Structured Programming and Program Verification in Java

werner.hett@hti.bfh.ch
Structured Programming and Program Verification in Java

This script makes references to the following books:


Structured Programming

What is Structured Programming?

The concept of structured programming has been introduced in the 1970s with the aim that a program should be written in such a way that it is both easily understood and free from errors. In recent years, the concept of object-oriented programming has gained a lot of interest, and one could think it has replaced structured programming altogether. However, this is not the case.

The two concepts are complementary in the sense that the principles of structured programming remain important for programming "in the small", while object-oriented programming is used for mastering the complexity involved in developing large program systems. Java is a good example for how the two concepts can be combined in a single programming language.

Structured programming can be said to be a set of rules and recommendations for how ‘good’ programs should be written. One of the most important rules is that there are no arbitrary jumps (no gotos) from one place in a program to another. Instead, structured programming defines a very limited set of ways in which an action can be decomposed into smaller actions.

An action is either

- an elementary (or atomic) operation, or
- a sequence of other actions, or
- a choice between two or more other actions, or
- a repetition of another action

There are two methods to represent structured programs graphically: Michael Jackson diagrams (MJ) and Nassi-Shneiderman diagrams (NS). Both methods guarantee that if a program can be represented graphically according to their rules then the program is necessarily well structured.

Flowcharts can also be used to depict programs. However, flowcharts can represent every sequential program, not just well structured ones. Therefore, we use flowcharts only to explain the details of programming language constructs, but not for algorithm design.
An overview of the graphical elements of the three methods is given in the following table.

<table>
<thead>
<tr>
<th>Action A</th>
<th>Michael Jackson</th>
<th>Nassi-Shneiderman</th>
<th>Flowchart</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action A is the sequence of the actions B, C, and D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Sequence Diagram" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Nassi-Shneiderman Sequence Diagram" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Flowchart Sequence Diagram" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Choice</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action A is either the action B or C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Choice Diagram" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Nassi-Shneiderman Choice Diagram" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Flowchart Choice Diagram" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Repetition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action A is a repetition of action B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Repetition Diagram" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Nassi-Shneiderman Repetition Diagram" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Flowchart Repetition Diagram" /></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Elementary Operations

In Java, there are essentially two kinds of elementary (or atomic) operations: assignment statement (in its various forms) and method invocation.

Examples: $x = y + 13$; System.out.println($x$);

By coincidence, those are the two most prominent statements that must be terminated with a semicolon in Java.

Sequence

In Java, a statement block, or simply block, is the means to express a sequence of actions: multiple statements are grouped within curly brackets or braces { and }.

Example:

```java
{ int x = 14;
 x = x / 2;
 x++;
 System.out.println("x = " + x);
}
```

Choices or Alternatives

Recommended reading: Horstmann: Chapter 5: Decisions

Conditional Actions

If an action $A$ is to be performed only when a certain condition $c$ is satisfied then $A$ is called a conditional action. The situation is depicted as follows:

Michael Jackson

Nassi-Shneiderman

Flowchart

Java:

```java
if (c) A
```

Example:

```java
daysInYear = 365;
if (year % 4 == 0)
daysInYear = 366;
```
Two Alternatives

An action that is a selection between two mutually exclusive alternatives \(A\) and \(B\) can be represented as shown in the following table. A condition \(c\) determines which alternative to choose.

<table>
<thead>
<tr>
<th>Michael Jackson</th>
<th>Nassi-Shneiderman</th>
<th>Flowchart</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>(B)</td>
<td>A</td>
</tr>
<tr>
<td>(Y)</td>
<td>(N)</td>
<td>B</td>
</tr>
<tr>
<td><strong>Java</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{if } (c) A \text{ else } B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: Printing two integer numbers in ascending order

```java
if (a < b)
    System.out.println(a + "," + b);
else
    System.out.println(b + "," + a);
```

Multiple Alternatives

If an action consists of more than two mutually exclusive alternatives then we can use nested two-way alternatives.

Example:

Compute the function \(\text{sign}(x)\)

```java
if (x > 0)
    sign = 1;
else if (x == 0)
    sign = 0;
else
    sign = -1;
```
There is a particular case of nested if statements which has to be handled carefully. Consider the following Java statement:

```java
if (c1) if (c2) A else B
```

Due to the fact that there are two if clauses but only a single else, the statement is ambiguous. It could be understood in two different ways:

- if (c1) { if (c2) A else B } 
- if (c1) { if (c2) A } else B

However, the Java language definition specifies that in this case the first way is the correct one. In other words:

An else clause is bound to the most recent if that does not have an else.

As shown in the interpretation (2) above, curly braces can be used to bind the else clause to the first if. The following drawings should make things clear:

Note that indentation is very important to clarify the situation for the human reader. However, the compiler does not care about indentation!

In some cases, one of several alternatives is selected according to an integer value. Consider, for instance the following program segment where the number of days in a given month is determined:

```java
if (month == 4 || month == 6 || month == 9 || month == 11)
    daysInMonth = 30;
else if (month == 2)
    if (leapYear)
        daysInMonth = 29;
    else
        daysInMonth = 28;
else
    daysInMonth = 31;
```
The same effect can be achieved with the Java \textit{switch} statement:

\begin{verbatim}
switch (month) {
    case 4: case 6: case 9: case 11:
        daysInMonth = 30;
        \textbf{break};
    case 2:
        if (leapYear)
            daysInMonth = 29;
        else
            daysInMonth = 28;
        \textbf{break};
    default:
        daysInMonth = 31;
}
\end{verbatim}

Unfortunately, the syntax of the Java \textit{switch} statement is influenced by the corresponding construct in C and does not guarantee a well-structured program. If you forget a \textit{break} clause then your code becomes an unstructured program!

We conclude the section with a Nassi-Shneiderman diagram for selection among multiple alternatives:

![Nassi-Shneiderman Diagram](image)

\section*{Repetition}

\textbf{Recommended reading:} Horstmann: Chapter 6: Iterations

The real power of computers comes from the fact that they can repeat the same operation over and over again. In Java, there are three ways to program repetition: \textit{while}, \textit{do-while}, and \textit{for}.

\textit{while} Loop

The \textit{while} loop is used to repeat a statement (or block of statements) $A$ as long as a particular condition $c$ is true. The condition $c$ is always checked \textit{before} the statement $A$ is executed. Therefore, if $c$ is false at the beginning, the body of the \textit{while}-loop isn't executed at all. In practice, the \textit{while} loop is typically used when the number of repetitions is not known in advance.
Example: Find the smallest power-of-two, which is greater than or equal to a given positive whole number.

```java
long number;            // given number
long power2 = 1;        // power of 2
while (power2 < number) // power of 2
    power2 = 2 * power2;
```

**do-while Loop**

The do-while loop is useful whenever a statement (or block of statements) is to be repeated at least once. In general, the number of repetitions is not known in advance, and the decision whether or not to terminate the repetition is typically taken after a certain processing step (the loop body) has been executed.
Example: Suppose a program asks the user to enter a positive number. If the user erroneously types in a negative number or zero she should be given another chance. This can be programmed as follows:

```java
long number;
System.out.print("Please enter a positive number > ");
do {
    number = Terminal.readLong(); // this is not standard Java!
    if (number <= 0)
        System.out.print("Oops - try again > ");
} while (number <= 0);
System.out.println("Thank you");
```

while and do-while are Equivalent

In principle, it would not be necessary to have both the while- and the do-while loop, because each of them can replace the other one:

```java
do A while (c); is equivalent to: A; while (c) A
while (c) A is equivalent to: if (c) do A while (c);
```

for Loop

In Java, a for loop is a very versatile construct. It is based on a while loop and can be defined by the following equivalence:

```java
for (I; c; E) A is equivalent to: I; while (c) {A; E}
```

In the above formula the symbols I, c, E, and A have the following meaning:

I: initialisation expression(s), separated by comma

c: loop condition; while c evaluates to true a next iteration is begun

E: increment expression(s), separated by comma

A: loop body

Despite the fact that it is possible to write any loop in the form of a for loop, good programming style dictates that this kind of loop should be used only in cases where the number of iterations is known in advance.

Typically, a for loop is a counting loop where some counter variable is incremented (or decremented) until it reaches some limit. The following diagrams depict such a typical case.
for i = 1 to 10

Java

for (i = 1; i <= 10; i++) A

Example: Calculate the sum of the squares of the numbers 1 to 10:

```java
long sum = 0;
int k;
for (k = 1; k <= 10; k++)
    sum = sum + k*k;
```

Another typical case are so-called *iterators* which are used to process all elements of an object collection.

**Local variables in for loops**

You can declare a local variable in the initialisation section of a *for* loop. Such a variable is then valid only in the body of the loop and within the initialisation, test, and increment expressions of the loop. In the example above, if the variable \( k \) is not used further down in the program, you can write:

```java
long sum = 0;
for (int k = 1; k <= 10; k++)
    sum = sum + k*k;
```

This is not only shorter, but also better programming style. Variable should always be declared such that they have their smallest possible scope.

**Breaking Out of a Loop and Other Bad Things**

It is possible in Java to terminate the execution of a loop abruptly with a *break* statement. The effect is that the program execution is continued immediately after the loop. However, this breaks the rules of structured programming and you should have very good reasons to do it.

From its ancestors in the C world, Java has inherited some evil things, *like labelled statements*, which are the Java variant of the old outlawed *goto*. There is just one rule to those constructs: *don't use them!*
Exercises

A Template for Simple Java Application Programs

You can use the following program as a template for some of the exercises. The program calculates the area of a rectangle when it is given width and length. Compile the program and start it from the command line with two integer numbers as arguments.

```java
/* Given width and length of a rectangle, * compute its area. * @author hans.muster@hti.bfh.ch * Time-stamp: <19-Nov-2004 21:49 hew> */

public class Rectangle {

    static final String USAGE =
        "usage: java Rectangle <a> <b>";

    public static void main (String[] args) {
        int a, b;    // the two input numbers
        int area;    // the area

        try {
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
        } catch (Exception e) {
            System.out.println(USAGE);
            return;
        }

        area = a * b;
        System.out.println("Area = " + area);
    }
}
```

Recommended Exercises

1) Identify and correct the logical and syntactic errors of the following. Explain (in advance) what exactly happens when you try to compile and/or execute the program segments.

a) if (age > 65):
   System.out.println("retired");

b) if (day = 7)
   System.out.println("Sunday");
   else;
   System.out.println("weekday");

c) if (x%7 < 2)
alpha = 25
else
alpha = 31;

d) int n = 21; int k = 1;
while (k < n);
k = 2 * k;

e) int x = 17, sum = 0;
while (x >= 0)
sum = sum + x;
x--;

f) for (int k = 100; k >= 0; x++)
    System.out.println(k);

g) The following should output the odd numbers from 19 to 1.
for (int x = 19, x > 0, x -=2)
    System.out.println(x);

h) The following should output the even numbers from 2 to 100.
counter = 2;
do {
    System.out.println(counter);
counter++;
} while (counter < 100);

i) The following should print whether x is negative, zero, or positive.
switch (Math.abs(x)) {
case -1:
    System.out.println("negative");
case 0:
    System.out.println("zero");
default:
    System.out.println("positive");
}

j) for (float z = 0.1f; z != 1; z = z + 0.1f)
    System.out.println(z);

k) for (int i = 1, k = 1, h = 0; i < 100;
h = i, i += k, k = h)
    System.out.println(i);

2) ThreeNumbers. Given three integer numbers, a Java program should decide whether their sequence is ascending, descending, unsorted, or whether they are all equal.

3) Triangle. Given three positive integers, find out whether they can be the sides of a triangle. If this is the case, determine the type of triangle: equilateral (gleichseitig), isosceles (gleichschenklig), rectangular (rechtwinklig), general = scalene (allgemein = schief).

4) TempConv. Convert temperatures between Celsius, Fahrenheit and Kelvin. The program shall be started with the command:

    java TempConv [option] <temperature>
with following options:
- c the given temperature value is in degrees Celsius
- k the given temperature value is in Kelvin
- f the given temperature value is in degrees Fahrenheit

If no option is given Celsius is assumed as default. The output should always be the temperature values in all three scales.

5) **StarsX.** Write Java programs that print the following patterns:

```
*     *     *     *
   ***   ***   ***
  *****  *****  *****
 ********* ********* *********
```

a) b) c) d) The number of rows of the pattern should be input as command line parameter. The asterisks (*) and the blanks ( ) should be printed by statements of the form `System.out.print('*')` and `System.out.print(' ')`.

6) **DayNumber.** By numbering the days, starting at 1 for the first of January, we can define a day number for each date of the year. Develop a Java program `DayNumber` that calculates this number for a given date.

7) **Ulam.** The American mathematician Ulam is said to have invented an interesting number sequence, which is defined as follows. We start with a given positive integer $a_0$. The successor $a_{k+1}$ of a number $a_k$ is equal to $a_k/2$ if $a_k$ is even, and $3a_k + 1$ if $a_k$ is odd. The sequence ends when $a_n = 1$ is reached. So far, it is not known whether the sequence terminates for every start value $a_0$. Write a Java program `Ulam` which prints the number sequence for a given start value.

8) **EasterDate.** Determine the Easter date for a given year. Easter is always the first Sunday after the first full moon in spring. The following algorithm was invented by the German mathematician C.F.Gauss (1777-1855) to calculate this date. It should work at least for years in the range 1583..2100.

Given the year $n$, we first calculate the following three numbers:

\[
a = n \mod 19, \quad b = n \mod 4, \quad c = n \mod 7.
\]

Furthermore, we calculate:

\[
x = (((n \div 100) - (n \div 400) - (n \div 300) + 15) \mod 30)
\]

\[
y = (((n \div 100) - (n \div 400) + 4) \mod 7)
\]

\[
d = (19 \times a + x) \mod 30
\]

\[
e = (2 \times b + 4 \times c + 6 \times d + y) \mod 7
\]

\[
m = 22 + d + e
\]

$(\text{div}$ stands for integer division and $\text{mod}$ for the division remainder)$

The Easter date is then the $m$'th day of March if $m$ is less than 32, otherwise it's the $(m-31)$th of April. There is an exception to this rules: if $e = 6$ and $d = 29$, or if $e = 6$ and $d = 28$ and $a > 10$. In these cases $m = 15 + d + e$. 

12
Further Exercises

1) **TrianglePlus.** Write a program `TrianglePlus` to solve the following problem:
   Given three positive integers, find out whether they can be the sides of a triangle.
   If this is the case, determine the type of triangle (equilateral, isosceles, rectangular, general).
   In addition, the area should be calculated and, in the case of an isosceles or rectangular triangle, the program should tell, which side is the base or hypotenuse, respectively.

2) **Quadrangle.** Write a program `Quadrangle` to solve the following problem:
   Given four positive integers a, b, c, and d. We want them to be the sides of a quadrangle, in counter-clockwise order. Find out whether the numbers can indeed be the sides of a quadrangle, and if so, what kind of quadrangle they represent: rhombus, parallelogram, or kite (Drachen/cerf-volant).

3) **LengthConv.** Write a program `LengthConv` to convert lengths between feet, kilometers and nautical miles. Use the same technique as in the program `TempConv` above. A nautical mile is 1.852 km and a foot is 0.3048 m.

4) **FuelConv.** Americans measure the fuel consumption of their cars in miles per gallon, Europeans in liters per 100 kilometers. Write a program `FuelConv` to convert between the two measures. One US gallon is 0.2642 liters and a km is 62.137 (statute) miles.

5) **EasterList.** Write a program `EasterList` to determine the Easter dates for a range of years.

6) **GoodFriday.** Write a program `GoodFriday` to calculate all Good Fridays (Karfreitag, vendredi saint) which fall on a 13th of April. Range of years: 1583 .. 2100.

7) **UlamStat.** As an additional task, write a program `UlamStat` that does some statistics of the Ulam sequence. Interesting questions are: What is the value of n for a given start value $a_0$ and what is the largest number in the sequence? The results should be presented as a table for a range of start values.

8) ??? Find out what the following program segment does:

   ```java
   double a;
   // input a
   double x = 1.0;
   for (int k = 0; k < 10; k++)
       x = (x + a/x) / 2;
   ```
Program Verification

Recommended reading: Gersting, Sections 1.6 and 2.3

Program Correctness, Verification and Testing

The most important characteristic of computer programs is their correctness. A program is correct if, given any input variables that satisfy certain specified properties, it produces output values that satisfy other specified properties. Or simply stated: A program is correct if it does what it promises to do.

In practice we further require that the program terminate after a finite number of steps.

Program verification should not be confused with program testing. Program testing seeks to show that particular input values produce acceptable output values; testing is done for a variety of input values. If the test sets are intelligently chosen and the program always gives the right result then we get confident that the program is probably ok. Testing is an important part of program development. And for big programs, there is probably no other way to make sure that a program does what it is intended to do.

However, testing is not a proof of correctness in the mathematical sense. Program verification tries to show by logical inference that the program satisfies its specifications for any allowed input values - not just for some test values.

We use some simple examples to show how program verification can be done.

A First Example: Calculating the n-th Power of a Real Number

Problem: Develop a program that calculates the power $a^n$ for $n \in \mathbb{N}_0$ (i.e. a non-negative integer) and $a \in \mathbb{R}$ (i.e. a real number).

Besides the variables $a$ and $n$ we introduce the real variable $p$ that should hold the final result $a^n$ when the program terminates. We can now formulate the program specification in an annotated flow chart as follows:

The predicates in curly brackets $\{\}$ are called assertions (Zusicherungen, assertions). They stand for facts that are supposed or proved to be true. We can abbreviate the assertions, e.g.:

$\{E\} = \{a \in \mathbb{R} \land n \in \mathbb{N}_0\}$ and $\{A\} = \{p = a^n\}$
and formulate the above flowchart as a so-called Hoare triple:

\[ \{E\} \text{ P } \{A\} \]

This is a program specification with the following meaning: If the assertion \{E\} is true when the program starts, then, after completion of the program P, we want \{A\} to be true as well. And it is meant that this should hold for all possible values of the variables in \{E\}; i.e. the variables in \{E\} are implicitly universally quantified. \{E\} is called precondition, \{A\} postcondition. A simple solution to our problem is the following Java program:

```java
double a;
int n;
// input a, n (n >= 0)
int m = n;
double p = 1.0;
while (m > 0) {
    p = p * a;       // or:  p *= a;
    m = m - 1;       // or:  m--;
}
// output p
```

We represent the above program as a flowchart:
The key to the verification of this program is the assertion \( \{Q\} = \{a^n = p \cdot a^m\} \). This assertion remains true when the body of the loop

\[
\begin{align*}
p & = p \ast a; \\
m & = m - 1;
\end{align*}
\]

is traversed. Suppose we denote the values of the variables with \( p \) and \( m \) before, and with \( p' \) and \( m' \) after the program segment. Then we get:

\[
p' \cdot a^{m'} = (p \cdot a) \cdot a^{m-1} = p \cdot (a \cdot a^{m-1}) = p \cdot a^m
\]

The assertion \( \{Q\} = \{a^n = p \cdot a^m\} \) is called a loop invariant (Schleifeninvariante, invariante de boucle). It is the key issue of the proof rule for while statements that we will introduce and discuss later in this chapter.

**A Second Example: Sorting Three Numbers**

**Problem:** Given three integer variables \( a, b, \) and \( c \). How can we exchange the values of the variables, such that at the end of the process, \( a \leq b \leq c \)?

The following (not very intuitive) algorithm does the job:

```java
int a, b, c, s;
// input a, b, c
if (a <= c) {
} else {
    s = a; a = c; c = s;
}
if (a <= b) {
    if (b <= c) {
    } else {
        s = b; b = c; c = s;
    }
} else {
    s = a; a = b; b = s;
}
// output a, b, c
```

**Exercise:** Draw a flow chart for the above algorithm and try to understand how it works.

**Exercise:** Use the Java program TripletSort.java to test the correctness of the algorithm, i.e. run the program with different input data. How many tests are necessary in order to be quite sure the algorithm is correct?

We now prove the correctness of the above algorithm by applying the rules of logic. In order to do so we use assertions as milestones in the proof.
```java
int a, b, c, s;
// input a, b, c
if (a <= c) {
}
else {
    s = a; a = c; c = s;
}
if (a <= b) {
    if (b <= c) {
    }
    else {
        s = b; b = c; c = s;
    }
}
else {
    s = a; a = b; b = s;
}
// output a, b, c
```

Since version 1.4 the Java language supports a keyword `assert` which can be used to check for the validity of assertions. In its simple form, `assert` takes a boolean expression as its single argument. At runtime, if assertion checking is enabled, the expression is evaluated and an exception is thrown if it evaluates to `false`. We can thus formulate the assertions in Java syntax as boolean expressions and stick them behind the keyword `assert`. See the program TripletSortAs.java on the next page for an example of a complete Java implementation of our algorithm. Using `assert` is, of course, *not* automatic program verification, but simply a useful form of *silent testing*. 
Important note: Java programs using `assert` must be compiled and run with special flags like this:

```bash
> javac -source 1.4 MyClass.java
> java -ea MyClass
```

-ea stands for 'enable assertions'

/**
 * Sort three numbers ascendingly.
 * @author werner.hett@htl.bfh.ch
 * Time-stamp: Sat Nov 23 11:33:23 2002 hew
 * Given three integer numbers; print them in ascending order.
 * This is an example for a systematic program verification.
 * Assertions are introduced into the code in order to prove the
 * correctness of the algorithm.
 */

public class TripletSortAs {

    static int a, b, c; // variables for the three numbers

    public static void main (String[] args) {
        int s; // auxiliary variable (swap)
        try {
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
            c = Integer.parseInt(args[2]);
        } catch (Exception e) {
            System.out.println("usage: java TripletSort <a> <b> <c>");
            return;
        }
        if (a <= c) {
            assert (a <= c);
        } else {
            assert (a > c);
            s = a; a = c; c = s;
            assert (c > a);
        }
        assert (a <= c);
        if (a <= b) {
            assert (a <= c && a <= b);
            if (b <= c) {
                assert (a <= c && a <= b && b <= c);
                assert (a <= b && b <= c);
            } else {
                assert (a <= c && a <= b && b > c);
                s = b; b = c; c = s;
                assert (a <= b && a <= c && b < c);
                assert (a <= b && b <= c);
            }
            assert (a <= b && b <= c);
        } else {
            assert (a <= c && a > b);
            s = a; a = b; b = s;
            assert (b <= c && b > a);
            assert (a <= b && b <= c);
        }
        assert (a <= b && b <= c);
        System.out.println(a + "  " + b + "  " + c);
    }
}
The two examples we have seen so far demonstrate how applying particular logical
inference rules can prove program correctness. We are now going to discuss these
rules in detail.

**Consequence Rule**

Let \( \{E\} P \{A\} \) be a valid specification of a program segment \( P \). Then, we can always
substitute \( E \) by a "more restrictive" precondition \( E' \), and we can substitute \( A \) by a "less
restrictive" postcondition \( A' \).

**Example:** \( \{x \leq y\} P \{x = y+2\} \) can be replaced with \( \{x < y\} P \{x \geq y\} \), because
\( \{x < y\} \Rightarrow \{x \leq y\} \) and \( \{x = y+2\} \Rightarrow \{x \geq y\} \).

**Consequence Rule**

**IF:** \( E' \Rightarrow E \) and \( \{E\} P \{A\} \) and \( A \Rightarrow A' \)

**THEN:** \( \{E'\} P \{A'\} \)

The consequence rule means in practice: When working through a program in *forward
direction*, we can always replace an assertion with a *weaker* (less restrictive) one. In
our example, \( \{x \leq y\} \) means \( \{x < y \text{ or } x = y\} \) which is weaker than \( \{x < y\} \). On the
other hand, when working *backwards* through a program segment, we can replace
any assertion with a *stronger* (more restrictive) one.

**Assignment Rule**

One of the most fundamental program statements is *assignment*. Consider the
following example:

\[
x = 3 \ast y + z;
\]

Suppose we know that before the statement, the variables \( y \) and \( z \) have the values 2
and 7, respectively, then we can conclude that after the assignment, \( x \) equals 13.
Expressed as a Hoare triple we get:

\[
\{y = 2 \land z = 7\} \quad x = 3 \ast y + z; \quad \{x = 13 \land y = 2 \land z = 7\}
\]

A less trivial example:

\[
\{u \geq 1\} \quad u = 5 - 3 \ast u; \quad \{u \leq 2\}
\]

We are now going to develop the inference rule for the assignment

\[
\{E\} \quad x = expr; \quad \{A\}
\]

First, we try to find out the precondition \( \{E\} \) in the following example:

\[
\{E\} \quad x = x + 7; \quad \{x = 2y\}
\]

The real problem when dealing with assignments of the form \( x = expr \) is the fact
that the left-hand side variable \( x \) may appear in the expression \( expr \) on the right-hand side, and, as a consequence, does not have the same value on both sides of the
Assignment sign. **An assignment is not an equation!** For this reason, we write $x$ and $x'$ for the value of the variable $x$ before and after the assignment operation, respectively.

In our example, the postcondition $\{A\}$ is $\{x' = 2y\}$. $x'$ is the result of the assignment, thus $x' = x + 7 = 2y$ and therefore the precondition $\{E\}$ is $\{x + 7 = 2y\}$ or $\{x = 2y - 7\}$.

What did we exactly do in this last example? We replaced the variable $x$ in the postcondition $\{A\}$ with the right-hand side expression $\text{expr}$ of our assignment and got the precondition $\{E\}$. This is what the following rule says.

**Assignment Rule**

IF: $E \Rightarrow A$ (with $x$ replaced with $\text{expr}$)  
THEN: $\{E\} x = \text{expr}; \{A\}$

**Sequence Rule**

Most programs are sequences of more elementary building blocks. Consider the following example:

$$
\begin{align*}
&x = x + 3; \\
&x = x \times x;
\end{align*}
$$

When starting with the precondition $\{x > -1\}$ we get $\{x > 2\}$ after the first assignment, and $\{x > 4\}$ after the second one. In general:

**Sequence Rule**

IF: $\{E\} P_1 \{B\}$ and $\{B\} P_2 \{A\}$  
THEN: $\{E\} P_1; P_2 \{A\}$

**Conditional Rules**

Suppose $x$ is an integer variable and $\text{odd}(x)$ is a method that returns true if and only if $x$ is an odd number. Consider the following program segment

$$
\begin{align*}
\text{if (odd(x)) } \\
&x = 3 \times x + 1; \\
\text{else } \\
&x = x/2;
\end{align*}
$$

Let us assume that we can guarantee $\{x > 1\}$ as the precondition $\{E\}$. What is the most restrictive postcondition $\{A\}$ we can derive?

We consider the two alternatives separately. If $x$ is odd at the beginning then

$$
\{x > 1 \text{ and odd}(x)\} \quad x = 3 \times x + 1 \quad \{x > 4 \text{ and even}(x)\} \Rightarrow \{x \geq 1\}
$$

On the other hand, if $x$ is even then
\{x > 1 \text{ and not odd}(x)\} \quad x = x/2 \quad \{x \geq 1\}

Therefore, the strongest postcondition we can conclude for both cases is \(\{x \geq 1\}\).

In general, this leads us to the following

**Conditional Rule (if-then-else)**

**IF:** \(\{E \text{ and } c\} \ P_1 \ {A}\) \ and \(\{E \text{ and not } c\} \ P_2 \ {A}\)

**THEN:** \(\{E\} \quad \text{if } (c) \ P_1 ; \ \text{else } P_2 \ {A}\)

There is also a version of this rule for single-sided if-statements

**Conditional Rule (if-then)**

**IF:** \(\{E \text{ and } c\} \ P \ {A}\) \ and \(\{E \text{ and not } c\} \Rightarrow \{A\}\)

**THEN:** \(\{E\} \quad \text{if } (c) \ P ; \ {A}\)

**Loop Rules**

Consider the following program:

```
// input int a, int b
// a and b are both positive
int r = a;
int q = 0;
while (r >= b) {
    r = r - b;
    q = q + 1;
}
// output q and r
```

We can find out what the program does by tracing its execution for some given input values. A powerful technique is the use of a so-called trace table. In the trace table we note the current values of all the variables which may change within the loop body. We observe what our example program does for \(a = 17\) and \(b = 5\); checkpoint is immediately before the entrance test of the while loop:
Another example $a = 28$, $b = 7$:

<table>
<thead>
<tr>
<th>$r$</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q$</td>
<td>0</td>
</tr>
</tbody>
</table>

Now comes the creative part of proof! What is the assertion we can formulate to be always true at the checkpoint? We look for a combination of the variables $r$ and $q$ with the problem data, i.e. with the given values of $a$ and $b$.

The solution is apparently $\{Q\} = \{q b + r = a \text{ and } 0 \leq r < b\}$.

When the while loop eventually terminates, we can conclude that $\{Q \text{ and not } (r \geq b)\}$ holds, i.e. $\{Q \text{ and } r < b\}$, or $\{q b + r = a \text{ and } 0 \leq r < b\}$.

Now, that we strongly believe in the correctness of our assertion $\{Q\}$, it is time to formally prove it. We need to demonstrate two things:

1) When the control flow of the program arrives at the checkpoint for the first time, $\{Q\}$ is true, and

2) if $\{Q\}$ is true at the beginning of the loop body and if the entrance test $(r \geq b)$ succeeds then $\{Q\}$ is still true at the end of the loop body.

Both parts are not very difficult to prove:

1) When the program execution first arrives at the checkpoint then $r = a$ and $q = 0$, and we know that $a \geq 0$ and hence $r \geq 0$. Therefore, $q b + r = a$ and $r \geq 0$; thus $\{Q\} = \{q b + r = a \text{ and } r \geq 0\}$ is true.

2) $\{Q \text{ and } r \geq b\} = \{q b + r = a \text{ and } r \geq 0 \text{ and } r \geq b\}$. When the loop body is traversed, we get new values for the variables $r' = r - b$ and $q' = q + 1$. When we calculate the expression in $Q$ with these new values, we get:

$$q' b + r' = (q + 1) b + (r - b) = q b + b + r - b = q b + r = a$$

Secondly, if $r \geq b$ at the beginning of the loop body, then $r' = r - b \geq 0$ holds after the loop body is traversed. This is the second part of the assertion $\{Q\}$.
The generalisation of these ideas leads us to the following proof rule:

**Loop Rule (while)**

**IF:**  
\{Q and c\} \[ P \] \{Q\}

**THEN:**  
\{Q\} \[ while(c) \] \[ P; \] \{Q and not c\}

As mentioned earlier, the assertion \{Q\} is called loop invariant. Determining the loop invariant is not always easy, but it is the most important part of the proof.

Note that, essentially, the while loop rule is nothing else than the application of the mathematical principle of complete induction.

There is a second kind of loops in Java, the do-while loops. The proof rule for do-while loops is the following:

**Loop Rule (do-while)**

**IF:**  
\{Q\} \[ P \] \{R\} \[ and \] \{R and c\} \[ \Rightarrow \] \{Q\}

**THEN:**  
\{Q\} \[ do \] \[ P \] \[ while(c) \] \{R and not c\}

And finally, for loops are a convenient variant of while loops. Their definition goes like this:

for \( (A; \ c; \ B)\) \[ P \] is equivalent to  
\[ A; \ while \ (c) \] \{P; B\}

In this formula, A stands for the initialisation block, B is the increment block, c is the condition (to continue), and P is the loop body. We use this equivalence when we prove the correctness of a program with for loops.

**Loop Termination**

Until now, we have only considered the so-called partial correctness. Loosely speaking, this means that, *if* the program terminates at all, it produces the correct result. However, when there are loops in a program, we have to make sure that their exit conditions are reached in a finite number of steps. The trick is to find an integer arithmetic expression \( t \), which brings us closer to the termination with each pass through the loop. If we find such an expression then we have proved the total correctness of the program.
A Final Example

We reconsider the problem of calculating the \( n \)th power of a real number \( a \). The following algorithm does the job:

```plaintext
// input double a, int n >= 0
double b = a;
int m = n;
double p = 1.0;

while (m > 0) { // checkpoint
    if (m % 2 == 1) { // m is odd
        p = p * b;
        m = m - 1;
    } else {
        b = b * b;
        m = m / 2;
    }
}

// output p
```

We trace the execution of the program for a particular value of \( n \), say \( n = 13 \).
Exercises

All programs can be found under http://www.hta-bi.bfh.ch/~hew/dma/

1) Prove the correctness of the algorithm of program TripletBubble.java that solves the same problem as TripletSort.java. Use the same technique as we did for the TripletSort.java example.

2) Four numbers can be sorted with the algorithm given in the program QuadrupletSort.java. Prove its correctness by introducing assertions. Use the Java keyword `assert` to do silent testing.

3) Take the division algorithm we discussed in the Loop Rule section of the course and write a corresponding Java program with `assert` statements at the right place.

In the following exercises, use trace tables to find out how things work. Write complete Java programs with `assert` statements in order to make sure your assertions are correct.

4) Two positive integers \(a\) and \(b\) can be multiplied using the following very old algorithm (expressed in Java):

```java
// input int a, int b (both positive)
int u = a;
int v = b;
int p = 0;
while (u > 0) {
    if (u % 2 == 1) {
        p = p + v;
        u = u - 1;
    }
    u = u / 2;
    v = 2 * v;
}
// output p
```

(a) Trace the program with some particular values for \(a\) and \(b\), say \(a = 13\), \(b = 7\). Prove the correctness of the algorithm. (b) What is the benefit of the method? (c) How many times is the loop body executed? (d) If you think in terms of machine programming, how could you implement the algorithm most efficiently?

5) The power \(a^n\) for \(n \in \mathbb{N}_0\) (i.e. a non-negative integer) and \(a \in \mathbb{R}\) (i.e. a real number) can be calculated using the following algorithm:

```java
// input double a, int n
double b = a;
int m = n;
double p = 1.0;
while (m > 0) {
    if (m % 2 == 1)
        p = p * b;
    m = m / 2;
    b = b * b;
}
// output p
```
(a) Prove the correctness of the program. (b) What is the difference between this algorithm and the one given in the course?

6) Find out what the following algorithm does. \(x\) and \(y\) are positive integers.

```c
// input int x, int y (both positive)
int r = x;
int q = 0;
int w = y;
while (w <= r)
    w = 2 * w;
while (w > y) {
    q = 2 * q;
    w = w / 2;
    if (w <= r) {
        r = r - w;
        q = q + 1;
    }
}
// output q and r
```

(a) Prove what you have found by verifying the program. (b) What would happen in the illegal case \(y = 0\)? (c) Discuss the difference between this algorithm and the one in course.

7) Find out what the following algorithm does. \(x\) and \(y\) are positive integers:

```c
// input int x, int y
int u = x;
int v = y;
while (v != 0) {
    int r = u % v;
    u = v;
    v = r;
}
// output u
```

Use program verification techniques to show the total correctness of the program.

**Further recommended exercises:** Gersting, Section 1.6: exercises 1 to 11; Section 2.3: exercises 1, 2, 9, 11.