Loop Invariants: Part 2

Maintaining the Loop Invariant

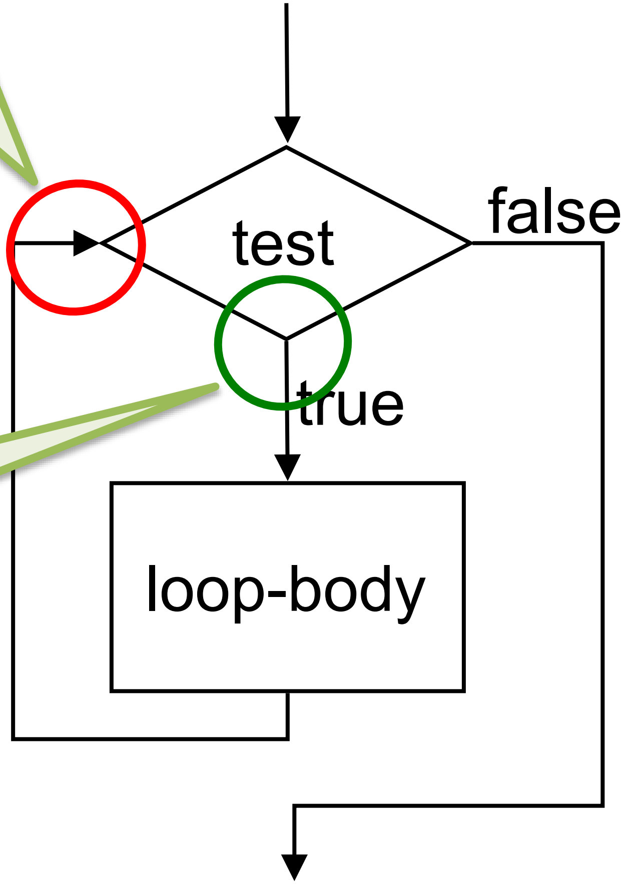
- A claimed ***loop invariant*** is valid only if the loop body actually maintains the property, i.e., the loop invariant remains true at the end of each execution of the loop body
- To show this, you may assume:
  - The loop invariant is valid at the start of the loop body
  - The loop condition is true

The Loop Invariant Picture

To show the loop invariant true at this red point...

```
while (test) {
    loop-body
}
```

...assume the invariant is true at this green point.



Isn’t This Reasoning Circular?

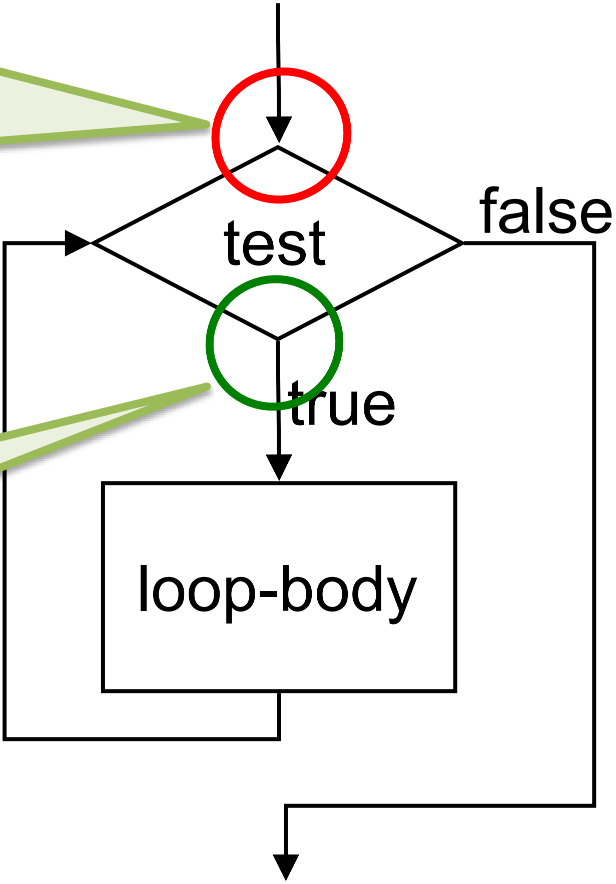
- To justify the assumption that the loop invariant holds just *after* the loop condition test, didn’t we argue that assumption was valid ***because*** the loop invariant holds just *before* the test?
- This is not circular reasoning but rather mathematical induction
  - See the confidence-building approach for reasoning about why recursion works

The Loop Invariant Picture

Show the loop invariant is true
at this red point...

```
while (test) {
   loop-body
 }
```

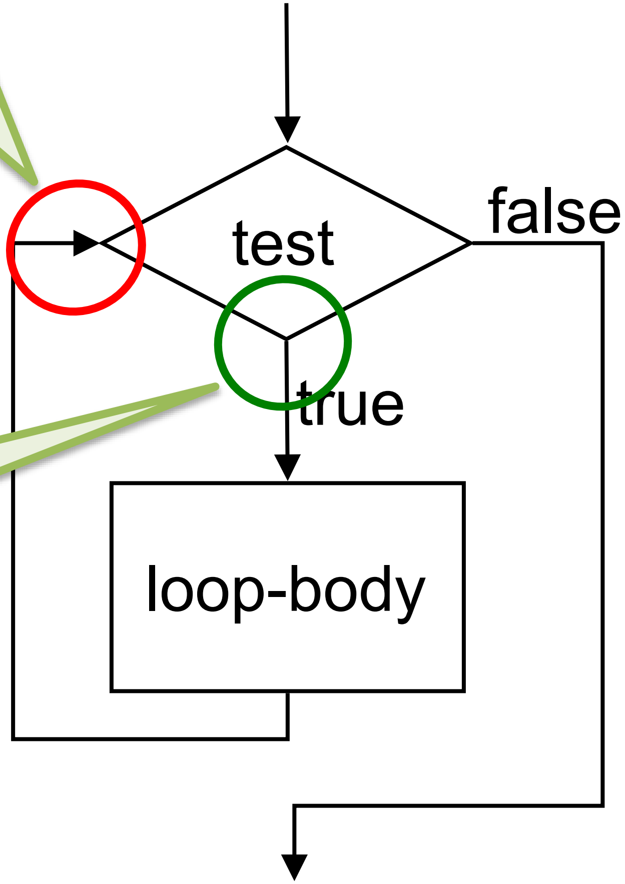
...which means it is true
at this green point
on the *first* iteration...



The Loop Invariant Picture

Show the loop invariant is true
at this red point
at the end of the *first* iteration...

```
while (test) {
  loop-body
}
```



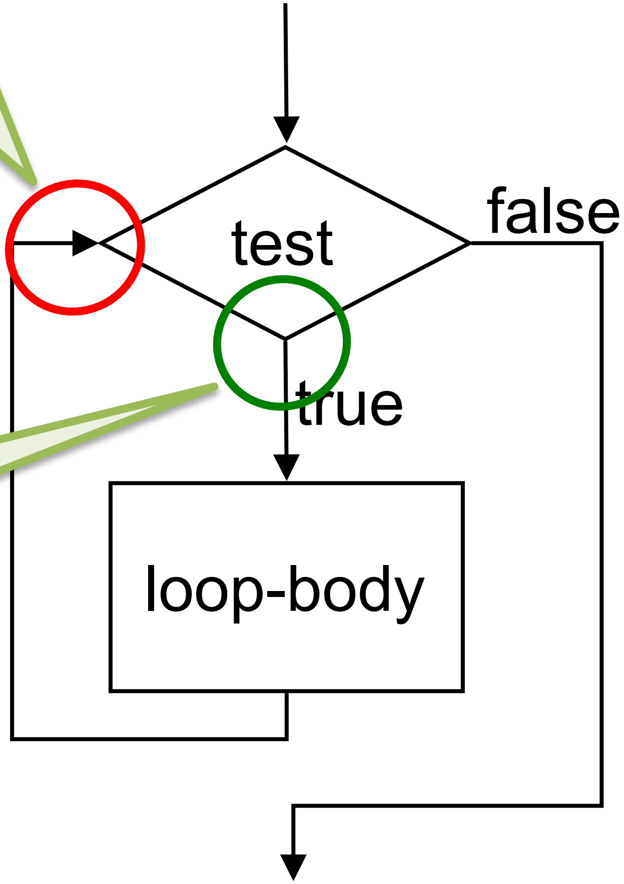
...which means it is true
at this green point
on the *second* iteration...

The Loop Invariant Picture

Show the loop invariant is true
at this red point
at the end of the *k-th* iteration...

```
while (test) {
  loop-body
}
```

...which means it is true
at this green point
on the *(k+1)-st* iteration...

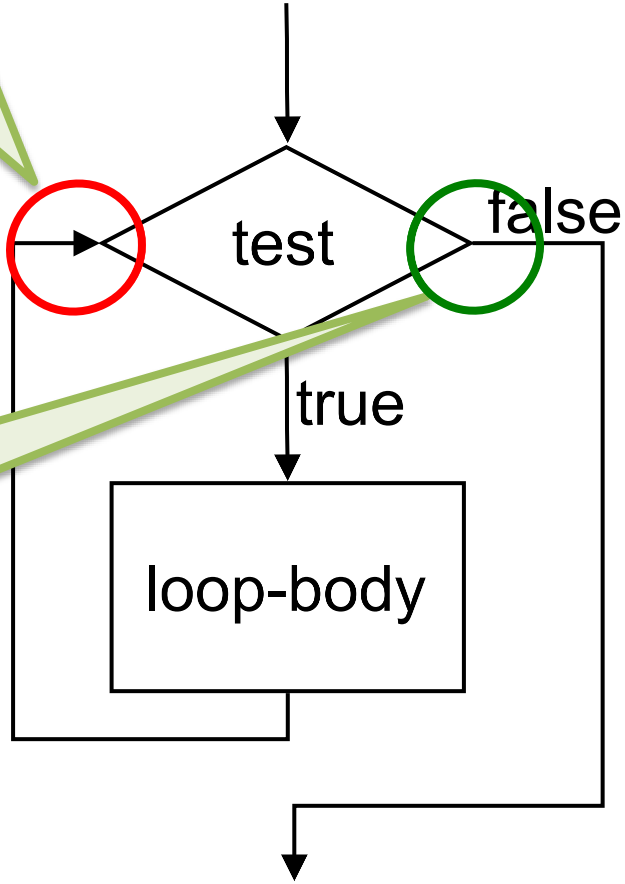


The Loop Invariant Picture

Show the loop invariant is true
at this red point
at the end of the *last* iteration...

```
while (test) {
  loop-body
}
```

...which means it is true
at this green point
when the loop terminates.



Example #2

```
double power(double x, int p)
```

- Returns `x` to the power `p`.
- Requires:

  p > 0

- Ensures:

  power = x^(p)

Example #2: Method Body

```
double result = 1.0;
double factor = x;
int pLeft = p;
/**
 * @updates result, factor, pLeft
 * @maintains
 * pLeft >= 0  and
 * result * factor^(pLeft) = x^(p)
 * @decreases
 * pLeft
 */
while (pLeft > 0) {
   ...
}
return result;
```

	x = 3.0
	p = 5
	result = 1.0
	factor = 3.0
	pLeft = 5

/** * @maintains * pLeft >= 0 and * result * factor^(pLeft) = x^(p) */ while (pLeft > 0) {	
...	
}	
	x = 3.0 p = 5 result = factor = pLeft =

What are the values of the other variables here?

What Loop Body Would Work?

- Observation: `pLeft` is positive at the start of the loop body, and the loop body has to decrease it
- How could you decrease `pLeft`?

Idea 1: Decrement pLeft

```
/**
  * @updates result, factor, pLeft
  * @maintains
  * pLeft >= 0  and
  * result * factor^(pLeft) = x^(p)
  * @decreases
  * pLeft
  */
 while (pLeft > 0) {
   ...
   pLeft--;
 }
```

The Rest of the Loop Body

- This is true at the start of the loop body (for each clause: why?):

```
pLeft >= 0  and
result * factor^(pLeft) = x^(p)  and
pLeft > 0
```

- This has to be true at the end of the loop body (for each clause: why?):

```
pLeft - 1 >= 0  and
result * factor^(pLeft - 1) = x^(p)
```

The Rest of

Since x and p do not change in the loop (why?), the two circled expressions must be equal at the end of the loop body.

- This is true at the (why?):

```
pLeft >= 0  and
result * factor^(pLeft) = x^(p)  and
pLeft > 0
```

- This has to be true at the end of the loop body (why?):

```
pLeft - 1 >= 0  and
result * factor^(pLeft - 1) = x^(p)
```

The Rest of the Loop Body

- We need to update `result` from $\mathtt{result}_i$ to $\mathtt{result}_f$, and/or update `factor` from $\mathtt{factor}_i$ to $\mathtt{factor}_f$, to make this true:

$$\mathtt{result}_i \texttt{ * } \mathtt{factor}_i\texttt{\^{}(pLeft) =}$$
$$\mathtt{result}_f \texttt{ * } \mathtt{factor}_f\texttt{\^{}(pLeft - 1)}$$

- How could you do that?

The Rest of the Loop Body

- We need to update `result` from $result_i$ to $result_f$, and/or update `factor` from $factor_i$ to $factor_f$, to make this true:

$$result_i \; * \; factor_i{}^\wedge(pLeft) \; = \\ result_f \; * \; factor_f{}^\wedge(pLeft - 1)$$

- How could you do that?

One line of code that updates `result`:

```
result *= factor;
```

Idea 2: Halve pLeft

```
/**
 * @updates result, factor, pLeft
 * @maintains
 * pLeft >= 0  and
 * result * factor^(pLeft) = x^(p)
 * @decreases
 * pLeft
 */
while (pLeft > 0) {
  ...
  pLeft /= 2;
}
```

The Rest of the Loop Body

- This is true at the start of the loop body (for each clause: why?):

```
pLeft >= 0  and
result * factor^(pLeft) = x^(p)  and
pLeft > 0
```

- This has to be true at the end of the loop body (for each clause: why?):

```
pLeft/2 >= 0  and
result * factor^(pLeft/2) = x^(p)
```

The Rest of the Loop Body

- We need to update `result` from $result_i$ to $result_f$, and/or update `factor` from $factor_i$ to $factor_f$, to make this true:

$$result_i * factor_i\text{^}(pLeft) = result_f * factor_f\text{^}(pLeft/2)$$

- How can you do that?
  - Remember: `pLeft` may be even or odd, but start with the simpler case where it is even