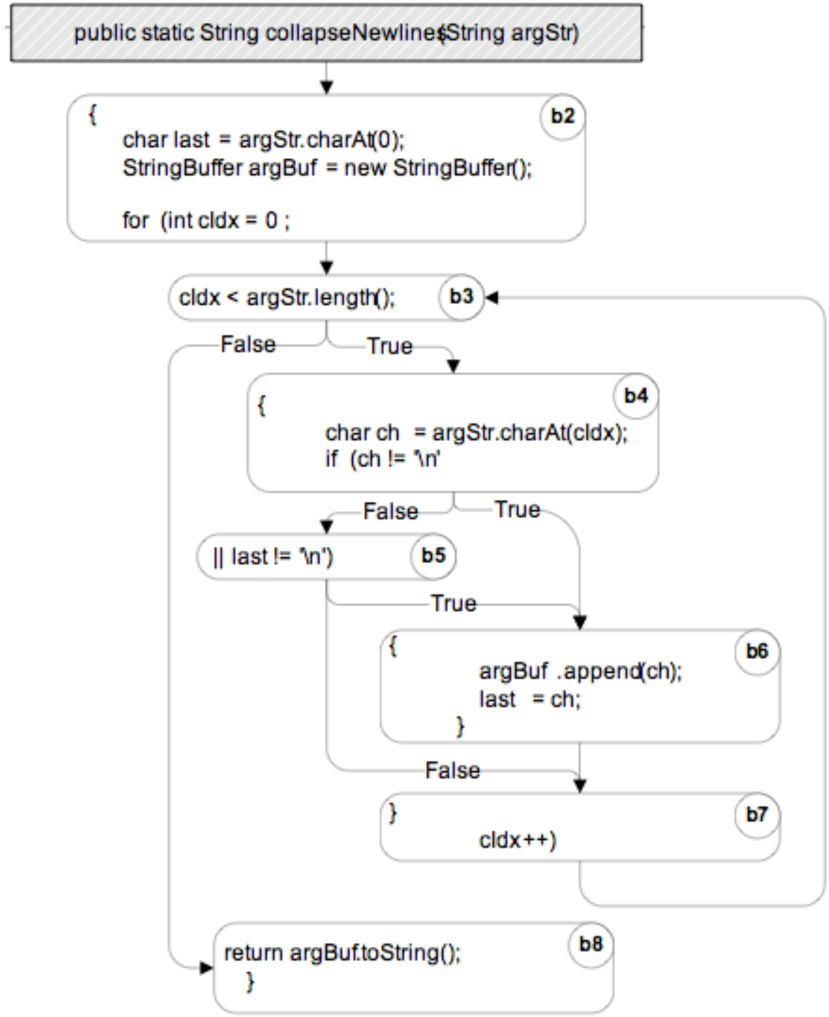


Loop Boundary Adequacy

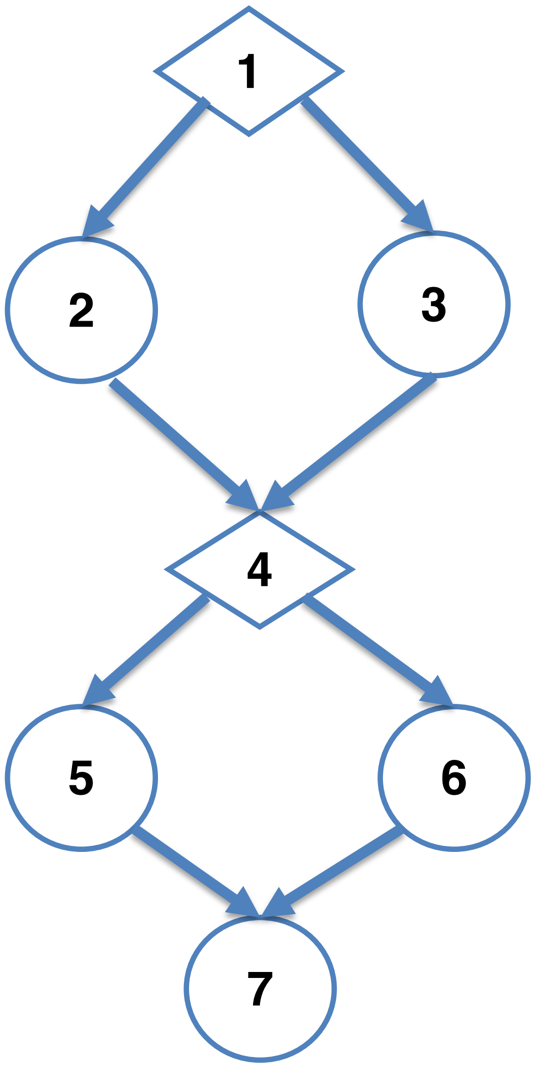
That's the challenge!

A test suite satisfies this criterion iff for every loop:

- a test case exercises the loop zero time
- a test case exercises the loop once
- a test case exercises the loop multiple times



McCabe's Cyclomatic Complexity

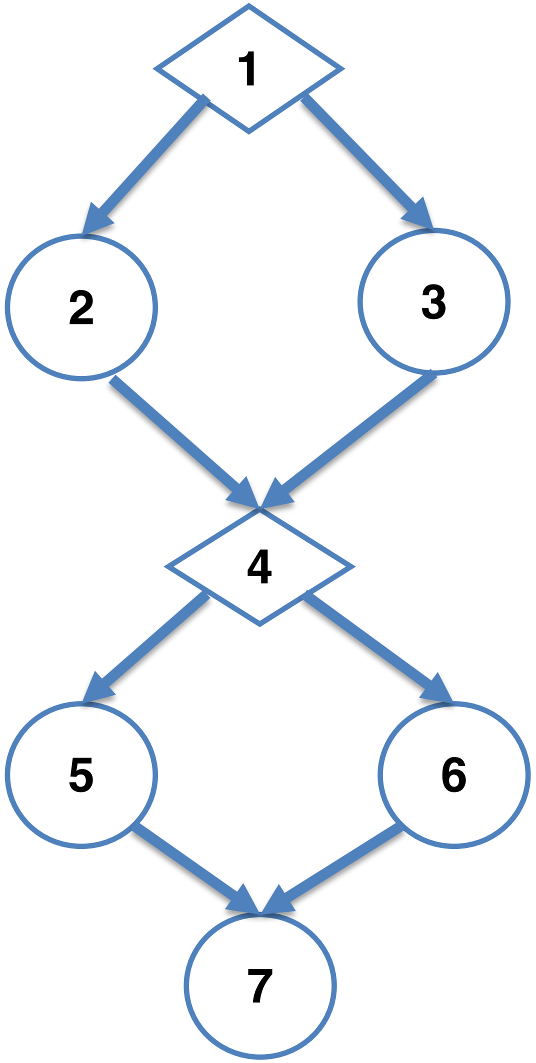


- C = |E| - |N| + 2
- C = # decision points + 1
- C = # of decision-statements + 1

C > 10: method too complex [McCabe, 1976]

[C correlated with #lines of code]

McCabe for Testing?



No empirical evidence that it is better than just decision coverage.

How many tests?

- Branch: **2 tests**
- All paths: **4 tests**
- McCabe: **3 tests**

McCabe: Easy to count, limited usefulness as coverage metric

Infeasible Paths

```
int example (int a) {
  int r = OK;

  if(a == -1) {
    r = ERROR_CODE;
    ERXA_LOG(r);
  }

  if(a == -2) {
    r = OTHER_ERROR_CODE;
    ERXA_LOG(r);
  }

  return r;
}
```

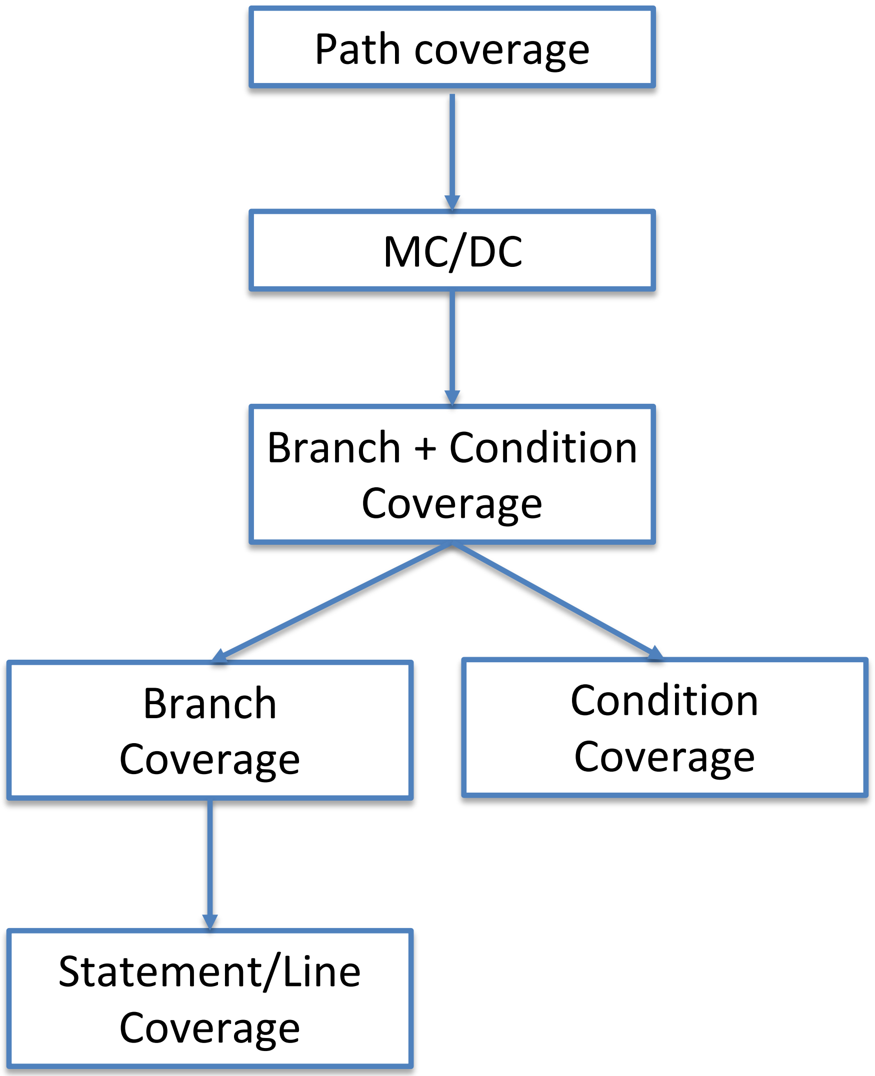
Three feasible paths:

1) a = -1;
2) a = -2
3) or any other a value

Infeasible path:

(a == -1) AND (a == -2)

Strategy **Subsumption**



(*) Although statement and line coverage have their differences, we are considering them to be similar when it comes to strategy subsumptions.

- Strategy X **subsumes** strategy Y if all elements that Y exercises are also exercised by X
- Example: 100% of branch coverage **implies** in 100% line coverage. 100% of line coverage **does not imply** in 100% branch coverage.

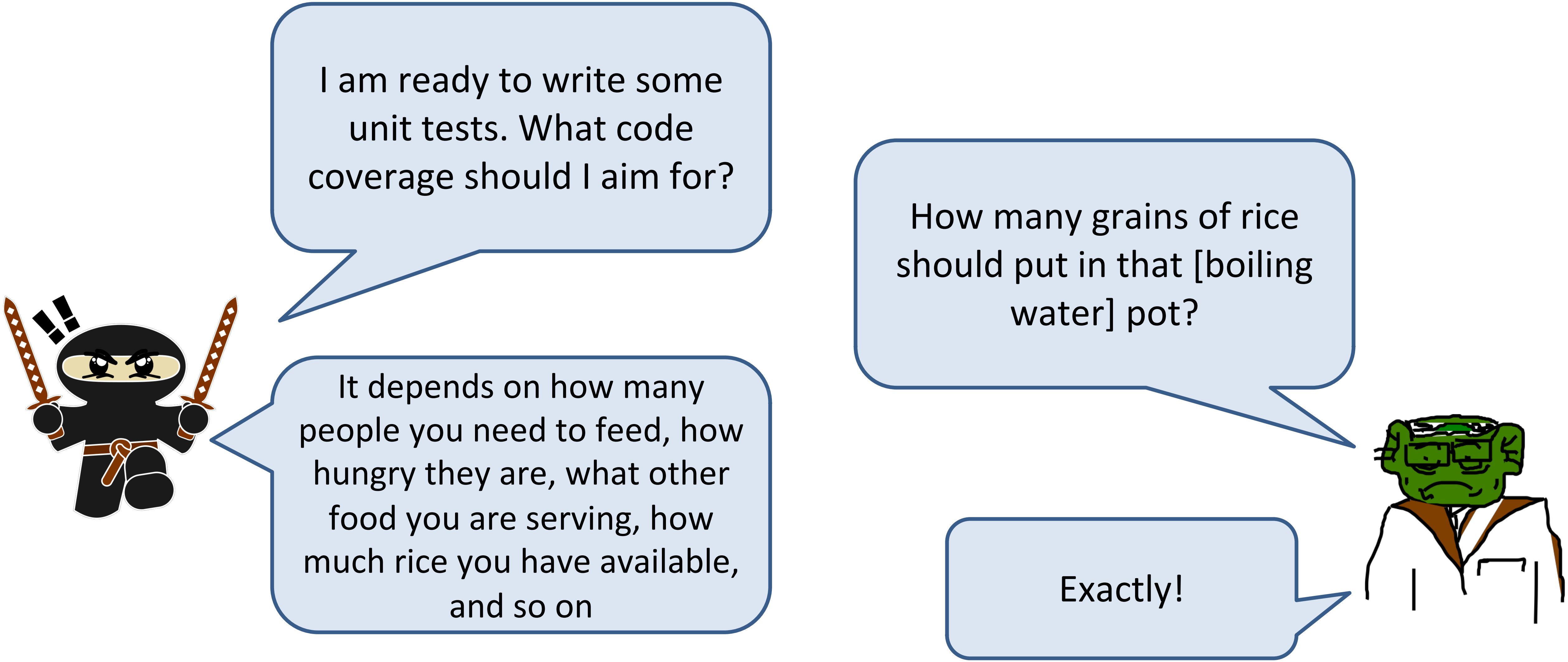
What do **YOU** think:

Do we need 100% code coverage?

I am ready to write some unit tests. What code coverage should I aim for?

Don’t worry about coverage, just write some good tests.

Testivus on Code Coverage. Alberto Savoia. https://www.artima.com/weblogs/viewpost.jsp?thread=204677



Testivus on Code Coverage. Alberto Savoia. https://www.artima.com/weblogs/viewpost.jsp?thread=204677

I am ready to write some unit tests. What code coverage should I aim for?

80% and no less!

Testivus on Code Coverage. Alberto Savoia. https://www.artima.com/weblogs/viewpost.jsp?thread=204677

The first programmer is new and just getting started with testing. Right now he has a lot of code and no tests. He has a long way to go; focusing on code coverage at this time would be depressing and quite useless. He’s better off just getting used to writing and running some tests. He can worry about coverage later.

Testivus on Code Coverage. Alberto Savoia. https://www.artima.com/weblogs/viewpost.jsp?thread=204677

The second programmer, on the other hand, is quite experience both at programming and testing. When I replied by asking her how many grains of rice I should put in a pot, I helped her realize that the amount of testing necessary depends on a number of factors, and she knows those factors better than I do – it’s her code after all. There is no single, simple, answer, and she’s smart enough to handle the truth and work with that.

Testivus on Code Coverage. Alberto Savoia. https://www.artima.com/weblogs/viewpost.jsp?thread=204677

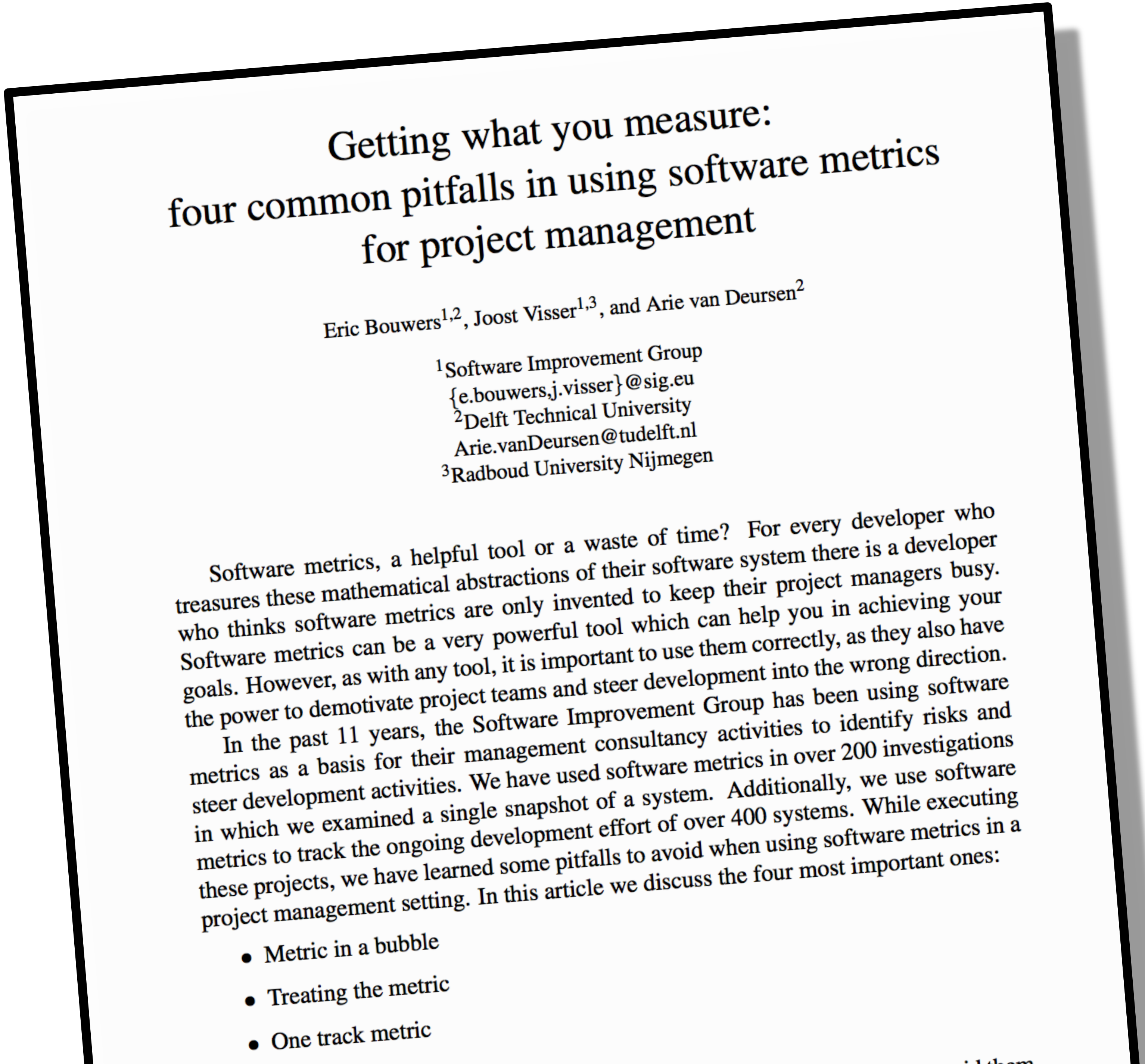
The third programmer wants only simple answers – even when there are no simple answers … and then does not follow them anyway.

Testivus on Code Coverage. Alberto Savoia. https://www.artima.com/weblogs/viewpost.jsp?thread=204677

Effectiveness of test coverage

- Hutchins et al. “Within the limited domain of our experiments, test sets achieving **coverage levels over 90% usually showed significantly better fault detection** than randomly chosen test sets of the same size. In addition**, significant improvements** in the effectiveness of coverage-based tests usually occurred as coverage **increased from 90% to 100%.** However, the results also indicate that **100% code coverage alone is not a reliable indicator** of the effectiveness of a test set.”
- Namin and Andrews: “Our experiments indicate that **coverage is sometimes correlated with effectiveness** when size is controlled for, and that using both size and coverage yields a more accurate prediction of effectiveness than size alone. This in turn suggests that both size and coverage are important to test suite effectiveness.”

Hutchins, M., Foster, H., Goradia, T., & Ostrand, T. (1994, May). Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering* (pp. 191-200). IEEE Computer Society Press.
Namin, A. S., & Andrews, J. H. (2009, July). The influence of size and coverage on test suite effectiveness. In Proceedings of the eighteenth international symposium on Software testing and analysis (pp. 57-68). ACM.



Getting what you measure: four common pitfalls in using software metrics for project management

Eric Bouwers[1,2], Joost Visser[1,3], and Arie van Deursen[2]

[1]Software Improvement Group
{e.bouwers,j.visser}@sig.eu
[2]Delft Technical University
Arie.vanDeursen@tudelft.nl
[3]Radboud University Nijmegen

Software metrics, a helpful tool or a waste of time? For every developer who treasures these mathematical abstractions of their software system there is a developer who thinks software metrics are only invented to keep their project managers busy. Software metrics can be a very powerful tool which can help you in achieving your goals. However, as with any tool, it is important to use them correctly, as they also have the power to demotivate project teams and steer development into the wrong direction.

In the past 11 years, the Software Improvement Group has been using software metrics as a basis for their management consultancy activities to identify risks and steer development activities. We have used software metrics in over 200 investigations in which we examined a single snapshot of a system. Additionally, we use software metrics to track the ongoing development effort of over 400 systems. While executing these projects, we have learned some pitfalls to avoid when using software metrics in a project management setting. In this article we discuss the four most important ones:

- Metric in a bubble
- Treating the metric
- One track metric

- Metric in a bubble
- Treating the metric
- One track metric
- Metrics galore

Compulsory reading!

Reading Material

- **Compulsory:** Chapter 4 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
- Chapter 12 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.
- Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. ACM computing surveys (csur), 29(4), 366-427.
- Cem Kaner on Code Coverage: http://www.badsoftware.com/coverage.htm
- Arie van Deursen on Code Coverage: http://avandeursen.com/2013/11/19/test-coverage-not-for-managers/

License

- You can use and share any of my material (lecture slides, website).
- You always have to give credits to the original author.
- You agree not to sell it or make profit in any way with this.

- Material that I refer has its own license. Please check it out.