

# Hoare Logic

Nico Roos

Data Science and Knowledge Engineering, Maastricht University



# Outline

Hoare Logic and PDL

Inference rules for Hoare Logic

Loop Invariants and Program Correctness

Practical use of Hoare Logic



# Hoare Logic

Hoare Logic can be viewed as a predecessor of PDL specially aimed at proving program correctness.



# Hoare Logic

Hoare Logic can be viewed as a predecessor of PDL specially aimed at proving program correctness.

Hoare Logic allows you to specify for each block of code  $\alpha$  of a programming language a pre-condition  $\varphi$  and a post-condition  $\psi$ .

$$\{\varphi\} \alpha \{\psi\}$$

# Hoare Logic

Hoare Logic can be viewed as a predecessor of PDL specially aimed at proving program correctness.

Hoare Logic allows you to specify for each block of code  $\alpha$  of a programming language a pre-condition  $\varphi$  and a post-condition  $\psi$ .

$$\{\varphi\} \alpha \{\psi\}$$

If the pre-condition  $\varphi$  holds before executing  $\alpha$ , then the post-condition  $\psi$  holds after executing  $\alpha$ .



# Hoare Logic

Some examples:

$\{x = 1\} \ x = x + 1 \ \{x = 2\}$  is a **correct** statement.



# Hoare Logic

Some examples:

$\{x = 1\} \ x = x + 1 \ \{x = 2\}$  is a **correct** statement.

$\{x = 1\} \ x = x + 1 \ \{x = 3\}$  is an **incorrect** statement.



# Hoare Logic

Some examples:

$\{x = 1\} \ x = x + 1 \ \{x = 2\}$  is a **correct** statement.

$\{x = 1\} \ x = x + 1 \ \{x = 3\}$  is an **incorrect** statement.

$\{x = 1, y = 5\} \ x = x + 1 \ \{x = 2, y = 5\}$  is a **correct** statement.





# Hoare Logic and PDL

A triple

$$\{\varphi\} \alpha \{\psi\}$$

in Hoare Logic is equivalent to

$$\varphi \rightarrow [\alpha]\psi$$

in PDL

# Hoare Logic and PDL

A triple

$$\{\varphi\} \alpha \{\psi\}$$

in Hoare Logic is equivalent to

$$\varphi \rightarrow [\alpha]\psi$$

in PDL

Standard Hoare Logic can only provide partial correctness proofs. We cannot say anything about the termination of a block of code  $\alpha$ , while in PDL, we can:

$$\varphi \rightarrow \langle \alpha \rangle \top$$



# Hoare Logic and PDL

A triple

$$\{\varphi\} \alpha \{\psi\}$$

in Hoare Logic is equivalent to

$$\varphi \rightarrow [\alpha]\psi$$

in PDL

Standard Hoare Logic can only provide partial correctness proofs. We cannot say anything about the termination of a block of code  $\alpha$ , while in PDL, we can:

$$\varphi \rightarrow \langle \alpha \rangle \top$$

Note that PDL does not and cannot provide a solution for the **halting problem**.



# Partial correctness example

The triple

$$\{x = 1\} \text{ while } (x > 0) \{ x = x + 1 \} \{x = 0\}$$

is a **correct** statement.



# Partial correctness example

The triple

$$\{x = 1\} \text{ while } ( x > 0 ) \{ x = x + 1 \} \{x = 0\}$$

is a **correct** statement.

The execution never terminates if the precondition  $x = 1$  holds before the execution of the while-loop.



# Auxiliary variables

We use **auxiliary variables** to remember the values of variables before the execution of a block of code.

$$\{x = u, y = v\} \quad x = x + 1 \quad \{x = u + 1, y = v\}$$



# Auxiliary variables

We use **auxiliary variables** to remember the values of variables before the execution of a block of code.

$$\{x = u, y = v\} \quad x = x + 1 \quad \{x = u + 1, y = v\}$$

**auxiliary variables** never change their value!



# Assignment rule

$$\frac{}{\{\varphi[e/x]\} \ x = e \ \{\varphi\}}$$

Here  $e$  denotes an expression and  $\varphi[e/x]$  denotes that the expression  $e$  is substituted for the variable  $x$  in  $\varphi$ .

The substitution in the pre-condition guarantees that the expression  $e$  is evaluated using the values of the variables before executing the assignment  $x = e$ .



# Assignment rule

$$\frac{}{\{\varphi[e/x]\} \ x = e \ \{\varphi\}}$$

Here  $e$  denotes an expression and  $\varphi[e/x]$  denotes that the expression  $e$  is substituted for the variable  $x$  in  $\varphi$ .

The substitution in the pre-condition guarantees that the expression  $e$  is evaluated using the values of the variables before executing the assignment  $x = e$ .

For example:

$$\{(x + 1) = 2\} \ x = x + 1 \ \{x = 2\}$$



# Assignment rule

$$\frac{}{\{\varphi[e/x]\} \ x = e \ \{\varphi\}}$$

Here  $e$  denotes an expression and  $\varphi[e/x]$  denotes that the expression  $e$  is substituted for the variable  $x$  in  $\varphi$ .

The substitution in the pre-condition guarantees that the expression  $e$  is evaluated using the values of the variables before executing the assignment  $x = e$ .

For example:

$$\begin{aligned} & \{(x + 1) = 2\} \ x = x + 1 \ \{x = 2\} \\ & \Rightarrow \{x = 1\} \ x = x + 1 \ \{x = 2\} \end{aligned}$$

# Assignment rule

Another example:

$$\{(x + y) = y + 2\} \ x = x + y \ \{x = y + 2\}$$



# Assignment rule

Another example:

$$\begin{aligned} &\{(x + y) = y + 2\} \ x = x + y \ \{x = y + 2\} \\ \Rightarrow &\{x = 2\} \ x = x + y \ \{x = y + 2\} \end{aligned}$$



# Assignment rule

Another example:

$$\begin{aligned} &\{(x + y) = y + 2\} \ x = x + y \ \{x = y + 2\} \\ \Rightarrow &\{x = 2\} \ x = x + y \ \{x = y + 2\} \end{aligned}$$

Note that practically, we substitute  $x = 2$  for  $x$  on the righthand side of the assignment  $x = x + y$  and next interpret the assignment symbol  $=$  as the mathematical equal symbol  $=$ .



# Assignment rule

The substitution on the pre-condition may seem odd.



# Assignment rule

The substitution on the pre-condition may seem odd.

Substitution in the post-condition does not work because we need the values before the execution of the assignment.



# Assignment rule

The substitution on the pre-condition may seem odd.

Substitution in the post-condition does not work because we need the values before the execution of the assignment.

The rule:

$$\frac{}{\{\varphi\} \text{ x = e } \{\varphi[e/x]\}}$$

can give the following **incorrect** result:

$$\{\text{x} = 0\} \text{ x = 1 } \{\text{1} = 0\}$$





# Consequence rule

$$\frac{\varphi' \rightarrow \varphi, \{\varphi\} \alpha \{\psi\}, \psi \rightarrow \psi'}{\{\varphi'\} \alpha \{\psi'\}}$$

The rule states that we can strengthen the pre-condition and that we can weaken the post-condition.

# Consequence rule

$$\frac{\varphi' \rightarrow \varphi, \{\varphi\} \alpha \{\psi\}, \psi \rightarrow \psi'}{\{\varphi'\} \alpha \{\psi'\}}$$

The rule states that we can strengthen the pre-condition and that we can weaken the post-condition.

For example, from:

$$\{x = 1\} \ x = x + 1 \ \{x = 2\}$$

we can derive:

$$\{x = 1, y = 5\} \ x = x + 1 \ \{x > 1\}$$



# Sequence rule

$$\frac{\{\varphi\} \alpha \{\psi\}, \{\psi\} \beta \{\eta\}}{\{\varphi\} \alpha; \beta \{\eta\}}$$

# Sequence rule

$$\frac{\{\varphi\} \alpha \{\psi\}, \{\psi\} \beta \{\eta\}}{\{\varphi\} \alpha; \beta \{\eta\}}$$

For example, from:

$$\begin{array}{l} \{x = 1\} x = x + 1 \{x = 2\} \\ \{x = 2\} x = x + 1 \{x = 3\} \end{array}$$

we can derive:

$$\{x = 1\} x = x + 1; x = x + 1 \{x = 3\}$$

# Conditional rule

$$\frac{\{\varphi \wedge \theta\} \alpha \{\psi\}, \{\varphi \wedge \neg\theta\} \beta \{\psi\}}{\{\varphi\} \text{ if } (\theta) \{ \alpha \} \text{ else } \{ \beta \} \{\psi\}}$$

# Conditional rule

$$\frac{\{\varphi \wedge \theta\} \alpha \{\psi\}, \{\varphi \wedge \neg\theta\} \beta \{\psi\}}{\{\varphi\} \text{ if } (\theta) \{ \alpha \} \text{ else } \{ \beta \} \{\psi\}}$$

For example, from:

$$\begin{aligned} \{x \geq y\} z = x \{z = \max(x, y)\} \\ \{x < y\} z = y \{z = \max(x, y)\} \end{aligned}$$

we can derive:

$$\{\} \text{ if } (x \geq y) \{ z = x \} \text{ else } \{ z = y \} \{z = \max(x, y)\}$$



# While rule

$$\frac{\{\phi \wedge \psi\} \alpha \{\psi\}}{\{\psi\} \text{while } (\phi) \{ \alpha \} \{\neg\phi \wedge \psi\}}$$

# While rule

$$\frac{\{\phi \wedge \psi\} \alpha \{\psi\}}{\{\psi\} \text{while } (\phi) \{ \alpha \} \{\neg\phi \wedge \psi\}}$$

For example, using the invariant:

$$\{r = x \times i\} r = r + x; i = i + 1 \{r = x \times i\}$$

we can derive:

$$\{r = x \times i\} \text{while } (i \neq y) \{ r = r + x; i = i + 1 \} \{r = x \times y\}$$



# While rule

$$\frac{\{\phi \wedge \psi\} \alpha \{\psi\}}{\{\psi\} \text{while } (\phi) \{ \alpha \} \{\neg\phi \wedge \psi\}}$$

For example, using the invariant:

$$\{r = x \times i\} r = r + x; i = i + 1 \{r = x \times i\}$$

we can derive:

$$\{r = x \times i\} \text{while } (i \neq y) \{ r = r + x; i = i + 1 \} \{r = x \times y\}$$

We can guarantee the pre-condition with the assignments:

$$r = 0; i = 0$$



# Conjunction and Disjunction rule

$$\frac{\{\varphi\} \alpha \{\psi\}, \{\varphi\} \alpha \{\psi'\}}{\{\varphi\} \alpha \{\psi \wedge \psi'\}}$$

# Conjunction and Disjunction rule

$$\frac{\{\varphi\} \alpha \{\psi\}, \{\varphi\} \alpha \{\psi'\}}{\{\varphi\} \alpha \{\psi \wedge \psi'\}}$$

$$\frac{\{\varphi\} \alpha \{\psi\}, \{\varphi'\} \alpha \{\psi\}}{\{\varphi \vee \varphi'\} \alpha \{\psi\}}$$

# Conjunction and Disjunction rule

$$\frac{\{\varphi\} \alpha \{\psi\}, \{\varphi\} \alpha \{\psi'\}}{\{\varphi\} \alpha \{\psi \wedge \psi'\}}$$

$$\frac{\{\varphi\} \alpha \{\psi\}, \{\varphi'\} \alpha \{\psi\}}{\{\varphi \vee \varphi'\} \alpha \{\psi\}}$$

These rules are redundant but convenient to have when proving the correctness of an algorithm.

# Example

A vase contains 35 white pebbles and 35 black pebbles. Proceed as follows to draw pebbles from the vase, as long as this is possible.

- ▶ Every round, draw two pebbles from the vase.
- ▶ If they have the same colour, then put a black pebble into the vase (you may assume that there are enough additional black pebbles outside of the vase).
- ▶ If they have different colours, then put the white pebble back.

In every round one pebble is removed from the vase, so after 69 rounds there is a single pebble left. What is the colour of this pebble?

How to answer this? Do we even have enough information?



# Example

The effect of drawing two pebbles:

Pebbles Drawn	White Pebbles	Black Pebbles
white white	-2	+1
white black	0	-1
black white	0	-1
black black	0	-1

Notice that the following properties hold:

- ▶ Initially, the number of white pebbles is odd.
- ▶ If the number of white pebbles is odd before drawing, then the number of white pebbles is odd after drawing.

Hence: if there is one pebble left, it must be white, for if it were black, there would have been an even number of white pebbles.



# Loop Invariants

The property  $\psi$ :

$\psi = \textit{The number of white pebbles is odd}$

is called a *loop invariant* for this scenario.

Characteristics of  $\psi$ :

- ▶  $\psi$  holds initially (before executing the loop)
- ▶ if  $\psi$  holds before executing the loop body then  $\psi$  holds after executing the loop body
- ▶  $\psi$ , together with the fact that the loop has ended, implies what we want to prove (i.e., the last pebble is white)



# Example

Consider the following function that computes the square of  $n$ :

```
int square (int n) {  
    int x = 0;  
    int k = 0;  
    while (k < n) {  
        x = x + 2*k + 1;  
        k = k + 1;  
    }  
    return x;  
}
```

How do we prove that it indeed returns  $n^2$ ?





# Example

We need to find the loop invariant  $\psi$  such that:

- ▶  $\psi$  is true before the *while* loop starts.
- ▶ The following Hoare triple is valid:

$$\{(k < n) \wedge \psi\} \ x = x + 2*k + 1; \ k = k + 1; \ \{\psi\} \quad (1)$$

- ▶  $\neg(k < n) \wedge \psi$  implies  $x = n^2$ , so that we can use (1) together with the rule

$$\frac{\{\phi \wedge \psi\} \alpha \ \{\psi\}}{\{\psi\} \ \text{while} \ (\phi) \ \{ \alpha \} \ \{\neg\phi \wedge \psi\}}$$

to derive

$$\{\psi\} \ \text{while} \ (k < n) \ \text{do} \ x = x + 2*k + 1; \ k = k + 1; \ \{\neg(k < n) \wedge \psi\}$$

which then implies that  $x = n^2$  after executing the loop.



# Example

Invariant:  $\psi = (x = k^2) \wedge (k \leq n)$

Justification:

- ▶  $\psi$  is true after `int x = 0; int k = 0;`
- ▶ The following Hoare triple is true

$$\{k < n \wedge \psi\} \ x = x + 2*k + 1; \ k = k + 1; \ \{\psi\}$$

- ▶ We can derive:

$$\{\psi\} \text{ while } (k < n) \text{ do } \{ \ x = x + 2*k + 1; \ k = k + 1; \ } \ \{(k = n) \wedge \psi\}$$

- ▶  $\neg(k < n) \wedge \psi$  implies  $x = n^2$



# Exercise 1

Consider the following function that computes the sum of the elements in an array:

```
int sum(int[] a) {  
    int s = 0;  
    int i = 0;  
    while (i != a.length) {  
        s = s + a[i];  
        i = i + 1;  
    }  
    return s;  
}
```

Prove that the algorithm returns sum of the elements in the array.

# Exercise 1

Loop invariant:  $\psi = (s = \sum_{n \in \{0, \dots, i-1\}} a[n])$ .

Justification:

►  $\psi$  is true after `int s = 0; int i = 0;`



# Exercise 1

Loop invariant:  $\psi = (s = \sum_{n \in \{0, \dots, i-1\}} a[n])$ .

Justification:

- ▶  $\psi$  is true after `int s = 0; int i = 0;`
- ▶ The following Hoare triple is true

$$\{(i < a.length) \wedge \psi\} \text{ s = s + a[i]; i = i + 1; } \{\psi\}$$



# Exercise 1

Loop invariant:  $\psi = (s = \sum_{n \in \{0, \dots, i-1\}} a[n])$ .

Justification:

- ▶  $\psi$  is true after `int s = 0; int i = 0;`
- ▶ The following Hoare triple is true

$$\{(i < a.length) \wedge \psi\} \text{ s = s + a[i]; i = i + 1; } \{\psi\}$$

- ▶ From this we can derive:

$$\{\psi\} \text{ while (i != a.length) } \{\dots\} \{\neg(i \neq a.length) \wedge \psi\}$$



# Exercise 1

Loop invariant:  $\psi = (s = \sum_{n \in \{0, \dots, i-1\}} a[n])$ .

Justification:

- ▶  $\psi$  is true after `int s = 0; int i = 0;`
- ▶ The following Hoare triple is true

$$\{(i < a.length) \wedge \psi\} \quad s = s + a[i]; \quad i = i + 1; \quad \{\psi\}$$

- ▶ From this we can derive:

$$\{\psi\} \quad \text{while } (i \neq a.length) \{ \dots \} \quad \{\neg(i \neq a.length) \wedge \psi\}$$

- ▶  $\neg(i \neq a.length) \wedge \psi$  implies

$$s = \sum_{n \in \{0, \dots, a.length-1\}} a[n].$$



## Exercise 2

Consider the following function that returns the factorial of  $n$ :

```
int Factorial (int n) {  
    int f = 1;  
    int i = 1;  
    while (i < n) {  
        i = i + 1;  
        f = f * i;  
    }  
    return f;  
}
```

Prove that the algorithm returns  $n!$ .





## Exercise 2

Loop invariant:  $\psi = (f = i!) \wedge i \leq n$

Justification:

►  $\psi$  is true after `int f = 1; int i = 1;`



## Exercise 2

Loop invariant:  $\psi = (f = i!) \wedge i \leq n$

Justification:

- ▶  $\psi$  is true after `int f = 1; int i = 1;`
- ▶ The following Hoare triple is true

$$\{(i < n) \wedge \psi\} \text{ i = i + 1; f = f * i; } \{\psi\}$$

## Exercise 2

Loop invariant:  $\psi = (f = i!) \wedge i \leq n$

Justification:

- ▶  $\psi$  is true after `int f = 1; int i = 1;`
- ▶ The following Hoare triple is true

$$\{(i < n) \wedge \psi\} \ i = i + 1; f = f * i; \ \{\psi\}$$

- ▶ From this we can derive:

$$\{\psi\} \text{ while } (i < n) \text{ do } \{ i = i + 1; f = f * i; \} \ \{\neg(i < n) \wedge \psi\}$$

## Exercise 2

Loop invariant:  $\psi = (f = i!) \wedge i \leq n$

Justification:

- ▶  $\psi$  is true after `int f = 1; int i = 1;`
- ▶ The following Hoare triple is true

$$\{(i < n) \wedge \psi\} \text{ i = i + 1; f = f * i; } \{\psi\}$$

- ▶ From this we can derive:

$$\{\psi\} \text{ while (i < n) do } \{ \text{ i = i + 1; f = f * i; } \} \{ \neg(i < n) \wedge \psi \}$$

- ▶  $\neg(i < n) \wedge \psi$  implies  $f = n!$ .



## Exercise 2

Loop invariant:  $\psi = (f = i!) \wedge i \leq n$

Justification:

- ▶  $\psi$  is true after `int f = 1; int i = 1;`
- ▶ The following Hoare triple is true

$$\{(i < n) \wedge \psi\} \ i = i + 1; f = f * i; \ \{\psi\}$$

- ▶ From this we can derive:

$$\{\psi\} \text{ while } (i < n) \text{ do } \{ i = i + 1; f = f * i; \} \ \{\neg(i < n) \wedge \psi\}$$

- ▶  $\neg(i < n) \wedge \psi$  implies  $f = n!$ .
- ▶ Note that here we need  $i \leq n$  as a conjunct of  $\psi$ , since  $i \leq n$  together with  $\neg(i < n)$  implies  $i = n$ .



# Program correctness

Hoare logic can be used to prove partial program correctness.



# Program correctness

Hoare logic can be used to prove partial program correctness.

Note that full correctness requires addressing the termination of a program, which is not address in standard Hoare Logic.



# Program correctness

Hoare logic can be used to prove partial program correctness.

Note that full correctness requires addressing the termination of a program, which is not address in standard Hoare Logic.

There are tools that support checking the correctness of a Java program. See for instance the [Java Modeling Language](#) (JML).





# Design by contract

Design by contract is a more pragmatic way of using Hoare Logic.



# Design by contract

**Design by contract** is a more pragmatic way of using Hoare Logic.

**Design by contract** consists of:

- ▶ specifying pre-conditions and post conditions for every method, function and procedure,
- ▶ specifying invariants for classes.



# Design by contract

**Design by contract** is a more pragmatic way of using Hoare Logic.

**Design by contract** consists of:

- ▶ specifying pre-conditions and post conditions for every method, function and procedure,
- ▶ specifying invariants for classes.

**Class invariants** are properties of a class that must hold before and after the execution of a method.



# Design by contract

**Design by contract** is a more pragmatic way of using Hoare Logic.

**Design by contract** consists of:

- ▶ specifying pre-conditions and post conditions for every method, function and procedure,
- ▶ specifying invariants for classes.

**Class invariants** are properties of a class that must hold before and after the execution of a method.

Programming language such as **Eiffel** support **design by contract**.



# Design by contract

**Design by contract** is a more pragmatic way of using Hoare Logic.

**Design by contract** consists of:

- ▶ specifying pre-conditions and post conditions for every method, function and procedure,
- ▶ specifying invariants for classes.

**Class invariants** are properties of a class that must hold before and after the execution of a method.

Programming language such as **Eiffel** support **design by contract**.

The **Java Modeling Language** (JML) enable the use of **design by contract** in Java programming.



# Loop invariants

Specifying **loop invariants** for (every) loop in your program is a pragmatic way to improve correctness of your program



# Loop invariants

Specifying **loop invariants** for (every) loop in your program is a pragmatic way to improve correctness of your program



# Example

```
{n > 1}                                     ⇐ pre-condition
int Factorial (int n) {
  int f = 1;
  int i = 1;
  {1 < i ∧ i ≤ n ∧ f = i!}                 ⇐ loop invariant
  while (i < n) {
    i = i + 1;
    f = f * i;
  }
  return f;
}
{f = n!}                                   ⇐ post-condition
```

Note that  $i! == (i-1)! * i$ .





# Example

Formulated in JML:

```
/*@ requires 1 <= n ;  
ensures \result == (\product int i; 1<=i && i<=n; i);  
@*/  
int Factorial (int n) {    int f = 1;  
    int i = 1;  
    /*@ loop_invariant i<=n &&  
f==(\product int j; 1<=j && j<=i; j); @*/  
    while (i < n) {  
        i = i + 1;  
        f = f * i;  
        return f;  
    }  
}
```



# Sorting example

```
public class Sort {
    public int[] a;

    /*@ public normal_behavior
       @ requires a.length > 0 && 0 <= start && start < a.length;
       @ ensures (\forall int i; start <= i && i < a.length; a[\result] >= a[i]);
       @ ensures start <= \result && \result < a.length;
    @*/
    int /*@ strictly_pure @*/ max(int start) {
        int counter = start;
        int idx = start;
        /*@ loop_invariant start <= counter && counter <= a.length &&
           @ start <= idx && idx < a.length && start < a.length &&
           @ (\forall int x; x >= start && x < counter; a[idx] >= a[x]);
           @ assignable \strictly_nothing;
           @ decreases a.length - counter;
        @*/
        while (counter < a.length) {
            if (a[counter] > a[idx])
                idx = counter;
            counter = counter + 1;
        }
        return idx;
    }
}
```

# Sorting example

```

/*@ public normal_behavior
   @ requires a.length > 0;
   @ ensures (\forall int i; 0 <= i && i<a.length-1; a[i] >= a[i+1]);
   @*/
void sort() {
  int pos = 0;
  int idx = 0;
  /*@ loop_invariant 0<=pos && pos<=a.length && 0<=idx && idx<a.length
     @ && (\forall int x; x>=0 && x<pos-1; a[x]>=a[x+1]) &&
     @ (pos>0 ==>(\forall int y; y>=pos && y<a.length; a[pos-1]>=a[y]));
     @ assignable a[*];
     @ decreases a.length - pos;
     @*/
  while (pos < a.length-1) {
    idx = max(pos);
    int tmp = a[idx];
    a[idx] = a[pos];
    a[pos] = tmp;
    pos = pos+1;
  }
}

```

# Tools

The **loop invariants**, as well as pre- and post conditions, and class invariants can be checked:

- ▶ during the execution of a program; i.e. runtime testing,
- ▶ before execution of a program; i.e., automated program verification.



# Tools

The **loop invariants**, as well as pre- and post conditions, and class invariants can be checked:

- ▶ during the execution of a program; i.e. runtime testing,
- ▶ before execution of a program; i.e., automated program verification.

Loop invariants, pre- and post conditions, and class invariants can be specified for Java programs as special comments in JML.



# Tools

A program with loop invariants, pre- and post conditions, and class invariants:

- ▶ can run as a normal Java program; i.e., the special comment are ignored,
- ▶ can run as a Java program that performs runtime testing, for instance: [jmlrac](#),
- ▶ can be verified at compile time, for instance: [ESC/Java2](#).

