252-0027
Einführung in die Programmierung
Übungen

Invarianten

Henrik Pätzold
Departement Informatik
ETH Zürich

Warm Up – Hoare Tripel

Gültig/Ungültig

Hoare Tripel – Prüfungsbeispiel HS18

```
{ b > c }
 if (x > b) {
     a = x;
} else {
     a = b;
}
{ a > c }
```

Hoare Tripel – Prüfungsbeispiel HS18

```
{ true }
 if (x > y) {
     y = x;
} else {
     y = -x;
}
{ y >= x}
```

Hoare Tripel – Prüfungsbeispiel HS18

```
{ x > 0 }
    y = x * x;
    z = y / 2;
{ z > 0 }
```

Warm Up – Finde die Weakest Precondition

Prüfungsbeispiel HS21

```
1. WP: {                                                    }

   k = m * 3;
   Q: { k > 0 }
```

Prüfungsbeispiel HS21

```
1. WP: { 3*m > 0                                          }



   k = m * 3;
   Q: { k > 0 }
```

Prüfungsbeispiel HS21

```
2. WP: {                                          }

   x = y * 2;
   x = x + 1;
   Q: {x > 2 }
```

Prüfungsbeispiel HS21

2. WP: {2*y > 1 }

```
x = y * 2;    {2*y > 1}
x = x + 1;    {x > 1}
Q: {x > 2 }
```

Prüfungsbeispiel HS21

```
3. WP: {                                                          }


   if (a > b) {
     c = (-2) * a;
   } else {
     c = a + 4;
   }
   Q: { c > 0 }
```

Prüfungsbeispiel HS21

```
3. WP: { (a>b => -2*a > 0) && (a<=b => a+4 > 0)          }

         (a>b && -2*a > 0) || (a<=b && a+4 > 0)

   if (a > b) {
     c = (-2) * a; {(-2)*a > 0}
   } else {
     c = a + 4;    {a+4 > 0}
   }
   Q: { c > 0 }
```

$$
\begin{aligned}
(b \implies R_1) \land (\neg b \implies R_2) &\equiv (\neg b \lor R_1) \land (\neg\neg b \lor R_2) \\
&\equiv (\neg b \lor R_1) \land (b \lor R_2) \\
&\equiv ((\neg b \lor R_1) \land b) \lor ((\neg b \lor R_1) \land R_2)) \\
&\equiv ((\neg b \land b) \lor (R_1 \land b)) \lor ((\neg b \land R_2) \lor (R_1 \land R_2)) \\
&\equiv (\bot \lor (R_1 \land b)) \lor ((\neg b \land R_2) \lor (R_1 \land R_2)) \\
&\equiv (R_1 \land b) \lor (\neg b \land R_2) \lor (R_1 \land R_2) \\
&\equiv (R_1 \land b) \lor (\neg b \land R_2) \lor ((R_1 \land R_2) \land \top) \\
&\equiv (R_1 \land b) \lor (\neg b \land R_2) \lor ((R_1 \land R_2) \land (\neg b \lor b)) \\
&\equiv (R_1 \land b) \lor (\neg b \land R_2) \lor ((R_1 \land R_2) \land \neg b) \lor ((R_1 \land R_2) \land b) \\
&\equiv (R_1 \land b) \lor (\neg b \land R_2) \lor ((\neg b \land R_2) \land R_1) \lor ((R_1 \land b) \land R_2) \\
&\equiv (R_1 \land b) \lor ((R_1 \land b) \land R_2) \lor (\neg b \land R_2) \lor ((\neg b \land R_2) \land R_1) \\
&\equiv ((R_1 \land b) \land (\top \lor R_2)) \lor ((\neg b \land R_2) \land (\top \land R_1)) \\
&\equiv (R_1 \land b) \lor (\neg b \land R_2) \\
&\equiv (b \land R_1) \lor (\neg b \land R_2)
\end{aligned}
$$

Loop - Invarianten

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

Loop-Invarianten

1. Die Loop-Invariante ist **vor der Schleife** erfüllt.

2. Die Loop-Invariante ist **nach jeder Ausführung** des while – body erfüllt.

3. Die Loop-Invariante ist **nach der Schleife** erfüllt.

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

Loop-Invarianten

1. Die Loop-Invariante ist **vor der Schleife** erfüllt.
2. Die Loop-Invariante ist **nach jeder Ausführung** des while – body erfüllt.
3. Die Loop-Invariante ist **nach der Schleife** erfüllt.

Wir wollen das folgende Hoare Triple beweisen:

```
{ Precondition }
  while ( Condition ) { Body };
{ Postcondition }
```

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

Wir wollen das folgende Hoare Triple beweisen:

```
{ Precondition }
  while ( Condition ) { Body };
{ Postcondition }
```

Dies können wir tun, falls eine Invariante existiert, für welche folgendes gilt:

1. Precondition ⇒ Invariante

2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel.

3. ¬ Condition ∧ Invariante ⇒ Postcondition

```
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	res	x

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1
2	b + 2	a - 2

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1
2	b + 2	a - 2
3	b + 3	a - 3

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1
2	b + 2	a - 2
3	b + 3	a - 3
…	…	…

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1
2	b + 2	a - 2
3	b + 3	a - 3
…	…	…
i	b + i	a - i

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1
2	b + 2	a - 2
3	b + 3	a - 3
…	…	…
i	b + i	a - i

`LI: res == b + a - x`

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

1. Precondition ⇒ Invariante
2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel.
3. ¬ Condition ∧ Invariante ⇒ Postcondition

LI: res == b + a - x

Reicht das?

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

1. Precondition ⇒ Invariante ✓
2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel. ✓
3. ¬ Condition ∧ Invariante ⇒ Postcondition ✗

LI: res == b + a - x

Reicht das?

```java
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
      x = x - 1;
      res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}
```

1. Precondition ⇒ Invariante ✓
2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel. ✓
3. ¬ Condition ∧ Invariante ⇒
   Postcondition ✓

```
res == b + a - x && x >= 0
```

Beispiel 2

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	result	counter

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	n

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	n
1	n	n – 1

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	n
1	n	n – 1
2	n * (n – 1)	n – 2

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	n
1	n	n – 1
2	n * (n – 1)	n – 2
3	…	n – 3

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	n
1	n	n – 1
2	n * (n – 1)	n – 2
3	…	n – 3
…	…	…

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	n
1	n	n – 1
2	n * (n – 1)	n – 2
3	…	n – 3
…	…	…
i	n! / (n – i)!	n – i

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}
```

Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	n
1	n	n – 1
2	n * (n – 1)	n – 2
3	…	n – 3
…	…	…
i	n! / (n – i)!	n – i

```
LI: result == n! / counter!
```

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}
```

1. Precondition ⇒ Invariante ✓
2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel. ✓
3. ¬ Condition ∧ Invariante ⇒ Postcondition ✗

LI: result == n! / counter!

Reicht das?

```java
public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}
```

1. `Precondition ⇒ Invariante` ✓
2. `{ Condition ∧ Invariante }`
   `Body;`
   `{ Invariante }` ist ein valides Tripel. ✓
3. `¬ Condition ∧ Invariante ⇒ Postcondition` ✓

```
LI: result == n! / counter! &&
    counter >= 0
```

```java
public int compute(int v, int n) {
    // Precondition: n >= 0
    int v = 2;
    int x;
    int tmp;

    x = 1;
    tmp = 1;
    // Loop-Invariante: ??
    while (x <= n) {
        tmp = tmp * v;
        x = x + 1;
    }
    // Postcondition: tmp == 2^n
    return tmp;
}
```

Wir wollen das folgende Hoare Triple beweisen:

```
{ Precondition }
  while ( Condition ) { Body };
{ Postcondition }
```

Dies können wir tun, falls eine Invariante existiert, für welche folgendes gilt:

1. Precondition ⇒ Invariante

2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel.

3. ¬ Condition ∧ Invariante ⇒ Postcondition

```java
public int compute(int v, int n) {
    // Precondition: n >= 0
    int v = 2;
    int x;
    int tmp;

    x = 1;
    tmp = 1;
    // Loop-Invariante: ??
    while (x <= n) {
        tmp = tmp * v;
        x = x + 1;
    }
    // Postcondition: tmp == 2^n
    return tmp;
}
```

1. Precondition ⇒ Invariante
2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel.
3. ¬ Condition ∧ Invariante ⇒ Postcondition

```
LI: v == 2 && tmp == 2^(x - 1)
    && x <= n + 1 && x >= 1
```

Rezept für Loop-Invarianten

Es gibt keine perfekten Rezepte!

- Loop-Invarianten richtig lösen können ist eine **Frage der Übung**.
- Das folgende Rezept hat sich in der Vergangenheit bei vielen Studenten als nützlich erwiesen.

```java
public int compute(int n){
    // Precondtion: n >= 0
    int k = 5;
    int i = 0;
    int result = 1;

    // Loop-Invariante: ??
    while (i < n) {
        result = result * k;
        i = i + 1;
    }
    //Postcondition result == 5^n
    return result;
}
```

1. **Loop Condition und Terminierung kombinieren.**

$$\{i <= n\}$$

2. **Postcondition und Loop Body kombinieren.**

$$\{result == 5^n\} \rightarrow \{result == 5^i\}$$

3. **Conditions wegen benutzter Methoden oder mathematischer Formeln.**

$$\{k == 5\}$$

```
{ i <= n && result == k^i && k == 5 }
```

```
public int compute(int n) {
  // Precondition:  n >= 0
  int x;
  int res;

  x = 1;
  res = 0;

  // Loop Invariante:
  while (x <= n) {
    res = res + 2 * x;
    x = x + 1;
  }
  // Postcondition:  res == n * (n + 1)
  return res;
}
```

1. Loop Condition und Terminierung kombinieren.

$$\{x <= n + 1\}$$

2. **Postcondition und Loop Body kombinieren.**

$$\{res == (x - 1) * x$$

3. **Conditions wegen benutzter Methoden oder mathematischer Formeln.**

$$x \geq 1$$

Bitte geben Sie die Loop Invariante an.

Loop Invariante: {1 <= x && x <= n + 1 && res == (x - 1) * x}

Übung 8 – Relevanteste Aufgaben (A.o.G)

- Aufgabe 1 (!!)
- Aufgabe 2 (!!!)
- Aufgabe 3 (!!)
- Bonus (!!!)