CIS 301: Lecture Notes on Program Verification

Torben Amtoft
Department of Computing and Information Sciences
Kansas State University

September 30, 2005

These notes are written as a supplement to [1, Sect. 16.5], but can be read independently. Section 6 is inspired by Chapter 16 in [3], an excellent treatise on the subject of program construction; also our Section 10 is inspired by that book. The proof rules in Section 7 are inspired by the presentation in [4, Chap. 4]. Section 8 is inspired by [2].

Contents

1 Hoare Triples

To reason about correctness we shall consider *Hoare triples*, of the form

$$\begin{array}{l} \{\phi\} \\ \quad P \\ \{\psi\} \end{array}$$

saying that if ϕ (the *precondition*) holds prior to executing program code P then ψ (the *postcondition*) holds afterwards. Here ϕ and ψ are *assertions*, written in First Order Logic.

Actually, the above description is ambiguous: what if P does not terminate? Therefore we shall distinguish between

partial correctness: *if* P terminates *then* ψ holds;

total correctness: P *does* terminate *and then* ψ holds.

In these notes, we shall interpret a Hoare triple as denoting *partial correctness*, unless stated otherwise.

2 Software Engineering

In light of the notion of Hoare triples, one can think of software engineering as a 3-stage process:

1. *Translate* the demands D of the user into a *specification* (ϕ_D, ψ_D).
2. *Write* a program P that satisfies the specification constructed in 1.
3. *Prove* that in fact it holds that

   $$\begin{array}{l} \{\phi_D\} \\ \quad P \\ \{\psi_D\} \end{array}$$

When it comes to software practice, 1 is a huge task (involving numerous discussions with the users) and hardly ever done completely. While 2 obviously has to be done, 3 is almost never carried out.

When it comes to academic discourse, 1 is an interesting task but only briefly touched upon (Section 3) in CIS 301. Instead, we shall focus on 3 (Sections 5 and 7), but also give a few basic heuristics for how to do 2 and 3 *simultaneously* (Section 6).

3 Specifications

3.1 Square root

Suppose the user demands

> Compute the square root of `x` and store the result in `y`.

As a first attempt, we may write the specification

$$P$$
$$\{\mathtt{y}^2 = \mathtt{x}\}$$

We now remember that we cannot compute the square root of negative numbers and therefore add a precondition:

$$\{\mathtt{x} \geq 0\}$$
$$P$$
$$\{\mathtt{y}^2 = \mathtt{x}\}$$

Then we realize that if `x` is not a perfect square then we have to settle for an approximation (since we are working with integers):

$$\{\mathtt{x} \geq 0\}$$
$$P$$
$$\{\mathtt{y}^2 \leq \mathtt{x}\}$$

On the other hand, this is too liberal: we could just pick `y` to be zero. Thus, we must also specify that `y` is the largest number that does the job:

$$\{\mathtt{x} \geq 0\}$$
$$P$$
$$\{\mathtt{y}^2 \leq \mathtt{x} \land (\mathtt{y}+1)^2 > \mathtt{x}\}$$

which seems a sensible specification of the square root program. (Which entails that `y` has to be non-negative. Why?)

3.2 Factorial

Now assume that the user demands

> Ensure that y contains the factorial[1] of x.

This might give rise to the specification

$\{\mathtt{x} \geq 0\}$
$\quad P$
$\{\mathtt{y} = \mathrm{fac}(\mathtt{x})\}$

Well, it's not hard to write a program satisfying this specification:

$\{\mathtt{x} \geq 0\}$
$\quad \mathtt{x} := 4;$
$\quad \mathtt{y} := 24$
$\{\mathtt{y} = \mathrm{fac}(\mathtt{x})\}$

The user may respond:

> Hey, that's cheating! You were not allowed to change x.

Well, if not, that better has to be part of the specification! But how to incorporate such demands?

The answer is that we shall allow our specifications to contain *logical* variables[2]: using the logical variable x_0 to denote the initial (and un-changed) value of the identifier x, a program computing factorials can be specified as follows:

$\{\mathtt{x} = x_0 \geq 0\}$
$\quad P$
$\{\mathtt{y} = \mathrm{fac}(x_0) \wedge \mathtt{x} = x_0\}$

Likewise, the specification of the square root program can be modified.

[1]Remember that the factorial function is defined by

$$\begin{aligned} \mathrm{fac}(0) &= 1 \\ \mathrm{fac}(n+1) &= (n+1)\mathrm{fac}(n) \text{ for } n \geq 0 \end{aligned}$$

and thus fac(0) = 1, fac(1) = 1, fac(2) = 2, fac(3) = 6, fac(4) = 24, etc.

[2]We shall write logical variables with a subscript, so as to emphasize that they do not occur in programs.

4 A Simple Language

For the next sections, we consider programs P written in a simple language, omitting[3] many desirable language features—such as procedures, considered in Section 9, and arrays, considered in Section 10.

A program P is (so far) just a command, with the syntax of commands given by

$$\begin{array}{rcl} C & ::= & x := E \\ & \mid & C_1;\ C_2 \\ & \mid & \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi} \\ & \mid & \texttt{while}\ B\ \texttt{do}\ C\ \texttt{od} \end{array}$$

We have employed some auxiliary syntactic constructs:

- x stands for *identifiers*[4] like `x`, `y`, `z`, etc;
- E stands for *integer expressions* of the form n (a constant), x (an identifier), $E_1 + E_2$, $E_1 - E_2$, etc;
- B stands for *boolean tests* of the form $E_1 < E_2$, $E_1 \leq E_2$, $E_1 \neq E_2$, etc.

Programs are thus constructed from assignments, sequential composition, conditionals, and while-loops[5].

Next, we shall discuss how to verify a claim that

$$\begin{array}{l} \{\phi\} \\ \quad P \\ \{\psi\} \end{array}$$

[3]Still, our language is "Turing-complete" in that it can *encode* all other features one can imagine!

[4]We shall use the term "identifier" for what is often called a "program variable", so as to avoid confusion with the variables of First Order Logic. To further facilitate that distinction, we shall always write identifiers in typewriter font.

[5]Note that we use the symbol `od` to end while-loops, rather than a curly bracket as that symbol is used for writing pre- and post-conditions. Similarly, we use `fi` as a delimiter for conditionals.

5 Loop Invariants

For the purpose of verification, the notion of *loop invariants* is crucial.

5.1 Motivating Example

We look at the following program for computing the factorial function[6] (cf. Section 3).

```
{x ≥ 0}
    y := 1;
    z := 0;
    while z ≠ x do
        z := z + 1;
        y := y * z
    od
{y = fac(x)}
```

There are many mistakes we could have made when writing that program: for instance we could have reversed the two lines in the loop body (in which case y would be assigned zero and keep that value forever), or we could have written the loop test as $z \leq x$ (in which case y would end up containing $\text{fac}(x+1)$).

Let us now convince ourselves that what we wrote is correct. We might first try a simulation: if say $x = 4$, the situation at the entry of the loop is:

	x	y	z
After 0 iterations	4	1	0
After 1 iterations	4	1	1
After 2 iterations	4	2	2
After 3 iterations	4	6	3
After 4 iterations	4	24	4

and then $z = x$ so that the loop terminates, with y containing the desired result $24 = \text{fac}(x)$. This may boost our confidence in the program, but still a general proof is needed. Fortunately, the table above may help us in that endeavor. For it suggests that it is always the case that $y = \text{fac}(z)$.

[6] Since the program does not change the value of x, we can safely write its specification without employing a logical variable x_0.

Definition 5.1. A property which holds each time the loop test is evaluated is called an *invariant* for the loop. □

We now annotate the program with our prospective loop invariant:

$\{\mathtt{x} \geq 0\}$
    $\mathtt{y} := 1;$
    $\mathtt{z} := 0;$
$\{\mathtt{y} = \mathrm{fac}(\mathtt{z})\}$
    $\mathtt{while}\ \mathtt{z} \neq \mathtt{x}\ \mathtt{do}$
        $\mathtt{z} := \mathtt{z} + 1;$
        $\mathtt{y} := \mathtt{y} * \mathtt{z}$
$\{\mathtt{y} = \mathrm{fac}(\mathtt{z})\}$
    $\mathtt{od}$
$\{\mathtt{y} = \mathrm{fac}(\mathtt{x})\}$

Of course, we must prove that what we have found is indeed a loop invariant:

Proposition 5.2. *Whenever the loop entry is reached, it holds that* $\mathtt{y} = \mathrm{fac}(\mathtt{z})$*.* □

The proof has two parts:

- Establishing the invariant;
- Maintaining the invariant.

Establishing the invariant. We must check that when the loop entry is first reached, it holds that $\mathtt{y} = \mathrm{fac}(\mathtt{z})$. But since the preamble assigns $\mathtt{y}$ the value 1 and assigns $\mathtt{z}$ the value 0, the claim follows from the fact that $\mathrm{fac}(0) = 1$.

Maintaining the invariant. Next we must check that if $\mathtt{y} = \mathrm{fac}(\mathtt{z})$ holds *before* an iteration then it also holds *after* the iteration. With $\mathtt{y}'$ denoting the value of $\mathtt{y}$ *after* the iteration, and $\mathtt{z}'$ denoting the value of $\mathtt{z}$ *after* the iteration, this follows from the following calculation:

$$\mathtt{y}' = \mathtt{y}\mathtt{z}' = \mathtt{y}(\mathtt{z}+1) = \mathrm{fac}(\mathtt{z})(\mathtt{z}+1) = \mathrm{fac}(\mathtt{z}+1) = \mathrm{fac}(\mathtt{z}').$$

For the third equality we have used the assumption that the invariant holds *before* the iteration, and for the fourth equality we have used the definition of the factorial function.

Completing the correctness proof. We have shown that every time the loop test is evaluated, it holds that $\mathtt{y} = \text{fac}(\mathtt{z})$. If (when!) we eventually exit the loop then the loop test is `false`, that is $\mathtt{z} = \mathtt{x}$. Therefore, if (when) the program terminates it holds that $\mathtt{y} = \text{fac}(\mathtt{x})$. This shows that our program satisfies its specification, in that partial correctness holds (cf. Section 1). Moreover, in this case total correctness is not hard to prove: since `x` is initially[7] non-negative, and since `z` is initialized to zero and incremented by one at each iteration, eventually `z` will equal `x`, causing the loop to terminate.

5.2 Proof Principles for Loop Invariants

From the previous subsection we see that three steps are involved when proving that a certain property ψ is indeed a useful invariant for a loop:

1. we must show that the code before the loop establishes ψ;
2. we must show that ψ is maintained after each iteration;
3. we must show that if the loop test evaluates to `false`, ψ is sufficient to establish the desired postcondition.

6 Developing a Correct Program

In Section 5, we considered the situation where we must prove the correctness of a program which has *already* been written for a given specification. This two-step approach has some drawbacks:

- it gives us no clue about how actually to *construct* programs;
- if the program in question has been developed in an unsystematic way, perhaps by someone else, it may be hard to detect the proper loop invariant(s).

In this section, we shall illustrate that it is often possible to write a program *together* with the proof of its correctness.

For that purpose, we look at the square root specification[8] from Section 3:

[7] It is interesting that the proof of partial correctness does not use the precondition $\mathtt{x} \geq 0$.

[8] We shall not bother to employ the device of logical variables, and must therefore solemnly promise that the program to be constructed will not modify the value of `x`.

$\{\mathtt{x} \geq 0\}$
    P
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$

It seems reasonable to assume that P should be a loop, possibly with some preamble. With ϕ the (yet unknown) invariant of that loop, and with B the (yet unknown) test of the loop, we have the skeleton

```
{x ≥ 0}
    ???
{φ}
    while B do
        ???
{φ}
    od
{y² ≤ x ∧ (y + 1)² > x}
```

We now face the main challenge: to come up with a suitable invariant ϕ, the form of which will direct the remaining construction process. In order to justify the postcondition, we must ensure that

$$\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x} \text{ is a logical consequence of } \phi \wedge \neg B. \quad (1)$$

There are at least two ways to achieve that, to be described in the next two subsections.

6.1 Deleting a Conjunct

A simple way to satisfy (1) is to define

$$\begin{aligned} \phi &= \mathtt{y}^2 \leq \mathtt{x} \\ B &= (\mathtt{y}+1)^2 \leq \mathtt{x} \end{aligned}$$

That is, we follow the following general recipe:

- let the loop test be the negation of one of the conjuncts of the post-condition;
- let the loop invariant be the remaining conjuncts of the postcondition.

Our prospective program now looks like

$\{\mathtt{x} \geq 0\}$
    ???
$\{\mathtt{y}^2 \leq \mathtt{x}\}$
    $\mathtt{while}\ (\mathtt{y}+1)^2 \leq \mathtt{x}\ \mathtt{do}$
        ???
$\{\mathtt{y}^2 \leq \mathtt{x}\}$
    $\mathtt{od}$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$

Thanks to the precondition $\mathtt{x} \geq 0$, initializing $\mathtt{y}$ to zero will establish the loop invariant. Thanks to the loop test $(\mathtt{y}+1)^2 \leq \mathtt{x}$, incrementing $\mathtt{y}$ by one will maintain the loop invariant. We end up with the program

$\{\mathtt{x} \geq 0\}$
    $\mathtt{y} := 0$
$\{\mathtt{y}^2 \leq \mathtt{x}\}$
    $\mathtt{while}\ (\mathtt{y}+1)^2 \leq \mathtt{x}\ \mathtt{do}$
        $\mathtt{y} := \mathtt{y}+1$
$\{\mathtt{y}^2 \leq \mathtt{x}\}$
    $\mathtt{od}$
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 > \mathtt{x}\}$

This program will clearly always terminate, but is rather inefficient. We shall now describe a method which in this case results in a more efficient program.

6.2 Replacing an Expression By an Identifier

Let us consider another way of satisfying (1). First observe that the postcondition involves the expression $\mathtt{y}$ as well as the expression $\mathtt{y}+1$. It might be beneficial to loosen the connection between these two entities, by introducing a new identifier $\mathtt{w}$ which eventually should equal $\mathtt{y}+1$ but in the meantime may roam more freely. Note that the postcondition is implied by the formula

$$\mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \wedge \mathtt{w} = \mathtt{y}+1$$

containing three conjuncts. It is thus tempting to apply the previous technique of "deleting a conjunct", resulting in

$$\begin{aligned} \phi &= \mathtt{y}^2 \leq \mathtt{x} \wedge \mathtt{w}^2 > \mathtt{x} \\ B &= \mathtt{w} \neq \mathtt{y}+1 \end{aligned}$$

Our prospective program now looks like

```
{x ≥ 0}
    ???
{y² ≤ x ∧ w² > x}
    while w ≠ y + 1 do
        ???
{y² ≤ x ∧ w² > x}
    od
{y² ≤ x ∧ (y + 1)² > x}
```

To establish the loop invariant, we must not only initialize `y` to zero but also initialize `w` so that $w^2 > x$: clearly, $x + 1$ will do the job.

For the loop body, it seems a sensible choice to modify *either* `y` *or* `w`. This can be expressed as a conditional of the form

```
if B' then
    y := E₁
else
    w := E₂
fi
```

We must check that each branch maintains the invariant, and therefore perform a case analysis:

- if B' is true, we must require that ${E_1}^2 \leq x$;
- if B' is false, we must require that ${E_2}^2 > x$.

Let E be an arbitrary expression; then these demands can be satisfied by stipulating

$$\begin{aligned} E_1 &= E \\ E_2 &= E \\ B' &= E^2 \leq x \end{aligned}$$

We have thus constructed the program

```
{x ≥ 0}
```

```
    y := 0;
    w := x + 1;
{y² ≤ x ∧ w² > x}
    while w ≠ y + 1 do
        if E² ≤ x
        then
            y := E
        else
            w := E
        fi
{y² ≤ x ∧ w² > x}
    od
{y² ≤ x ∧ (y + 1)² > x}
```

which is partially correct, no matter how E is chosen! But of course, we also want to ensure termination, and hopefully a quick such! For that purpose, we pick

$$E \quad = \quad (\mathtt{y} + \mathtt{w}) \ \mathtt{div}\ 2$$

where a `div` b (for positive b) is the largest integer c such that $bc \leq a$. With that choice, it is not difficult to see that `y` and `w` will get closer to each other for each iteration, until eventually $\mathtt{w} = \mathtt{y} + 1$. This shows total correctness. Even more, the program runs much faster than our first attempt!

7 Well-Annotated Programs and Valid Assertions

We have argued that annotating a program with loop invariants is essential for the purpose of verification (and also to understand how the program works!) It is often beneficial to provide more fine-grained annotations.

Example 7.1. For the factorial program from Sect. 5.1, a fully annotated version looks like

```
{x ≥ 0}                        (A)
{1 = fac(0)}                   (B)
    y := 1;
{y = fac(0)}                   (C)
    z := 0;
{y = fac(z)}                   (D)
```

```
     while z ≠ x do
{y = fac(z) ∧ z ≠ x}                    (E)
{y(z + 1) = fac(z + 1)}                 (F)
          z := z + 1;
{yz = fac(z)}                           (G)
          y := y * z
{y = fac(z)}                            (H)
     od
{y = fac(z) ∧ z = x}                    (I)
{y = fac(x)}                            (J)
```

We shall soon see that this program is in fact *well-annotated*. □

We first define what it means for an assertion to be *valid*. There are several cases:

Logical consequence. If the assertion $\{\psi\}$ immediately follows the assertion $\{\phi\}$, and ψ is a logical consequence of ϕ, then ψ is valid.

Trying to conform with the notation used in [1], we can write this rule as

	$\{\phi\}$	
$\triangleright$	$\{\psi\}$	**Implies** (if ψ logical consequence of ϕ)

saying that the marked assertion is valid.

Of course, in order to trust that ψ holds, we must at some point also establish that ϕ is valid!

EXAMPLE 7.2. Referring back to Example 7.1, note that thanks to this rule

- assertion (B) is valid, since it is a mathematical fact and therefore surely a logical consequence of assertion (A);
- assertion (F) is valid, since if by assertion (E) we have $y = \text{fac}(z)$ then $y(z+1) = \text{fac}(z)(z+1) = \text{fac}(z+1)$;
- assertion (J) is valid, since it is a logical consequence of assertion (I). □

Rule for While loops. We have the rule

```
    {ψ}
        while B do
▷   {ψ ∧ B}                 WhileTrue
        ...
    {ψ}
        od
▷   {ψ ∧ ¬B}                WhileFalse
```

saying that if ψ is a loop invariant then

- at the beginning of the loop body, the loop test has just evaluated to `true` and therefore $\psi \wedge B$ will hold;
- immediately after the loop, the loop test has just evaluated to `false` and therefore $\psi \wedge \neg B$ will hold.

Note that we are still left with the obligation to show that the two ψ assertions (one before the loop, the other at the end of the loop body) are valid.

EXAMPLE 7.3. Referring back to Example 7.1, note that assertions (E) and (I) are valid, thanks to this rule. □

Rule for Conditionals. We have the rule

```
    {φ}
        if B
        then
▷   {φ ∧ B}                 IfTrue
        ...
    {ψ}
        else
▷   {φ ∧ ¬B}                IfFalse
        ...
    {ψ}
        fi
▷   {ψ}                     IfEnd
```

saying that if ϕ holds before a conditional command then

- at the beginning of the `then` branch, $\phi \wedge B$ will hold;

- at the beginning of the `else` branch, $\phi \land \neg B$ will hold;

and also saying that ψ holds after the conditional command if ψ holds at the end of both branches.

Again, we are left with the obligation to show that the initial ϕ assertion is valid, and that the ψ assertions concluding each branch are valid.

Observe that this rule is quite similar to the rule $\lor$ **Elim** from propositional logic!

Rule for Assignments We would surely expect that for instance it holds that

$\{\mathtt{y} = 5\}$
  $\mathtt{x} := \mathtt{y} + 2$
$\{\mathtt{x} = 7 \land \mathtt{y} = 5\}$

and it seems straightforward to go from precondition to postcondition. But now consider

$\{\mathtt{y} + 2\mathtt{z} \leq 3 \land \mathtt{z} \geq 1\}$
  $\mathtt{x} := \mathtt{y} + \mathtt{z}$
$\{???\}$

where it is by no means a simple mechanical procedure to fill in the question marks: what does the precondition imply concerning the value of $\mathtt{y} + \mathtt{z}$?

It turns out that we shall formulate the proper rule *backwards*: if we assign `x` the expression E, and we want $\psi(x)$ to hold *after* the assignment, we better demand that $\psi(E)$ holds *before* the assignment! This motivates the rule[9]

  $\{\psi(E)\}$
    $\mathtt{x} := E$
$\triangleright$ $\{\psi(x)\}$ **Assignment**

Referring back to our first example, we have

[9] We let $\psi(x)$ denote a formula where x is possibly free, and let $\psi(E)$ denote the result of substituting E for *all* free occurrences of x.

$\{\mathtt{y} = 5\}$
$\{\mathtt{y} + 2 = 7 \wedge \mathtt{y} = 5\}$ **Implies**
  $\mathtt{x} := \mathtt{y} + 2$
$\{\mathtt{x} = 7 \wedge \mathtt{y} = 5\}$ **Assignment**

And referring back to our second example, we have

$\{\mathtt{y} + 2\mathtt{z} \leq 3 \wedge \mathtt{z} \geq 1\}$
$\{\mathtt{y} + \mathtt{z} \leq 2\}$ **Implies**
  $\mathtt{x} := \mathtt{y} + \mathtt{z}$
$\{\mathtt{x} \leq 2\}$ **Assignment**

since it is easy to check that if $y + 2z \leq 3$ and $z \geq 1$ then $y + z \leq 2$.

EXAMPLE 7.4. Referring back to Example 7.1, note that assertions (C), (D), (G), and (H) are valid, thanks to this rule. □

Well-annotation. We are now done with all the rules for validity. Note that there is no need for a rule for sequential composition $C_1; C_2$, since in

$\{\phi\}$
  $C_1;$
$\{\phi_1\}$
  C_2
$\{\phi_2\}$

the validity of each ϕ_i $(i = 1, 2)$ must be established using the form of C_i. But there is a rule for all other language constructs, and also a rule **Implies** that is not related to any specific language construct.

We are now ready to assemble the pieces:

Definition 7.5. We say that an annotated program

$\{\phi\}$
  $\ldots$
$\{\psi\}$

is *well-annotated* iff all assertions, *except* for the precondition ϕ, are valid. □

Theorem 7.6. *Assume that the annotated program*

$\{\phi\}$
$\quad\ldots$
$\{\psi\}$

is in fact well-annotated. Then the program is partially correct wrt. the specification (ϕ,ψ). □

EXAMPLE 7.7. The program in Example 7.1 is well-annotated. This follows from Examples 7.2, 7.3, and 7.4. We can write

$\{\mathtt{x} \geq 0\}$	
$\{1 = \text{fac}(0)\}$	**Implies**
$\quad \mathtt{y} := 1;$	
$\{\mathtt{y} = \text{fac}(0)\}$	**Assignment**
$\quad \mathtt{z} := 0;$	
$\{\mathtt{y} = \text{fac}(\mathtt{z})\}$	**Assignment**
$\quad \mathtt{while}\ \mathtt{z} \neq \mathtt{x}\ \mathtt{do}$	
$\{\mathtt{y} = \text{fac}(\mathtt{z}) \wedge \mathtt{z} \neq \mathtt{x}\}$	**WhileTrue**
$\{\mathtt{y}(\mathtt{z}+1) = \text{fac}(\mathtt{z}+1)\}$	**Implies**
$\qquad \mathtt{z} := \mathtt{z}+1;$	
$\{\mathtt{yz} = \text{fac}(\mathtt{z})\}$	**Assignment**
$\qquad \mathtt{y} := \mathtt{y} * \mathtt{z}$	
$\{\mathtt{y} = \text{fac}(\mathtt{z})\}$	**Assignment**
$\quad \mathtt{od}$	
$\{\mathtt{y} = \text{fac}(\mathtt{z}) \wedge \mathtt{z} = \mathtt{x}\}$	**WhileFalse**
$\{\mathtt{y} = \text{fac}(\mathtt{x})\}$	**Implies**

□

EXAMPLE 7.8. The program developed in Section 6.1 can be well-annotated:

$\{\mathtt{x} \geq 0\}$	
$\{0^2 \leq \mathtt{x}\}$	**Implies**
$\quad \mathtt{y} := 0$	
$\{\mathtt{y}^2 \leq \mathtt{x}\}$	**Assignment**
$\quad \mathtt{while}\ (\mathtt{y}+1)^2 \leq \mathtt{x}\ \mathtt{do}$	
$\{\mathtt{y}^2 \leq \mathtt{x} \wedge (\mathtt{y}+1)^2 \leq \mathtt{x}\}$	**WhileTrue**
$\{(\mathtt{y}+1)^2 \leq \mathtt{x}\}$	**Implies**
$\qquad \mathtt{y} := \mathtt{y}+1$	

$\{y^2 \leq x\}$ **Assignment**

od

$\{y^2 \leq x \wedge (y+1)^2 > x\}$ **WhileFalse**

□

EXAMPLE 7.9. The program developed in Section 6.2 can be well-annotated:

$\{x \geq 0\}$

$\{0^2 \leq x \wedge (x+1)^2 > x\}$ **Implies**

y := 0;

$\{y^2 \leq x \wedge (x+1)^2 > x\}$ **Assignment**

w := x + 1;

$\{y^2 \leq x \wedge w^2 > x\}$ **Assignment**

while $w \neq y + 1$ do

$\{y^2 \leq x \wedge w^2 > x \wedge w \neq y + 1\}$ **WhileTrue**

if $E^2 \leq x$

then

$\{y^2 \leq x \wedge w^2 > x \wedge w \neq y + 1 \wedge E^2 \leq x\}$ **IfTrue**

$\{E^2 \leq x \wedge w^2 > x\}$ **Implies**

y := E

$\{y^2 \leq x \wedge w^2 > x\}$ **Assignment**

else

$\{y^2 \leq x \wedge w^2 > x \wedge w \neq y + 1 \wedge E^2 > x\}$ **IfFalse**

$\{y^2 \leq x \wedge E^2 > x\}$ **Implies**

w := E

$\{y^2 \leq x \wedge w^2 > x\}$ **Assignment**

fi

$\{y^2 \leq x \wedge w^2 > x\}$ **IfEnd**

od

$\{y^2 \leq x \wedge w^2 > x \wedge w = y + 1\}$ **WhileFalse**

$\{y^2 \leq x \wedge (y+1)^2 > x\}$ **Implies**

□

8 Secure Information Flow

Assume we are dealing with two kinds of identifiers: those of high security (classified); and those of low security (non-classified). Our goal is that users with low clearance should not be able to gain information about the values of the classified identifiers. In the following, this notion will be made precise.

For the sake of simplicity, let us assume that there are only two identifiers in play: `l` (for low) and `h` (for high). We want to protect ourselves against an attacker (spy) who

- knows the initial value of `l`;
- knows the program that is running;
- can observe the final value of `l`;
- can *not* observe intermediate states of program execution.

A program is said to be *secure* if such an attacker cannot detect anything about the initial value of `h`.

8.1 Examples

The program below is *not* secure.

$$\texttt{l} := \texttt{h} + 7 \tag{2}$$

For by subtracting 7 from the final value of `l`, the attacker gets the initial value of `h`. On the other hand, the program below is clearly secure.

$$\texttt{l} := \texttt{l} + 47 \tag{3}$$

One rotten apple does not always spoil the whole barrel; having the insecure program in (2) as a preamble may still yield a secure program as in

$$\texttt{l} := \texttt{h} + 7;\ \texttt{l} := 27 \tag{4}$$

since we assumed that the attacker cannot observe intermediate values of `l`. Also the following program is secure:

$$\texttt{h} := \texttt{l} \tag{5}$$

For even though the attacker learns the *final* value of `h` (as it equals the initial value of `l` which is known), he is still clueless about the *initial* value of `h`.

The following program is just a fancy way of writing $\texttt{l} := \texttt{h} + 7$ (since we do not care about the final value of `h`)

$$\texttt{l} := 7;\ \texttt{while}\ \texttt{h} > 0\ \texttt{do}\ \texttt{h} := \texttt{h} - 1;\ \texttt{l} := \texttt{l} + 1\ \texttt{od} \tag{6}$$

and is therefore insecure. Also, the following program is insecure

$$\texttt{if h} = 6789 \texttt{ then l} := 0 \texttt{ else l} := 1 \texttt{ fi} \tag{7}$$

since if the final value of `l` is zero, we know that `h` was initially 6789.

8.2 Specification

By putting quantifiers in front of Hoare triples, we can express security formally:

Definition: The program P is secure iff

$$\forall l_0\ \exists l_1\ \forall h_0\ \exists h_1$$
$$\{\texttt{l} = l_0 \land \texttt{h} = h_0\}$$
$$P$$
$$\{\texttt{l} = l_1 \land \texttt{h} = h_1\}$$

To put it another way, the final value (l_1) of `l` must depend only on the initial value (l_0) of `l` and *not* on the initial value (h_0) of `h`.

By negating this definition (and applying de Morgan's laws repeatedly), we arrive at:

Observation: The program P is insecure iff

$$\exists l_0\ \forall l_1\ \exists h_0\ \neg\exists h_1$$
$$\{\texttt{l} = l_0 \land \texttt{h} = h_0\}$$
$$P$$
$$\{\texttt{l} = l_1 \land \texttt{h} = h_1\}$$

To put it another way, a program is insecure if for all possible final values of `l`, there exists an initial value of `h` that produces a different final value for `l`.

8.3 Examples Revisited

We first address the programs that are secure, and show that they do indeed meet the requirement stated in our Definition. In each case, we are given some l_0 and must find l_1 such that

$\forall h_0 \; \exists h_1$

$$\begin{array}{l} \{\mathtt{l} = l_0 \wedge \mathtt{h} = h_0\} \\ \qquad P \\ \{\mathtt{l} = l_1 \wedge \mathtt{h} = h_1\} \end{array}$$

For the program in (3), we choose l_1 as $l_0 + 47$; this does the job since

$\forall h_0 \; \exists h_1$

$$\begin{array}{l} \{\mathtt{l} = l_0 \wedge \mathtt{h} = h_0\} \\ \qquad \mathtt{l} := \mathtt{l} + 47 \\ \{\mathtt{l} = l_0 + 47 \wedge \mathtt{h} = h_1\} \end{array}$$

For the program in (4), we can choose l_1 as 27; for the program in (5), we simply choose l_1 as l_0.

We next address the programs that are *not* secure, and show (cf. our Observation) that no matter how l_1 has been chosen, we can find h_0 such that it does *not* hold that

$$\begin{array}{l} \{\mathtt{l} = l_0 \wedge \mathtt{h} = h_0\} \\ \qquad P \\ \{\mathtt{l} = l_1\} \end{array}$$

For the programs in (2) and (6), we can just pick an h_0 different from $l_1 - 7$, say $h_0 = l_1$. For clearly it does not hold that

$$\begin{array}{l} \{\mathtt{l} = l_0 \wedge \mathtt{h} = l_1\} \\ \qquad \mathtt{l} := \mathtt{h} + 7 \\ \{\mathtt{l} = l_1\} \end{array}$$

For the program in (7), we proceed by cases on l_1: if l_1 is zero, then we can choose (among many possibilities) h_0 to be 2345 since it does not hold that

$$\begin{array}{l} \{\mathtt{l} = l_0 \wedge \mathtt{h} = 2345\} \\ \qquad \mathtt{if}\ \mathtt{h} = 6789\ \mathtt{then}\ \mathtt{l} := 0\ \mathtt{else}\ \mathtt{l} := 1\ \mathtt{fi} \\ \{\mathtt{l} = 0\} \end{array}$$

Alternatively, if l_1 is one, then we choose h_0 to be 6789 since it does not hold that

$$\{\mathtt{l} = l_0 \wedge \mathtt{h} = 6789\}$$
$$\mathtt{if}\ \mathtt{h} = 6789\ \mathtt{then}\ \mathtt{l} := 0\ \mathtt{else}\ \mathtt{l} := 1\ \mathtt{fi}$$
$$\{\mathtt{l} = 1\}$$

(If l_1 is neither zero nor one, we can choose any value for h_0.)

8.4 Declassification

A severe limitation of our theory is exposed by the last example (7) which is considered insecure even though very little information may actually be leaked to the attacker. Think of `h` as denoting a PIN code, with the attacker testing whether it happens to be 6789; if the PIN codes were selected randomly, the chance of the test revealing the PIN code is very small (1 to 10,000). It is currently an important challenge for research in (language based) security to formalize these considerations!

8.5 Data Integrity

We might consider an alternative interpretation of the identifiers `l` and `h`: `l` denotes a licensed entity, whereas `h` denotes a hacked (untrustworthy) entity. The integrity requirement is now:

> Licensed data should *not* depend on hacked data.

It is interesting to notice that the framework described on the preceding pages covers also that situation! In particular, a program satisfies the above integrity requirement if and only if it is considered secure (according to our Definition). For example, (4) is safe as the licensed identifier `l` will eventually contain 27 which does not depend on hacked data, whereas (7) is unsafe as the value of the hacked identifier `h` influences the value of the licensed identifier.

9 Procedures

A convenient feature, present in almost all programming languages, is the ability to define *procedures*; these are "named abstractions" of commonly used command sequences. In these notes, we shall consider procedure declarations of the form[10]

```
proc p (var x, y)
    local ...
    begin
        C
    end
```

where the procedure p has *body* C and *formal parameters* `x` and `y`; the body may refer to these parameters and possibly also to the *local identifiers* (declared after `local`) but *not* to any other ("global") identifiers.

A program P is now a sequence of procedure declarations, followed by a command (running the program amounts to executing that command). The syntax of commands was defined in Section 4 and is now extended to include *procedure calls*:

$$\begin{array}{rcl} C & ::= & \ldots \\ & \mid & \texttt{call}\ p(x_1, x_2) \end{array}$$

Here x_1 and x_2 are the *actual parameters*; note that they must be identifiers and we shall even require them to be distinct.

As an example, consider the procedure `swap` with declaration

```
proc swap (var x, y)
    local t
    begin
        t := x;
        x := y;
        y := t
    end
```

The following code segment contains a call of `swap`; after the call, we would expect that $\texttt{z} = 7$ and that $\texttt{w} = 3$.

```
    z := 3;
    w := 7;
    call swap(z, w)
```

[10] The generalization to an arbitrary number of formal parameters is immediate.

This example shows that our parameter-passing mechanism[11] is "call-by-reference" (as indicated by the keyword `var`): what is passed to the procedure is the "location" of the actual parameter, not just its value.

The body of a procedure may contain calls to other procedures. A procedure may even call itself (directly or indirectly), in which case we say that it is *recursive*. In Section 5.1 we implemented the factorial function using *iteration* (that is, `while` loops); below is an implementation which uses recursion and which thus more closely matches the recursive definition (given in Footnote 1) of the factorial function.

```
proc fact (var x, y)
    local t,r
    begin
        if x = 0
        then
            y := 1
        else
            t := x − 1;
            call fact(t, r);
            y := x ∗ r
        fi
    end
```

9.1 Contracts

As is the case for a program, also a procedure should come with a specification, which can be viewed as a "contract" for its use. For example, we might want a procedure `twice` with the contract[12]

proc twice (var x, y)
$\forall a, b$
$\{\mathtt{x} = a \wedge \mathtt{y} = b\}$
    C
$\{\mathtt{x} = 2a \wedge \mathtt{y} = 2b\}$

This contract promises that for a call to `twice`, the following property holds for the identifiers provided as actual parameters: no matter what their values

[11]It is not difficult to extend our theory to other parameter passing mechanisms.
[12]Alternatively, one often uses the term "summary".

were *before* the call, their values *after* the call will be twice as big.

The natural way to implement `twice` is

$$
\begin{array}{l}
\texttt{proc twice (var x, y)} \\
\quad \texttt{begin} \\
\quad\quad \texttt{x} := 2 * \texttt{x}; \\
\quad\quad \texttt{y} := 2 * \texttt{y} \\
\quad \texttt{end}
\end{array}
$$

Note that this would *not* work if we had not required the actual parameters to be *distinct* identifiers, as the command `call twice(w, w)` would in effect multiply `w` by 4.

The contract for `swap` is as follows:

$$
\begin{array}{l}
\texttt{proc swap (var x, y)} \\
\forall a, b \\
\{\texttt{x} = a \wedge \texttt{y} = b\} \\
\quad C \\
\{\texttt{x} = b \wedge \texttt{y} = a\}
\end{array}
$$

and we can easily verify that its implementation fulfills that contract: for arbitrary a and b, we have

$$
\begin{array}{ll}
\{\texttt{x} = a \wedge \texttt{y} = b\} & \\
\{\texttt{y} = b \wedge \texttt{x} = a\} & \textbf{Implies} \\
\quad \texttt{t} := \texttt{x}; & \\
\{\texttt{y} = b \wedge \texttt{t} = a\} & \textbf{Assignment} \\
\quad \texttt{x} := \texttt{y}; & \\
\{\texttt{x} = b \wedge \texttt{t} = a\} & \textbf{Assignment} \\
\quad \texttt{y} := \texttt{t} & \\
\{\texttt{x} = b \wedge \texttt{y} = a\} & \textbf{Assignment}
\end{array}
$$

The contract for `fact` is as follows:

$$
\begin{array}{l}
\texttt{proc fact (var x, y)} \\
\forall a \\
\{\texttt{x} = a \wedge \texttt{x} \geq 0\} \\
\quad C \\
\{\texttt{y} = \mathrm{fac}(a)\}
\end{array}
$$

To verify that the implementation of `fact` satisfies that specification, we must first address how to reason about procedure calls.

9.2 Rule for Procedure Calls

Given a procedure p with contract

$\texttt{proc}\ p\ (\texttt{var x}, \texttt{y})$
$\forall a_1, a_2$
$\{\phi_1(\texttt{x}, \texttt{y}, a_1, a_2)\}$
$\quad C$
$\{\phi_2(\texttt{x}, \texttt{y}, a_1, a_2)\}$

We might expect that for calls of p, we have the rule

$$\begin{array}{ll} & \{\phi_1(x_1, x_2, c_1, c_2)\} \\ & \quad \texttt{call}\ p(x_1,x_2) \\ \triangleright & \{\phi_2(x_1, x_2, c_1, c_2)\} \end{array}$$

While this rule is sound (since x_1 and x_2 denote distinct identifiers), it is not immediately useful, in that assertions unrelated to the procedure call are forgotten afterwards. To allow such an assertion ψ to be remembered, we propose the rule

$$\begin{array}{ll} & \{\phi_1(x_1, x_2, c_1, c_2) \wedge \psi\} \\ & \quad \texttt{call}\ p(x_1, x_2) \\ \triangleright & \{\phi_2(x_1, x_2, c_1, c_2) \wedge \psi\} \end{array}$$

We must require that ψ is indeed unrelated to the procedure call; due to our assumption that the body C manipulates no global identifiers, it is sufficient to demand that the identifiers denoted by x_1 and x_2 do not occur in ψ. To see the need for this restriction, consider the purported annotation below (where the role of ψ is played by the assertion $2\texttt{w} = 14$):

$\{\texttt{z} = 3 \wedge \texttt{w} = 7 \wedge 2\texttt{w} = 14\}$
$\quad \texttt{call swap}(\texttt{z}, \texttt{w})$
$\{\texttt{z} = 7 \wedge \texttt{w} = 3 \wedge 2\texttt{w} = 14\}$

Clearly, this annotation is not correct, since after the call, $2\texttt{w}$ equals 6 rather than 14.

As an extra twist, it is convenient (as we shall see in our examples) to allow c_1 and c_2 to be existentially quantified. We are now ready for

Definition 9.1. Assuming that x_1 and x_2 denote *distinct* identifiers which are *not* free in ψ, we have the following proof rule for procedure calls:

$$
\begin{array}{lll}
 & \{\exists c_1 \exists c_2 (\phi_1(x_1, x_2, c_1, c_2) \wedge \psi)\} & \\
 & \quad \texttt{call } p(x_1, x_2) & \\
\triangleright & \{\exists c_1 \exists c_2 (\phi_2(x_1, x_2, c_1, c_2) \wedge \psi)\} & \textbf{Call}
\end{array}
$$

□

EXAMPLE 9.2. Calling `twice` with arguments `z` and `w` satisfying $\texttt{z} \leq 4$ and $\texttt{w} \geq 7$, establishes $\texttt{z} \leq 8$ and $\texttt{w} \geq 14$. This is formally verified by the following well-annotation, where in the application of **Call**, the role of ψ is played by the assertion $c_1 \leq 4 \wedge c_2 \geq 7$.

$$
\begin{array}{ll}
\{\texttt{z} \leq 4 \wedge \texttt{w} \geq 7\} & \\
\{\exists c_1 \exists c_2 (\texttt{z} = c_1 \wedge \texttt{w} = c_2 \wedge c_1 \leq 4 \wedge c_2 \geq 7)\} & \textbf{Implies} \\
\quad \texttt{call twice}(\texttt{z}, \texttt{w}) & \\
\{\exists c_1 \exists c_2 (\texttt{z} = 2c_1 \wedge \texttt{w} = 2c_2 \wedge c_1 \leq 4 \wedge c_2 \geq 7)\} & \textbf{Call} \\
\{\texttt{z} \leq 8 \wedge \texttt{w} \geq 14\} & \textbf{Implies}
\end{array}
$$

□

EXAMPLE 9.3. Calling `swap` with arguments `z` and `w` such that $\texttt{z} > \texttt{w}$, establishes $\texttt{w} > \texttt{z}$. This is formally verified by the following well-annotation, where in the application of **Call**, the role of ψ is played by the assertion $c_1 > c_2$.

$$
\begin{array}{ll}
\{\texttt{z} > \texttt{w}\} & \\
\{\exists c_1 \exists c_2 (\texttt{z} = c_1 \wedge \texttt{w} = c_2 \wedge c_1 > c_2)\} & \textbf{Implies} \\
\quad \texttt{call swap}(\texttt{z}, \texttt{w}) & \\
\{\exists c_1 \exists c_2 (\texttt{z} = c_2 \wedge \texttt{w} = c_1 \wedge c_1 > c_2)\} & \textbf{Call} \\
\{\texttt{w} > \texttt{z}\} & \textbf{Implies}
\end{array}
$$

□

We are now ready to prove that `fact` fulfills its contracts. That is, given a, we must prove

$$
\begin{array}{l}
\{\texttt{x} = a \wedge \texttt{x} \geq 0\} \\
\qquad \texttt{if x} = 0
\end{array}
$$

$\quad\quad \texttt{then}$

$\quad\quad\quad \texttt{y} := 1$

$\quad\quad \texttt{else}$

$\quad\quad\quad \texttt{t} := \texttt{x} - 1;$

$\quad\quad\quad \texttt{call fact}(\texttt{t}, \texttt{r});$

$\quad\quad\quad \texttt{y} := \texttt{x} * \texttt{r}$

$\quad\quad \texttt{fi}$

$\{\texttt{y} = \text{fac}(a)\}$

But this follows from the following well-annotation:

$\{\texttt{x} = a \wedge \texttt{x} \geq 0\}$	
$\quad\quad \texttt{if x} = 0$	
$\quad\quad \texttt{then}$	
$\{\texttt{x} = a \wedge \texttt{x} \geq 0 \wedge \texttt{x} = 0\}$	**IfTrue**
$\{1 = \text{fac}(a)\}$	**Implies**
$\quad\quad\quad \texttt{y} := 1$	
$\{\texttt{y} = \text{fac}(a)\}$	**Assignment**
$\quad\quad \texttt{else}$	
$\{\texttt{x} = a \wedge \texttt{x} \geq 0 \wedge \texttt{x} \neq 0\}$	**IfFalse**
$\{\exists c(\texttt{x} - 1 = c \wedge \texttt{x} - 1 \geq 0 \wedge \texttt{x} = a \wedge \texttt{x} = c + 1)\}$	**Implies**
$\quad\quad\quad \texttt{t} := \texttt{x} - 1;$	
$\{\exists c(\texttt{t} = c \wedge \texttt{t} \geq 0 \wedge \texttt{x} = a \wedge \texttt{x} = c + 1)\}$	**Assignment**
$\quad\quad\quad \texttt{call fact}(\texttt{t}, \texttt{r});$	
$\{\exists c(\texttt{r} = \text{fac}(c) \wedge \texttt{x} = a \wedge \texttt{x} = c + 1)\}$	**Call**
$\{\texttt{xr} = \text{fac}(a)\}$	**Implies**
$\quad\quad\quad \texttt{y} := \texttt{x} * \texttt{r}$	
$\{\texttt{y} = \text{fac}(a)\}$	**Assignment**
$\quad\quad \texttt{fi}$	
$\{\texttt{y} = \text{fac}(a)\}$	**IfEnd**

10 Arrays

Until now, we have only considered simple data structures like integers; in this section we shall consider *arrays*. An array can hold a sequence of values (just like a linked list can), where each element of that sequence can be accessed, and mutated, directly (unlike what is the case for a linked list, where one has to follow a chain of pointers).

Below is depicted an array a with 5 elements: 7,3,9,5,2.

0	1	2	3	4
7	3	9	5	2

We thus have $a[0] = 7$, $a[1] = 3$, etc.

Individual elements of arrays can be updated; after issuing the command $\texttt{a}[3] := 8$ the array a will now look like

0	1	2	3	4
7	3	9	8	2

We shall talk about two arrays being *permutations* of each other if they contain the same elements, though perhaps in different order. This is, e.g., the case for the two arrays given below:

0	1	2	3	4
8	3	9	8	2

0	1	2	3	4
3	9	2	8	8

We shall write $\mathrm{perm}(a_1, a_2)$ if a_1 and a_2 are permutations of each other.

10.1 Verifying Programs Reading Arrays

Let us first consider programs which are *read-only* on arrays. For such programs, the verification principles from the previous sections carry through unchanged[13].

As an example, let us construct a program that stores in `m` the maximum of the first `k` elements of the array `a`, that is the maximum of $\texttt{a}[0], \ldots, \texttt{a}[\texttt{k} - 1]$. We assume that $\texttt{k} \geq 1$, and that `a` indeed has at least `k` elements.

Assuming that all identifiers have non-negative values (greatly improving readability, as otherwise assertions of the form $j \geq 0$ would have to be inserted numerous places), the desired postcondition can be expressed as

$$\forall j (j < \texttt{k} \rightarrow \texttt{a}[j] \leq \texttt{m}) \quad \wedge \quad \exists j (j < \texttt{k} \wedge \texttt{a}[j] = \texttt{m})$$

We shall need a loop, and it seems reasonable to guess that its test should be $\texttt{i} \neq \texttt{k}$ and its invariant should be

$$\phi : \forall j (j < \texttt{i} \rightarrow \texttt{a}[j] \leq \texttt{m}) \quad \wedge \quad \exists j (j < \texttt{i} \wedge \texttt{a}[j] = \texttt{m})$$

[13] For programs manipulating arrays, loop invariants and other properties will almost certainly contain quantifiers, whereas for programs without arrays, invariants can often be expressed in propositional logic.

since then the loop invariant, together with the negation of the loop test, will imply the postcondition. With the aim of establishing and maintaining the invariant ϕ, we construct the following program:

```
i := 1;
m := a[0];
while i ≠ k do
    if a[i] > m
    then
        m := a[i];
        i := i + 1
    else
        i := i + 1
    fi
od
```

To prove the correctness of this program, we annotate it:

$\{\mathtt{k} \geq 1\}$

$\{\forall j(j < 1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[0]) \wedge$
$\exists j(j < 1 \wedge \mathtt{a}[j] = \mathtt{a}[0])\}$ **Implies**(A)

    i := 1;

$\{\forall j(j < \mathtt{i} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[0]) \wedge$
$\exists j(j < \mathtt{i} \wedge \mathtt{a}[j] = \mathtt{a}[0])\}$ **Assignment**

    m := a[0];

$\{\phi\}$ **Assignment**

    while i ≠ k do

$\{\phi \wedge \mathtt{i} \neq \mathtt{k}\}$ **WhileTrue**

        if a[i] > m
        then

$\{\phi \wedge \mathtt{i} \neq \mathtt{k} \wedge \mathtt{a}[\mathtt{i}] > \mathtt{m}\}$ **IfTrue**

$\{\forall j(j < \mathtt{i}+1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[\mathtt{i}]) \wedge$
$\exists j(j < \mathtt{i}+1 \wedge \mathtt{a}[j] = \mathtt{a}[\mathtt{i}])\}$ **Implies**(B)

            m := a[i];

$\{\forall j(j < \mathtt{i}+1 \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \wedge$
$\exists j(j < \mathtt{i}+1 \wedge \mathtt{a}[j] = \mathtt{m})\}$ **Assignment**

            i := i + 1

$\{\phi\}$ **Assignment**

`else`

$\{\phi \land \mathtt{i} \neq \mathtt{k} \land \mathtt{a}[\mathtt{i}] \leq \mathtt{m}\}$ **IfFalse**

$\{\forall j(j < \mathtt{i}+1 \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \land$
$\exists j(j < \mathtt{i}+1 \land \mathtt{a}[j] = \mathtt{m})\}$ **Implies**(C)

$\mathtt{i} := \mathtt{i}+1$

$\{\phi\}$ **Assignment**

`fi`

$\{\phi\}$ **IfEnd**

`od`

$\{\phi \land \mathtt{i}{=}\mathtt{k}\}$ **WhileFalse**

$\{\forall j(j < \mathtt{k} \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \land$
$\exists j(j < \mathtt{k} \land \mathtt{a}[j] = \mathtt{m})\}$ **Implies**

Below we shall show the validity of (A) and (B) and (C); it is then an easy exercise to check the validity of the rest of the assertions.

To see that (A) is valid, observe that 0 is the only j such that $j < 1$.

To see that (B) is valid, we must prove that

$$\forall j(j < \mathtt{i} \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \text{ and} \quad (1)$$
$$\exists j(j < \mathtt{i} \land \mathtt{a}[j] = \mathtt{m}) \text{ and} \quad (2)$$
$$\mathtt{a}[\mathtt{i}] > \mathtt{m} \quad (3)$$

implies

$$\forall j(j < \mathtt{i}+1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[\mathtt{i}]) \text{ and} \quad (4)$$
$$\exists j(j < \mathtt{i}+1 \land \mathtt{a}[j] = \mathtt{a}[\mathtt{i}]). \quad (5)$$

To establish (4), let j be given such that $j < \mathtt{i}+1$: if $j = \mathtt{i}$, the claim is trivial; otherwise, $j < \mathtt{i}$ and the claim follows from (1) and (3). For (5), we can use $j = \mathtt{i}$.

To see that (C) is valid, we must prove that

$$\forall j(j < \mathtt{i} \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \text{ and} \quad (6)$$
$$\exists j(j < \mathtt{i} \land \mathtt{a}[j] = \mathtt{m}) \text{ and} \quad (7)$$
$$\mathtt{a}[\mathtt{i}] \leq \mathtt{m} \quad (8)$$

implies

$$\forall j(j < \mathtt{i}+1 \rightarrow \mathtt{a}[j] \leq \mathtt{m}) \text{ and} \tag{9}$$
$$\exists j(j < \mathtt{i}+1 \wedge \mathtt{a}[j] = \mathtt{m}). \tag{10}$$

To establish (9), let j be given such that $j < \mathtt{i}+1$. If $j = \mathtt{i}$, the claim follows from (8). Otherwise, $j < \mathtt{i}$ and the claim follows from (6). Finally, (10) follows from (7).

10.2 Verifying Programs Updating Arrays

Next we consider programs which also *write* on arrays, that is, contain commands of the form $\mathtt{a}[\mathtt{i}] := E$. For such assignments, we want to apply the proof rule

$\{\psi(E)\}$
    $\mathtt{x} := E$
▷ $\{\psi(x)\}$ **Assignment**

But if we apply that rule *naively* to the assignment $\mathtt{a}[2] := \mathtt{x}$ and the post-condition $\forall j(j < 10 \rightarrow \mathtt{a}[j] > 5)$, substituting the right hand side of the assignment for the left hand side, we would infer (since $\mathtt{a}[2]$ does not occur in the postcondition) that the following program is well-annotated:

$\{\forall j(j < 10 \rightarrow \mathtt{a}[j] > 5)\}$
    $\mathtt{a}[2] := \mathtt{x}$
$\{\forall j(j < 10 \rightarrow \mathtt{a}[j] > 5)\}$

This is clearly unsound, as can be seen by taking $\mathtt{x} = 3$.

Instead, the proper treatment is to interpret an assignment $\mathtt{a}[\mathtt{i}] := E$ as being really the assignment

$$\mathtt{a} := \mathtt{a}\{\mathtt{i} \mapsto E\}$$

That is, we assign to $\mathtt{a}$ an array that is like $\mathtt{a}$, except that in position $\mathtt{i}$ it behaves like E. More formally, we have

$$\begin{aligned} \mathtt{a}\{\mathtt{i} \mapsto E\}[j] &= E && \text{if } j = \mathtt{i} \\ \mathtt{a}\{\mathtt{i} \mapsto E\}[j] &= \mathtt{a}[j] && \text{if } j \neq \mathtt{i} \end{aligned}$$

Then, in the above example, we get the well-annotated program

$\{\forall j(j < 10 \to \mathtt{a}\{2 \mapsto \mathtt{x}\}[j] > 5)\}$
    $\mathtt{a}[2] := \mathtt{x}$
$\{\forall j(j < 10 \to \mathtt{a}[j] > 5)\}$

where the precondition can be simplified to

$$\forall j((j < 10 \land j \neq 2) \to \mathtt{a}[j] > 5) \land \mathtt{x} > 5$$

which is as expected.

As a larger example, let us construct a program that rearranges the first `k` elements of an array `a` such that the highest element is placed in position number 0.

The desired postcondition can be expressed as follows:

$$\forall j(j < \mathtt{k} \to \mathtt{a}[j] \leq \mathtt{a}[0]) \ \land \ \mathrm{perm}(\mathtt{a}, a_0)$$

where the logical variable a_0 denotes the initial value of a; the latter condition $\mathrm{perm}(\mathtt{a}, a_0)$ is also part of the precondition. We shall need a loop, and it seems reasonable to guess that its test should be $\mathtt{i} \neq \mathtt{k}$ and its invariant should be

$$\psi : \forall j(j < \mathtt{i} \to \mathtt{a}[j] \leq \mathtt{a}[0]) \ \land \ \mathrm{perm}(\mathtt{a}, a_0)$$

since then the loop invariant, together with the negation of the loop test, will imply the postcondition. With the aim of establishing and maintaining the invariant ψ, we construct the following program:

```
i := 1;
while i ≠ k do
    if a[i] > a[0]
    then
        t := a[0];
        a[0] := a[i];
        a[i] := t;
        i := i + 1
    else
        i := i + 1
    fi
od
```

To prove the correctness of this program, we annotate it, as done in Fig. 1. Below we shall show the validity of (D); it is then an easy exercise to check the validity of the rest of the assertions.

Let $a' = \mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{a}[0]\}$; we must prove that

$$\forall j(j < \mathtt{i} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[0]) \text{ and} \tag{11}$$
$$\text{perm}(\mathtt{a}, a_0) \text{ and} \tag{12}$$
$$\mathtt{a}[\mathtt{i}] > \mathtt{a}[0] \tag{13}$$

implies

$$\forall j(j < \mathtt{i} + 1 \rightarrow a'[j] \leq a'[0]) \text{ and} \tag{14}$$
$$\text{perm}(a', a_0) \tag{15}$$

Clearly a' is a permutation of $\mathtt{a}$, so (15) follows from (12). To show (14), let $j < \mathtt{i}+1$ be given; we must show that $a'[j] \leq a'[0]$ which is trivial if $j = 0$ so assume that $0 < j < \mathtt{i} + 1$. Since $a'[0] = \mathtt{a}[\mathtt{i}]$, our task can be accomplished by showing that

$$a'[j] \leq \mathtt{a}[\mathtt{i}].$$

We do so by a case analysis on the value of j. If $j = \mathtt{i}$, the claim follows from (13) since $a'[j] = \mathtt{a}[0]$. Otherwise, $0 < j < \mathtt{i}$ and therefore $a'[j] = \mathtt{a}[j]$; the claim thus boils down to showing $\mathtt{a}[j] \leq \mathtt{a}[\mathtt{i}]$ which follows from (11) and (13).

References

[1] Jon Barwise and John Etchemendy. *Language, Proof and Logic.* CSLI Publications, 1999.

[2] Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In *Workshop on Issues in the Theory of Security (WITS'03). Affiliated to ETAPS 2003, Warsaw, Poland.*, 2003.

[3] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[4] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2000.

$\{\text{perm}(\mathtt{a}, a_0)\}$

$\{\forall j(j < 1 \to \mathtt{a}[j] \leq \mathtt{a}[0]) \ \wedge \ \text{perm}(\mathtt{a}, a_0)\}$ **Implies**

$\mathtt{i} := 1;$

$\{\psi\}$ **Assignment**

$\mathtt{while}\ \mathtt{i} \neq \mathtt{k}\ \mathtt{do}$

$\{\psi \wedge \mathtt{i} \neq \mathtt{k}\}$ **WhileTrue**

$\mathtt{if}\ \mathtt{a}[\mathtt{i}] > \mathtt{a}[0]$

$\mathtt{then}$

$\{\psi \wedge \mathtt{i} \neq \mathtt{k} \wedge \mathtt{a}[\mathtt{i}] > \mathtt{a}[0]\}$ **IfTrue**

$\{\forall j(j < \mathtt{i} + 1 \to \mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{a}[0]\}[j] \leq \mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{a}[0]\}[0])$
$\wedge\ \text{perm}(\mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{a}[0]\}, a_0)\}$ **Implies**(D)

$\mathtt{t} := \mathtt{a}[0];$

$\{\forall j(j < \mathtt{i} + 1 \to \mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{t}\}[j] \leq \mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{t}\}[0])$
$\wedge\ \text{perm}(\mathtt{a}\{0 \mapsto \mathtt{a}[\mathtt{i}]\}\{\mathtt{i} \mapsto \mathtt{t}\}, a_0)\}$ **Assignment**

$\mathtt{a}[0] := \mathtt{a}[\mathtt{i}];$

$\{\forall j(j < \mathtt{i} + 1 \to \mathtt{a}\{\mathtt{i} \mapsto \mathtt{t}\}[j] \leq \mathtt{a}\{\mathtt{i} \mapsto \mathtt{t}\}[0])$
$\wedge\ \text{perm}(\mathtt{a}\{\mathtt{i} \mapsto \mathtt{t}\}, a_0)\}$ **Assignment**

$\mathtt{a}[\mathtt{i}] := \mathtt{t};$

$\{\forall j(j < \mathtt{i} + 1 \to \mathtt{a}[j] \leq \mathtt{a}[0])$
$\wedge\ \text{perm}(\mathtt{a}, a_0)\}$ **Assignment**

$\mathtt{i} := \mathtt{i} + 1$

$\{\psi\}$ **Assignment**

$\mathtt{else}$

$\{\psi \wedge \mathtt{i} \neq \mathtt{k} \wedge \mathtt{a}[\mathtt{i}] \leq \mathtt{a}[0]\}$ **IfFalse**

$\{\forall j(j < \mathtt{i} + 1 \to \mathtt{a}[j] \leq \mathtt{a}[0])$
$\wedge\ \text{perm}(\mathtt{a}, a_0)\}$ **Implies**

$\mathtt{i} := \mathtt{i} + 1$

$\{\psi\}$ **Assignment**

$\mathtt{fi}$

$\{\psi\}$ **IfEnd**

$\mathtt{od}$

$\{\psi \wedge \mathtt{i} = \mathtt{k}\}$ **WhileFalse**

$\{\forall j(j < \mathtt{k} \to \mathtt{a}[j] \leq \mathtt{a}[0]) \ \wedge \ \text{perm}(\mathtt{a}, a_0)\}$ **Implies**

Figure 1: A well-annotated program for putting the highest array value first.

A Some Previous Exam Questions

Question I (Fall 2002)

Given a positive integer x, we may define its integer logarithm as the largest integer y with the property that $2^y \leq x$. (Example: the integer logarithm of 10 is 3, since $2^3 = 8 \leq 10$ but $2^4 = 16 > 10$.)

We claim that if the program below terminates then `y` will denote the integer logarithm of `x`. Prove this claim by well-annotating the program. This includes

a. formalizing the desired postcondition of the program;

b. coming up with a suitable invariant for the `while` loop.

You can ignore the implicit demand that the value of `x` should not change (that is, don't bother about introducing a logical variable x_0).

```
y := 0;

w := 2;

while w ≤ x do

    y := y + 1;

    w := 2 * w

od
```

This question carried 8 out of 25 points in a 50 minutes test.

Proposed Answer for I

The desired postcondition is

$$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge 2^{\mathtt{y}+1} > \mathtt{x}\}$$

which can be established using the loop invariant

$$2^{\mathtt{y}} \leq \mathtt{x} \wedge \mathtt{w} = 2^{\mathtt{y}+1}$$

The program can now be well-annotated:

$\{\mathtt{x} \geq 1\}$	Precondition
$\{2^0 \leq \mathtt{x} \wedge 2 = 2^{0+1}\}$	**Implies**
$\quad \mathtt{y} := 0;$	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge 2 = 2^{\mathtt{y}+1}\}$	**Assignment**
$\quad \mathtt{w} := 2;$	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge \mathtt{w} = 2^{\mathtt{y}+1}\}$	**Assignment**
$\quad \mathtt{while}\ \mathtt{w} \leq \mathtt{x}\ \mathtt{do}$	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge \mathtt{w} = 2^{\mathtt{y}+1} \wedge \mathtt{w} \leq \mathtt{x}\}$	**WhileTrue**
$\{2^{\mathtt{y}+1} \leq \mathtt{x} \wedge 2\mathtt{w} = 2^{\mathtt{y}+1+1}\}$	**Implies**
$\qquad \mathtt{y} := \mathtt{y} + 1;$	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge 2\mathtt{w} = 2^{\mathtt{y}+1}\}$	**Assignment**
$\qquad \mathtt{w} := 2 * \mathtt{w}$	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge \mathtt{w} = 2^{\mathtt{y}+1}\}$	**Assignment**
$\quad \mathtt{od}$	
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge \mathtt{w} = 2^{\mathtt{y}+1} \wedge \mathtt{w} > \mathtt{x}\}$	**WhileFalse**
$\{2^{\mathtt{y}} \leq \mathtt{x} \wedge 2^{\mathtt{y}+1} > \mathtt{x}\}$	**Implies**

Question II (Spring 2003)

Assume we have a predicate `divides` with the property that

$\texttt{divides}(x, w) = \textsf{TRUE}$ if and only if there exists an integer z with $w = xz$.

(For example, $\texttt{divides}(3, 6) = \textsf{TRUE}$ but $\texttt{divides}(3, 7) = \textsf{FALSE}$.) Next we define a predicate `P` as follows:

$$\texttt{P}(n, q) = \textsf{TRUE} \text{ if and only if } \forall i[(1 < i \wedge i < q) \rightarrow \neg\texttt{divides}(i, n)]$$

(For example, $\texttt{P}(25, 5) = \textsf{TRUE}$ but $\texttt{P}(25, 6) = \textsf{FALSE}$.) By this definition,

for $n \geq 2$, n is a prime if and only if $\texttt{P}(n, n)$.

The following program is supposed to decide whether its input `n` is a prime, and store the answer in the boolean identifier `prime`. (In the last program line, if $\texttt{q} = \texttt{n}$ then $\textsf{TRUE}$ is assigned to `prime`, otherwise $\textsf{FALSE}$ is assigned to `prime`.)

```
{n ≥ 2}
    q := 2;
    while ¬divides(q, n) do
        q := q + 1
    od;
    prime := (q = n)
{prime ↔ P(n, n)}
```

It turns out that a suitable invariant for the `while` loop is

$$2 \leq \texttt{q} \wedge \texttt{q} \leq \texttt{n} \wedge \texttt{P}(\texttt{n}, \texttt{q})$$

Given that invariant, write down a well-annotated version of the program so as to demonstrate that the program does indeed satisfy its specification.

We shall be interested in total correctness, so you must also argue that the program always terminates. And for each instance of the rule **Implies**, you must explicitly argue that the assertion is indeed a logical consequence of the previous assertion.

This question carried 8 out of 25 points in a 90 minutes test.
In retrospective, though, this question was way too hard.

Proposed Answer for II

$$
\begin{array}{ll}
\{\mathtt{n} \geq 2\} & \\
\{2 \leq 2 \leq \mathtt{n} \wedge \mathtt{P}(\mathtt{n}, 2)\} & \textbf{Implies}^1 \\
\quad \mathtt{q} := 2; & \\
\{2 \leq \mathtt{q} \leq \mathtt{n} \wedge \mathtt{P}(\mathtt{n}, \mathtt{q})\} & \textbf{Assignment} \\
\quad \mathtt{while} \ \neg\mathtt{divides}(\mathtt{q}, \mathtt{n}) \ \mathtt{do} & \\
\{2 \leq \mathtt{q} \leq \mathtt{n} \wedge \mathtt{P}(\mathtt{n}, \mathtt{q}) \wedge \neg\mathtt{divides}(\mathtt{q}, \mathtt{n})\} & \textbf{WhileTrue} \\
\{2 \leq \mathtt{q} + 1 \leq \mathtt{n} \wedge \mathtt{P}(\mathtt{n}, \mathtt{q} + 1)\} & \textbf{Implies}^2 \\
\qquad \mathtt{q} := \mathtt{q} + 1 & \\
\{2 \leq \mathtt{q} \leq \mathtt{n} \wedge \mathtt{P}(\mathtt{n}, \mathtt{q})\} & \textbf{Assignment} \\
\quad \mathtt{od}; & \\
\{2 \leq \mathtt{q} \leq \mathtt{n} \wedge \mathtt{P}(\mathtt{n}, \mathtt{q}) \wedge \mathtt{divides}(\mathtt{q}, \mathtt{n})\} & \textbf{WhileFalse} \\
\{(\mathtt{q} = \mathtt{n}) \leftrightarrow \mathtt{P}(\mathtt{n}, \mathtt{n})\} & \textbf{Implies}^3 \\
\quad \mathtt{prime} := (\mathtt{q} = \mathtt{n}) & \\
\{\mathtt{prime} \leftrightarrow \mathtt{P}(\mathtt{n}, \mathtt{n})\} & \textbf{Assignment}
\end{array}
$$

Here **Implies**^{1} is justified, since $\mathtt{P}(\mathtt{n}, 2)$ is vacuously true (as there is no integer i such that $1 < i < 2$).

To see that **Implies**^{2} is justified, observe that

- $\mathtt{P}(\mathtt{n}, \mathtt{q}) \wedge \neg\mathtt{divides}(\mathtt{q}, \mathtt{n})$ implies $\mathtt{P}(\mathtt{n}, \mathtt{q} + 1)$;
- $\neg\mathtt{divides}(\mathtt{q}, \mathtt{n})$ implies $\mathtt{q} \neq \mathtt{n}$.

To see that **Implies**^{3} is justified, observe that

- if $\mathtt{q} = \mathtt{n}$ then clearly $\mathtt{P}(\mathtt{n}, \mathtt{n})$ is a consequence of $\mathtt{P}(\mathtt{n}, \mathtt{q})$;
- if $\mathtt{q} \neq \mathtt{n}$, then $\mathtt{q} < \mathtt{n}$ and from $\mathtt{divides}(\mathtt{q}, \mathtt{n})$ we infer that $\mathtt{P}(\mathtt{n}, \mathtt{n})$ does not hold.

We now show termination, using a proof by contradiction: assume that $\mathtt{q}$ keeps on being incremented. Since initially $\mathtt{q} \leq \mathtt{n}$, at some point it will hold that $\mathtt{q} = \mathtt{n}$, implying that $\mathtt{divides}(\mathtt{q}, \mathtt{n})$. Then we exit the loop, contradicting our assumption.

Question III (Fall 2003)

Below is written a program which given x and n raises x to the power of n and stores the result in y (without changing x or n). An invariant for the while loop is given. Prove that the program is partially correct, by completing the assertions so as to produce a well-annotated program. (You do not need to argue for the validity of any application of **Implies**.)

```
{n ≥ 0}
    y := 1;
    w := x;
    k := n;
{y · w^k = x^n}
    while k ≠ 0 do
        if k is even
        then
            k := k/2;
            w := w * w
        else
            k := k − 1;
            y := y * w
        fi
    od
{y = x^n}
```

This question carried 6 out of 25 points in a 90 minutes test.

Proposed Answer for III

$\{\mathtt{n} \geq 0\}$	
$\{\mathtt{1} \cdot \mathtt{x}^{\mathtt{n}} = \mathtt{x}^{\mathtt{n}}\}$	**Implies**
$\quad \mathtt{y} := 1;$	
$\{\mathtt{y} \cdot \mathtt{x}^{\mathtt{n}} = \mathtt{x}^{\mathtt{n}}\}$	**Assignment**
$\quad \mathtt{w} := \mathtt{x};$	
$\{\mathtt{y} \cdot \mathtt{w}^{\mathtt{n}} = \mathtt{x}^{\mathtt{n}}\}$	**Assignment**
$\quad \mathtt{k} := \mathtt{n};$	
$\{\mathtt{y} \cdot \mathtt{w}^{\mathtt{k}} = \mathtt{x}^{\mathtt{n}}\}$	**Assignment**
$\quad$ `while` $\mathtt{k} \neq 0$ `do`	
$\{\mathtt{y} \cdot \mathtt{w}^{\mathtt{k}} = \mathtt{x}^{\mathtt{n}} \wedge \mathtt{k} \neq 0\}$	**WhileTrue**
$\qquad$ `if` `k` is even	
$\qquad$ `then`	
$\{\mathtt{y} \cdot \mathtt{w}^{\mathtt{k}} = \mathtt{x}^{\mathtt{n}} \wedge \mathtt{k} \neq 0 \wedge \mathtt{k} \text{ is even}\}$	**IfTrue**
$\{\mathtt{y} \cdot (\mathtt{w}^2)^{\mathtt{k}/2} = \mathtt{x}^{\mathtt{n}}\}$	**Implies**
$\qquad\quad \mathtt{k} := \mathtt{k}/2;$	
$\{\mathtt{y} \cdot (\mathtt{w}^2)^{\mathtt{k}} = \mathtt{x}^{\mathtt{n}}\}$	**Assignment**
$\qquad\quad \mathtt{w} := \mathtt{w} * \mathtt{w}$	
$\{\mathtt{y} \cdot \mathtt{w}^{\mathtt{k}} = \mathtt{x}^{\mathtt{n}}\}$	**Assignment**
$\qquad$ `else`	
$\{\mathtt{y} \cdot \mathtt{w}^{\mathtt{k}} = \mathtt{x}^{\mathtt{n}} \wedge \mathtt{k} \neq 0 \wedge \mathtt{k} \text{ is odd}\}$	**IfFalse**
$\{\mathtt{y} \cdot \mathtt{w} \cdot \mathtt{w}^{\mathtt{k}-1} = \mathtt{x}^{\mathtt{n}}\}$	**Implies**
$\qquad\quad \mathtt{k} := \mathtt{k} - 1;$	
$\{\mathtt{y} \cdot \mathtt{w} \cdot \mathtt{w}^{\mathtt{k}} = \mathtt{x}^{\mathtt{n}}\}$	**Assignment**
$\qquad\quad \mathtt{y} := \mathtt{y} * \mathtt{w}$	
$\{\mathtt{y} \cdot \mathtt{w}^{\mathtt{k}} = \mathtt{x}^{\mathtt{n}}\}$	**Assignment**
$\qquad$ `fi`	
$\{\mathtt{y} \cdot \mathtt{w}^{\mathtt{k}} = \mathtt{x}^{\mathtt{n}}\}$	**IfEnd**
$\quad$ `od`	
$\{\mathtt{y} \cdot \mathtt{w}^{\mathtt{k}} = \mathtt{x}^{\mathtt{n}} \wedge \mathtt{k} = 0\}$	**WhileFalse**
$\{\mathtt{y} = \mathtt{x}^{\mathtt{n}}\}$	**Implies**

Question IV (Spring 2004)

We consider an array a with n (> 0) elements, $\mathtt{a}[0] \ldots \mathtt{a}[n-1]$. Below is a program that decides whether a is sorted: if it is, the identifier k ends up having the value $n-1$; if it is not, k ends up having the value n.

The loop invariant, and the postcondition, have been given. Complete the assertions, thus demonstrating that the program satisfies its specification (which actually proves only one part of correctness.) You do not need to argue that the applications of the **Implies** rule are valid.

$\{n > 0\}$

```
    k := 0;
{k < n → ∀j(j < k → a[j] ≤ a[j + 1])}
    while k < n − 1 do

        if a[k] ≤ a[k + 1]
        then


            k := k + 1

        else


            k := n

        fi

    od
```

$\{\mathtt{k} < n \rightarrow \forall j(j < n-1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])\}$

This question carried 6 out of 25 points in a 100 minutes test.

Proposed Answer for IV

$\{n > 0\}$
$\{0 < n \rightarrow \forall j(j < 0 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])\}$ **Implies**
  $\mathtt{k} := 0;$
$\{\mathtt{k} < n \rightarrow \forall j(j < \mathtt{k} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])\}$ **Assignment**
  $\mathtt{while}\ \mathtt{k} < n-1\ \mathtt{do}$
$\{(\mathtt{k} < n \rightarrow \forall j(j < k \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])) \wedge \mathtt{k} < n-1\}$ **WhileTrue**
    $\mathtt{if}\ \mathtt{a}[\mathtt{k}] \leq \mathtt{a}[\mathtt{k}+1]$
    $\mathtt{then}$
$\{(\mathtt{k} < n \rightarrow \forall j(j < \mathtt{k} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])) \wedge \mathtt{k} < n-1 \wedge \mathtt{a}[\mathtt{k}] \leq \mathtt{a}[\mathtt{k}+1]\}$ **IfTrue**
$\{\mathtt{k}+1 < n \rightarrow \forall j(j < \mathtt{k}+1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])\}$ **Implies**
      $\mathtt{k} := \mathtt{k}+1$
$\{\mathtt{k} < n \rightarrow \forall j(j < \mathtt{k} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])\}$ **Assignment**
    $\mathtt{else}$
$\{(\mathtt{k} < n \rightarrow \forall j(j < \mathtt{k} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])) \wedge \mathtt{k} < n-1 \wedge \mathtt{a}[\mathtt{k}] > \mathtt{a}[\mathtt{k}+1]\}$ **IfFalse**
$\{n < n \rightarrow \forall j(j < n \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])\}$ **Implies**
      $\mathtt{k} := n$
$\{\mathtt{k} < n \rightarrow \forall j(j < \mathtt{k} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])\}$ **Assignment**
    $\mathtt{fi}$
$\{\mathtt{k} < n \rightarrow \forall j(j < \mathtt{k} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])\}$ **IfEnd**
  $\mathtt{od}$
$\{(\mathtt{k} < n \rightarrow \forall j(j < \mathtt{k} \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])) \wedge \mathtt{k} \geq n-1\}$ **WhileFalse**
$\{\mathtt{k} < n \rightarrow \forall j(j < n-1 \rightarrow \mathtt{a}[j] \leq \mathtt{a}[j+1])\}$ **Implies**

Question V (Fall 2004)

Consider the program below, which given a positive integer y multiplies it by two (in a silly way), and stores the result in $\mathtt{x}$.

$\{\mathtt{y} = y_0 \wedge y_0 > 0\}$

```
    x := 0;

    while y ≠ 0 do

        y := y − 1

        x := x + 2

    od
```

$\{\mathtt{x} = 2y_0\}$

The main issue in proving correctness is to come up with a suitable invariant. First argue why the following suggested invariants will *not* work:

- $\mathtt{x} \geq 0$
- $\mathtt{x} + 2\mathtt{y} = 2y_0 \wedge \mathtt{x} > 0$
- $\mathtt{x} + (2\mathtt{y})^{(\mathtt{x}+1)} = 2y_0$

It turns out that a suitable invariant is $\underline{\mathtt{x} + 2\mathtt{y} = 2y_0}$. Using that invariant, give a formal proof of (partial) correctness of the above program, by well-annotating it.

This question carried 10 out of 25 points in a 50 minutes test.

Proposed Answer for V

Each of the purported invariants satisfies 2 of the 3 requirements, but...

- $\mathtt{x} \geq 0$ is not strong enough to let us infer the postcondition;
- $\mathtt{x} + 2\mathtt{y} = 2y_0 \wedge \mathtt{x} > 0$ is not established by the preamble;
- $\mathtt{x} + (2\mathtt{y})^{(\mathtt{x}+1)} = 2y_0$ is not maintained by the loop body.

That $\mathtt{x} + 2\mathtt{y} = 2y_0$ is a suitable invariant follows from the well-annotated program

$\{\mathtt{y} = y_0 \wedge y_0 > 0\}$	
$\{0 + 2\mathtt{y} = 2y_0\}$	**Implies**
$\mathtt{x} := 0;$	
$\{\mathtt{x} + 2\mathtt{y} = 2y_0\}$	**Assignment**
`while` $\mathtt{y} \neq 0$ `do`	
$\{\mathtt{x} + 2\mathtt{y} = 2y_0 \wedge \mathtt{y} \neq 0\}$	**WhileTrue**
$\{\mathtt{x} + 2 + 2(\mathtt{y} - 1) = 2y_0\}$	**Implies**
$\mathtt{y} := \mathtt{y} - 1$	
$\{\mathtt{x} + 2 + 2\mathtt{y} = 2y_0\}$	**Assignment**
$\mathtt{x} := \mathtt{x} + 2$	
$\{\mathtt{x} + 2\mathtt{y} = 2y_0\}$	**Assignment**
`od`	
$\{\mathtt{x} + 2\mathtt{y} = 2y_0 \wedge \mathtt{y} = 0\}$	**WhileFalse**
$\{\mathtt{x} = 2y_0\}$	**Implies**