1 Introduction

The hallmark of a science is the avoidance of error.
J. Robert Oppenheimer

- The topic of this course is to present a methodology for

  proving existing computer algorithms and programs correct, and even

  constructing them so that they come out correct "out of the box".

- The reason for the name "Programming techniques" is that the presented methdology

  stems from programming (rather than software engineering) considerations

  develops programming from an ad-hoc activity into something with a solid foundation — namely, applied logic.

1.1 Testing vs. Proving

- Edsger W. Dijkstra (a famous Dutch computer scientist who invented the classic shortest path algorithm (Cormen et al., 2001, Chapter 24.3) bearing his name, among other things), one of the earliest and most authoritative proponents of proving programs correct, stated in his ACM Turing Award lecture (coloring mine):

  > Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand.
  >
  > (Dijkstra, 1972)

  This course examines this approach using tools whose development was initiated by him and his colleagues.

- On the other hand, proving program properties has its own problems, such as:
  - Fully detailed proofs need stating a surprising amount of "obvious" background information. E.g. to prove a sorting algorithm correct, we need that
    - the ordering relation is transitive

      $$\forall x, y, z. x < y \land y < z \Longrightarrow x < z$$

    - or even that the output must consist of the same elements as the input, but in a different order.

    The same problem hinders logical methods in Artificial Intelligence too.
  - Some programming tasks are infeasible or even impossible to describe in sufficient detail needed for proving.

    E.g. in what level of detail would you want to discuss the "correctness" of a GUI? Is it a fixed concept independent of the users? Does it even make sense?

- Hence proving program properties is now seen as complementing (rather than replacing) testing:
  - Proving well-defined program *parts* correct can be done even by hand with the methods described here.

    Then if subsequent testing reveals a bug in the whole program, we can start hunting it in the other, unproved parts — or the unsatisfied and/or unstated assumptions of the proof.
  - Those (parts of) programs which are mission- or even life-critical *must* also be proved correct, not just tested exhaustively.
  - Failing to prove a property may reveal bugs which go unnoticed in testing.

    E.g. Intel tests its microprocessor designs extermely heavily. But in addition, a separate group of engineers selects parts of the designs, and tries to verify their correctness using semi-automatic tools. When they cannot find a proof, they often find a *counterexample:* the part is incorrect, because in such-and-such circumstances it does the wrong thing. Such counterexamples may be situations which would arise only in exceptional circumstances, and might hence be missed by mere testing.

1.2 Logic – Why, How?

- We start with a *specification* for an algorithm or program: a (more or less precise) description of what we want it to do.
- Following Dijkstra, we would like to

  prove that the algorithm we have invented is *correct* wrt. the given specification, or even

  construct our algorithm so that it is guaranteed.
- To prove anything, we need a language where the concept

  “this *claim* <u>entails</u> that *claim*”

  (”tästä *väitteestä* <u>seuraa</u> tuo *väite*”)

  is well and precisely defined.
- Logic in general means the development, study and use of such languages.
- Hence this course will use logic for expressing specifications and properties of its algorithms.
- Different particular logics arise since the *claims* and <u>entailments</u> can differ, e.g.:

  Temporal logics are used when our *claims* must contain time-related words like “eventually”,. . .

  Constructive (a.k.a. intensional) logics arise if we read “. . . <u>entails</u> that there is some x such that. . . ” as requiring “give me a concrete example of such an x, otherwise I won’t believe you!”.
- Computer Science has (at least) three distinct approaches for reasoning about program correctness using logic:

Pre- and postconditions is the approach in this course.

- It is intended for "classical" algorithms which
  1. are called with some input...
  2. run silently until they stop...
  3. and then the caller can see the output.
- Suitable logical conditions are attached to key points in the program source.
- The proof checks that the previous condition entails the next.

Temporal approaches are intended for "non-classical" programs which may run forever and/or interact concurrently with their environment: telecommunication protocols, operating system services, embedded systems, microprocessors,...

- Here it is no longer enough to state and prove that a certain condition holds at a certain point of a program.
- Instead, here the verifier must be able to state and prove claims about whole execution sequences, e.g. "every incoming message must be eventually followed by a reply to its sender (but there can be other messages in-between)".
- Model checking is often used instead of proving:
  1. The externally observable behaviour of the program is expressed as a familiar finite automaton (suitably extended).
  2. Ditto for the opposite of the claim to verify.
  3. Then we check that their intersection automaton is empty. (If not, we have found a counterexample: a possible run which exposes the bug.)

Type-theoretical approaches state the classical conditions as the types for the variables and expressions in the program.

- Hence the programming/specification language must have a very rich type system — and therefore also a very complex compiler.
- It turns out that if the compiler accepts the source code, then the code is itself a proof for the conditions stated in its types — so coding becomes proving and vice versa.
- This kind of coding is best supported in languages where it is easy to treat code as pure mathematical expressions — whereas we usually think of code as a mixture of some math and control commands to the underlying machine.
- This leads naturally into *functional programming languages* such as Haskell. (Peyton Jones, 2003)
- These approaches highlight the connection between modern programming language design and proof theory, which is another subfield of logic. This connecting "specifications-as-types, proofs-as-programs" idea is formally called the *Curry-Howard correspondence* (Sørensen and Urzyczyn, 2006).

- We shall even use logic as a guide in program *design:*
  - A programming problem is often described in natural language and imprecise or even ambiguous words.

  - To get started, we must first turn this description into a precise and unambiguous specification. Logics are good languages for expressing them.
  - We should also have some idea about an approach which might yield an algorithm meeting this specification.

    But such an idea is really a hazy handwaving argument (to yourself at least, if not others) that this approach might work. Why not try to elaborate it into a precise proof?
  - It turns out that the correct code often *writes itself* as a by-product of this elaboration:
    * Coding in this style means repeatedly asking yourself, "What kind of code I need here to get ahead in my proof?"
    * The code needed here can in turn often be seen from the logical formula you are trying to prove here!
  - Logic does not help the programmer to find an idea — but once an idea has been found, it helps in turning it into concrete and correct code.

    (Ideally, we should be debugging our ideas, not our code.)

1.3 Binary Search, Informally

- As an introductory example, let us reason about the correctness of the venerable *binary search* informally, using a mixture of natural language and simple mathematics.

  Later, we shall switch to more formal tools and reasoning steps – to logic.
- Although the algorithmic idea is simple to explain, getting its details just right is notoriously tricky.
- The problem statement is as follows:

  Inputs: An array $A[1 \ldots N]$ in nondescending order; that is, $A[i] \leq A[i+1]$ for all its indices $1 \leq i < N$.

  An item b to look for.

  Output: Either an index r such that $A[r] = b$ or 0 if b does not occurr in A.

  This is actually the *general* search problem.
- What makes it binary is the extra performance requirement that only $\mathcal{O}(\log N)$ computation steps should be used, but that is not part of the correctness criterion we are proving.

- We must show the following two claims:
  1. "If b occurrs in A, then it occurs in the subpart $A[l \ldots u]$."

     This is called a *loop invariant* because it must remain true throughout the **while** loop, even though the lower and upper indices l and u change.
  2. "This subpart must get strictly smaller on each round of the **while** loop."

     This is needed to see that the **while** loop does not keep on running forever.
- We must check claim 1 at certain crucial points:

```
1  $l \leftarrow 1$; $u \leftarrow N$;
2  while $l < u$
3        do $m \leftarrow (l + u) \operatorname{div} 2$
4           if $b \leq A[m]$
5              then $u \leftarrow m$
6              else $l \leftarrow m + 1$
7  $r \leftarrow$ if $b = A[u]$ then $u$ else $0$.
```

Figure 1: Binary Search

- Just before the **while** loop ("on line $1\frac{1}{2}$") to see that it holds before the first execution of the loop body.
- Just after the body (on line $6\frac{1}{2}$) to see that it continues to hold even after giving l and u *new values* during the body.

  However, here we can (and must) *assume*

  1. the loop condition $l < u$, because we are inside the loop, and
  2. claim 1 itself

  for the *old* values for l and u (the ones which they had on line $2\frac{1}{2}$ just before the loop body).
- Assumption 2 is in turn justified by an *inductive* argument:
  - If the loop body has not yet been executed, then it holds by line $1\frac{1}{2}$.
  - If it has been executed, then it holds by line $6\frac{1}{2}$ of the preceding execution.

  Thus we see that iteration and induction are related.

These crucial points are the ones where we are about to test the loop condition to see whether we should execute its body.

- Let us check claim 1 for line 5, and leave the rest as exercises for the reader...
  - The **if** condition $b \leq A[m]$ can now be used as a third assumption, since it was tested on the way to this line.
  - If $b = A[m]$, then truncating the subpart to $A[l \ldots m]$ does not eliminate this occurrence of b, so the assignment $u \leftarrow m$ is indeed permitted in this case...

    ... except for a small detail: Can $m < l$? Then we would be eliminating this occurrence of b after all!

    Fortunately this cannot happen: Line 3 ensures that $l \leq m \leq u$. But this needs to be checked too.
  - The remaining case is $b < A[m]$. Here we must argue that the discarded subpart $A[m+1 \ldots u]$ cannot contain any occurrence of b. This in turn follows from the problem statement, which required that the input array A is nondescending.
- Thus on line 7 we have claim 1 (by these checks) and $l = u$ (because the loop is over). Hence claim 1 reduces to "if b occurs in A, then it occurs in the one-element subpart $A[u \ldots u]$", and this is indeed what the **if**-expression states...

  ... except for a special kind of input we forgot, and must leave as another exercise for the reader!

- After doing the omitted checks for claim 1 and 2, we indeed do have a correctness proof for our version binary search.
- However, even this small example showed how easy it is miss details and special cases.
- We also skipped over some uninteresting routine parts. These are identified in natural language proofs by such catchphrases as "follows from", "it is clear that",... and so on.

  But what if we skipped too much? What if it does *not* follow if we look more closely?
- Formal logic(s) can help here: The proof simply does not go through all the way, if some necessary details are missing, or if the next reasoning step does not follow from the ones before.
- What if the correctness is *not* clear to the *other* persons reading our proofs? If they are not convinced, then they will not use our code.
- Formal logic(s) can "help" here too, in the technical sense that each single reasoning step is short enough to be checked independently of the others. So the others can convince themselves slowly but surely...

2 Pre- and Postcondition Programming

- We present here a programming notation (= an abstract programming language, which is not implemented as such, but which is easy to transliterate into implemented languages) where logic plays a prominent dual node:

  Syntactically our programs combine code with logical formulas.

  These formulas express properties of the code, which we are obliged to prove, and entitled to use in other proofs.

  Semantically even the meanings of the code constructs are defined via logic (and not by referring to any particular execution model, say).

  More generally, programming language designers have adopted logical methods for specifying new languages and studying existing ones. The most comprehensive example of this programming language specification style is the definition for the functional language Standard ML (Milner et al., 1997).
- Our notation is the *Guarded Command Language (GCL)* initially introduced by the aforementioned Edsger W. Dijkstra. These lectures are based on its presentations in the books by Backhouse (2003, Chapters 9–13), Dromey (1989, Part 2) and Gries (1981, Part II).

2.1 States and Their Names

- The GCL approach models the *procedural* programming paradigm, where the elementary computation step is "modify the *working memory* and move to the next instruction".
- This idea naturally stems already from Turing.

- Also object-oriented programming is procedural in this sense: its fundamental step is to update the objects' data fields — inheritance etc. just lets us specify the correct update method more flexibly.

- Other paradigms choose their elementary steps differently:

  Functional: Replacing a function name with its definition. Formally, doing one so-called β reduction of the λ calculus (Sørensen and Urzyczyn, 2006, Chapter 1).

  Historically it is interesting to note that this idea was published already in the same year (1936) as the Turing machines, and they were both intended as answers to the same then open problem in mathematical logic.

  Logical: One step in a logical proof. The main language is Prolog (Clocksin and Mellish, 1987).

  Data-Flow: Compute the next value to the output channel using the current values in the input channels (without remembering what their previous values were).

  and so on.

Definition 1 (States)**.** A *state* is a function f from the set of all possible variable names into the set of all possible values for these variables. The value $f(x)$ is the contents of variable x at this state f of the computation.

The notation $f[x_1 \leftarrow v_1, x_2 \leftarrow v_2, x_3 \leftarrow v_3, \ldots, x_n \leftarrow v_n]$ stands for the state f' such that

$$f'(y) = \begin{cases} v_1 & \text{if } y \text{ is the variable name } x_1 \\ v_2 & \text{if } y \text{ is the variable name } x_2 \\ v_3 & \text{if } y \text{ is the variable name } x_3 \\ & \vdots \\ v_n & \text{if } y \text{ is the variable name } x_n \\ f(y) & \text{otherwise.} \end{cases}$$

In other words, f' is the state obtained from f by *assigning* the value v_1 into the variable x_1, v_2 into x_2, v_3 into x_3,... and v_n into x_n.

- We adopt the vector notation as a shorthand, so that this notation for f' can be abbreviated as $f[\vec{x} \leftarrow \vec{v}]$, and so on.

- Hence a state f and $f[\vec{x} \leftarrow \vec{v}]$ are mathematical abstractions for our intuitive concepts "the current contents of the working memory" and "store these v_is into those x_is".

- In this course, we restrict our attention to those values v_i which are not functions themselves.

  – Hence our programming language (GCL) and logic are *first-order* (except for arrays (Section 2.4.7)).

  – Here 0th order consists of this function-free data, and $(k+1)$th order of operations whose inputs and output are (at most) kth order.

  – E.g. numbers $0, 1, 2, \ldots$ are 0th order data, addition '+' is a 1st order function since it takes two numbers as input and produces a third number as its output, and comparison '<' is a 1st order predicate since it takes two numbers as input and produces a *Boolean* value as output and these are 0th order too.

  - This rules out some programming idioms, such as passing procedure names as parameters to other procedures, since that is a *higher*-order idiom.

- However, we want to consider whole *sets of states*, not single states: e.g. on line 5 of our informal binary search example (Section 1.3) we considered *all the infinitely many* different states f such that

$$\overbrace{f(l) < f(u)}^{\text{line 2}} \text{ and } \overbrace{f(m) = (f(l) + f(u)) \operatorname{div} 2}^{\text{line 3}} \text{ and}$$
$$\underbrace{f(b) \leq f(A)[f(m)]}_{\text{line 4}} \text{ and Claim 1 wrt. this } f$$

  (The part corresponding to line 4 reads "index the value of variable A, which is an array, with the value of variable m".)

- We can use logic for this purpose: the formula

$$(l < u) \wedge (m = (l + u) \operatorname{div} 2) \wedge (b \leq A[m]) \wedge (\text{Claim 1})$$

  can be considered to be a *name for* the set of all those states f for which this formula is true.

- I assume that you are already familiar with the standard definition of "the formula ϕ in *First Order Logic (FOL)* is true in the model $\mathcal{M}$ under the assignment θ". It is also called First Order *Predicate Calculus (FOPC)* or *Logic (FOPL)*.

  Here is just a quick reminder of its salient points:

$$\begin{aligned}
\mathcal{M} \models (\forall x.\varphi)\theta &\text{ iff } \mathcal{M} \models \varphi(\theta[x \leftarrow v]) \text{ for all possible values } v \text{ in } \mathcal{M} \\
\mathcal{M} \models (\varphi \wedge \psi)\theta &\text{ iff both } \mathcal{M} \models \varphi\theta \text{ and } \mathcal{M} \models \psi\theta \\
\mathcal{M} \models (\neg\varphi)\theta &\text{ iff it is not the case that } \mathcal{M} \models \varphi\theta
\end{aligned}$$

- We *evaluate* the truth value of an atomic φ using $\mathcal{M}$ and θ:

$$\mathcal{M} \models \varphi\theta \text{ iff its value } \operatorname{eval}_{\mathcal{M},\theta}(\varphi) \text{ equals `}\textsc{true}\text{'}$$

  where

$$\begin{aligned}
\operatorname{eval}_{\mathcal{M},\theta}(x) &= \theta(x) \text{ for variable names } x \\
\operatorname{eval}_{\mathcal{M},\theta}(c) &= \mathcal{M}(c) \text{ for constant names } c \\
&\qquad \text{like `7',`+',`<',\ldots} \\
\operatorname{eval}_{\mathcal{M},\theta}(g(h_1, \ldots, h_k)) &= \underbrace{\operatorname{eval}_{\mathcal{M},\theta}(g)}_{=\mathcal{M}(g)}(\operatorname{eval}_{\mathcal{M},\theta}(h_1), \ldots, \operatorname{eval}_{\mathcal{M},\theta}(h_k)).
\end{aligned}$$

- In programming, assignments θ and states f are exactly the same thing.

- The model $\mathcal{M}$ is in turn a mathematical abstraction for the predefined constants, functions and predicates that you can expect to find built into your programming language.

  - In contrast to usual FOL, we restrict our attention into just this one *intended* model $\mathcal{M}$.

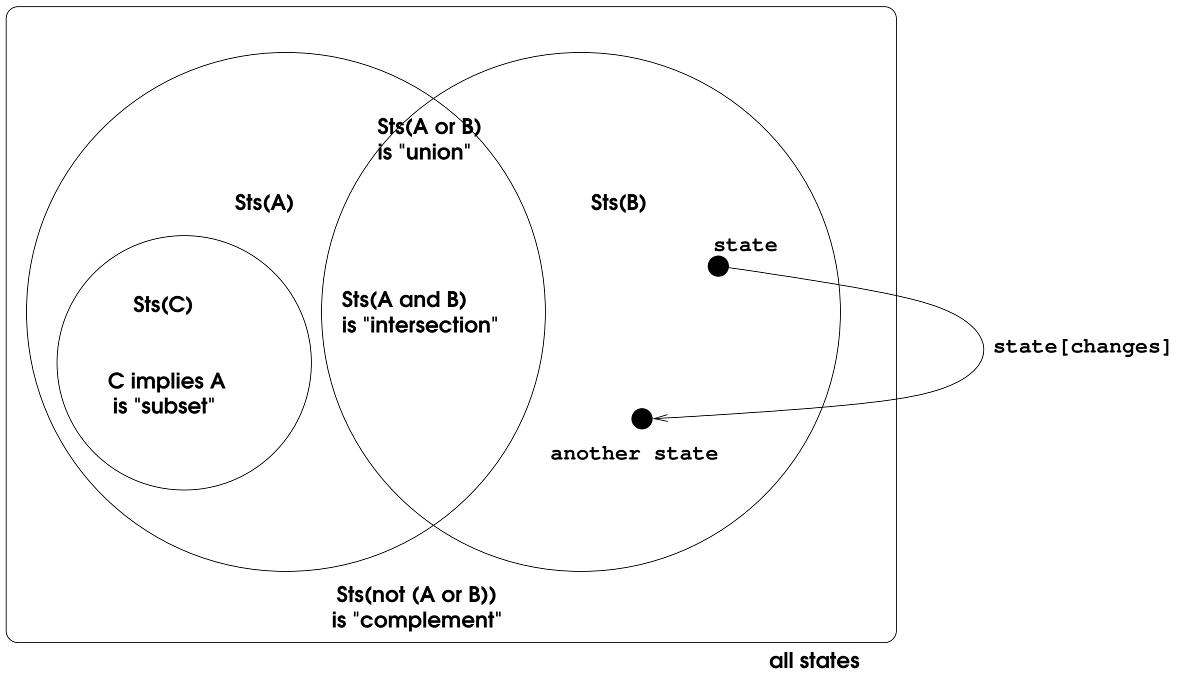

Figure 2: A Venn Diagram for States.

- To be completely formal, we should of course spell out the contents of our $\mathcal{M}$ explicitly…
- Let us instead just assume that $\mathcal{M}$ contains "all the usual math". We will especially need the *arithmetic and induction over the natural numbers* $\mathbb{N}$.

Definition 2 (Formula as a Set of States)**.** The FOL formula ϕ corresponds to the set

$$\mathrm{Sts}(\phi) = \{f \colon \mathcal{GCL} \models \phi f\}$$

of states f which satisfy ϕ in our intended model $\mathcal{GCL}$ which informally means "the GCL standard library" (whatever that turns out to be).

- Our formulae will have three kinds of variables:

  Bound variables and quantifies will be used for expressing generalized properties such as

  $$\forall i \colon \mathbb{N}.1 \leq i \land i < N \implies A[i] \leq A[i+1] \tag{1}$$

  which states that the input array A is nondescending in our binary search code (Figure 1).

  Program variables are **free** variables appearing *both* in the code and in the formulae. They stand for the current values of the corresponding mutable program variables (named storage locations).

  Ghost (also known as *rigid* or *auxiliary* or *initial*) variables are **free** variables appearing only in the formulae *but not* in the code. They must be used occasionally to remember the *past* values of variables before they were changed by the code.

  E.g. Claim 2 can be proved by proving "the *old* value of $u - l$ on line $3\frac{1}{2}$ was larger than its *new* value is on line $6\frac{1}{2}$". But the code itself does not compute the value of $u - l$ anywhere.

- The following short-circuiting Boolean connectives are very useful in programming:

  φ **cand** ψ is like the conjunction $\varphi \wedge \psi$, except that if its first part φ turns out to be FALSE, then the whole thing is FALSE, and its second part ψ is not even checked.

  E.g. $(1 \leq m < N)$ **cand**$(A[m] = b)$ would first check that m is a valid index for the array A, and hence avoid the "out of bounds" run-time error.

  φ **cor** ψ is the same for the disjunction $\varphi \wedge \psi$: if its first part φ turns out to be TRUE, then the whole thing is TRUE, and its second part ψ is not even checked.

  Hence they are in GCL too.

- Because they are so useful in programming, we add them into our FOL too. Technically, our logic becomes three-valued: TRUE, FALSE and UNDEFINED. Then FALSE **cand** UNDEFINED is FALSE, and so on. We skip the details.

2.2 The Hoare Triple

(They were proposed by the eminent British computer scientist sir C. Anthony R. Hoare, whose many contributions to computing such as the classic quicksort algorithm (Cormen et al., 2001, Chapter 7) even earned him his knighthood.)

- We need a connection between logical formulae and program code. For this we will use the so-called Hoare triples.
- For a practical programmer, this approach is similar to writing run-time sanity checks like the `assert` in the Java (Sestoft, 2005, Section 12.7) and C (Kernighan and Ritchie, 1988, Section B6) programming languages, except that these `assert`ions can be general FOL formulae.
- Then the proof amounts to checking statically (beforehand, "at compile time") that these asserted formulae are always TRUE and therefore these `assert`ions do not halt our program in a run-time error.

Definition 3 (The Hoare Triple)**.** The *Hoare triple* $\{\ \phi_{\text{pre}}\ \}S\{\ \phi_{\text{post}}\ \}$ is the claim that "if the program code S is started in an arbitrary initial state $f \in \text{Sts}(\phi_{\text{pre}})$ then its execution terminates normally and has reached some halting state $f' \in \text{Sts}(\phi_{\text{post}})$".

Precondition formula ϕ_{pre} describes when code S can be trusted to work correctly.

For our binary search (Figure 1) it would contain Formula (1).

Postcondition formula ϕ_{post} describes what the desired outcome of S is when it works correctly.

For our binary search code (Figure 1) it would be

$$(b = A[r])\,\mathbf{cor}(r = 0) \tag{2}$$

if we assume that $A[r] =$ UNDEFINED when r is not a valid index for array A.

Total correctness is required here: S must execute without run-time errors to the finish.

```
     { Formula (1) }
1    $l \leftarrow 1$; $u \leftarrow N$;
     { Formula (1) $\wedge$ Claim 1 which can be written as:
             $(\exists j : \mathbb{N}. 1 \le j \wedge j \le N \wedge b = A[j]) \Longrightarrow$
                 $(\exists j : \mathbb{N}. l \le j \wedge j \le u \wedge b = A[j])$ }
2    **while** $l < u$
           **do** { Formula (1) $\wedge$ Claim 1 $\wedge$ $(l < u)$ }
3             $m \leftarrow (l + u) \operatorname{div} 2$
4             **if** $b \le A[m]$
                **then** { Formula (1) $\wedge$ Claim 1 $\wedge$ $(l < u)$ $\wedge$
                      $m = (l + u) \operatorname{div} 2 \wedge b \le A[m]$ }
5                   $u \leftarrow m$
                    { Formula (1) $\wedge$ Claim 1 }
6               **else** $l \leftarrow m + 1$
              { Formula (1) $\wedge$ Claim 1 }
     { Formula (1) $\wedge$ Claim 1 $\wedge$ $\neg(l < u)$ }
7    $r \leftarrow$ **if** $b = A[u]$ **then** $u$ **else** $0$.
     { Formula (2) }
```

Figure 3: Our Binary Search with Some Assertions

Partial correctness would be "… and … then …": S could also terminate abnormally or loop forever.

Hoare's original notation (1960) was in fact $\phi_{\text{pre}}\{\ S\ \}\phi_{\text{post}}$ and meant partial correctness.

- The "arbitrary f" creates an implicit '$\forall$' for all the free (= program and ghost) variables over the whole triple.
- The idea is to
  1. first add assertions { … } as pre- and postcondition *brackets* around different parts of code (Figure 3)
  2. and then prove each of these resulting Hoare triples separately.
- Using *structured* programming means that this bracketing can be developed to mirror the structure of the code:
  - E.g. the pre- and postconditions around the body of the **while** loop (lines $2\frac{1}{2}$ and $6\frac{1}{3}$) can be determined from the *outside in* (until we get to assignments) from the pre- and postconditions surrounding the loop (lines $1\frac{1}{2}$ and $6\frac{2}{3}$).

    "The assertions line up with the indentation level of the code."
  - Conversely, the proofs of the resulting Hoare triples can proceed from the *inside out* (starting from the assignments):

    To prove the loop, prove its body; to prove its body, prove the branches of the **if** in the body, and so on.

  - E.g. `break`ing out from the middle of the loop is efficient and sometimes simplifies the loop *code* — but can complicate its *proof!* For instance, it bypasses testing the loop condition, so we cannot assume its negation after the loop.

- Let us now verify the Hoare triple for line 5.

  Here we proceed more formally than before (Section 1.3) by splitting our reasoning into so small formula manipulation steps that we can "see" that each step must be correct.

  1. We have $(l < u) \land m = (l + u) \operatorname{div} 2$ in the precondition. Straightforward arithmetic reasoning from them gives us $l \leq m \land m < u$.
  2. The result of Step 1 allow us to split the consequent of Claim 1 into two parts

     $$\overbrace{(\exists j : \mathbb{N}. l \leq j \land j \leq m \land b = A[j])}^{\text{low part}} \lor \underbrace{(\exists j : \mathbb{N}. m < j \land j \leq u \land b = A[j])}_{\text{high part}}.$$

     Without Step 1 one part might be empty and the other larger than $A[l \dots u]$, so this split formula would not mean the same as the original.
  3. We also have $b \leq A[m]$ — that is, $b = A[m] \lor b < A[m]$ — in the precondition. Consider these two cases separately:
     - If $b = A[m]$ then the low part is TRUE: Just choose $j = m$. Hence we have shown the formula

       $$(\text{antecedent of Claim 1}) \Longrightarrow \text{low part} \tag{3}$$

       from

       $$\underbrace{(l < u) \land m = (l + u) \operatorname{div} 2}_{\text{Used in Step 1,}} \land \underbrace{\text{Claim 1}}_{\text{Step 2,}} \land \underbrace{b = A[m]}_{\text{and here.}}.$$
     - If $b < A[m]$ then the high part is FALSE:

       $$b < A[m] \land \underbrace{A[m] \leq A[m+1]}_{\text{Formula (1) with } i = m} \text{ and so } b < A[m+1] \text{ too.}$$

       Then repeating this with $i = m + 1, m + 2, m + 3, \dots, u - 1$ (with a small induction) shows that for every j in the high part we have $b < A[j]$ and hence not $b = A[j]$. Hence we have shown the same Formula (3) from

       $$(l < u) \land m = (l + u) \operatorname{div} 2 \land \text{Claim 1} \land b < A[m] \land \text{Formula (1)}.$$
  4. By joining the two cases of Step 3 back together, we have shown Formula (3) from

     $$(l < u) \land m = (l + u) \operatorname{div} 2 \land \text{Claim 1} \land b \leq A[m] \land \text{Formula (1)}$$

     which is the precondition. Hence Formula (3) is also true at the precondition.
  5. Formula (3) is otherwise exactly like Claim 1, except that it has m in place of u.

     Since this m is the value assigned to u on line 5, we conclude that Claim 1 holds in its original form just after line 5.

We have thus verified one part of the postcondition, namely Claim 1.

- The other part of the postcondition is Formula (1). Intuitively, it holds because it holds for the precodition and line 5 does not change array A, since it does not even mention A.

  However, note that this assumes tacitly that the variable u is not one of the variables $A[1], A[2], A[3], \ldots, A[N]$. We shall later forbid such *aliasing* for this reason.

- *Developing* code proceeds from the outside in, but here assertions are known, the code unknown:

  1. We start with the input and output specifications:

     { Formula (1) }
     What code gets us from the assertion above to the one below?
     { Formula (2) }

  2. Programming experience might suggest us to try Claim 1:

     { Formula (1) }
     $l \leftarrow 1;\ u \leftarrow N;$
     { Formula (1) $\land$ Claim 1 }
     What code moves our l and u closer to each other?
     { Formula (1) $\land$ Claim 1 $\land \neg(l < u)$ }
     $r \leftarrow \textbf{if } b = A[u] \textbf{ then } u \textbf{ else } 0.$
     { Formula (2) }

     and so on.

  The details of the code can be extracted from the assertions.

- Code thus annotated with pre- and postcondition assertions is called a (Hoare) *tableau* (French; plural *tableaux*).

- This name might come from a method of constructing and presenting logical proofs.

  In this method, sets { ... } of formulae are decomposed into simpler sets using the rules of the logic in question, until we reach sets which are "self-evident" by the semantics of the logic.

  Then the proof can be collected by tracing back the decompositions made.

- Here we can analogically view the code parts as (names for) the decomposition steps which allowed us to get from the precondition into the postcondition in the specification.

  This view is similar to the "proofs-as-programs" view of the Curry-Howard correspondence (Section 1.2).

- Hoare triples and tableaux are not without problems, e.g.:

  – They have no way of saying "you can modify these variables, but *not* those".
  – They have no formal way of limiting *how* you can modify the output variables.
  – Ghost variables (Section 2.1) are often needed, but we have no formal rules for introducing them and getting rid of them when they are no longer needed.
    The issue here is how to limit the *scope* of a ghost variable:

        * It does not appear in the code, so the scope rules of the programming language cannot be used.
        * An FOL quantifier does have its own scope rule, but it applies only to a single formula. Here we need instead a scope which consists of several assertions and the code between them.

- Such problems could be solved by switching to *specification* statements instead. The book by Morgan (1994) is devoted to this approach. The solution for ghost variables (Morgan, 1994, Chapter 23.3.5) does not distribute nicely over disjunction (Theorem 9), though.
    - There programming becomes step-py-step *refinement* of the initially altogether unexecutable specification into a form which is altogether executable — that is, final program code.
    - Hence that specification refinement approach is "top-down software engineering" while this Hoare triple approach is "bottom-up programming" — a logical treatment for the good habit of writing `assert`ions in the code.
    - Even in the specification refinement approach, the final program code must have a logical semantics, so that we can justify the refinement steps — hence Hoare triples or some analogous device is required there as well.